Vol.2017-ARC-224 No.14
Vol.2017-SLDM-178 No.14
2017/1/24

A Framework for Tree-based Checkpointing Architecture on FPGAs

HOANG GIA VU[†1] SHINYA TAKAMAEDA-YAMAZAKI[†2]
TAKASHI NAKADA[†1] YASUHIKO NAKASHIMA[†1]

Abstract: The integration of FPGAs into computing systems 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 to automatically generate checkpointing infrastructure for applications in order to reduce programmer effort. In this paper, first we present a checkpoint/restart architecture for FPGA-based computing. Second, we provide a Python-based framework to generate checkpointing functionality for applications. Our experimental results show that the checkpointing functionality generated by the framework causes less than 9.73% maximum clock frequency degradation, while the LUT overhead varies from 5.92 % (Dijkstra) to 147.07 % (Matrix Multiplication).

Keywords: FPGA, Framework, Tree-based Checkpointing

1. 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. Second, a software tool to convert HDL source code from original source code to the source code with checkpoint/restart functionality also need to be developed. Our main contributions in this work are as follows:

1) We present a tree-based architecture for hardware checkpointing along with a checkpointing mechanism that are transparent to hardware structure.

2) We provide a Python-based framework to analyze the original Verilog HDL source code and insert checkpointing functionality.

The rest of the paper is organized as follows: Section II describes a tree-based checkpointing architecture for FPGAs. Section III presents the Python-based framework for checkpointing insertion. Section IV shows the evaluation. Section V discusses related works. Conclusion is summarized in section VI.

†1 Nara Institute of Science and Technology
†2 Hokkaido University

2. Tree-based Checkpointing Architecture

2.1 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. 1. 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. 2. 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.

2.1.1 *CPR Gate*: CPR gate of all CPR nodes except the CPR node of the top module is defined in Verilog HDL as in Fig. 3. 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

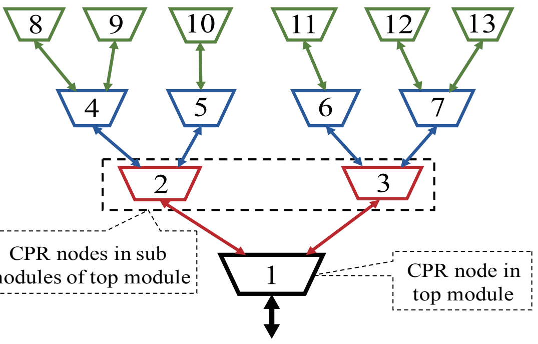

Fig. 1. Checkpointing tree

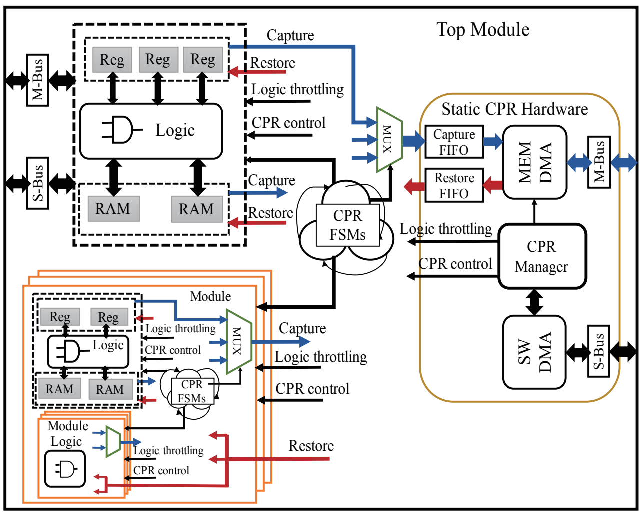

Fig. 2. Tree-based checkpointing architecture on FPGA

```
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. 3. 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. 4. CPR interface

pause the application when checkpoint/restart. d) Controlling checkpoint/restart 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.

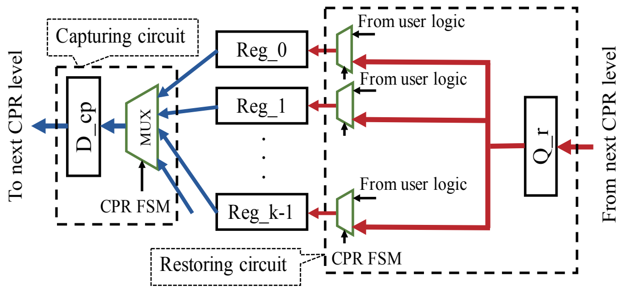

Fig. 5. MUX-based capturing/restoring circuit for registers

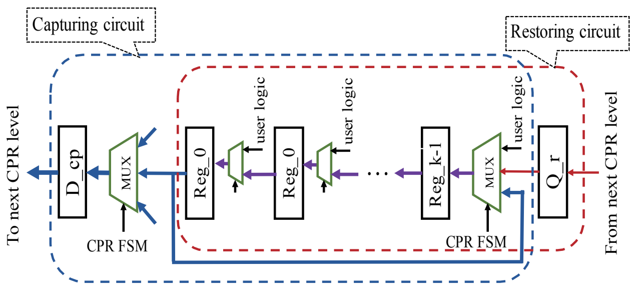

Fig. 6. Shift-Reg-based capturing/restoring circuit for registers

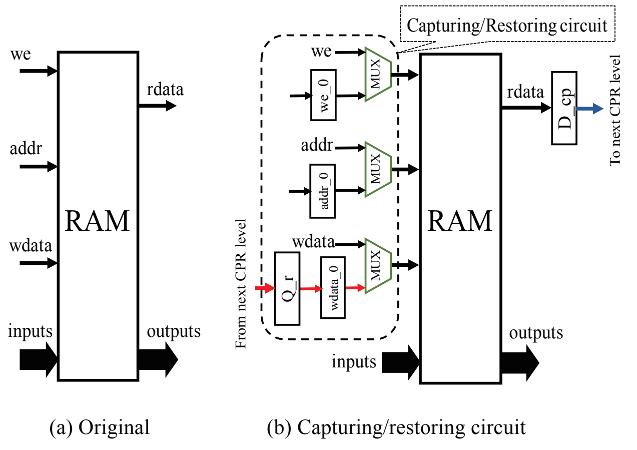

Fig. 7. Adding capturing/restoring circuit to RAM

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.

2.1.2 *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. 4. 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.

2.1.3 *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.

2.1.3.1 *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. 5. 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. 6. To satisfy the requirement that the values of registers are kept unchanged after capturing, the value of Reg_0 is looped back 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.

2.1.3.2 *RAM capturing/restoring circuit*: Fig. 7 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.

2.1.4 *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:

2.1.4.1 *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.

2.1.4.2 *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.

2.2 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. 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

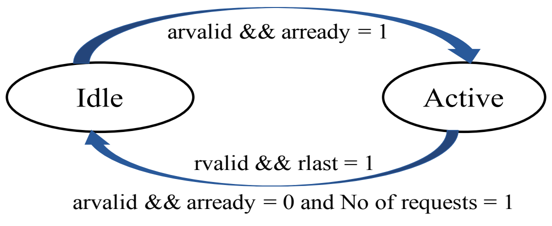

Fig. 8. Channel finite state machine

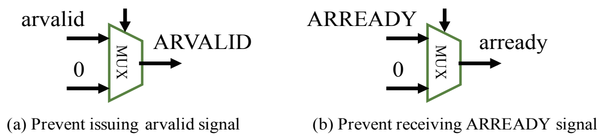
arvalid
0
MUX
ARVALID
(a) Prevent issuing arvalid signal
ARREADY
0
MUX
arready
(b) Prevent receiving ARREADY signal

Fig. 9. Prevent issuing requests on the mater side

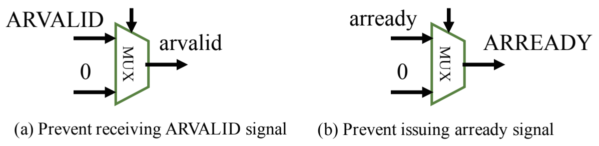
ARVALID
0
MUX
arvalid
(a) Prevent receiving ARVALID signal
arready
0
MUX
ARREADY
(b) Prevent issuing arready signal

Fig. 10. Prevent receiving requests on the slave side

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.

2.2.1 *Channel Finite State Machine*: Fig. 8 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'.

2.2.2 *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. 9 and Fig. 10.

3. Framework for FPGA Checkpointing

3.1 Proposed design flow

Since we chose HDL-based checkpointing to investigate, the

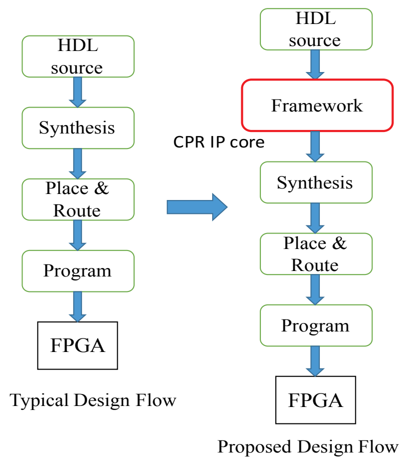

Fig. 11. Modification in design flow

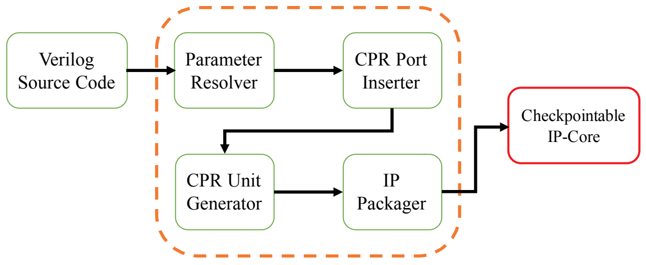

Fig. 12. Structure of the framework

framework must be inserted before synthesis in the proposed design flow as in Fig.11. The input of the framework is Verilog source code. Due to the location of the framework in the design flow, our checkpointing methodology is portable across hardware platforms and not dependent on technology.

3.2 Structure of the framework

The structure of the framework includes four blocks. The first is Parameter Resolver. This block analyzes the input Verilog source code, then produces the abstract syntax tree of the source code. Parameters from the source code are also abstracted and resolved. The second is CPR Port Inserter. This block inserts a CPR gate as ports in each module to connect CPR levels. The third is CPR Unit Generator. This block is to modify always blocks and insert CPR finite state machines in each module. The fourth block is IP Packager. This block packages the Verilog source code with checkpointing functionality to create an IP core, called CPR IP cores. For checkpointing purpose, this section presents the two blocks: CPR Port Inserter and CPR Unit Generator.

3.2.1 CPR Port Inserter

The output source code is as in Fig.3. The input *CPR_request* is to request checkpointing mode. There are four modes: *prepare, capture, restore*, and *signal virtualization*. The input *DRIVE* is to throttle user logic, and inserted in always blocks. *CPR_state* inform the CPR manager about the progress of checkpointing procedures. *capture_flag* is to enable the capturing procedure in a module.

3.2.2 CPR Unit Generator

CPR Unit Generator includes three tasks. The first is to modify always blocks to insert register capturing/restoring circuits and throttling signals as in Fig.13. The second is to realize modules that will be synthesized as dedicated blocks, such as distributed RAM and block RAM. These modules should not be inserted CPR ports or modified always blocks. In case of RAMs, all parameters including *data width, address width,* and *signal groups* for ports are abstracted from the definition of the module. The third is to insert two CPR finite state machines, including one for capturing and the other for restoring. Fig.14 shows the CPR finite state machine for capturing, including both register capturing (blue) and BRAM capturing (red).

```
always @(posedge CLK) begin
    if(RST) begin
        cyclecount <= 0;
    end else if DRIVE begin
        if(state == 2) begin
            cyclecount <= 0;
        end else begin
            cyclecount <= cyclecount + 1;
        end
    end else if shift_enable begin
        cyclecount <= { computation_size[31:0] };
    end
end
```

Fig. 13. Modifying always block

```
if(CPR_request == 2) begin
  case(cp_state)
    0: begin
      if(capture_flag && !mem_cpr_out_almost_full) begin
        D_cp_valid <= 1;
        D_cp <= {ram_addr_0, ROOM_DEQ};
        restore_cnt <= restore_cnt + 1;
        if(restore_cnt == reg_cnt - 1) begin
          restore_cnt <= 0;
          cp_state <= 1;
        end
      end
    end
    1: begin
      if(capture_flag && !mem_cpr_out_almost_full) begin
        ram_addr_cpr_1 <= ram_addr_cpr_1 + 1;
        ram_deq <= 1;
      end
      if(ram_deq) begin
        D_cp_valid <= 1;
        D_cp <= ram_Q_cpr;
        if(ram_addr_cpr_1 == 0) begin
          CPR_state <= 1;
          cp_state <= 2;
        end
      end
    end
    2: begin
    end
  endcase
end
```

Fig. 14. Inserting CPR finite state machines

TABLE I. Experimental Setup

EDA Tool	Vivado 2014.4, ISE 147
FPGA	Xilinx Zynq-7000 XC7z020clg484-1
Clock frequency	100 MHz

TABLE II. LUT Utilization and Max Clock Frequency

Apps	Additional LUTs (handwriting)	Additional LUTs (framework)	Max Clock frequency (handwriting)	Max Clock frequency (framework)
Mat-Mul	160.67 %	147.07 %	103.875 MHz	103.875 MHz
Dijkstra	17.98 %	5.92 %	161.589 MHz	161.589 MHz

4. Evaluation

In Table II, our evaluation on two realistic applications: matrix multiplication (Mat-mul) and Dijkstra graph processing (Dijkstra) shows that checkpointing functionality generated by our framework consume less hardware resources than that from handwriting. In addition, the max clock frequency degradation is the same between by handwriting and by framework.

5. 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. We also provided a framework for a tree-based checkpointing architecture on FPGAs. Our experimental results show that the checkpointing functionality generated by the framework causes less than 9.73% maximum clock frequency degradation, while the LUT overhead varies from 5.92 % (Dijkstra) to 147.07 % (Matrix Multiplication).

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.

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