IEICE TRANS. INF. & SYST., VOL.E101–D, NO.2 FEBRUARY 2018

PAPER *Special Section on Reconfigurable Systems*

# A Tree-Based Checkpointing Architecture for the Dependability of FPGA Computing

**Hoang-Gia VU**[†a)], ***Nonmember***, **Shinya TAKAMAEDA-YAMAZAKI**[††], **Takashi NAKADA**[†], ***Members***, ***and*** **Yasuhiko NAKASHIMA**[†], ***Senior Member***

**SUMMARY** Modern FPGAs have been integrated in computing systems as accelerators for long running applications. This integration puts more pressure on the fault tolerance of computing systems, and the requirement for dependability becomes essential. As in the case of CPU-based system, checkpoint/restart techniques are also expected to improve the dependability of FPGA-based computing. Three issues arise in this situation: how to checkpoint and restart FPGAs, how well this checkpoint/restart model works with the checkpoint/restart model of the whole computing system, and how to build the model by a software tool. In this paper, we first present a new checkpoint/restart architecture along with a checkpointing mechanism on FPGAs. We then propose a method to capture consistent snapshots of FPGA and the rest of the computing system. Third, we provide "fine-grained" management for checkpointing to reduce performance degradation. For the host CPU, we also provide a stack which includes API functions to manage checkpoint/restart procedures on FPGAs. Fourth, we present a Python-based tool to insert checkpointing infrastructure. Experimental results show that the checkpointing architecture causes less than 10% maximum clock frequency degradation, low checkpointing latencies, small memory footprints, and small increases in power consumption, while the LUT overhead varies from 17.98% (Dijkstra) to 160.67% (Matrix Multiplication).

***key words:*** *checkpointing, FPGA, dependability, tree-based*

## 1. Introduction

Field Programmable Gate Arrays (FPGAs) have been integrated and played an important role in high performance computing thanks to their high reconfigurability, high performance for parallel applications, and fully customized hardware. However, this integration compounds the problem of increasing failure rate because of the growing size and complexity in the computing system [1], [2]. As a consequence, fault tolerance and resilience 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 languages (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 widest use and independence in technology, we chose to move ahead with the HDL-based checkpointing. In this work, we assume that the original HDL source code is pure, that means the source code consists of HDL statements only. It is assumed that the user hardware operates with single clock domain.

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 hardware platforms. To provide such properties, we make the following contributions:

1) We propose CPRtree - a new checkpointing architecture for FPGAs 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 new concept: reduced set of state-

Manuscript received May 2, 2017.
Manuscript revised July 31, 2017.
Manuscript publicized November 17, 2017.

[†]The authors are with the Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.
[††]The author is with the Hokkaido University, Sapporo-shi, 060–0808 Japan.
a) E-mail: vu.hoang_gia.uw9@is.naist.jp

DOI: 10.1587/transinf.2017RCP0010

Copyright © 2018 The Institute of Electronics, Information and Communication Engineers

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.

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

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

5) We introduce a Python-based tool to modify the user HDL source code and insert checkpointing infrastructure.

The first four contributions were presented at the Third International Symposium on Computing and Networking (CANDAR 2016) [6]. After that, a Python-based software to generate checkpointing source code was developed. In this paper, we introduce such Python-based tool and use this tool to demonstrate and evaluate the proposed architecture on two more realistic applications: 9-point Stencil Computation and String Search.

The rest of the paper is organized as follows: Section 2 discusses the challenges of HDL-based checkpointing. Section 3 presents CPRtree: checkpointing architecture for FPGAs. Section 4 describes the checkpoint/restart flow from the host side (software) to FPGA (hardware). Section 5 introduces the Python-based tool. Section 6 shows the evaluation. Section 7 discusses related works. Conclusion is summarized in Sect. 8.

## 2. 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 (disk). 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 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.

**Separating 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 constraints and complexity on the capturing and restoring circuits.

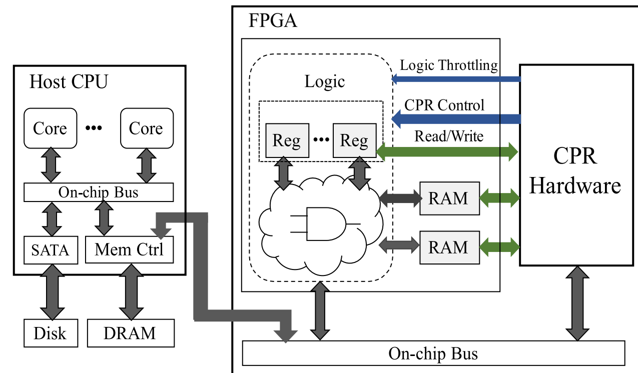


**Fig. 1** Computing node with checkpointing hardware.

**How to checkpoint 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 [7]. 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.

## 3. CPRtree: Checkpointing Architecture for FPGAs

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

### 3.1 Reduced Set of State-Holding Elements

In order to checkpoint hardware, the context of the application in HDL source code must be defined. In other words, a set of elements defining the state of the application, called set of state-holding elements in this paper, must be determined. This set must satisfy the following property: only if all of elements of the set are recovered from a snapshot priorly taken, 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 one element in the set is interpolated from any of the others, this element can be removed from the set to make a reduced set of state-holding elements.

There are three cases in which wires can be removed from the set. First, wires as inputs from the 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 also be 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, regarding wires as outputs of a module which 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 or not the outputs are delayed when compared with the inputs. In the first case, 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($t$), so the output wire can be removed from the set of state-holding elements. The second case is more complicated in that the outputs are delayed responses 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($t$), and thus the output wires cannot be removed from the set. It is much more difficult to restore a value to a

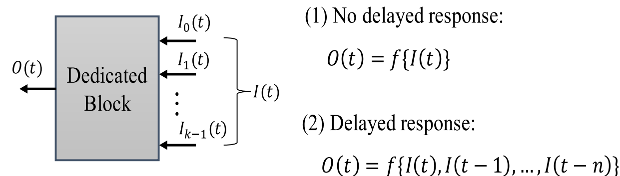


**Fig. 2** Dedicated block.

wire than to a register. Even if the output O($t$) is recovered before restarting, this 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 a single checkpointing is not our intention. 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 in order to store values of I at the time $t$-1, $t$-2, ..., $t$-$n$. These registers are called additional registers in this paper. If the data width of I is $w$, then the number of additional 1-bit registers required is $n * w$. However, to guarantee the normal operation of the hardware, the values stored in these registers must be restored to the input I at consecutive clock cycles before resuming the operation. It is noted that in normal operation, the values of the input I are copied to the additional registers. This does not affect the circuit behavior. In capturing process, the values of these additional registers are captured then written to the off-chip memory. This also has no impact on the behavior of the user circuit. However, in restoring process, the values stored in these registers are restored to the input I at consecutive clock cycles. Therefore, in order to guarantee the correctness of the circuit behavior, it is required to ensure that the restoring circuit has no impact on the input I in normal operation. We propose to use a multiplexer in front of the input I in order to separate the restoring circuit from the user circuit in normal operation. 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 is now 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 a single checkpointing. Second, our proposal removes all wires from the set, so that its recovery becomes much simpler.

### 3.2 Checkpointing Architecture

#### 3.2.1 Tree-Based Structure

It should be noted that a structure of nested modules can be considered as a model of a tree, in which the top module is the foot of the tree while sub-modules are nodes of the tree. Therefore, a checkpointing architecture based on the model of a tree is an approach to deal with complicated structure of nested modules. Each module has its own corresponding

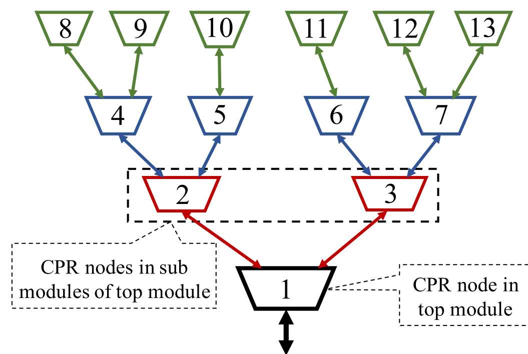


**Fig. 3** CPRtree.

checkpoint/restart infrastructure, called CPR node, and the CPR nodes of all modules form a checkpointing tree, as in Fig. 3. The structure of a CPR node is the same among modules in the user hardware and is composed of parts: a CPR gate to the next CPR level, CPR interfaces with CPR nodes of the previous CPR level, its own state-holding elements, context capturing/restoring circuits, and two CPR finite state machines (FSMs) - a capturing FSM and a restoring FSM. The tree-based structure is determined recursively according to the module structure of the original HDL source code. It starts from the CPR node of the top module (root node). In the original HDL source code if a module A includes $m$ sub-module instances $M_0, M_1, M_2, \ldots, M_{m-1}$, which are not synthesized as dedicated blocks, then the CPR node of module A is the parent node of the CPR nodes of module instances $M_0, M_1, M_2, \ldots, M_{m-1}$. In other words, the CPR nodes of module instances $M_0, M_1, M_2, \ldots, M_{m-1}$ are the children nodes of the CPR node of module A. It should be noted that $m$ can be an arbitrary value. Thus, the tree-based structure is not limited to binary trees. The CPR level of each node can be determined by the following rule. The CPR level of the root node is 0. If the CPR level of a CPR node is $n$, then the CPR level of each children node of this node is $n + 1$. Therefore, in Fig. 3 the CPR level of node 1 is 0. The CPR level of node 2 and 3 is 1. That of node 4, 5, 6, and 7, is 2. That of node 8, 9, 10, 11, 12, and 13, is 3. In the figure, node 1 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 order to connect two checkpointing infrastructures, only a connection "parent node" - "children node" is required. In other words, to link checkpointing infrastructures, only port insertion in modules is required. Therefore, it is believed that the tree is an efficient structure for dealing with nested modules.

#### 3.2.2 CPR Gate

For capturing path, checkpoints in a CPR node are moved from their node through their next CPR levels to the foot node before being written to the off-chip memory. For restoring path, in contrast, checkpoints are read from the off-chip memory to the foot node before being moved to

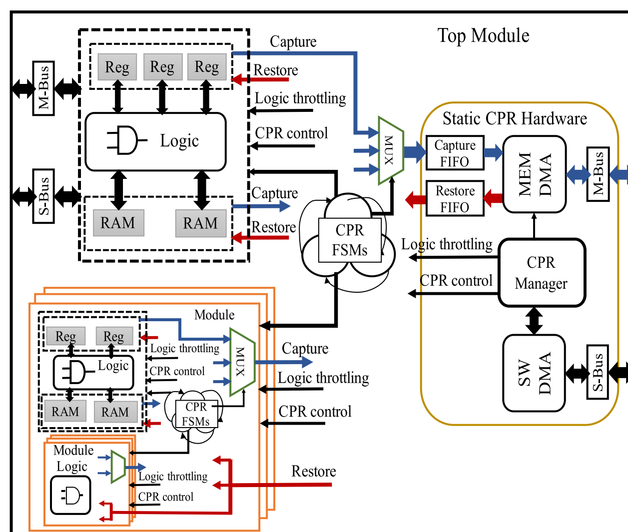


**Fig. 4** Tree-based checkpointing architecture on FPGA.

the corresponding node through its next CPR levels. After that, the checkpoints are assigned to the corresponding state-holding elements. For example, the capturing path and restoring path for node 5 are 5-2-1 and 1-2-5, respectively. The capturing path and restoring path for node 12 are 12-7-3-1 and 1-3-7-12, respectively. While in this tree model the capturing path for each node includes its next CPR levels only, the capturing path for each node in the scan chain method [3] includes all registers, thus including all nodes. As a result, the data movements when capturing in this tree model are less than that in the full scan chain. Therefore, this tree model, compared with the full scan chain method, is expected to reduce the energy consumption when capturing.

From another point of view, checkpointing hardware is divided into 2 parts: static CPR hardware, which is the CPR gate of the top module, and the rest of the checkpointing tree, called user-logic-based CPR hardware, as shown in Fig. 4. The static part is fixed and independent of the user hardware and thus transparent to applications. Meanwhile, the user-logic-based part depends on the user hardware. The user-logic-based part includes the state-holding elements of the foot node, such as registers and RAMs, the two CPR FSMs of the foot node, and other nodes as sub-modules. The figure also reveals the connectivity between a node and other nodes as its previous CPR level.

CPR gate of all CPR nodes except the CPR 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 [8], control signals, synchronous signals, and data signals for capturing and restoring. The logic throttling signal is used to pause sequential circuits, thus pausing the application for checkpointing.

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.

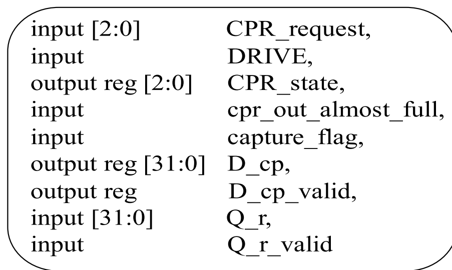


**Fig. 5** CPR gate.

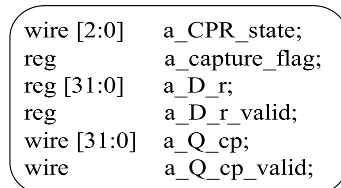


**Fig. 6** CPR interface.

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 checkpoint/restart procedures. As in [9], 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. It is noted that CPR manager does not wait until all checkpoints are stored in Capture FIFO, but issues a write request to the off-chip memory as soon as it detects that the FIFO is not empty. As a result, while checkpoints are being stored into the FIFO, the checkpoints already stored in the FIFO can be written to the off-chip memory. Therefore, a large size of the FIFO for holding all checkpoints is not required. In our implementation, we choose the size of 16 items for both Capture FIFO and Restore FIFO. This size is insignificant. 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 [10], 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.

#### 3.2.3 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. Figure 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.

#### 3.2.4 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 are 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

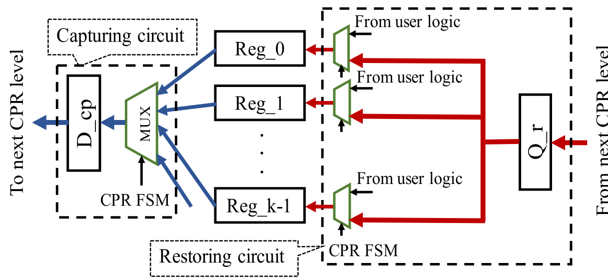


**Fig. 7** MUX-based capturing/restoring circuit for registers.

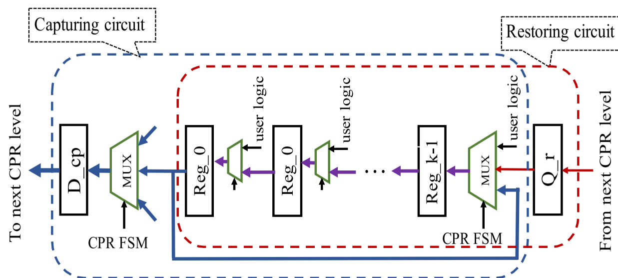


**Fig. 8** Shift-Reg-based capturing/restoring circuit for registers.

is looped back to the Reg_$k$-1 via its input multiplexer. For the restoring circuit, context is consecutively shifted from *Q_r* to all the 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 more 32-bit input is added to the 32-bit multiplexer in front of the registers: *D_cp*, Reg_0, Reg_1,..., Reg_$k$-2, while two more 32-bit inputs 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 $k$ is equal to 1, there is no shifting structure in the shifting circuit, thus these two circuits are the same. When $k$ is 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 $k$ is greater than 2, 2$k$ is greater 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*: Figure 9 shows how to add a 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 entire RAM 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 to maintain unchanged after capturing in order to guarantee the ability to resume the hardware, and sometimes these inputs are controlled from outside, not inside, the module containing this type of 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

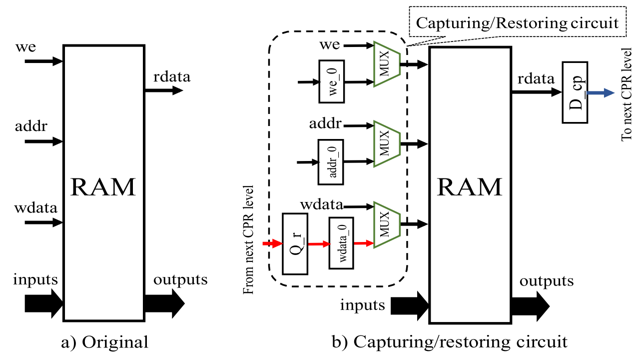


**Fig. 9** Adding capturing/restoring circuit to RAM.

three signals: write enable (*we*), address (*addr*), and write data (*wdata*), to control the port via multiplexers.

#### 3.2.5 CPR FSMs

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

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

*b) FSM for restoring*: This FSM also has two tasks, but the reverse of the FSM for capturing. The first task is to control the context-restoring circuits to get checkpoints from the next CPR level and then restore them to the state-holding elements. The second task 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.

### 3.3 Consistent Snapshot with FPGA

#### 3.3.1 Overview of Consistent Snapshot

This section answers the question mentioned in Sect. 1: How does the CPR model on FPGA work with the CPR model of the whole computing system? The answer is that

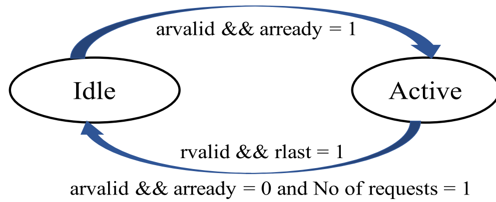


**Fig. 10** Channel finite state machine.

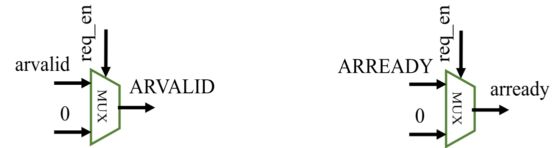


**Fig. 11** Preventing issuing requests on the mater side.

**Fig. 12** Preventing receiving requests on the slave side.

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 [11], [12]. 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 the 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 communication channels are idle. This virtual consistent global state is achieved by preventing the application on FPGA issuing/receiving new requests on all communication channels to/from other components, and waiting until all the channels become idle. This method of making all the channels idle is called request throttling in this paper. It is noted that this throttling changes the flow of execution but does not change the execution result, so that this global state still satisfies two properties of a consistent global state mentioned in Sect. 2. To know whether the state of a channel is idle or active, two finite state machines are required, called channel finite state machines (FSMs) in this paper. Since the most popular protocol used on FPGA to communicate with other components is AXI4, it is chosen to illustrate operation of these two hardware classes.

#### 3.3.2 Channel Finite State Machine

Figure 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 order to determine when a new read request is issued and when a read transaction finishes. Particularly, when *arvalid* = *arready* = '1', a new read request is issued. When *rvalid* = *rlast* = '1', a read transaction finishes. 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 increases 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'.

#### 3.3.3 Request Throttling

To throttle new requests, a control signal named *req_en* (request enable) from CPR manager is required to prevent issuing new requests on the master side, and to prevent receiving new requests on the slave side of the communication channel. In this section, we discuss request throttling for read transactions, the mechanism for write transactions is similar. It is noted that the lower-case letters, *arvalid* and *arready*, are signals used in user circuit, whereas the upper-case letters, *ARVALID* and *ARREADY*, are signals on the side of communication channels. When the user application on FPGA plays a role as master side in a communication channel, the signal *ARVALID* must not be asserted during a request throttling period. Therefore, we propose to use a multiplexer in order to fasten this signal to '0' as in Fig. 11. When *req_en* = '1', *ARVALID* = *arvalid*. When *req_en* = '0', *ARVALID* = '0'. At the same time *arready* should be also kept at '0' in order to ensure that the user circuit does not receive an acknowledgement on the channel. As a result, another multiplexer with the same control signal is employed. Similarly, on the slave side, *arvalid* and *ARREADY* must be fastened to '0' during the request throttling period as in Fig. 12.

## 4. 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-grained" control through a simple software stack represented as an application programming interface (API) in Fig. 13, CPR Manager provides "fine-grained" management for CPR procedures on FPGAs, as shown in Fig. 14.

```
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.

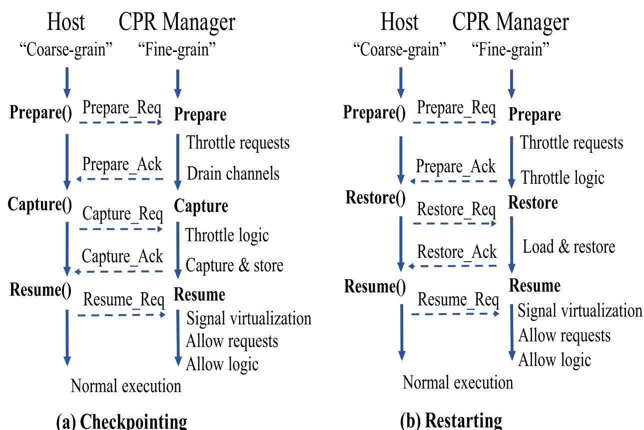


**Fig. 14** Checkpointing/restarting timing diagram.

### 4.1 Prepare

Instead of pausing an application and waiting for channels to be idle, as in [7], the host calls CPRtree_prepare() and passes the allocated address of checkpointing data and a request command to CPR Manager. The address of the 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 the issuing of requests on the master side and preventing the receiving of requests on the slave side of channels and then waits until all channels become idle. While CPR Manager is waiting, the user logic is still operating. Finally, CPR manager sends an acknowledgement back to the host to inform that the user logic is ready to be captured.

### 4.2 Capture

After the prepare procedure finishes, the host calls CPRtree_capture() to send a request command to CPR Manager. Then, first of all, CPR Manager throttles logic to pause the operation of the original hardware. The system snapshot taken is now a virtual consistent global state because all channels have become idle. To capture context, CPR Manager issues a request to all checkpointing nodes of the checkpointing tree, and checkpointing data flow continuously from nodes 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 upon detecting that the FIFO is not empty. In addition, since the checkpointing tree does not use a handshaking procedure between CPR levels, the delay in the data flow in the tree is minimized. This therefore 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 the context is written to off-chip memory.

### 4.3 Restore

When the host needs to restart the FPGA operation from the most recent saved snapshot of checkpointing, it must complete the prepare procedure before starting to restore data. This procedure is required because, in order to separate FPGA from other distributed processes, communication channels must be idle and no request can be issued. 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 the context by throttling logic and requesting MEM DMA to read all the context data from off-chip memory. The data are then stored in Restore FIFO before being pumped to nodes of the checkpointing tree. At the CPR nodes, checkpointing data is restored to stateholding 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.

### 4.4 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, whereas the signal virtualization consumes the number of clock cycles equal to the number of clock cycles of delay between the outputs and the inputs of the dedicated blocks. Finally, after several clock cycles, the operation of the user hardware is resumed.

## 5. Tool for Checkpointing Insertion

### 5.1 Proposed Design Flow

Since we chose HDL-based checkpointing to investigate, the tool must be inserted before synthesis in the proposed design flow as in Fig. 15. The input of the tool is Verilog source code. Due to the location of the tool in the design flow, our checkpointing methodology is portable across hardware platforms and not dependent on technology.

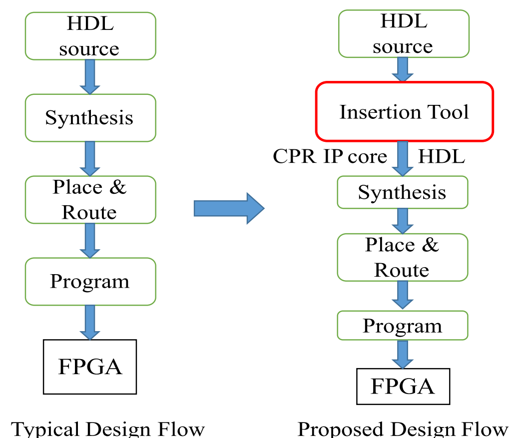


**Fig. 15** Modification in design flow.

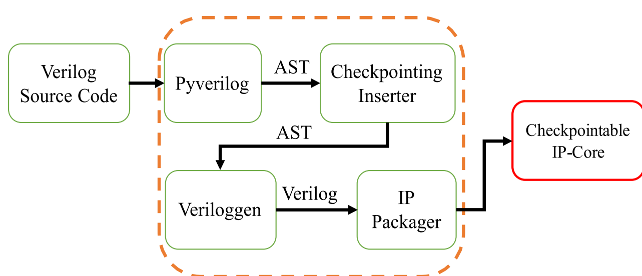


**Fig. 16** Structure of the tool.

### 5.2 Structure of the Tool

The structure of the tool includes four blocks, as in Fig. 16. The first block is Pyverilog [20], a Python-based analysis and synthesis tool of Verilog HDL source code. The parser in Pyverilog is a fundamental tool to analyze Verilog HDL source code. The parser generates an abstract syntax tree (AST) in the form of nested class objects in Python. The AST includes AST nodes presenting all information about the Verilog HDL source code, such as module definitions, parameters, ports, instances, always blocks. Parameters from the source code are also abstracted and resolved. The second block is Checkpointing Inserter. This block modifies and inserts more nodes to the AST that is the output of Pyverilog. The third block is Veriloggen [21], a Python-based hardware description and hardware customization library. This block is to generate Verilog HDL source code from the modified AST. The fourth block is IP Packager. By using PyCoram [22], this block packages the Verilog source code with checkpointing functionality to create an IP core, called CPR IP core. For the checkpointing purpose, this section focuses on the block: Checkpointing Inserter.

### 5.3 Algorithm of Checkpointing Insertion

Based on the proposed tree-based checkpointing architecture, Algorithm 1 has been proposed to insert checkpointing functionality into an HDL module. In this algorithm, if the

**Algorithm 1** Inserting checkpointing functionality

```
1  m = top_module
2  call insert_checkpointing(m)
3  //-----------------------------------------------------------
4  function insert_checkpointing(m)
5      if m is top_module then
6          call insert_static_CPR_part(m)
7      else:
8          call insert_CPR_gate(m)
9      for all inst ∈ m.instances do
10         inst_type = check_instance(inst)
11         if inst_type is normal_inst then
12             call insert_checkpointing(inst.module)
13             call insert_CPR_interface(inst)
14         if inst_type is RAM then
15             call insert_RAM_checkpointing(inst)
16     call modify_always_blocks(m)
17     call insert_capturing_FSM(m)
18     call insert_restoring_FSM(m)
```

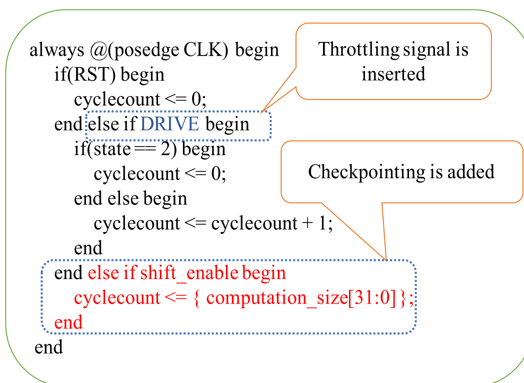


**Fig. 17** Modifying always block.

module is the top module, then the static CPR part will be inserted. Otherwise, a CPR gate will be inserted as in line 6. After that, all instances of the module will be visited and checked as in lines 9 and 10. The check_instance() function returns one of two values: *normal_inst* (a normal instance) and RAM (a module instance that will be synthesized as a distributed RAM or a block RAM). The check_instance() function matches the module definition of this instance with the module definitions that will be synthesized as given dedicated blocks, such as distributed RAMs and block RAMs. If the module instance is matched with a RAM, then all parameters, such as data width and address width, will be realized. The tool will not modify the module of the instance. Instead, in line 15 the tool will insert a circuit outside as in Fig. 9 in order to checkpoint the RAM. If the RAM is a block RAM that the output is delayed response to the input, then the tool will insert "additional registers" to store values of the input at consecutive clock cycles. These registers are also captured, and they will be used to resume the operation of the dedicated block later. If the module instance is not matched with a dedicated block, then the module will be inserted checkpointing infrastructure as in line 12. Also a CPR interface will be created in order to connect with the

**Table 1** Experimental Setup.

| EDA Tool | Vivado 2014.4 and ISE 14.7 |
|---|---|
| FPGA | Xilinx Zynq-7000 XC7z020clg484-1 |
| Evaluation Board | Zedboard |
| Clock frequency | 100 MHz |
| Host CPU | ARM Cortex-A9 |
| Operating system | Debian 8.0 |

checkpointing infrastructure as in line 13.

After visiting all instances of the module, the tool modifies always blocks to insert the logic throttling signal and circuits for register checkpointing. Figure 17 shows an always block after being inserted checkpointing functionality. The logic throttling signal - *DRIVE* in this figure is the same as the *DRIVE* signal in Fig. 5. The modifications in this always block are for a register checkpointing circuit only. Finally, the tool inserts a capturing FSM and a restoring FSM into the module.

## 6. Evaluation

Our experiments are set up as in Table 1 to evaluate hardware resource utilization, performance degradation, memory footprint, and maximum clock frequency degradation caused by the proposed checkpointing architecture.

### 6.1 Hardware Resource Utilization

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

#### 6.1.1 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. Figure 18 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.

Figure 18 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

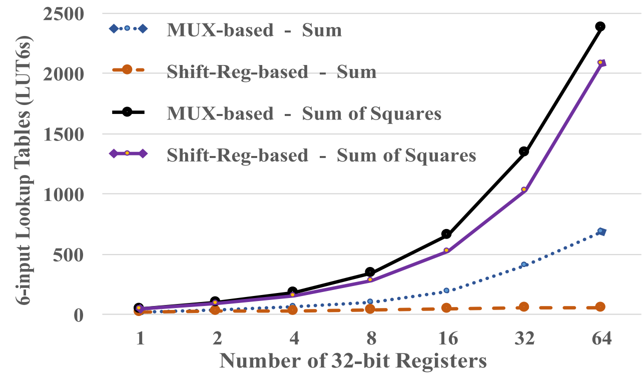

**Fig. 18** LUT utilization for capturing/restoring registers.

remains steady in the Shift-Reg-based circuit. The difference can be explained as follows. It is noted that in the following explanations LUT$k$ is defined as $k$-input LUT. In Sum, each register bit has only one input pattern and a carry input. Thus, an LUT2 is used for each register bit. When capturing/restoring circuits are added to the original hardware, one more input pattern is added for each register bit. Totally, each register bit has two input patterns, one carry input, and one 1-bit selector. Thus, instead of LUT2, an LUT4 is employed in front of each register bit. 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, 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 consumption 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 LUT2, LUT3, LUT4, and LUT5 to become 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 depends 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 either 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 addi-

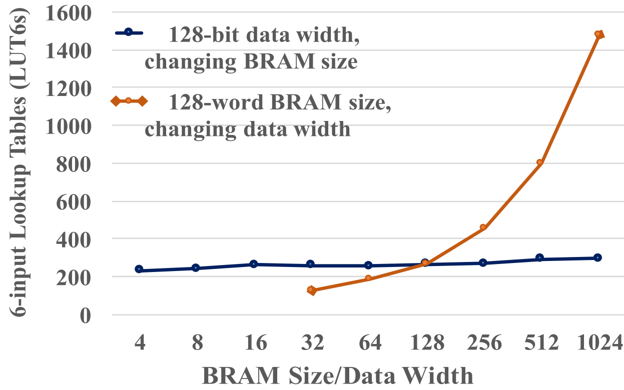


**Fig. 19** LUT utilization of capturing/restoring BRAM.

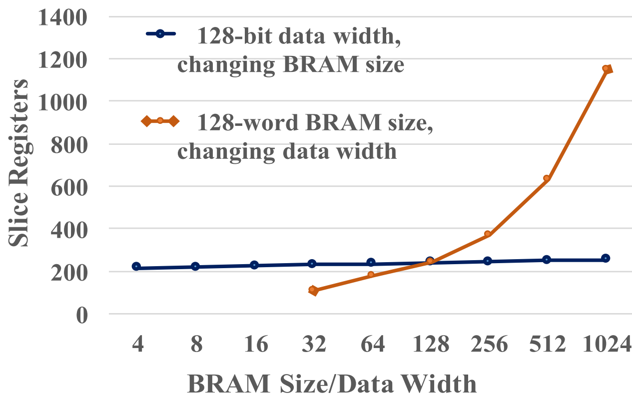


**Fig. 20** Slice register utilization for checkpointing BRAM.

tional slice registers for counters and CPR finite state machines.

6.1.2 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. 19 and Fig. 20 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 due to the fact that the LUT utilization comes from the multiplexer for the input data of BRAM and from the input multiplexer for $D_cp$. These two multiplexers depend on the data width of BRAM and the number of 32-bit inputs, respectively. Meanwhile, the increase in the register consumption is linear because the register consumption is caused mainly by the register $wdata_0$, which has the same

**Table 2** Additional 1-bit Registers for Resuming Dedicated Blocks.

| Apps | Distributed RAM (bits) | Block RAM (bits) | Additional 1-bit registers |
|---|---|---|---|
| Mat-Mul | 0 | 57158 | 61 |
| Dijkstra | 7168 | 0 | 0 |
| Stencil | 512 | 104672 | 73 |
| S-Search | 0 | 17632 | 46 |

**Table 3** Tree Structures after Checkpointing Insertion.

| Apps | CPR nodes | CPR levels | Nodes level 0 | Nodes level 1 | Nodes level 2 | Nodes level 3 |
|---|---|---|---|---|---|---|
| Mat-Mul | 28 | 4 | 1 | 6 | 17 | 4 |
| Dijkstra | 6 | 2 | 1 | 5 | - | - |
| Stencil | 23 | 4 | 1 | 5 | 15 | 2 |
| S-Search | 16 | 3 | 1 | 5 | 10 | - |

**Table 4** 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.) |
| Stencil | 6748 | 7395 (109.59%) | 1835 (3.45% avail.) |
| S-Search | 4056 | 3771 (92.97%) | 1468 (2.76% avail.) |

data width as BRAM.

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

6.1.3 Additional Registers for Resuming Dedicated Blocks

We apply the checkpointing mechanism to four realistic applications - pipelined SIMD matrix multiplication (Mat-Mul), Dijkstra graph processing (Dijkstra), 9-point Stencil Computation (Stencil), and String Search (S-Search). Table 2 shows the number of additional 1-bit registers required for resuming dedicated blocks in each application. These numbers are really small. It is noted that the response delay for distributed RAM and block RAM is 0 and 1 clock cycle, respectively. In Dijkstra application, only distributed RAMs are employed. In distributed RAMs, their outputs have no response delay to their inputs. Therefore, additional registers are not required. In the other applications, block RAMs are employed. These RAMs are dedicated blocks that the data outputs are one-clock-cycle delayed responses to their input addresses. As a result, additional 1-bit registers must be inserted in order to resume the operation of block RAMs.

6.1.4 Resource Utilization for the Whole Checkpointing Tree

Table 3 reveals the tree structures of the four application benchmarks after inserting checkpointing functionality. The synthesis results are shown in Table 4. As can be seen in the table, the static CPR hardware in the applications consumes small amounts of slice LUTs compared with the total amount of the device. Since the design of the static CPR part

**Table 5** Maximum Clock Frequency Degradation.

| Apps | $F_{max}$ (MHz) (Original) | $F_{max}$ (MHz) (Checkpointing) | Degradation |
|---|---|---|---|
| Mat-Mul | 115.075 | 103.875 | 9.73% |
| Dijkstra | 161.627 | 161.589 | 0.0235% |
| Stencil | 200.844 | 202.184 | −0.667% |
| S-Search | 188.929 | 188.644 | 0.15% |

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.

### 6.2 Maximum Clock Frequency & Data Footprint

Table 5 shows the synthesis results from Xilinx ISE 14.7. When adding checkpointing hardware to the four realistic applications, the maximum clock frequency decreases from 115.075 MHz to 103.875 MHz (a decline of 9.733%) for Mal-Mul while it is kept nearly unchanged in the three other applications. It is believed that input patterns added for checkpointing could find an available room in utilized LUTs in the critical paths of the three applications, thus no multiplexer is required in the critical paths. In contrast, in Mat-Mul input patterns could not find a room in utilized LUTs, thus an additional dedicated MUX or LUT are required to insert in the critical paths. This leads to the significant degradation in maximum clock frequency. It is noted that the measured speed reduction caused by scan chain insertion in [18] is slightly lower than 20%. Therefore, the proposed HDL-based checkpointing mechanism is better than the scan chain methodology in terms of maximum clock frequency.

Since our checkpointing mechanism is based on the definition of the reduced set of state-holding elements, the data footprint in our mechanism is also reduced significantly compared with the readback method [4]. If the four applications are implemented on the same device (XC7z020clg484-1), the readback method needs to read up to 4 Mbyte. Our method, however, needs to read only 8.3 kbyte, 1.1 kbyte, 14.3 kbyte, and 2.8 kbyte for Mat-Mul, Dijkstra, Stencil, and S-Search, respectively. Since the reduced set of state-holding elements is regconized as registers and RAMs, the memory footprint for checkpointing is approximately the total amount of registers and RAMs used in the application.

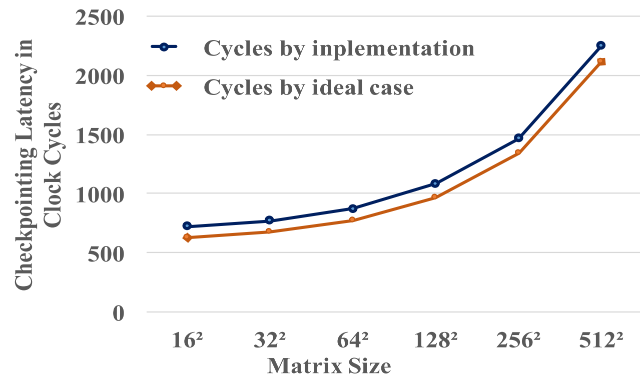


**Fig. 21** Checkpointing latency.

### 6.3 Performance Degradation

To evaluate performance degradation, we evaluate the checkpointing latency *cp_l*, which is the time to complete the extraction of hardware context. The checkpointing latencies are measured in Mat-Mul while scaling the matrix size. Changing the matrix size leads to changing the size of BRAMs, thus changing the amount of checkpointing data and the memory footprint. The ideal checkpointing latency in clock cycles is defined as the number of 32-bit checkpointing words. Figure 21 reveals that while scaling the matrix size, the checkpointing latency is always higher than the ideal case but has the same shape. It is likely that the checkpointing latency can be represented as: $cp_l = ideal_case + C$, with $C$ as a constant. Our experiments on Zedboard show that $C$ is about 120 clock cycles (1.2 us). The checkpointing latency cannot reach the ideal case because the constant $C$ is the representative of the delay due to off-chip memory requests and the delay due to the "'coarse-grained" control from the host, and thus $C$ cannot be removed. Therefore, the checkpointing latency seems to be linear with the number of checkpointing words and therefore linear with the number of register bits and with the total amount of RAM capacity used in the application. Let $t_{exe}$ be the execution time of an application without checkpointing request. Let $f_{cp}$ be the checkpointing rate, which is the number of checkpointing times per second. Let $t_{cp}$ be the checkpointing interval. Let $cp_o$ be the checkpointing overhead, which is the increased execution time of the application due to checkpointing. Let $n_{cp}$ be the number of checkpointing times during the execution time. Let $P_o$ be the performance overhead of the application.

$f_{cp}$ is defined by $f_{cp} = 1/t_{cp}$
$n_{cp}$ is defined by $n_{cp} = t_{exe} * f_{cp} = t_{exe}/t_{cp}$
$cp_o$ is defined by $cp_o = n_{cp} * cp_l = t_{exe} * cp_l/t_{cp}$
$P_o$ is defined by $P_o = cp_o/t_{exe} = cp_l/t_{cp}$

In our evaluation, *cp_l* is about 2300 clock cycles (0.023 ms) for the matrix size of 512 * 512. We choose the checkpoiting interval $t_{cp}$ to be one second. Therefore, the performance overhead is 0.23%. This value is small. Table 6 shows the context sizes and checkpointing latencies

**Table 6** Context Size and Checkpointing Latency.

| Apps | Register context (kbyte) | RAM context (kbyte) | Total context (kbyte) | Checkpointing latency (ms) |
|---|---|---|---|---|
| Mat-Mul | 0.95 | 7.31 | 8.26 | 0.023 |
| Dijkstra | 0.22 | 0.87 | 1.09 | 0.005 |
| Stencil | 1.28 | 13.06 | 14.34 | 0.038 |
| S-Search | 0.46 | 2.38 | 2.84 | 0.008 |

**Table 7** Power Consumption (W).

| Apps | Original | CPRtree + static | CPRtree + capturing | CPRtree + restoring |
|---|---|---|---|---|
| Mat-Mul | 3.496 | 3.520 (0.69%) | 3.576 (2.29%) | 3.554 (1.66%) |
| Dijkstra | 3.345 | 3.426 (2.42%) | 3.555 (6.28%) | 3.588 (7.26%) |
| Stencil | 3.507 | 3.607 (2.85%) | 3.562 (1.57%) | 3.545 (1.08%) |
| S-Search | 3.430 | 3.460 (0.87%) | 3.523 (2.71%) | 3.502 (2.1%) |

of the four benchmarks. It can be seen that checkpointing latencies depend linearly on the context sizes.

### 6.4 Power Consumption

The current sense on board was used to evaluate the power consumption caused by the whole Zedboard with our checkpointing architecture in the four realistic applications. There are three modes for the evaluation: static checkpointing - checkpointing hardware is inserted but the hardware runs in normal operation without any checkpointing request, capturing mode, and restoring mode. Table 7 shows that the power consumption of the board in static checkpointing is nearly the same as that of the board with original hardware (biggest increase of 2.85% for Stencil). The table also shows the power consumption when the board run in capturing and restoring mode. However, compared with the power of the board with the original hardware, the increases are quite small, which are from 1.08% (restoring mode in Stencil) to 7.26% (restoring mode in Dijkstra). Furthermore, the normal operation accounts for the majority of the execution time. Therefore, the proposed architecture in terms of power consumption is acceptable.

## 7. Related Work

While the concept of checkpointing is well known for software systems [13], checkpointing in hardware is underdeveloped. 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 [14]. 2) Kernel-based checkpointing is not portable across platforms but transparent to applications. BLCR [15] is a typical tool of this type.

For system-level FPGA checkpointing, some works have presented effective checkpoint/restart techniques on FPGAs to improve the dependability of FPGA computing. The first approach is the bitstream-based method. [4], [5] presented the bitstream-based method to read back 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 [16] by reading only used frames of bitstream. However, this method has several drawbacks. First, since the the formats of configuration bitstream are not the same for different types of FPGA, a bitstream read from an FPGA device cannot be used to resume the normal operation on other types of FPGA. Thus, the bitstream-based method is technology-dependent. Second, the data footprint of this method is high because only less than 8% of the data in the stream represent the hardware state. Third, the bitstream-based method cannot manage the channels of communication between FPGA and other devices. Thus, this method cannot guarantee consistent snapshots of FPGA and other devices, such as host CPU and off-chip memory. Fourth, previous works on this method focused on reading flip-flops only, whereas reading RAMs was under consideration.

The second approach is the netlist-based method. The scan-chain structure was employed in this 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: the MUX-based capturing/restoring circuit and the 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 the HDL level. Therefore, in their methodologies, new LUTs were inserted as multiplexers before flip-flops regardless of the 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 [17], scan-chain was also employed, but, by analyzing finite state machines, checkpoints were selected. As a result, instead of a full scan-chain, only partial scan-chains were used to capture the value of flip-flops. Therefore, the hardware overhead and memory footprint decreased significantly. Scan-chain was also used in [18] to observe the state of the full chip and to control internal signals, but not for checkpointing functionality. To access quickly 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, the netlist-based method also has several drawbacks. First, this method is generally tool-dependent because the formats and syntax of netlist files are

not the same for different tools. To generate technology-independent netlists from HDL source code, in [3], the authors needed to use a special tool - Synopsis Design Compiler at the front-end synthesis. After that, they used their own tool named "StateAccess" to identify all flip-flops in the netlist. Second, the netlist-based method cannot allow the simulation and the verification of a design with checkpointing functionality. Third, netlist is not a language. Thus, netlist nodes cannot be abstracted as a syntax abstract tree. As a result, behavior and parameters of circuits cannot be analyzed in netlist level. That may be the reason why the StateAccess tool used in [3] could not recognize RAM instances, provide RAM checkpointing, or manage communication channels for consistent snapshots.

The third approach is the HDL-based method. In [19], 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, even though 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 also be utilized by users. Furthermore, they did not propose a particular architecture to deal with structures of nested modules, even though dealing with these structures is more complicated than checkpointing a single module.

Our proposed method (HDL-based) solved all above problems. First, it is technology-independent and tool-independent because it works on HDL source code. Thus, the output of our method can be used by any tools and any technology. Also, no special tool is required in our method. Second, the extraction time of our method is low (0.038 ms) compared with bitstream-based method which is up to 1.2 ms [4] and 13 ms [16] although these previous works did not consider the extraction of RAM content. Third, the HDL-based method allows us to analyze the behavior of user hardware, customize hardware in order to manage communication channels for consistent snapshots. Fourth, the HDL-based method allows user designs with checkpointing functionality to possibly be simulated and verified. Fifth, the HDL-based method allows us to develop a tool inserting checkpointing functionality based on the abstract syntax tree of the original HDL source code. The tool can be seamlessly integrated into typical system design flows.

## 8. 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-grained" control to reduce delays in requests from the host, thus reducing performance degradation. We also provided a software stack with API functions for the host to "coarse-grained" manage checkpointing procedures. We believe that this is the first work concerned with the two issues: how to checkpoint dedicated blocks with outputs delayed compared with inputs; and how to guarantee consistent snapshots of FPGA and other components (e.g., the host CPU and the external memory). We solved these problems by proposing a new concept, a reduced set of state-holding elements, and by proposing a method to manage state of communication channels and throttle channel requests. To provide the property of automatic (system-level) checkpointing for the proposed checkpointing methodology, we have introduced a Python-based tool to generate checkpointing infrastructure automatically. Our evaluation shows that the checkpointing architecture causes less than 10% maximum clock frequency degradation, low checkpointing latencies, small memory footprints, and small increases in power consumption, while the LUT overhead varies from 17.98% (Dijkstra) to 160.67% (Matrix Multiplication).

## Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers JP17H00730, JP16K12407.

## References

[1] B. Schroeder and G.A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," IEEE Trans. Dependable and Secure Comput., vol.7, no.4, pp.337–350, Oct-Dec. 2010.

[2] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward Exascale Resillience - 2014 Update," J. Supercomputing Frontiers and Innovations, vol.1, no.1, 2014.

[3] D. Koch, C. Haubelt, and J. Teich, "Efficient Hardware Checkpointing - Concepts, Overhead Analysis, and Implementation," FPGA'07, pp.188–196, Feb. 18-20, 2007.

[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. Männer, "Multitasking on FPGA Coprocessors," Proc. 10rd International Conference on Field Programmable Logic and Application (FPL'00), vol.1896, pp.121–130, 2000.

[6] H.G. Vu, S. Kajkamhaeng, S. Takamaeda-Yamazaki, and Y. Nakashima, "CPRtree: A Tree-based Checkpointing Architecture for Heterogeneous FPGA Computing," 2016 4th International Symposium on Computing and Networking (CANDAR 2016), pp.57–66, Nov. 2016.

[7] A. Rezaei, G. Coviello, C.-H. Li, S. Chakradhar, and F. Mueller, "Snapify: Capturing Snapshots of Offload Applications on Xeon Phi Manycore Processors," HPDC'14, pp.1–12, June 2014.

[8] S. Takamaeda-Yamazaki and K. Kise, "A Framework for Efficient Rapid Prototyping by Virtually Enlarging FPGA Resources," 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig 2014), pp.1–8, Dec. 2014.

[9] A.A. Mendon, R. Sass, Z.K. Baker, and J.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), pp.1–6, 2012.

[10] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B.R. de Supinski, and S. Matsuoka, "Design and Modeling of a Non-blocking Checkpointing System," SC12, pp.1–10, Nov. 10-16, 2012.

[11] K.M. Chandy and L. Lamport, "Distributed snapshots: Determining

global states of distributed systems," ACM Trans. Comput. Syst., vol.3, no.1, pp.63–75, Feb. 1985.
[12] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," IEEE Trans. Softw. Eng., vol.SE-13, no.1, pp.23–31, Jan. 1987.
[13] 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., vol.34, no.3, pp.375–408, 2002.
[14] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop," 2009 IEEE International Symposium on Parallel & Distributed Processing, pp.1–12, May 23-29, 2009.
[15] P.H. Hargrove and J.C. Duell, "Berkeley lab checkpoint/restart (BLCR) for linux cluster," Proc. SciDAC, vol.46, pp.494–499, 2006.
[16] A. Morales-Villanueva and A. Gordon-Ross, "On-chip Context Save and Restore of Hardware Tasks on Partially Reconfigurable FPGAs," FCCM'13, pp.61–64, 2013.
[17] A. Bourge, O. Muller, and F. Rousseau, "Automatic High-Level Hardware Checkpoint Selection for Reconfigurable Systems," pp.155–158, FCCM 2015.
[18] I. Mavroidis, I. Mavroidis, and I. Papaefstathiou, "Accelerating Emulation and Providing Full Chip Observability and Controlability," IEEE Des. Test. Comput., vol.26, no.6, pp.84–94, Dec. 2009.
[19] A.G. Schmidt, B. Huang, R. Sass, and M. French, "Checkpoint/Restart and Beyond: Resilient High Performance Computing with FPGAs," pp.162–169, FCCM 2011.
[20] S. Takamaeda-Yamazaki, "Pyverilog: A Python-based Hardware Design Processing Toolkit for Verilog HDL," 11th International Symposium on Applied Reconfigurable Computing (ARC 2015) (Poster), Lecture Notes in Computer Science, vol.9040/2015, pp.451–460, April 2015.
[21] M. Watanabe, K. Sano, S. Takamaeda, T. Miyoshi, and H. Nakajo, "Japanese High-level Synthesis Tools for FPGA Hardware Acceleration," IEICE Trans. Commun., vol.J100-B, no.1, pp.1–10, 2016 (in Japanese).
[22] S. Takamaeda-Yamazaki, K. Kise, and J.C. Hoe, "PyCoRAM: Yet Another Implementation of CoRAM Memory Architecture for Modern FPGA-based Computing," Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2013) (Co-located with MICRO-46), Dec. 2013.

**Hoang Gia Vu** received the B.E. and M.E. degrees in Electrical Engineering from Le Quy Don Technical University, Vietnam in 2007 and 2010, respectively. During 2007–2014, he was a lecturer and research assistant in Le Quy Don Technical University. Since 2015, he has been a PhD candidate in Graduate School of Information Science, Nara Institute of Science and Technology. His research interests include reconfigurable computing, parallel computing, and processor architecture. He is a member of IEEE.

**Shinya Takamaeda-Yamazaki** received the B.E, M.E and D.E degrees from Tokyo Institute of Technology, Japan in 2009, 2011 and 2014 respectively. From 2011 to 2014, he was a JSPS research fellow (DC1). From 2014 to 2016, he was an assistant professor in Graduate School of Information Science, Nara Institute of Science and Technology, Japan. Since 2016, he has been an associate professor in Graduate School of Information Science and Technology, Hokkaido University, Japan. His research interests include computer architecture, reconfigurable system, and high level synthesis. He is a member of IEEE and IPSJ.

**Takashi Nakada** received his M.E. and Ph.D. degrees from Toyohashi University of Technology in 2004 and 2007 respectively. He has been an Associate Professor at the Nara Institute of Science and Technology since 2016. His research interests include Normally-Off Computing, processor architecture and related simulation technologies. He is a member of IEEE, ACM, and IPSJ.

**Yasuhiko Nakashima** received the B.E., M.E., and PhD degrees in Computer Engineering from Kyoto University in 1986, 1988, and 1998, respectively. He was a computer architect in the Computer and System Architecture Department, FUJITSU Limited from 1988 to 1999. From 1999 to 2005, he was an associate professor in the Graduate School of Economics, Kyoto University. Since 2006, he has been a professor in the Graduate School of Information Science, Nara Institute of Science and Technology. His research interests include processor architecture, emulation, CMOS circuit design, and evolutionary computation. He is a member of IEEE CS, ACM, and IPSJ.