CPRtree: A Tree-based Checkpointing Architecture for Heterogeneous FPGA Computing

Hoang Gia Vu[1], Supasit Kajkamhaeng[1], Shinya Takamaeda-Yamazaki[2], Yasuhiko Nakashima[1]
[1] Graduate School of Information Science
Nara Institute of Science and Technology
Nara, Japan
E-mail: {vu.hoang_gia.uw9, supasit.kajkamhaeng.sd3, nakashim}@is.naist.jp
[2] Graduate School of Information Science and Technology
Hokkaido University
Hokkaido, Japan
E-mail: takamaeda@ist.hokudai.ac.jp

***Abstract*—FPGAs provide reconfigurability and high performance for parallel applications. Modern FPGAs can be integrated in computing systems as accelerators so that they can combine with host CPU to execute offload applications. This integration puts more pressure on the fault tolerance of computing systems and the question how to improve the dependability becomes crucial. Similar to CPU-based system, checkpoint/restart techniques are expected to be developed and applied to FPGA-based computing systems. There are two issues rising in this situation: how to checkpoint and restart FPGA, and how this checkpoint/restart model works well with the checkpoint/restart model of the whole computing system. In this paper, first we propose a new checkpoint/restart architecture along with a checkpointing mechanism on FPGA. Second, we propose "fine-grain" management for checkpointing to reduce performance degradation. Third, we propose a technique to capture consistent snapshots of FPGA and the rest of the computing system. For host software, we also provide CPRtree stack including API functions to manage checkpoint/restart procedures on FPGA. Our experimental results show that the checkpointing architecture causes up to 9.73% maximum clock frequency degradation, small breakdown, and small data footprint, while the LUT overhead varies from 17.98% (Dijkstra) to 160.67% (Matrix Multiplication).**

Keywords—Checkpointing; FPGA; dependability; tree-based

I. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are expected to play a more important role in high performance computing system. They do not only provide reconfigurability and high performance for parallel applications, but also show great advantages of exploiting memory bandwidth to increase memory throughput and accelerate data-intensive applications. Therefore, the integration of FPGAs into high performance computing architectures becomes indispensable in the future. However, this trend compounds the problem of increasing failure rate because of growing size and complexity in the computing system [1, 2]. As a consequence, fault tolerance becomes more essential in FPGA operation. The most dominant technique used to deal with faults in CPU-based systems is checkpoint/restart, and this technique is also expected to improve the dependability of FPGA-based computing systems. There are two types of checkpointing on FPGA: user-level checkpointing and system-level checkpointing. While user-level checkpointing requires more effort from programmers to write additional code along with applications, system-level checkpointing is performed automatically by provided checkpointing infrastructure. Conversely, system-level checkpointing is predicted to be more complicated and consumes more hardware resource than user-level one. However, in this paper we choose to go forward system-level checkpointing to remove effort from programmers.

In system-level checkpointing, there are several approaches to exploit properties of automatic checkpointing, depending on where checkpointing infrastructure is inserted in the hardware design flow. First, checkpointing infrastructure can be written and inserted in high-level languages, such as C/C++, Java, or Python. There are many high-level synthesis tools, such as Vivado HLS and OpenCL, that can support to do so. Second, checkpointing infrastructure can be written and inserted in hardware description language (HDL), called *HDL-based checkpointing* in this paper. Third, checkpointing technique can be integrated in the hardware design flows at the netlist level as in [3]. Fourth, checkpointing technique can also be employed by using configuration tools to read back and then filter the configuration bitstream to get the values of flip-flops and RAMs used in the hardware [4, 5]. While the first approach shows an advantage of exploiting hardware abstract in high-level language, it requires knowledge in specific high level languages and specific tools as well. The third and the fourth approaches also depend much on tools and technology. For the most global and popular use, we choose HDL-based checkpointing to investigate.

However, to satisfy the properties of system-level checkpointing, the HDL-based checkpointing technique must cover all situations of hardware behavior, transparent to applications and technology, and portable across computing platforms. There are two issues rising in this situation. First, a common checkpointing mechanism is required along with rules to convert HDL source code from original source code to the source code with checkpoint/restart functionality. Second, the checkpointing technique itself must ensure that the checkpoint/restart model on FPGA must work well with the checkpoint/restart model of the whole computing system. Our main contributions in this work are as follows:

1) We propose CPRtree – a new architecture of hardware

2379-1896/16 $31.00 © 2016 IEEE
DOI 10.1109/CANDAR.2016.57

checkpointing along with a checkpointing mechanism that can be applied to any hardware structure. This architecture forms a checkpointing tree connecting all checkpointed elements to the checkpointing on-chip storage with a continuous flow.

2) We propose a technique to guarantee the consistent snapshots of FPGA and others in a computing system by managing communication channels between user logic and other components.

3) We propose a new concept: *reduced set of state-holding elements* that represents the full state of hardware operation. By capturing and restoring only this set of elements, FPGA operation can be resumed correctly.

4) We propose a checkpoint/restart (CPR) manager on FPGA that receives *"coarse-grain"* control from the host but provides *"fine-grain"* management for checkpoint/restart procedures. CPRtree stack – a software stack including API functions and library is also provided for *"coarse grain"* management from the host. The manager and the stack are also transparent to applications.

The rest of the paper is organized as follows: Section II presents challenges of FPGA checkpointing. Section III describes CPRtree: checkpointing architecture for FPGAs. Section IV explains the checkpoint/restart flow from the host side (software) to FPGA (hardware). Section V shows the evaluation. Section VI discusses related works. Conclusion is summarized in section VII.

II. Challenges

We assume a computing node with checkpointing hardware as in Fig. 1. This computing node model consists of a host CPU, an FPGA, a unified main memory, and a non-volatile storage device. When checkpointing, context on FPGA is written to the main memory before being copied to the non-volatile storage. Conversely, when restarting, context is copied from the non-volatile storage (disk) to the main memory before being read to FPGA and restored to state-holding elements, such as registers and RAMs. In HDL-based FPGA checkpointing, several challenges must be overcome to make a computing system checkpoint-able and restart-able. We summarize the challenges as follows:

How to define a checkpointing architecture on FPGAs. Original hardware may have an arbitrary structure from simple as a single module to complicated as a structure of many nested modules. State-holding elements located in modules may have arbitrary sizes or data widths. To provide a network model of checkpointing, which is transparent to structures, to capture and restore all state-holding elements is a challenges.

Saperating the operation of the checkpointing hardware and the user logic hardware. As the nature of a checkpoint/restart procedure, before the context is captured or restored, the user hardware must be paused. After checkpointing, the user hardware is resumed. To guarantee correct operation of the user hardware after resuming, the values of all signals, including wires, registers, and RAMs, must be kept unchanged after checkpointing. For example, in order to capture RAM content, the input address signal of RAM must be changed to read memory words in different addresses. After capturing, this address signal must be returned to the original value that it holds before capturing. This puts more contraints and complexity on the capturing and restoring circuits.

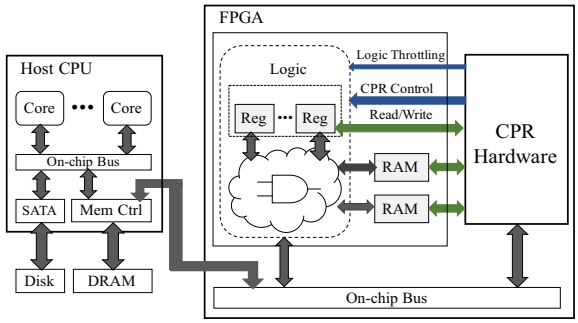


Fig. 1. Computing node with checkpointing hardware

Checkpointing HDL modules that will be synthesized as dedicated blocks. For dedicated blocks in which the ouputs are delayed responses to the inputs, such as BRAM and pipelined DSP blocks, we cannot insert HDL code to capture/restore the inside states. Instead, an analysis of the relationship between the states and the input values is required.

Ensuring that snapshots of FPGA is consistent with others. FPGA-based heterogeneous computing systems can be considered as distributed systems, in which there are distributed components, such as FPGA, off-chip memory, host CPU, that we cannot ensure that the states of all components will be taken at the same instant because they do not share a global clock. It is necessary to separate FPGA snapshot from the rest of system and ensure this snapshot together with snapshots of the host CPU and external memory form a consistent global state. A global state consists of states of the host CPU, FPGA, main memory, and states of communication channels between these components as well. A consistent global state must satisfy two properties. First, this state can be reached in the normal operation of the application. Second, the application can be restarted and resumed correctly from this state [6]. In case that the host snapshot is captured before the host sends a message to FPGA, and the FPGA snapshot is taken after receiving the message, then this pair of snapshots does not form a consistent global state. For another counter example, if the FPGA snapshot is captured after FPGA issuing a memory access request to the off-chip memory, and the content of the off-chip memory is captured before that, then this pair of snapshots cannot form a consistent snapshot, and it cannot be used to resume the application.

III. CPRtree: Checkpointing Architecture for FPGAs

Before discussing checkpointing architecture, the concept of context must be defined first.

A. Reduced Set of State-holding Elements

In order to checkpoint hardware, a set of elements defining context in HDL source code, called *set of state-holding*

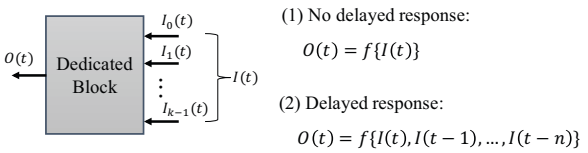


Fig. 2. Dedicated block

elements in this paper, must be determined. This set must satisfy the property: if only its all elements are recovered from a snapshot taken before, the operation of the hardware will be resumed correctly. It is noted that a set of all objects defined in the HDL source code, including all registers, RAMs, and wires, is a set of state-holding elements, and we call this set as a full set of state-holding elements. However, if an element in the set is interpolated from others, this element can be removed from the set to make a reduced set of state-holding elements.

There are three cases that wires can be removed from the set. First, wires as inputs from outside can be removed because the hardware is only checkpointed or restarted when these inputs are guaranteed to be inactive, thus the values of these inputs do not affect the operation of the hardware when checkpointing/restarting. Second, wires as outputs from a combinational circuit can be also removed because these outputs are interpolated from the inputs of the circuits. Third, wires as outputs of a checkpointed module can be removed as well, because these outputs are interpolated from the recovered inside of the module.

However, for wires as outputs of a module that is synthesized as a dedicated block as in Fig. 2, such as distributed RAM, Block RAM, and dedicated DSP, there are two cases depending on whether the outputs are delayed compared with the inputs or not. First, the outputs are immediate responses to the inputs. For example, the output data *O(t)* of a distributed RAM at the time *t* is a function of the corresponding input address *I(t)* at the time *t* without any delay. In this case, the output *O(t)* is immediately generated from the input *I(t)*, thus the output wire can be removed from the set of state-holding elements. Second, this case is more complicated that the outputs are delayed response to the inputs. For example, the output data of a block RAM is a one-clock-cycle delayed response to the corresponding input address. For another example, the output data of a dedicated four-stage pipelined multiplier is a four-clock-cycle delayed response to the inputs. In this case, the output *O(t)* cannot be interpolated from the current inputs *I(t)*, thus the output wires cannot be removed from the set. It is much more difficult to restore a value to a wire compared to a register. Even if the output *O(t)* is recovered before restarting, it cannot ensure that the output *O(t+1)* will hold the expected value since it depends on *I(t+1), I(t), I(t-1), ...,* and *I(t-(n-1))*. Furthermore, the values of *I(t-1), I(t-2), ...,* and *I(t-(n-1))* belong to previous snapshots of the hardware, and taking many consecutive snapshots for once checkpointing is not our expectation. To deal with this issue, this paper proposes a method to replace the output *O(t)* in the set of state-holding elements by adding registers to store values of *I* at the time *t-1, t-2, ..., t-n*. These

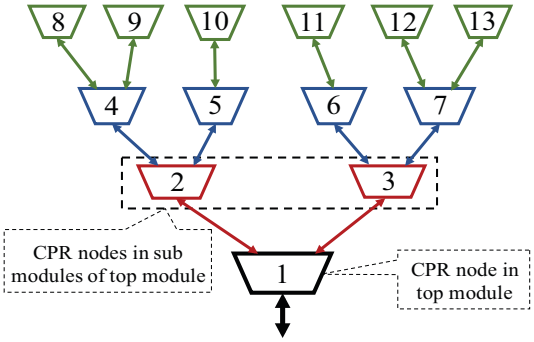


Fig. 3. Checkpointing tree

registers are called *additional registers* in this paper. However, to guarantee the normal operation of the hardware, the values stored in these registers must be restored to the input *I* at the consecutive times before resuming. Finally, all wires are removed from the set, and the set now includes only memory elements, such as user registers, additional registers, and RAMs. The set now is called *reduced set of state-holding elements*. Our proposal solves two problems. First, it presents a method to checkpoint dedicated blocks without need to capture multiple snapshots for once checkpointing. Second, it removes all wires from the set, thus its recovery becomes much simpler.

B. Checkpointing Architecture

It is noted that a structure of nested modules can be considered as a model of tree, in which the top module is the foot of the tree while sub-modules are branches of the tree. Therefore, a checkpointing architecture based on the model of tree is an approach to deal with complicated structures of nested modules. Each hardware module on FPGA has its own corresponding checkpoint/restart infrastructure, called *CPR node*, and the CPR nodes of all modules form a checkpointing tree as Fig. 3. In the figure, node 1 of the top module is called the next CPR level of node 2 and node 3, while node 4 and node 5 are called the previous CPR level of node 2, and node 6 and node 7 are the previous CPR level of node 3. In tree model, both capturing and restoring processes are performed sequentially through branches of the tree. This tree model is expected to reduce the data movement and energy consumption when capturing and restoring. The structure of CPR node is the same among modules in the user hardware and composed of parts: a CPR gate to the next CPR level, CPR interfaces with CPR nodes of the previous CPR level, context capturing/restoring circuits, and two CPR finite state machines (FSMs) – a capturing FSM and a restoring FSM. In another point of view, checkpointing hardware is divided into 2 parts: *static CPR hardware* that is the CPR gate of the top module, and the rest of the checkpointing tree, called *user-logic-based CPR hardware,* as showed in Fig. 4. The static part is fixed and independent from the user hardware, thus transparent to applications. Meanwhile, the user-logic-based part depends on the user hardware. To find out the rules to insert this part to user logic is one of our research purposes.

1) CPR Gate: CPR gate of all CPR nodes except the CPR

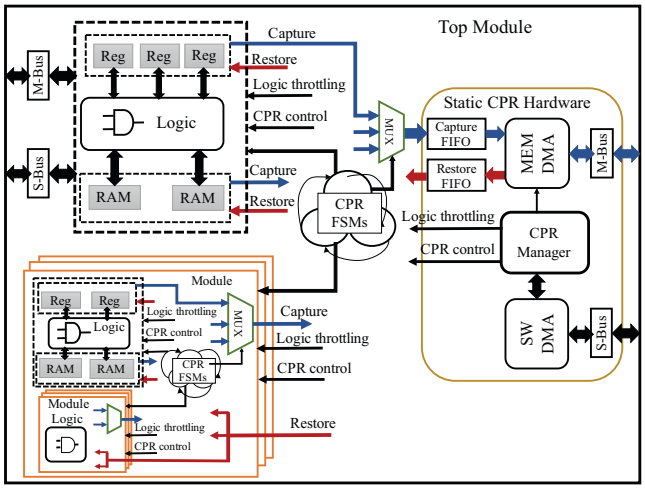


Fig. 4. Tree-based checkpointing architecture on FPGA

```
input [2:0]        CPR_request,
input              DRIVE,
output reg [2:0]   CPR_state,
input              cpr_out_almost_full,
input              capture_flag,
output reg [31:0]  D_cp,
output reg         D_cp_valid,
input [31:0]       Q_r,
input              Q_r_valid
```

Fig. 5. CPR gate

```
wire [2:0]   a_CPR_state;
reg          a_capture_flag;
reg [31:0]   a_D_r;
reg          a_D_r_valid;
wire [31:0]  a_Q_cp;
wire         a_Q_cp_valid;
```

Fig. 6. CPR interface

node of the top module is defined in Verilog HDL as in Fig. 5. The gate consists of a logic throttling signal – DRIVE as in [7], control signals, synchronous signals, and data signals for capturing and restoring.

It is noted that while the CPR gate described above is quite simple, structure of the CPR gate of the CPR node in the top module is much more complicated. This CPR gate is the static CPR hardware part as mentioned above. This part of checkpointing hardware is portable across platforms since it is fixed and does not depend on any parameter of the user hardware. This part includes: 1) *SW DMA* - a direct memory access (DMA) engine for AXI4-Lite protocol to communicate with the software in the host CPU via slave bus (S-Bus). 2) *Capture FIFO* – a FIFO to store checkpointing data captured from the user hardware. 3) *Restore FIFO* - a FIFO to store checkpointing data read from off-chip memory before restoring to the state-holding elements. 4) *MEM DMA* - a DMA engine for AXI4 protocol to write FPGA context from Capture FIFO to off-chip memory and read the context from off-chip memory to Restore FIFO via master bus (M-Bus). 5) *CPR Manager* - a checkpoint/restart (CPR) manager with functions as follows: a) Reading control code/writing status code and address of checkpoints stored in off-chip memory from/to SW DMA. b) Controlling MEM DMA to write and read checkpoints to/from off-chip memory. c) Throttling user logic to pause the application when checkpointing/restarting. d) Controlling checkpointing/restarting procedures. As in [8], using hardware core to manage CPR procedures provides considerable performance advantage over software-only methods, our CPR manager is also expected to improve the CPR performance over the direct control from the host.

Capture FIFO and Restore FIFO can be considered as on-chip storage for checkpoints on FPGA. Checkpointing process in a computing node now including 3 levels, called *multi-level checkpointing*: First, checkpoints are captured and written to the on-chip storage. Second, checkpoints in the on-chip storage are written to main memory. Third, checkpoints are copied from main memory to the non-volatile storage of the node. Since, a combination between multi-level and non-blocking checkpointing can benefit the performance of checkpointing [9], in our checkpointing architecture, FPGA does not wait until its all checkpoints are written to the non-volatile storage of the node, but resumes the normal operations immediately after the all checkpoints are written to Capture FIFO.

2) CPR Interface with the Previous CPR Level: As simple as the CPR gate in a module, a CPR interface consists of wires and registers to communicate with a CPR node of the previous CPR level. Fig. 6. shows the definition of CPR signals for a sub-module named "a", for example. This group of signals is mapped to corresponding signals of the CPR gate of the sub-module and does not include handshaking signals. Therefore, the checkpointing data movement is not interrupted by handshaking procedures.

3) Context Capturing/Restoring Circuit: As mentioned in the definition of the reduced set of state-holding elements, the context finally consists of registers and RAMs. In this paper, we propose methods to capture/restore registers and RAMs.

a) Register capturing/restoring circuit: It is assumed that there are *n* registers with arbitrary bit length: *Reg_0, Reg_1, ..., Reg_n-1*. To align the data in these registers with the 32-bit data width of checkpointing, these registers are concatenated and scaled again to form 32-bit registers: *Reg_0, Reg_1, ..., Reg_k-1*. It should be noted that the bit length of *Reg_k-1* may be less than *32* if the bit-length sum of the registers is not a multiple of *32*. We have two alternative approaches to capture/restore registers.

MUX-based capturing/restoring circuit: The values of these registers are assigned to D_cp (a buffer register of CPR gate) in consecutive states of the capturing FSM, and the values of Q_r (data wire from the next CPR level for restoring) are consecutively assigned to the registers in states of the restoring FSM. This, when synthesized, will generate a capturing circuit and a restoring circuit as in Fig. 7. In this case, the capturing circuit creates *k* 32-bit inputs more for the 32-bit multiplexer in front of D_cp. In addition, the restoring circuit creates one 32-bit input more for the 32-bit multiplexer in front of each register. Totally, *2k* 32-bit inputs are added to 32-bit multiplexers.

Shift-Reg-based capturing/restoring circuit: If the bit length of *Reg_k-1* is less than *32*, a padding register is inserted to guarantee the 32-bit data width of *Reg_k-1*. In the capturing circuit, the data in the *k* 32-bit registers is step by step shifted to the 32-bit multiplexer in front of D_cp as in Fig. 8. To satisfy the requirement that the values of registers are kept unchanged after capturing, the value of *Reg_0* is looped back

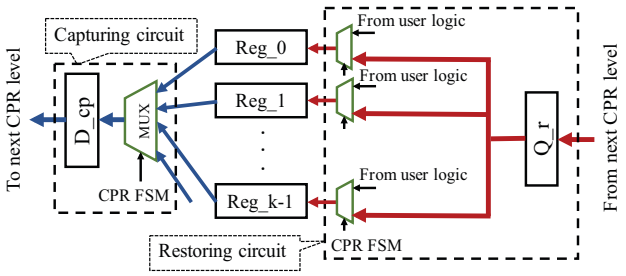


Fig. 7. MUX-based capturing/restoring circuit for registers

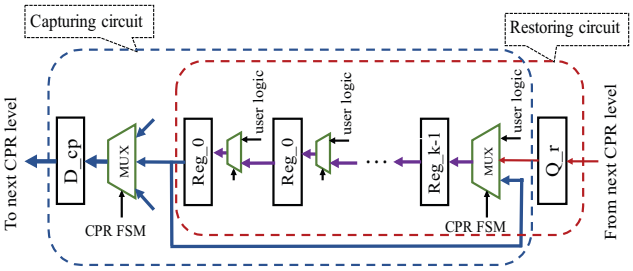


Fig. 8. Shift-Reg-based capturing/restoring circuit for registers

to the *Reg_k-1* via its input multiplexer. For the restoring circuit, context is consecutively shifted from Q_r to the all registers via 32-bit multiplexers. It is realized that the capturing circuit and the restoring circuit can share the register shifting circuit, thus saving hardware resource consumption, and we consider this as an advantage of this approach in this paper. In this case, one 32-bit input more is added to the 32-bit multiplexer in front of registers: D_cp, *Reg_0, Reg_1, ..., Reg_k-2*, while two 32-bit inputs more are added to the 32-bit multiplexer in front of *Reg_k-1*. Totally, *k+2* 32-bit inputs are added to 32-bit multiplexers.

When *k* equal to *1*, there is no shifting structure in the shifting circuit, thus these two circuits are the same. When *k* equal to *2,* the MUX-based circuit may be better than the Shift-Reg-based circuit in terms of resource consumption if a padding register is required. When *k* more than *2*, *2k* is more than *k+2*. Therefore, the Shift-Reg-based capturing/restoring circuit is expected to be better than the MUX-based circuit.

b) RAM capturing/restoring circuit: Fig. 9 shows how to add capturing/restoring circuit to the original RAM to make it checkpoint-able. Since the size of RAM can be determined in the HDL source code, the context of RAM can be captured and restored by iterating reading and writing through the whole its address space. Therefore, one port of RAM must be selected to read and write when capturing and restoring. However, the inputs of this port are expected unchanged after capturing to guarantee ability of resuming hardware, and sometimes this inputs are controlled from outside, not inside the module containing such RAM. For these reasons, instead of using a port of RAM directly to read and write, three registers: *we_0, addr_0,* and *wdata_0* are added along with the three signals: *write enable (we), address (addr),* and *write data (wdata)*, to control the port via multiplexers.

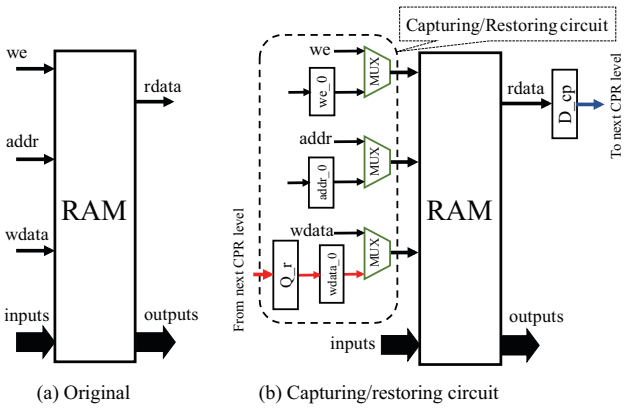


Fig. 9. Adding capturing/restoring circuit to RAM

4) CPR FSMs: The two CPR finite state machines (CPR FSMs) include one for capturing and the other for restoring. Both of the two FSMs are controlled by signals from the CPR manager and the next CPR level. There are several rules to design these two FSMs:

a) FSM for capturing: The FSM for capturing has two tasks. The first is to control the context capturing circuits of the current CPR node to assign the values of state-holding elements to the register D_cp of the CPR gate, and set the value of D_cp_valid to *'1'*. The second is to connect the previous CPR level to the next CPR level by copying the checkpointing data from the previous CPR level to the register D_cp, and set the value of D_cp_valid to *'1'*. The difference between the two tasks is about the condition to capture. While the first task requires Capture FIFO to have some rooms available, the second task ignores this condition to force the current CPR node to serve checkpointing data from the previous CPR level. In this case, to ensure Capture FIFO not overflowed when MEM DMA gets stuck, the guard gap of the signal *almost_full* from Capture FIFO should be more than the number of CPR levels in the user hardware.

b) FSM for restoring: This FSM also has two tasks but contrast to the FSM for capturing. The first is to control the context restoring circuits to get checkpoints from the next CPR level then restore to the state-holding elements. The second is to connect the next CPR level to the previous CPR level by copying checkpoints from Q_r to the CPR interfaces with the CPR nodes of the previous CPR level.

C. Consistent Snapshot with FPGA

This section answers the question mentioned in section I: How does the CPR model on FPGA work with the CPR model of the whole computing system? The answer is that the snapshot of FPGA must be consistent with the snapshot of the rest of the computing system to form a consistent global state. A global state of a distributed system is a set of component process and communication channel states [10, 11]. In order to get a global state, the states of all components and channels between them must be captured. Unfortunately, we cannot capture/restore the physical state of communication channels.

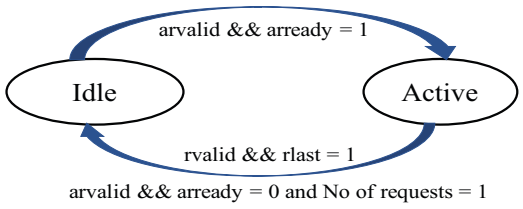
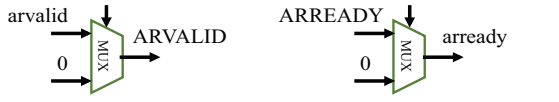
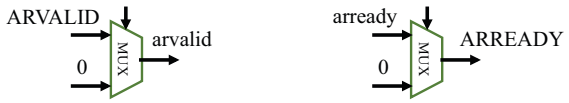


Fig. 10. Channel finite state machine



(a) Prevent issuing arvalid signal (b) Prevent receiving ARREADY signal

Fig. 11. Prevent issuing requests on the mater side



(a) Prevent receiving ARVALID signal (b) Prevent issuing arready signal

Fig. 12. Prevent receiving requests on the slave side

Therefore, the simplest way to make a consistent global state is to capture the states of all components when all communication channels are idle. In this case, the states of channels are all *empty*, and the global state now consists of only states of distributed components. However, this case rarely occurs because at the time a channel is idle, others may be active. In this paper, we propose a new concept named *virtual consistent global state,* in which all channels are idle. This global state is created by throttling channel requests and waiting until all channels become idle. It is noted that this throttling changes the flow of execution but does not change the execution result, thus this global state still satisfies two properties of a consistent global state mentioned in section II. To know the state of a channel to be idle or active, two finite state machines are required, called *channel finite state machines* (FSMs) in this paper. To throttle new requests, a unit is required to prevent issuing new requests on the master side, and prevent receiving new requests on the slave side of the channel, called *request throttling unit* in this paper. Since the most popular protocol used on FPGA to communicate with others is AXI4, it is chosen to illustrate operation of these two hardware classes.

1) Channel Finite State Machine: Fig. 10 shows a channel FSM for read transaction, the channel FSM for write transaction is similar. In this FSM, we use two pairs of signals: *arvalid & arready* and *rvalid & rlast*. In addition, we also use a register to count the number of read requests in the channel. The FSM is composed of two states: *Idle* and *Active*. The state will switch from *Idle* to *Active* if the condition *arvalid = arready = 1* is satisfied. In this case, the number of requests increase from *'0'* to *'1'*. Conversely, if both *rvalid* and *rlast* are equal to *'1'*, *arvalid* or *arready* are equal to *'0'*, and the number of requests is equal to *'1'*, the state will transit from *Active* to *Idle* and the number of requests will decrease from *'1'* to *'0'*.

2) Request Throttling Unit: For AXI4 protocol, we propose a method to prevent issuing new requests on the master side and prevent receiving requests on the slave side. In this method, the *arvalid, arready, awvalid,* and *awready* signals are fastened to *'0'*. The simplest way to do that is to use 2-to-1 multiplexers as showed in Fig. 11 and Fig. 12.

IV. Checkpoint/Restart flow

CPR procedures on FPGA are clock-cycle-level controlled by CPR Manager, while CPR Manager is controlled directly by the host. However, while the host only provides *"coarse-grain"* control through a simple software stack represented as an application programming interface (API) in Fig. 13, CPR Manager provides *"fine-grain"* management for CPR procedures on FPGA as showed in Fig. 14.

A. Prepare

Instead of pausing application and waiting for channels to be idle as in [6], the host calls *CPRtree_prepare()* and passes the allocated address of checkpointing data and a request command to CPR Manager. The address of checkpointing data is stored at a register in CPR Manager and used when writing FPGA context to or reading from off-chip memory. After that, CPR Manager throttles channel requests by preventing issuing requests on the master side and preventing receiving requests on the slave side of channels, then waits until all channels become idle. While waiting for all channels to be idle, user logic is still operating. Finally, CPR manager sends an acknowledgement back to the host to inform that FPGA is ready to be captured.

B. Capture

After the *prepare* procedure finishes, the host calls *CPRtree_capture()* to send a request command to CPR Manager. Then first, CPR Manager throttles logic to pause the operation of the original hardware. The system snapshot taken now is a virtual consistent global state because all channels have been idle. To capture context, CPR Manager issues a request to all checkpointing nodes of the checkpointing tree, and checkpointing data consecutively flows from branches of the tree to Capture FIFO before being written to off-chip memory. It is worth noting that CPR Manager does not wait until the FIFO is full, but issues a write request to off-chip memory immediately when realizing the FIFO is not empty. In addition, since the checkpointing tree does not use handshaking procedure between CPR levels, the delayed time of data flow in the tree is minimized. Therefore, it also minimizes the checkpointing time, which is defined as the total time from when the capture request is issued by host until the last word of context is written to off-chip memory.

C. Restore

When the host needs to restart FPGA operation from a saved image of checkpointing, it must complete the *prepare*

```
void CPRtree_prepare(int* cpr_data);
void CPRtree_capture();
void CPRtree_restore();
void CPRtree_wait(int cpr_n);
void CPRtree_resume();
void CPRtree_copy(int* cpr_data, char* context_path);
```

Fig. 13. CPRtree API

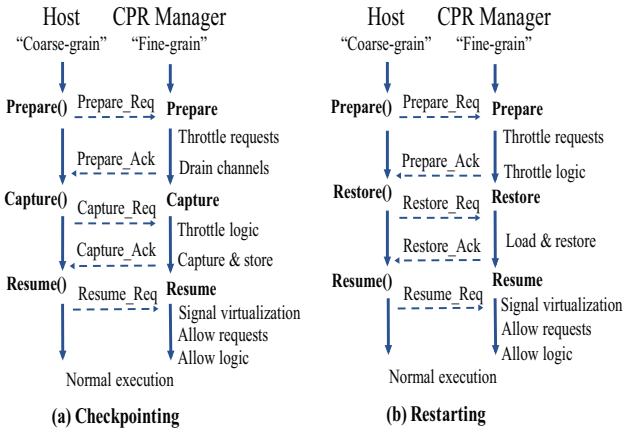


Fig. 14. Checkpointing/restarting timing diagram

procedure before starting to restore data. This procedure is required because communication channels must be idle and no request is issued to separate FPGA from other distributed processes. Furthermore, the allocated address must be passed from the host to FPGA to determine the location of the context in off-chip memory. Then the API function *CPRtree_restore()* is called, which sends a request command to CPR Manager. CPR Manager starts to restore by throttling logic and requesting MEM DMA to read the whole context data from off-chip memory. The data is then stored in Restore FIFO before being pumped to branches of the checkpointing tree. At the CPR nodes of the branches, checkpointing data is restored to state-holding elements, such as registers and RAMs. When the last word is restored to the corresponding elements, CPR Manager sets an acknowledgement to inform the host that the *restore* procedure is complete.

D. Resume

The *resume* procedure is used in both checkpointing and restarting processes. First, the host calls *CPRtree_resume()* to send a request command to CPR Manager. After that, CPR Manager virtualizes outputs of dedicated blocks by restoring the inputs in consecutive clock cycles before allowing channel requests and logic operation. It takes only one clock cycle to allow requests and logic operation, while the signal virtualization consumes a number of clock cycles equal to the number of clock cycles delayed between the outputs and the inputs of the dedicated blocks. Finally, after several clock cycles, the operation of the user hardware is resumed.

TABLE I. EXPERIMENTAL SETUP

EDA Tool	Vivado 2014.4
FPGA	Xilinx Zynq-7000 XC7z020clg484-1
Clock frequency	100 MHz
Host CPU	ARM Cortex-A9
Operating system	Debian 8.0

V. EVALUATION

Our experiments are set up as in TABLE I to evaluate hardware resource utilization, performance degradation, data footprint, and maximum clock frequency degradation caused by the proposed checkpointing architecture.

A. Hardware Resource Utilization

Since our checkpointing architecture is based on the model of tree with CPR nodes, to evaluate resource utilization of the checkpointing architecture, we evaluate resource utilization in nodes. The resource consumption in each node is mainly caused by circuits for capturing/restoring registers and RAMs.

1) Resource ultilization for capturing/restoring registers in a CPR node: We have two alternative methods to capture/restore as presented in Fig. 7 and Fig. 8. To compare the resource utilization of these two methods, we evaluate on two simple applications: *Sum* – sum of registers, and *Sum of Squares* – sum of squares of registers. Each application is written in a single Verilog HDL module. Fig. 15 shows the synthesis result in both applications. LUT consumption of the MUX-based circuit is higher than that of the Shift-Reg-based circuit when the number of registers more than *2*. It is explained that in the MUX-based circuit, *2k* inputs are added to multiplexers while in the Shift-Reg-based circuit, the corresponding number is only $k + 2$, with k is the number of 32-bit registers, as mentioned above. Therefore, the paper recommends that when the number of 32-bit registers is more than *2*, Shift-Reg-based circuit should be used.

Fig. 15 also reveals that the LUT utilization in *Sum of Squares* increases dramatically in both the MUX-based circuit and the Shift-Reg-based circuit, while the LUT utilization in *Sum* rises slightly in the MUX-based circuit and remains steady in the Shift-Reg-based circuit. The difference can be explained as follows. In *Sum*, each register has only one input pattern and an LUT1 is used for each bit of each register. When capturing/restoring circuits are added to the original hardware, one more input pattern is added for each register. Totally, each bit of these registers has two input patterns, thus an LUT3 is used to create an 1-bit 2-input multiplexer instead of LUT1. In this case, the number of used slice LUT is kept unchanged. Meanwhile, in *Sum of Squares*, each register has *3* input patterns and most of bits of registers have *3* input patterns. As a consequence, an LUT5 with 2-bit selector is employed for the 1-bit 3-input multiplexer for each of these bits. When one more input pattern is added to each register for checkpointing functionality, an additional LUT3 is used to make an 1-bit 2-input multiplexer. Totally, two slice LUTs,

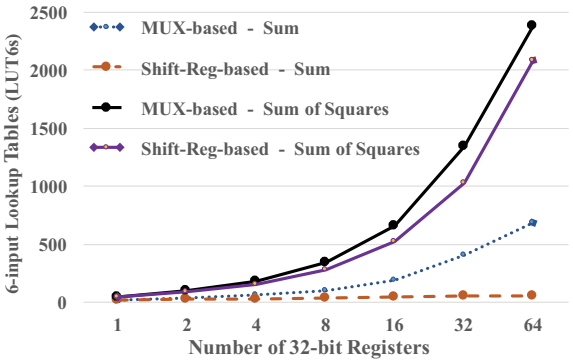


Fig. 15. LUT utilization for capturing/restoring registers

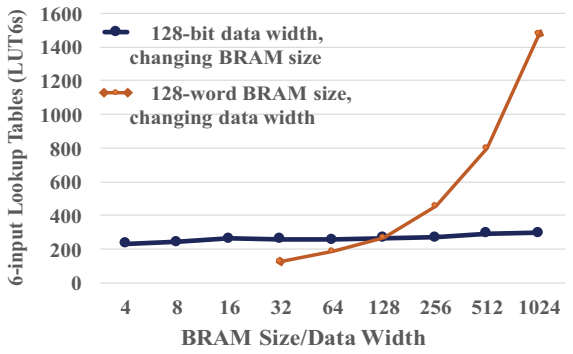


Fig. 16. LUT utilization of capturing/restoring BRAM

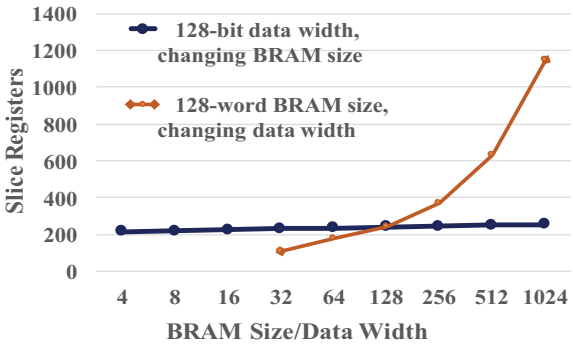


Fig. 17. Slice register utilization for checkpointing BRAM

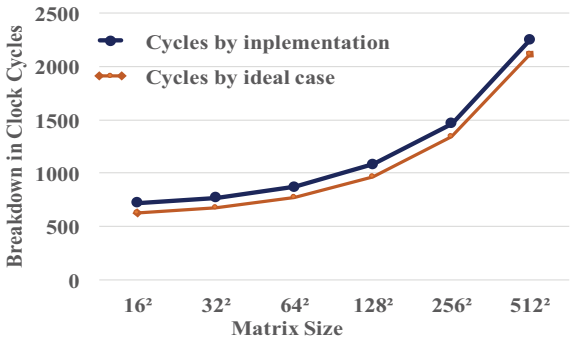


Fig. 18. Breakdown in checkpointing

including one LUT5 and one additional LUT3, are employed. That is the reason why the LUT utilization in *Sum of Squares* increases dramatically. For more optimal case, an LUT6 can be used to replace these two LUTs to create an 1-bit 4-input multiplexer. In this case, two more slice LUTs must be employed to generate the 2-bit selector of the 1-bit 4-input multiplexer from the 2-bit selector of the LUT5 and the 1-bit selector of the LUT3 , and this 2-bit selector may be shared with other 1-bit 4-input multiplexers. As a result, the slice LUT cunsumption is reduced significantly. The same optimization is achievable for the case that there are available inputs in slice LUTs to add more input patterns. For example, an input can be added to LUT1, LUT2, LUT3, LUT4, and LUT5 to become LUT2, LUT3, LUT4, LUT5, and LUT6, respectively, thus no additional slice LUT is used for checkpointing functionality. In short, the LUT utilization caused by checkpointing does not depend on how many LUTs consumed in the original hardware, but depend on how many registers used in the user logic and whether there is available input in LUTs used as multiplexers in front of registers or not.

Since registers are not duplicated in both the MUX-based circuit and the Shift-Reg-based circuit, the slice register utilization for checkpointing is insignificant. It includes only *32* slice registers for the 32-bit D_cp register and slice registers for counters and CPR finite state machines.

2) Resource utilization for capturing/restoring RAMs in a CPR node: We evaluate in two cases. First, BRAM size is kept at *128* words, and data width is changed. Second, data width is kept at *128* bits, while BRAM size varies. The synthesis results in Fig. 16 and Fig. 17 show that both the slice LUT and slice register utilization for checkpointing remains almost constant when BRAM size changes while data width is fixed. The very small increase in both LUT and register consumption is caused by using more bits for counters and the address of BRAM.

Conversely, the two figures also illustrate a dramatic increase in both LUT and register utilization when BRAM size is fixed at *128* words while data width increases. The increase is nearly linear with data width. The linear increase is because the LUT utilization comes from the multiplexer for the input data of BRAM and the input multiplexer for D_cp. These two multiplexers depend on the data width of BRAM in terms of data width of inputs and number of 32-bit inputs, respectively. Meanwhile, the increase in the register consumption is linear because the register consumption is mainly caused by the register *wdata_0*, which has the same data width as BRAM.

In summary, LUT consumption for checkpointing RAMs does not depend on RAM size but depends linearly on data width.

3) Resource utilization for the whole checkpointing tree: We apply the checkpointing mechanism to two realistic applications – pipelined SIMD matrix multiplication (Mat-Mul) and Dijkstra graph processing. The synthesis results are showed in TABLE II. As can be seen in the table, the static CPR hardware in both applications consumes small amounts

TABLE II. LUT UTILIZATION

Apps	LUTs	Additional LUTs (User-logic-based)	LUTs (Static CPR part)
Mat-Mul	3323	5339 (160.67 %)	2030 (3.73% avail.)
Dijkstra	8126	1461 (17.98 %)	1812 (3.4% avail.)

of slice LUTs compared with the total amount of the device. Since the design of the static CPR part is fixed and transparent to applications, the amount of LUTs consumed for this part is compared with the total amount of the device instead of the amount of utilized LUTs in the original hardware. The context in Mat-Mul includes *242* 32-bit registers and the total data width of used BRAMs is *885* bits. While these numbers in Dijkstra are much smaller, which are only *56* 32-bit registers and *448* bits, respectively. This explains why the LUT overhead (*160.67%*) in the user-logic-based part of Mat-Mul is much higher than that of Dijkstra (*17.98%*). The point of the table is to show that the LUTs consumed by checkpointing depends on the amount of registers used to define the context of application and the total data width of utilized RAMs.

B. Maximum Clock Frequency & Data Footprint

The synthesis results in Xilinx ISE 14.7 show that when adding checkpointing hardware to the two applications, the maximum clock frequency decreases from *115.075* MHz to *103.875* MHz (a decline of *9.733%*) for Mal-Mul and from *161.627* MHz to *161.589* MHz (a slight decline of *0.0235%*) for Dijkstra. This degradation is caused by inserting the throttling signal, other control signals for checkpointing, and capturing/restoring circuits. For checkpointing functionality, this degradation is acceptable.

Since our checkpointing mechanism is based on the definition of reduced set of state-holding elements, the data footprint in our mechanism is also reduced significantly compared with the readback method [4]. In Mat-Mul and Dijkstra applications, the readback method needs to read up to *4* Mbyte, while our method needs to read only *8.3* kbyte and *1.1* kbyte, respectively. As the reduced set of state-holding elements is realized as registers and RAMs, the data footprint for checkpointing is approximately the total amount of registers and RAMs used in application.

C. Performance Degradation

To evaluate performance degradation, we evaluate the breakdown caused by once checkpointing. In our method, the breakdown is the period of time user logic is throttled for capturing context. The breakdown is measured in Mat-Mul while scaling the matrix size. It is noted that changing the matrix size leads to changing the size of BRAMs, thus changing the amount of checkpointing data and data footprint. The ideal breakdown in clock cycles is defined as the number of 32-bit checkpointing words. Fig. 18 reveals that while scaling the matrix size, the breakdown is always higher than the ideal case but has the same shape. It is likely that the breakdown can be represented as: *breakdown = ideal case + C*, with *C* is a constant. Our experiments on Zedboard show that *C* is about *120* clock cycles (*0.12* us). The breakdown cannot reach to the ideal case because the constant *C* is the representative of the delayed off-chip memory request and delayed *"coarse-grain"* control from the host, thus cannot be removed. Therefore, the breakdown seems linear with the number of checkpointing words, thus linear with the amount of registers and RAMs used in the application.

VI. RELATED WORKS

While the concept of checkpointing is well known for software systems [12], checkpointing in hardware is under developed. For system-level checkpointing, software checkpointing is classified into two types: 1) *Library-based checkpointing* is portable across platforms and transparent to applications. A typical tool of this type is Distributed Multithreaded Checkpointing (DMTCP) [13]. 2) *Kernel-based checkpointing* is not portable across platforms but transparent to applications. BLCR [14] is a typical tool of this type.

For system-level FPGA checkpointing, some works has presented effective checkpoint/restart techniques on FPGAs to improve the dependability of FPGA computing. [4, 5] presented the readback method to read the configuration bitstream, and then filter the stream to get the state information. The report indicated that less than *8%* of the data in the bitstream is useful. The data footprint and performance were improved in [15] by reading only used frames of bitstream. However, the improvement was only about reading flip-flops while reading RAMs was under consideration.

The second approach is the scan-chain method as described in [3]. In this work, they introduced three methodologies to access the state of a hardware module: *Memory-Mapped State Access, Scan Chain based State Access,* and *Shadow Scan Chain based State Access*. The first and the second methodologies are quite similar to our two methods: MUX-based capturing/restoring circuit and Shift-Reg-based capturing/restoring circuit. The difference is that all these methodologies used tools to modify hardware modules at the netlist level while our methods insert checkpointing hardware at HDL level. Therefore, in their methodologies, new LUTs were inserted as multiplexers before flip-flops regardless of possibility of exploiting available inputs of LUTs. As a result, the LUT overhead in our experiments is smaller than the overhead estimation of their methodologies. Meanwhile, their third methodology duplicates all flip-flops of the original hardware to make a chain of additional flip-flops. As a consequence, the flip-flop overhead and LUT overhead increase dramatically while the checkpoint efficiency decreases much. In another work [16], scan-chain was also employed but by analyzing finite state machines, checkpoints were selected. As a result, instead of full scan-chain, only partial scan-chains were used to capture value of flip-flops. Therefore, the hardware overhead and data footprint decreased significantly. Scan-chain was also used in [17] to observe the state of full chip and to control internal signals, but not for checkpointing functionality. To quickly access any flip-flop in a design, they used multiple scan chains instead of a single scan chain to reduce the scan chain length. This scan chain model is similar to the methodology *Scan Chain based State*

Access mentioned above. However, all of these works focused on extracting state of flip-flops only but ignored state of on-chip RAMs, while on-chip RAMs are employed much in FPGA computing.

The third approach is the HDL-based method. In [18], the authors revealed methods to capture/restore state-holding elements, such as registers, BRAMs, finite state machines, and FIFOs by providing a context interface. However, when evaluating LUT utilization of additional hardware, they only evaluated LUT consumption in the context interface, while LUT consumption caused by multiplexers inserted along side with registers for restoring context was significant. They used the second port of BRAM as a dedicated port for checkpointing without considering that this port may be also utilized by users. Furthermore, they did not propose a particular architecture to deal with structures of nested modules, which is more complicated than checkpointing a single module.

VII. Conclusion

This paper has presented a new checkpointing architecture along with a checkpointing mechanism on FPGAs that is transparent to applications and portable across hardware platforms. For checkpoint/restart management, we provided *"fine-grain"* control to reduce delays in requests from the host, thus reduce performance degradation. We believe that this is the first work concerned about the two issues: how to checkpoint dedicated blocks with outputs delayed compared with inputs, and how to guarantee consistent snapshots of FPGA and others (host CPU and external memory). We solve these problems by proposing a new concept: reduced set of state-holding elements, and proposing a technique to manage state of communication channels and throttle channel requests. Our evaluation shows that checkpointing hardware causes up to *9.73%* degradation of maximum clock frequency, small data footprint, and small breakdown, while the LUT overhead varies from *17.98%* (Dijkstra) to *160.67%* (Matrix Multiplication).

Acknowledgment

This work is supported in part by Mazda foundation and JSPS KAKENHI Grant Number JP16K16026.

References

[1] Bianca Schroeder and Garth A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," IEEE transactions on Dependable and Secure Computing, VOL. 7, NO. 4, Oct-Dec 2010.

[2] F. Cappello, Al Geist, W. Gropp, S. Kale, B. Kramer, M. Snir, "Toward Exascale Resillience – 2014 Update," Journal of Supercomputing Frontiers and Innovations, Vol. 1, No. 1, 2014.

[3] Dirk Koch, Christian Haubelt and J¨urgen Teich, "Efficient Hardware Checkpointing - Concepts, Overhead Analysis, and Implementation," FPGA'07, pp.188-196, February 18–20, 2007, Monterey, California, USA.

[4] H. Kalte and M. Porrmann, "Context Saving and Restoring for Multitasking in Reconfigurable Systems," International Conference on Field Programmable Logic and Applications, pp. 223-228, 2005.

[5] I H. Simmler, L. Levinson, and R. Manner, "Multitasking on FPGA Coprocessors," In Proceedings of the 10rd International Conference on Field Programmable Logic and Application (FPL'00), pages 121–130, 2000.

[6] Arash Rezaei, Giuseppe Coviello, Cheng-Hong Li, Srimat Chakradhar, and Frank Mueller, "Snapify: Capturing Snapshots of Offload Applications on Xeon Phi Manycore Processors," HPDC'14, June 23–27, Vancouver, BC, Canada.

[7] Shinya Takamaeda-Yamazaki and Kenji Kise, "A Framework for Efficient Rapid Prototyping by Virtually Enlarging FPGA Resources," 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig 2014), December 2014.

[8] Ashwin A. Mendon, Ron Sass, Zachary K. Baker, and Justin L. Tripp, "Design and Implementation of a Hardware Checkpoint/Restart Core," 2012 IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops (DSN-W).

[9] Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka, "Design and Modeling of a Non-blocking Checkpointing System," SC12, November 10-16, 2012.

[10] K. Mani Chandy and Leslie Lamport, "Distributed snapshots: Determining global states of distributed systems," ACM Transactions on Computer Systems, Volume 3 Issue 1: 63-75, Feb. 1985.

[11] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," IEEE trans. on Software Engineering, SE-13(1): 23-31, Jan. 1987.

[12] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," ACM Comput. Surv., 34(3), 2002.

[13] Jason Ansel, Kapil Arya, and Gene Cooperman, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop," Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on 23-29 May 2009.

[14] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux cluster," in Proceedings of SciDAC, 2006.

[15] Aurelio Morales-Villanueva and Ann Gordon-Ross, "On-chip Context Save and Restore of Hardware Tasks on Partially Reconfigurable FPGAs," FCCM 2013, pp.61-64.

[16] Alban Bourge, Olivier Muller and Frederic Rousseau, "Automatic High-Level Hardware Checkpoint Selection for Reconfigurable Systems," FCCM 2015, pp.155-158.

[17] Iakovos Mavroidis, Ioannis Mavroidis, and Ioannis Papaefstathiou, "Accelerating Emulation and Providing Full Chip Observability and Controlability," IEEE Design & Test of Computers, Dec. 2009.

[18] Andrew G. Schmidt, Bin Huang, Ron Sass, and Matthew French, "Checkpoint/Restart and Beyond: Resilient High Performance Computing with FPGAs," FCCM 2011, pp.162-169.