NAIST-IS-DD1561036

Doctoral Dissertation

Dependable and Scalable FPGA Computing Using HDL-based Checkpointing

Hoang-Gia Vu

February 01, 2018

Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Hoang-Gia Vu

Thesis Committee:

Professor Yasuhiko Nakashima	(Supervisor)
Professor Michiko Inoue	(Co-supervisor)
Associate Professor Takashi Nakada	(Co-supervisor)
Assistant Professor Renyuan Zhang	(Co-supervisor)
Assistant Professor Tran Thi Hong	(Co-supervisor)

Dependable and Scalable FPGA Computing Using HDL-based Checkpointing[*]

Hoang-Gia Vu

Abstract

Thanks to high computational capabilities, reconfigurability, power efficiency, and the great advantages of customizing hardware for domain-specific applications, Field Programmable Gate Arrays (FPGAs) are now widely deployed in modern datacenters and high-performance computing systems. However, this deployment compounds the dependability of the computing systems due to their growing size and complexity. On the other hand, it challenges designers to scale computing systems.

In this doctoral dissertation, we present how FPGA computing can be dependable and scalable using checkpointing in hardware description language (HDL) level. First, we study a method to guarantee the consistency of snapshots between FPGA and other components. Such consistency is essential for the snapshots to be resumed correctly on FPGA. We then propose two checkpointing architectures along with a checkpointing mechanism on FPGA: CPRtree - a tree-based checkpointing architecture, and CPRflatten – a ring-based flattened checkpointing architecture. The two checkpointing architectures are transparent to applications and portable across different hardware platforms. Third, we investigate a static analysis of the original HDL source code for CPRflatten from fundamentals to algorithms in order to re-use hardware resources for the checkpointing purpose, thus reducing hardware consumption caused by checkpointing functionality. Fourth, we introduce two Python-based tools in structures and algorithms to generate checkpointing infrastructures according to CPRtree and CPRflatten so that designers' task in writing checkpointing source code can be removed completely.

[*]Doctoral Dissertation, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD1561036, February 01, 2018.

The two tools can be integrated seamlessly into hardware design flows. The position of the tools in design flows ensures that our checkpointing architectures are independent of other tools and technology. Fifth, we study a checkpoint/restart scheme for dependability of FPGA computing. In this scheme, we also introduce a software stack with application programming interface (API) functions for “coarse-grained” management from the host. The stack is also transparent to applications and portable across hardware platforms. Sixth, we present two schemes for scalability of FPGA computing employing our above checkpointing architectures. The first scheme – on-the-fly multitasking on FPGA allows multiple users to efficiently share a limited reconfigurable fabric. The second scheme – on-the-fly hardware task migration in heterogeneous FPGA computing allows a hardware task to be migrated between different FPGA fabrics with different technology.

We evaluate our proposals from hardware overhead, maximum clock frequency degradation, data footprints, and performance overhead to power consumption. Although the hardware overhead is still significant, the performance degradation and the additional power consumption is small. Our proposals show a potential for bringing FPGAs to hyper-scale computing, such as hyper-scale data centers and hyper-scale clouds while taking advantages of software-based computing.

Keywords:

FPGA computing, checkpointing, multitasking, task migration, dependability, scalability

Contents

List of Figures

List of Tables

1. Introduction

1.1 Overview

Thanks to high computational capabilities, reconfigurability, power efficiency, and the great advantages of customizing hardware for domain-specific applications, Field Programmable Gate Arrays (FPGAs) are integrated in high-performance computing and now widely deployed in modern datacenters [1, 2, 3, 4, 5]. Catapult with the Microsoft hyperscale cloud [2], Amazon's EC2 F1 [6], Baidu's SDA [7], IBM's FPGA fabric [8], and Novo-G [9] are some examples of FPGA deployments. However, this deployment raises two issues: dependability and scalability of FPGA computing. For the dependability, this integration compounds the problem of increasing failure rate because of the growing size and complexity in the computing system [10, 11]. 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. For the scalability, first, FPGA fabrics are expected to be shared by multiple tasks using task switching and time multiplexing. Second, FPGA computing is expected to be used in clusters/clouds, in which the flexibility of run-time task scheduling should be achieved by live hardware task migration in order to save energy, balance the load, and prepare production servers for maintenance. Fortunately, task switching and task migration also require a checkpoint/restart technique as the dependability does.

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 dissertation 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 [12]. 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 [13, 14]. 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. Furthermore, inserting checkpointing circuits always brings with it hardware overhead for the application circuit. However, we believe that hardware resource utilization can be reduced by efficient checkpointing architectures. In many cases, hardware resources utilized for normal operation can be also employed for checkpointing functionality in order to reduce the total hardware consumption.

1.2 Contributions

Contributions of this thesis are as follows:

1. We propose a method to guarantee the consistency of snapshots between FPGA and other components. Such consistency is essential for the snapshots to be resumed correctly on FPGA.

2. We propose a new concept: reduced set of state-holding elements that represents the full state of hardware operation. By capturing and restoring only

this set of elements, FPGA operation can be resumed correctly. This set can be used to checkpoint dedicated blocks.

3. We present two checkpointing architectures along with a checkpointing mechanism on FPGA: CPRtree – a tree-based checkpointing architecture, and CPRflatten – a ring-based flattened checkpointing architecture. The two checkpointing architectures are transparent to applications and portable across different hardware platforms.

4. We propose a static analysis of the original HDL source code for CPRflatten from fundamentals to algorithms in order to re-use hardware resources for the checkpointing purpose, thus reducing hardware consumption caused by checkpointing functionality.

5. We introduce two Python-based tools in structures and algorithms to generate checkpointing infrastructures according to CPRtree and CPRflatten so that designers' task in writing checkpointing source code can be removed completely. The two tools can be integrated seamlessly into hardware design flows. The position of the tools in design flows ensures that our checkpointing architectures are independent of other tools and technology.

6. We present a checkpoint/restart scheme for dependability of FPGA computing. In this scheme, we also introduce a software stack with application programming interface (API) functions for "coarse-grained" management from the host. The stack is also transparent to applications and portable across hardware platforms.

7. We propose two schemes for scalability of FPGA computing employing our above checkpointing architectures. The first scheme – on-the-fly multitasking on FPGA allows multiple users to efficiently share a limited reconfigurable fabric. The second scheme – on-the-fly hardware task migration in heterogeneous FPGA computing allows a hardware task to be migrated between different FPGA fabrics with different technology.

These contributions have been published in [15, 16, 17, 18]. We combine all the above proposals and demonstrate on four application benchmarks. Our proposals and evaluation show a potential for bringing FPGAs to hyper-scale computing, such as hyper-scale data centers and hyper-scale clouds while taking advantages of software-based computing.

1.3 Thesis Outline

The remainder of this thesis is structured as follows. Section 2 discusses the dependability and scalability of FPGA computing from concepts, motivation to challenges. Section 3 presents CPRtree - a tree-based checkpointing architecture and the corresponding Python-based tool. Section 4 describes CPRflatten - a Ring-based flattened checkpointing architecture, the static analysis, and the corresponding Python-based tool. Section 5 presents a checkpoint/restart flow for dependability of FPGA computing. Section 6 proposes the multitasking scheme on FPGA. Section 7 describes the hardware task migration scheme in heterogeneous FPGA computing. Conclusion with potential future work and applications is summarized in section 8.

2. Dependability and Scalability of FPGA computing

2.1 Definition and Motivation

Dependability of FPGA computing is a concept referring to the resilience of applications running on FPGA after failures. In other words, it refers to fault tolerance of FPGA computing. With many great advantages, FPGAs are not only employed as accelerators for computational iterations, but are also deployed for long running applications, such as graph processing, scientific simulation, and big data applications. For such long running applications, dependability should be taken into account so that the applications do not need to restart from beginning. In addition, although FPGAs themselves are reliable devices, they become more fault-injectable when being deployed in a large scale. Furthermore, the integration into heterogeneous computing requires the dependability of FPGA computing to keep the whole system reliable. Therefore, the dependability of FPGA computing must be investigated thoroughly.

For the dependability of FPGA computing, every kinds of faults or failures are considered. Our checkpointing method brings the resilience to the computing systems, even the faults or failures may come from CPU, DRAM, storage, or power supply. The faults or failures may also come from the bitstream, that including configuration of FPGA. However, in my method, we only need the initial bitstream, which can be restored any time from the non-volatile storage.

Scalability of FPGA computing is a concept referring to the ability of adapting the computing to meet greater needs or demands. Two aspects of scalability can be pointed out in FPGA computing. First, a single FPGA fabric can serve multiple tasks so that limited hardware resources can be shared by an increasing number of users. Second, multiple/many FPGA fabrics, even with different architectures, technologies, or vendors, can works together to form a lager scale computing system. The hardware task migration between different nodes can adapt the computing system for load balancing and energy savings. Therefore, the scalability of FPGA computing should be also studied deeply.

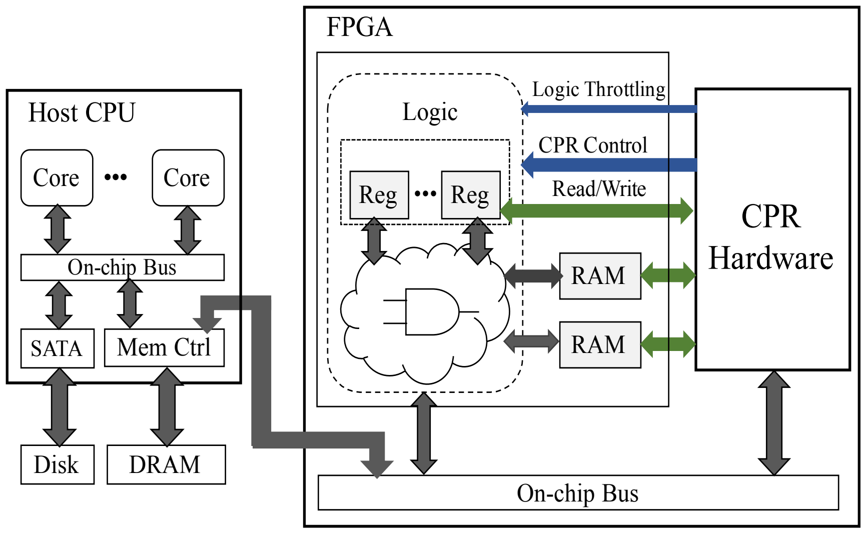


Figure 1. Computing node with checkpointing hardware.

2.2 Challenges

We assume a computing node with checkpointing hardware as in Figure 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.

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 [19]. 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.

2.3 Consistent snapshots in FPGA checkpointing

2.3.1 Overview of Consistent Snapshot

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 [20, 21]. 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 section II. 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.

2.3.2 Channel Finite State Machine

Figure 2 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

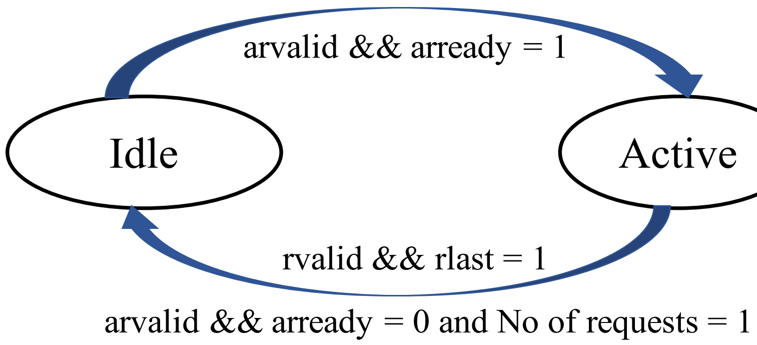


Figure 2. Channel finite state machine.

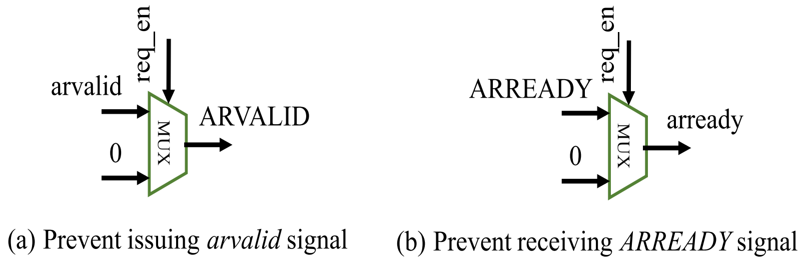

arvalid
req_en
0
MUX
ARVALID
(a) Prevent issuing *arvalid* signal
ARREADY
req_en
0
MUX
arready
(b) Prevent receiving *ARREADY* signal

Figure 3. Preventing issuing requests on the mater side.

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'.

2.3.3 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

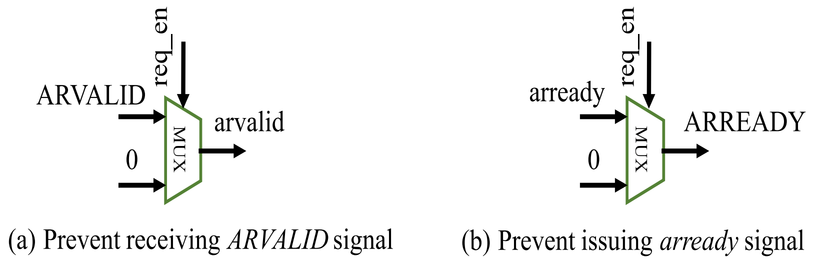

req_en
ARVALID
0
MUX
arvalid
(a) Prevent receiving *ARVALID* signal
req_en
arready
0
MUX
ARREADY
(b) Prevent issuing *arready* signal

Figure 4. Preventing receiving requests on the slave side.

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 Figure 3. 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 Figure 4.

3. CPRtree - A Tree-based checkpointing architecture

3.1 Related Work

While the concept of checkpointing is well known for software systems [22], 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 [23]. 2) Kernel-based checkpointing is not portable across platforms but transparent to applications. BLCR [24] 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. [13, 14] 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 [25] 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 [12]. 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 [26], 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 [27] 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 [12], 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 [12] could not recognize RAM instances, provide RAM checkpointing, or manage communication channels for consistent snapshots.

The third approach is the HDL-based method. In [28], 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.

The fourth approach is the high level synthesis based (HLS-based) method. Alban Bourge el al [26, 29] presented a high level synthesis design flow manipulating the intermediate representation of an HLS tool to insert a scan chain into the initial circuit. The main contribution of this work is checkpoint selection, in which only in some states of the finite state machine, checkpointing can be performed. In the checkpoint selection, the tool can find live variables, and only these live variables are checkpointed. As a result, the context size can be reduced, thus reducing the hardware overhead. However, this work did not consider the issue about consistent snapshots of FPGAs and other components. Once taking it into account, the state that checkpointing can be performed should depend on the state of communication channels between FPGA and others. Therefore, the checkpoint selection may be no longer feasible. Furthermore, the authors limited their application benchmarks to using a specific HLS tool generating application circuits with a single finite state machine. This constraint may prevent developers from designing complicated applications.

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 [13] and 13 ms [25] although these previous works did not consider the extraction of RAM content. Third, the HDL-based method al-

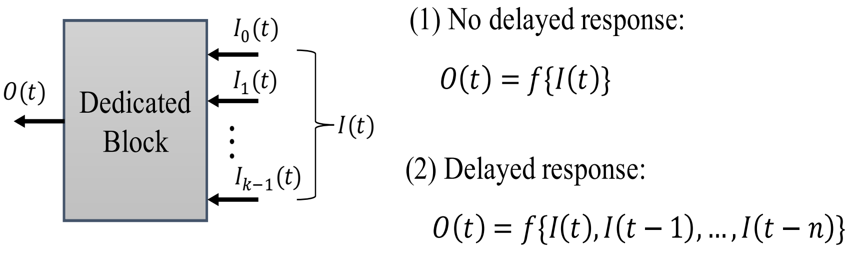


Figure 5. Dedicated block.

lows 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.

3.2 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 Figure 5, 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 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

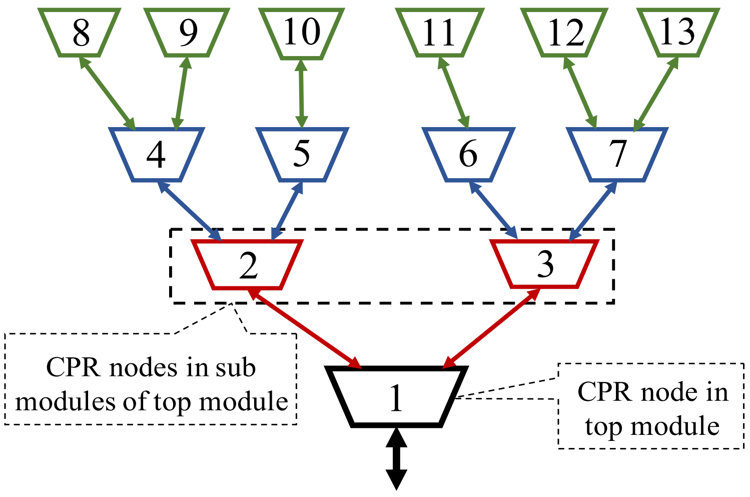


Figure 6. CPRtree.

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.

3.3 Checkpointing Architecture

3.3.1 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

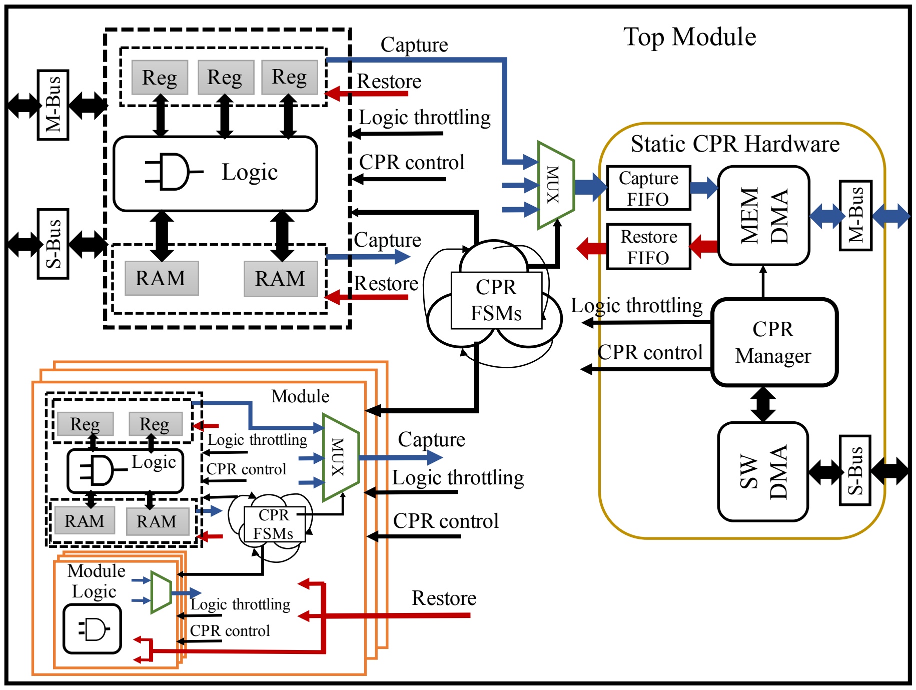
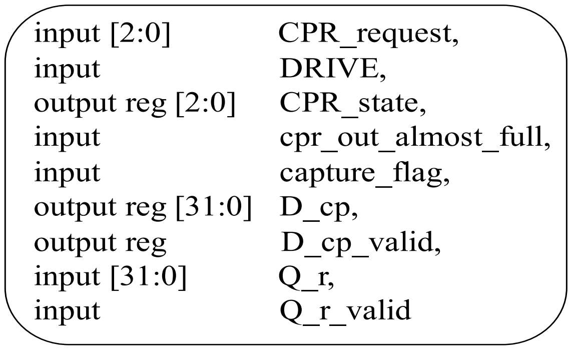


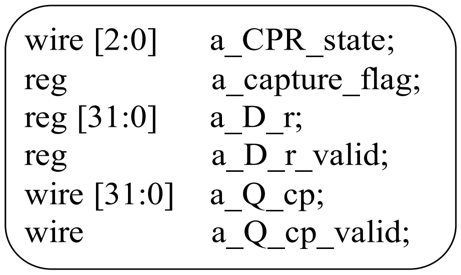
Figure 7. Tree-based checkpointing architecture on FPGA.

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 checkpoint/restart infrastructure, called CPR node, and the CPR nodes of all modules form a checkpointing tree, as in Figure 6. 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, . . ., 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, . . ., M_{m-1}. In other words, the CPR nodes of module instances M_0, M_1, M_2, . . ., M_{m-1} are the children nodes of the CPR node of module A. It



```
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
```

Figure 8. 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;
```

Figure 9. CPR interface.

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 Figure 6, 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.

3.3.2 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 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 [12] 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 Figure 7. 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 Figure 8. The gate consists of a logic throttling signal - *DRIVE* as in [30], 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. 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 [31], 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 [32], 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

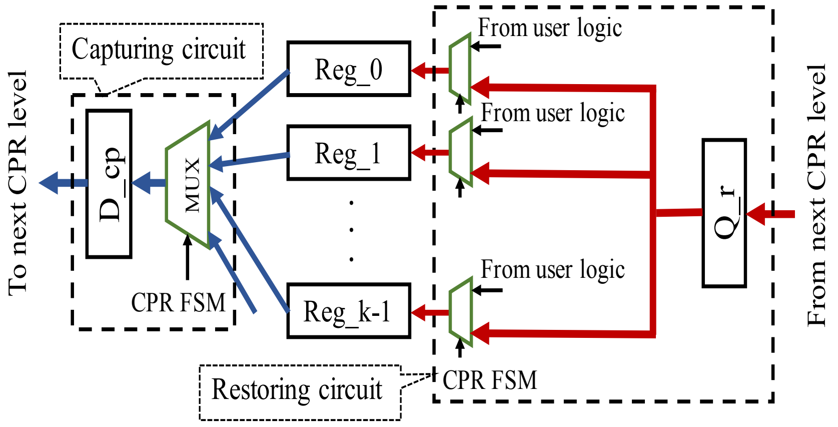


Figure 10. MUX-based capturing/restoring circuit for registers.

immediately after the all checkpoints are written to Capture FIFO.

3.3.3 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 9 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.

3.3.4 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

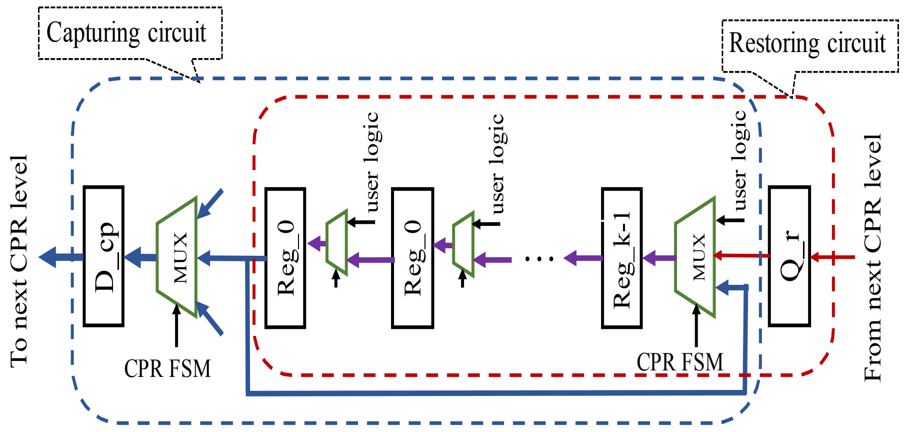


Figure 11. Shift-Reg-based capturing/restoring circuit for registers.

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 Figure 10. 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. It takes n clock cycles to capture/restore in this circuit.

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 Figure 11. 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 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.

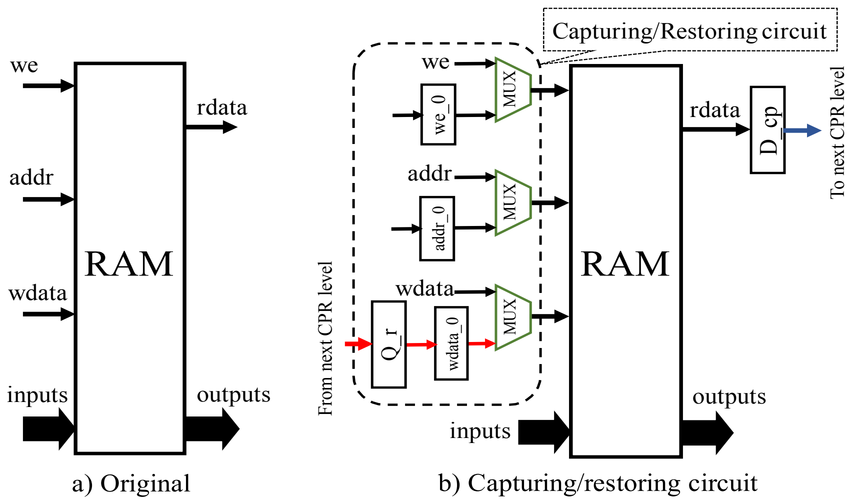


Figure 12. Adding capturing/restoring circuit to RAM.

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. It takes n clock cycles to capture/restore registers in this circuit. Therefore, compared with the MUX-based circuit, this circuit is the same in terms of latency. It should be noted that several registers may be used for the control of the checkpointing functionality. However, these registers are not captured or restored.

Changing the structure of registers does not affect the logic function of the user circuits. However, it may affect routing process, thus changing maximum clock frequency.

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, 2k 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 12 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

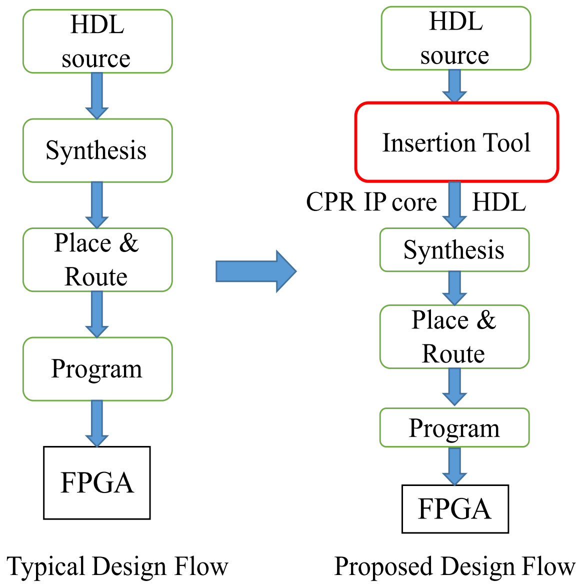


Figure 13. Modification in design flow.

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

3.3.5 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

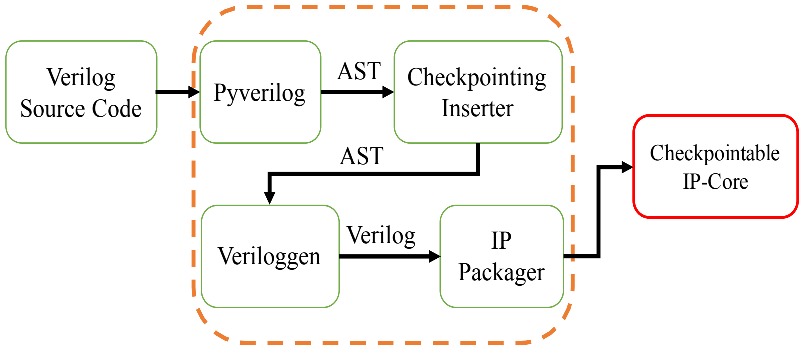


Figure 14. Structure of the tool.

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.

3.4 Tool for checkpointing insertion

3.4.1 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 Figure 13. The input of the tool is Verilog source code. Due to the location of the tool in the design flow,

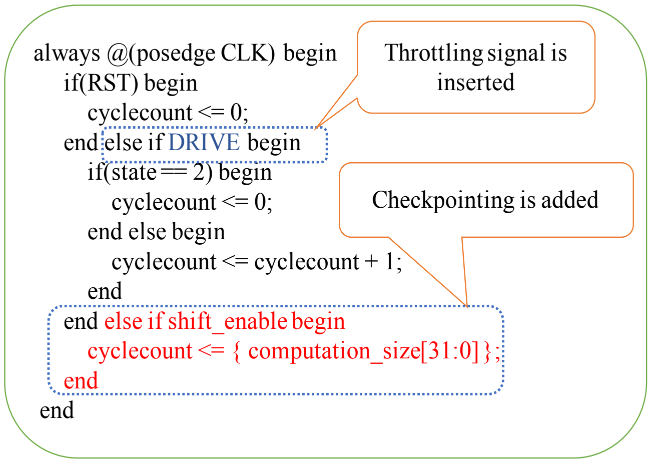
Algorithm 1 Inserting checkpointing functionality

```
m = top_module
call insert_checkpointing(m)
//-------------------------------------------------------
function insert_checkpointing(m)
    if m is top_module then
        call insert_static_CPR_part(m)
    else:
        call insert_CPR_gate(m)
    for all inst ∈ m.instances do
        inst_type = check_instance(inst)
        if inst_type is normal_inst then
            call insert_checkpointing(inst.module)
            call insert_CPR_interface(inst)
        if inst_type is RAM then
            call insert_RAM_checkpointing(inst)
    call modify_always_blocks(m)
    call insert_capturing_FSM(m)
    call insert_restoring_FSM(m)
```

our checkpointing methodology is portable across hardware platforms and not dependent on technology.

3.4.2 Structure of the Tool

The structure of the tool includes four blocks, as in Figure 14. The first block is Pyverilog [33], 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 [34], 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 [35], this block packages the Verilog source code with checkpointing functionality to create an IP core, called CPR IP core. For the checkpointing purpose, this



```
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
```

Figure 15. Modifying always block.

section focuses on the block: Checkpointing Inserter.

3.4.3 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 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 Figure 12 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

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

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 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 15 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 Figure 8. 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.

3.5 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.

3.5.1 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.

Resource Ultilization for Capturing/Restoring Registers in a CPR Node:

We have two alternative methods to capture/restore as presented in Figure 10 and Figure 11. 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 16 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 16 also reveals that the LUT utilization in Sum of Squares increases dramatically in both the MUX-based circuit and the Shift-Reg-based circuit, while the LUT utilization in Sum rises slightly in the MUX-based circuit and remains steady in the Shift-Reg-based circuit. The difference can be explained as follows. It is noted that in the following explanations LUTk 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

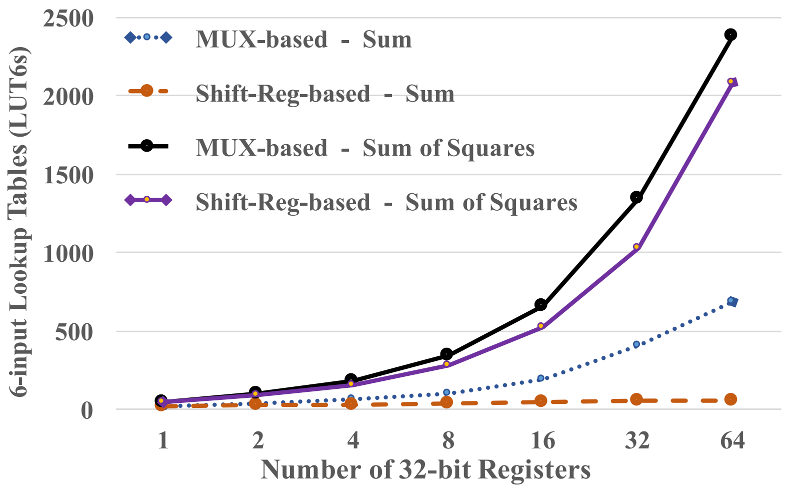


Figure 16. LUT utilization for capturing/restoring registers.

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 additional slice registers for counters and CPR finite state machines.

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 Figure 17 and Figure 18 show that both the slice LUT and

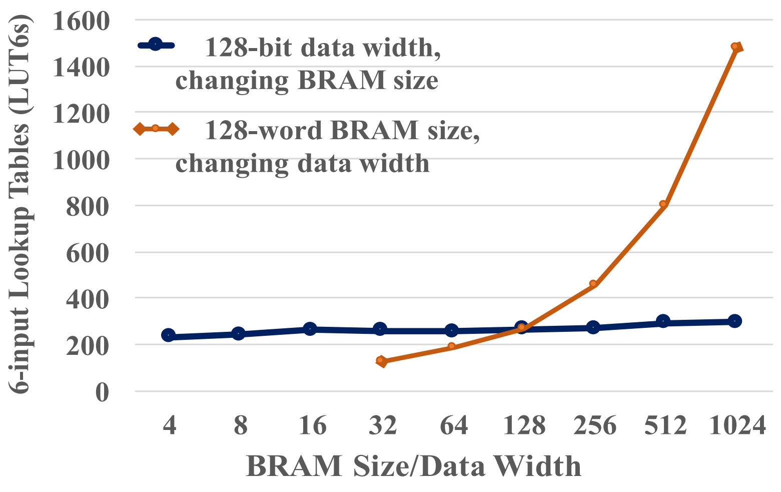


Figure 17. LUT utilization of capturing/restoring BRAM.

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 data width as BRAM.

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

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

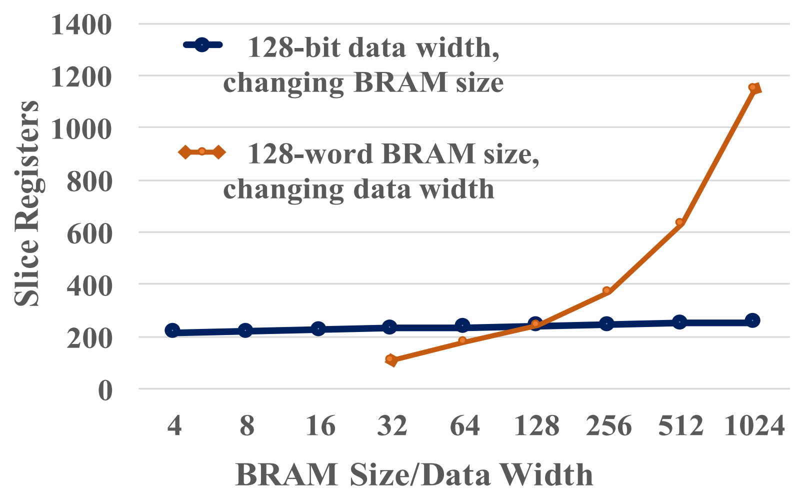


Figure 18. Slice register utilization for checkpointing BRAM.

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.

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 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

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	-

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.

3.5.2 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

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.)

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%

Table 6. 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%)

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 [27] 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. It can be seen from the table that the maximum clock frequency of Stencil even increase after inserting checkpointing functionality. The results are just estimated by the tool. This may come from the optimization of the tool so that the physical distance of logic elements reduces, thus reducing the critical path.

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 [13]. 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.

3.5.3 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 6 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.

4. CPRflatten: A Ring-based Flattened Checkpointing Achitecture

4.1 Checkpointing Architecture

4.1.1 Overview of CPRflatten

It is assumed that hardware structures are flattened in HDL level. That means two objects can be connected without the need for considering complicated structures of nested modules. Our checkpointing architecture as in Figure 19 is derived from the idea of removing all connectors between checkpoint/restart (CPR) levels of CPRtree. In this case, Mux-based capturing/restoring circuits and Shift-Reg-based capturing/restoring circuits can no longer be used anymore, and, instead, a shifting matrix of register bits must be employed in register checkpointing. However, the output of the matrix (CPR_out) is looped back to the input of the matrix to make sure that the values of registers after capturing are kept unchanged. Thus, this forms a shifting ring, and this architecture is called *ring-based flattened checkpointing architecture* in this paper. In the architecture, the reduced set of state-holding elements is divided into two sets: register set and RAM set. While capturing/restoring circuits for registers are configured as a shifting ring, capturing/restoring circuits for RAMs is separated from each other and is separated from the shifting ring. The proposed architecture can be divided into two parts: *static CPR hardware part* and the rest, called *user-logic-based part*. The later includes a shifting ring for register checkpointing, capturing/restoring circuits for RAMs, and two CPR finite state machines (CPR FSMs) for capturing and restoring.

4.1.2 Shifting Ring

The shifting ring is formed from the shifting matrix of register bits with outputs looped back to inputs. Let W be the data width of checkpointing, B be the number of register bits in the reduced set of state-holding elements, and C be B/W. If B is not a multiple of W, then a padding register is added to guarantee a multiple of W. As showed in Figure 20, the shifting matrix of register bits is a W-by-C matrix. There are two advantages of using shifting ring in checkpointing.

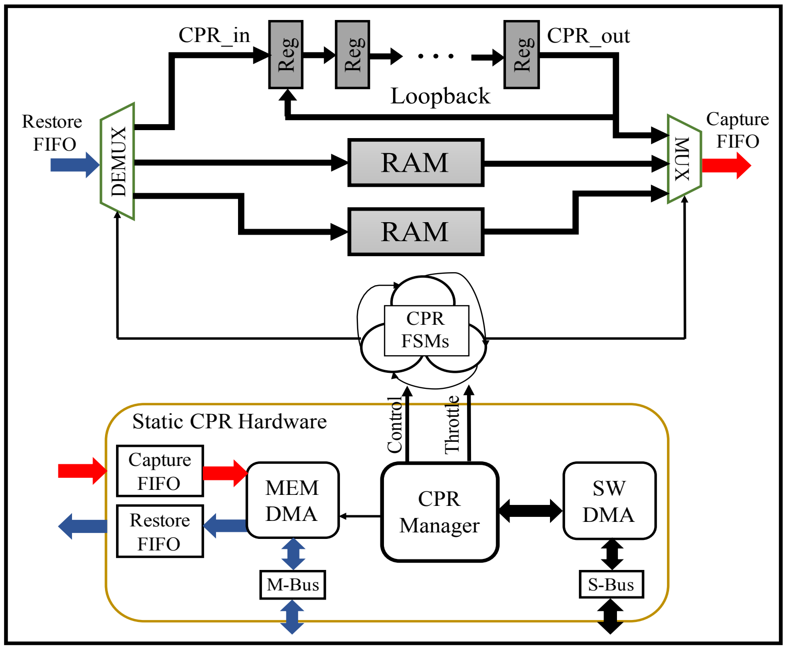


Figure 19. Ring-based flattened checkpointing architecture

First, it ensures that the content of registers is kept unchanged after capturing. Second, the shifting ring can be used for both capturing and restoring processes, thus saving hardware resources.

However, employing a shifting ring leads to complexity in hardware. In the worst case, particularly, one more input pattern is added to each register bit, thus one input is added to the multiplexer in front of the bit. As a result, additional LUTs may be used for such logic functionality. if the multiplexer is re-structured including more levels, the degradation of maximum clock frequency will be more serious due to the increase in the critical path.

4.1.3 RAM Capturing/Restoring Circuit

On-chip RAM context can be captured and restored by iterating reading and writing through its whole address space. To keep the inputs and outputs of a RAM unchanged after capturing, multiplexers are employed to separate checkpointing from the normal operation. However, instead of writing the context read from a RAM to the next CPR level when capturing as in CPRtree, the context is written directly to Capture FIFO in our architecture as in Figure 21. Con-

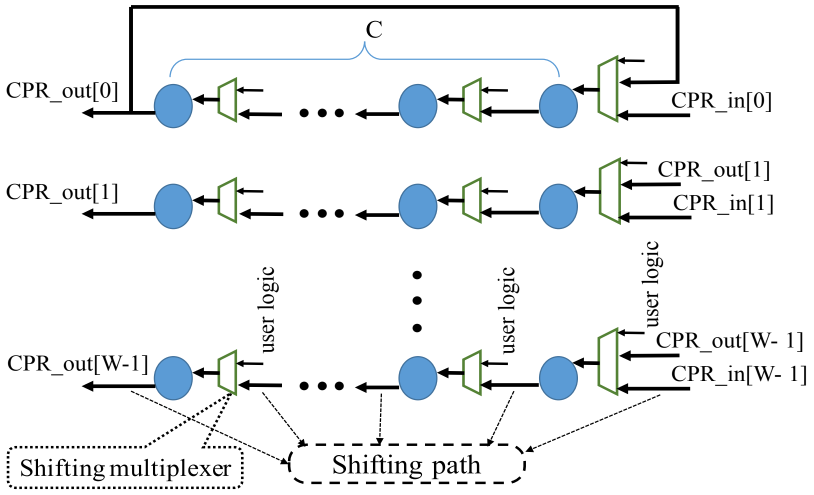


Figure 20. Shifting ring of register bits

versely, when restoring, context is read from Restore FIFO, and then written to the RAM. This implementation is possible by flattening HDL modules for RAM checkpointing.

A RAM port includes four signals: write enable (we), address (addr), write data (wdata), and read data (rdata). While the read data signal (output) can be shared between the normal operation and capturing, the other signals (inputs) require multiplexers to be shared between the normal operation and the restoring process. Therefore, three registers: we_0, addr_0, and wdata_0 are added for RAM checkpointing. Let DW be the data width of the RAM, and A be the number of address bits of the RAM. As can be seen from Figure 21, the capturing/restoring circuit requires a 1-bit register for we_0, an A-bit register for addr_0, and a DW-bit register for wdata_0. In total, it requires DW + A + 1 register bits. When synthesized, it is estimated to consume (DW + A + 1) slice registers. Since all the multiplexers are 2-input, (DW + A + 1) 2-input 1-bit multiplexers are required. It is also noted that all these 2-input multiplexers use the same select bit – throttling signal. Furthermore, an LUT6 can also be configured as two 5-input LUTs (32-bit ROMs) with separate outputs but common logic inputs. As a consequence, (DW + A + 1) 2-input 1-bit multiplexers can be mapped onto (DW + A + 1)/2 LUT6s. Therefore, if there are k RAMs in hardware structure with data widths DW0, DW1, . . . , DWk-1, and numbers of address bits A0, A1,

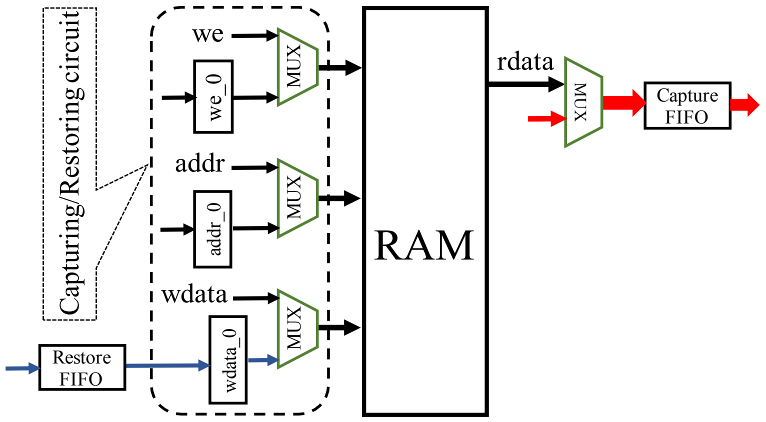


Figure 21. RAM capturing/restoring circuit

…, Ak-1, respectively, then the estimated number of utilized slice registers and the estimated number of utilized slice LUTs caused by RAM capturing/restoring circuits can be presented respectively as follows:

$$\mathrm{F_{RAM}} = \sum_{l=0}^{k-1}(DW_1 + A_1 + 1)$$

$$\mathrm{L_{RAM}} = \sum_{l=0}^{k-1}(DW_1 + A_1 + 1)/2$$

4.1.4 CPR Finite State Machines

Checkpoint/restart procedures require two finite state machines, the first for capturing and the second for restoring. The number of states in the two finite state machines depends on the number of utilized RAMs and their data widths. In both CPR FSMs, the slice register consumption comes mainly from counter registers. While the LUT ultilization in the capturing FSM is mainly caused by the multiplexer in front of the Capture FIFO, the LUT consumption in the restoring FSM is primarily from the comparators for selecting state-holding elements to be restored.

4.2 Static Analysis of Original HDL Source Code

4.2.1 Fundamentals of Static Analysis

In the shifting ring in Figure 20, a shifting path may bring with it not only an additional input to the shifting multiplexer in front of each register bit but also

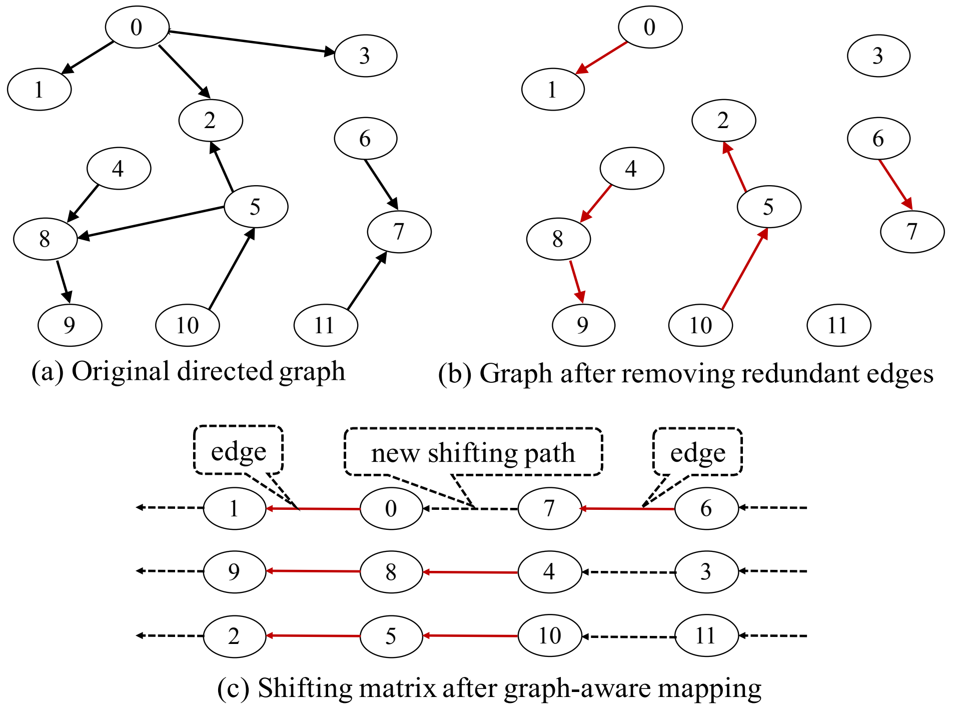

0
1
2
3
4
5
6
7
8
9
10
11
(a) Original directed graph
(b) Graph after removing redundant edges
edge
new shifting path
edge
(c) Shifting matrix after graph-aware mapping

Figure 22. Static analysis of original HDL source code

cause complexity for the combinational circuit generating select bits. This input may require more LUTs for the multiplexing functionality, whereas the complexity of the combinational circuit may also consume more LUTs. While the complexity cannot be avoided, the additional input will be not required if the shifting path coincides with one of the inputs from the user logic. Such coincidence is achieved when the preceding register bit (F1) in the shifting path is used to determine the value of the register bit (F0) in the next clock cycle. In this case F0 at the time t is a function of F1 at the time t-1, written as F0(t) = f(F1(t-1)). If register bits are considered as vertices of a directed graph, then F0 and F1 are two of the vertices of a graph and there is an edge from F1 to F0. Therefore, we believe that the LUT consumption caused by a shifting ring can be reduced if the shifting ring is designed in such a way that some shifting paths coincide with edges of the graph of register bits.

However, if there are several edges from one vertex, then only one edge can be mapped onto a shifting path. Conversely, if there are several edges to one vertex, then, too, only one edge can be used as a shifting path. Therefore, there are three steps to design a shifting matrix. The first step is to analyze the original HDL source code to identify the graph of register bits. The second step is to find groups of edges to the same vertex, then keep only one edge in each group while removing the rest of the group. The rest is called *redundant edges*. The third step is to map all remaining edges onto shifting paths of the shifting matrix. These three steps are called static analysis of original HDL source code. They are described in an example in Figure 22. This mapping is called *graph-aware mapping*. The original graph has 12 vertices and 10 directed edges. After removing redundant edges, there are 6 remaining edges. Therefore, after mapping vertices and edges onto the shifting matrix, only 3 additional shifting paths are required. As a result, the graph-aware mapping eliminates 6 shifting paths.

4.2.2 Algorithms

The Pseudo code to describe how to remove redundant edges and how to map register bits onto a shifting matrix are outlined in Algorithm 2 and Algorithm 3.

- Let *bitSet* be the set of register bits in the module.

- Let *unvisitedBitSet* define the set of register bits that has not been visited.

- Let *rightBitSet* of register bit B be the set of register bits used to determine B in the next clock cycle.

- Let *leftBitSet* of register bit B be the set of register bits that are determined by B in the next clock cycle.

- Let *preceding* of register bit B define the preceding register bit of B in the shifting path. *preceding* will be None if the shifting path does not coincide with any edge.

- Let *following* of register bit B be the following register bit of B in the shifting path. *following* will be None if the shifting path does not coincide with any edge.

- Let *noFollowBitSet* define the set of register bits that have no *following* but have *preceding* in the shifting path.

- Let *noFollowNoPrecBitSet* be the set of register bits that have no *following* and no *preceding* in the shifting path.

- Let *matrixBitList* define the list of bits in the shifting matrix. The bit index increases by 1 in the same column and increases by W in the same row.

Algorithm 2 Removing redundant edges

```
1: unvisitedBitSet ← bitSet
2: while unvisitedBitSet is not Ø do
3:     min_length ← min{length(B.rightBitSet), B ∈ unvisitedBitSet}
4:     for all B ∈ unvisitedBitSet do
5:         if length(B.rightBitSet) == min_length then
6:             for b ∈ B.rightBitSet do
7:                 if b.following is None and b.preceding is not B then
8:                     B.preceding ← b
9:                     b.following ← B
10:                    for C ∈ b.leftBitSet do
11:                        C.rightBitSet ← C.rightBitSet − {b}
12:                    break
13:            unvisitedBitSet ← unvisitedBitSet − {B}
```

- Let *unmappedIndexList* be the list of indexes of bits in *matrixBitList* that has not been mapped onto.

- Let Nb be the number of register bits in the module.

In Algorithm 2, register bits are consecutively visited to remove redundant edges (line 4). It is noted that after removing such redundant edges, all edges starting from a register bit can be removed, while one of them is expected to be mapped onto a shifting path. To avoid this case, register bits having a *rightBitSet* with fewer elements should be visited first (line 3, 5). After removing the redundant edges of a group, only one edge remains in the group (the edge from b to B). The vertex b cannot be used anymore, thus it is removed from *rightBitSets* (line 10, 11).

The next step to map register bits and remaining edges onto the shifting matrix is presented in Algorithm 3. Since the register bits with edges must be mapped onto the most left-side column first, the *noFollowBitSet* must be visited first as in line 3. The visit to bits in the *noFollowBitSet* also leads to a visit to bits that have both following and preceding by tracing the preceding on the bit chain (line 7 when B0.*preceding* is not None). It finally leads to a visit to the bits

Algorithm 3 Graph-aware Mapping Algorithm

```
1: noFollowBitSet ← {B ∈ bitSet, B.preceding is not None, B.following is None }
2: noFollowNoPrecBitSet ← {B ∈ bitSet, B.preceding is None, B.following is None }
3: for all B ∈ noFollowBitSet do
4:     k ← unmappedIndexList.Pop()
5:     matrixBitList[k] ← B
6:     B0 ← B
7:     while B0.preceding is not None do
8:         if (k + W) < Nb then
9:             matrixBitList[k + W] ← B0.preceding
10:            unmappedIndexList.Remove(k + W)
11:            k ← k + W
12:            B0 ← B0.preceding
13:        else
14:            if B0.preceding.preceding is None then
15:                noFollowNoPrecBitSet ← noFollowNoPrecBitSet ∪ {B0.preceding}
16:            else
17:                noFollowBitSet ← noFollowBitSet ∪ {B0.preceding}
18: for all B ∈ noFollowNoPrecBitSet do
19:     k ← unmappedIndexList.Pop()
20:     matrixBitList[k] ← B
```

that have *following* but no *preceding* (line 7 when B0.*preceding* is None). After that, it continues to visit the *noFollowNoPrecBitSet* to cover all bits in the *bitSet* (line 18).

4.3 Tool for Checkpointing Insertion

Based on CPRflatten, Algorithm 4 has been proposed to insert checkpointing functionality into an HDL module. In this algorithm, if the module is the top module, then the static CPR part will be inserted. Otherwise, control ports will be inserted (line 8). After that, all instances of the module will be visited and checked (line 9, 10). The check_instance() function matches the corresponding module definition with RAM templates. If it is matched, the func-

Algorithm 4 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_control_port(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         if inst_type is RAM_inst then
14             call insert_RAM_checkpointing(inst)
15     call visit_always_block(m)
16     if m is top_module then
17         call modify_always_blocks(m)
18         call insert_capturing_FSM(m)
19         call insert_restoring_FSM(m)
```

tion returns RAM_inst and the function insert_RAM_checkpointing() is called (line 14). Otherwise, the check_instance() returns normal_inst and the function insert_checkpointing() is called as a recursive algorithm (line 12). Then, the visit_always_block() (line 15) visits all always blocks in the module and counts registers driven by the always blocks. By calling this function in the recursive function insert_checkpointing(), all always blocks and registers in the design will be visited. After the recursive process, the modify_always_blocks() function is called (line 17) to modify always blocks based on the above static analysis. Finally, a capturing FSM and a restoring FSM are inserted in the HDL top module (line 18, 19).

Table 7. LUT Utilization.

Apps	LUTs (Original)	Additional LUTs (CPRtree)	Additional LUTs (CPRflatten)	Decline of Additional LUTs
Mat-Mul	3323	5339 (160.67%)	2593 (78.03%)	51.43%
Dijkstra	8126	1461 (17.98%)	1020 (12.55%)	30.18%
Stencil	6748	7395 (109.59%)	3883 (57.54%)	47.49%
S-Search	4056	3771 (92.97%)	1210 (29.83%)	67.91%
Mean				49.25%

Table 8. Maximum Clock Frequency Degradation.

Apps	F_{max} (MHz) (Original)	F_{max} (MHz) (CPRtree) & Degradation	F_{max} (MHz) (CPRflatten) & Degradation
Mat-Mul	115.075	103.875 (9.73%)	103.875 (9.73%)
Dijkstra	161.627	161.589 (0.02%)	165.822 (-2.6%)
Stencil	200.844	202.184 (-0.67%)	202.184 (-0.67%)
S-Search	188.929	188.644 (0.15%)	188.644 (0.15%)

4.4 Evaluation

4.4.1 Hardware Resource Ultilization

We demonstrate CPRflatten on four realistic applications: - pipelined SIMD matrix multiplication (Mat-Mul), Dijkstra graph processing (Dijkstra), 9-point Stencil Computation (Stencil), and String Search (S-Search). Since the static CPR hardware part is fixed and transparent to applications, to evaluate the efficiency of checkpointing architectures, only the hardware overhead of the user-logic-based part is considered. Table 7 shows that the hardware overhead is reduced the most from 92.97% (CPRtree) to 29.83% (CPRflatten) in the S-Search application (a decline of 67.91% of LUT overhead). The average decline of 49.25% of LUT overhead shows that the proposed checkpointing architecture (CPRflatten) with the proposed static analysis is much better than CPRtree in terms of hardware overhead.

Table 9. Dummy Register Bits & Memory Footprint

Apps	Dummy register bits	Register context (kB)	RAM context (kB)	Total context (kB)
Mat-Mul	05	0.70	7.31	8.01
Dijkstra	27	0.20	0.87	1.07
Stencil	04	1.26	13.06	14.32
S-Search	26	0.44	2.38	2.82

```
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);
```

Figure 23. CPRtree API.

4.4.2 Maximum Clock Frequency Degradation

The estimation results in Table 8 show that both CPRflatten and CPRtree have negligible impact on the maximum clock frequency. The average degradation is only 2.31% and 1.65%, respectively. The most significant degradation is 9.73% (Mat-Mul). These average values are really small compared with the value of around 20% in the scan-chain netlist-based method [27].

4.4.3 Dummy Register Bits & Memory Footprint

Table 9 shows the number of dummy register bits and the total context size (memory footprint) for each benchmark. For CPRtree, the number of dummy register bits is quite high since each node on the tree requires a dummy register. For CPRflatten, the number of dummy register bits is always less than W = 32 bit – the checkpointing data width.

5. Checkpoint/Restart flow

5.1 Checkpoint/Restart Timing Diagram

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 Figure 23, CPR Manager provides "fine-grained" management for CPR procedures on FPGAs, as shown in Figure 24.

5.1.1 Prepare

Instead of pausing an application and waiting for channels to be idle, as in [19], 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.

5.1.2 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

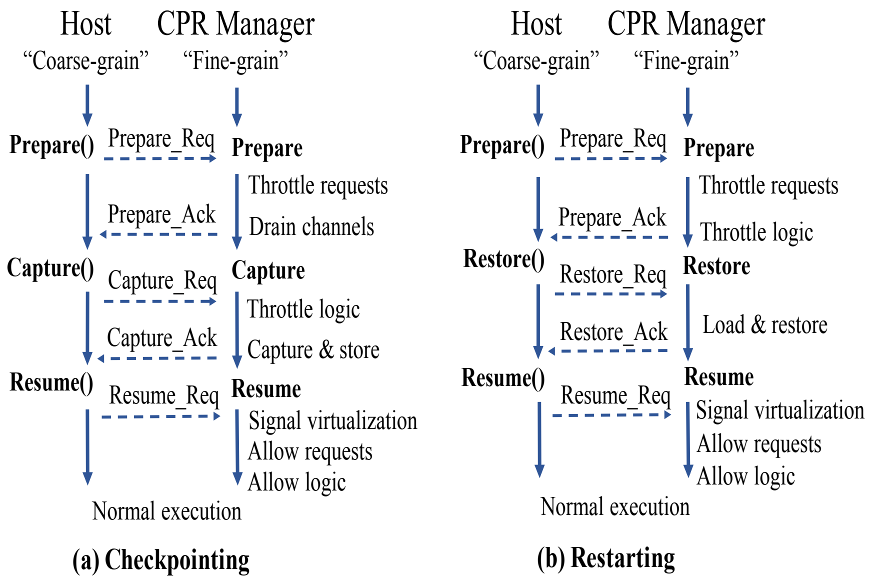


Figure 24. Checkpointing/restarting timing diagram.

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.

5.1.3 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

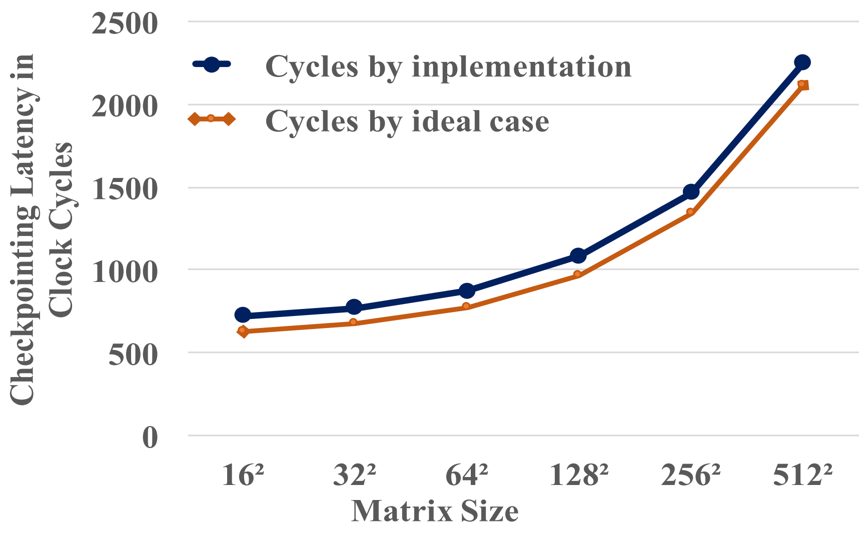


Figure 25. Checkpointing latency.

checkpointing tree. At the CPR nodes, checkpointing data is restored to state-holding elements, such as registers and RAMs. When the last word is restored to the corresponding elements, CPR Manager sets an acknowledgement to inform the host that the restore procedure is complete.

5.1.4 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.

Table 10. 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

5.2 Evaluation

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 25 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 10 shows the context sizes and checkpointing latency of the four benchmarks. It can be seen that checkpointing latencies depend linearly on the context sizes.

6. Multitasking on FPGA

6.1 Related Work

Multitasking on FPGA is not a new idea. Previous approaches to multitasking [13, 14, 36, 37, 38] using the readback-of-bitstream method for task switching have several serious drawbacks, that can prevent multitasking from being deployed in reconfigurable computing. First, the readback-of-bitstream method cannot ensure that a readback bitstream (a taken snapshot) is consistent with other components. For example, a bitstream, taken while FPGA is accessing off-chip memory, is not consistent with the off-chip memory. Thus, the application cannot be resumed correctly on FPGA. Second, the readback bitstream is not enough to resume the normal operation of dedicated blocks, which have outputs delayed compared with inputs. Third, the report in [13] indicated that only less than 8% of the data in the bitstream is useful, thus 92% of readback time is a waste of time.

6.2 Multitasking Structure

A multitasking system is organized in an FPGA-based computing node as Figure 26. Initial bitstreams are stored in the local storage. The host CPU will load an initial bitstream and configure FPGA when the corresponding task is required. FPGA can execute only one task at a time. Multiple tasks can share the reconfigurable fabric by using task switching. Each task is allocated a time period to run before being replaced by another task. For task switching, HDL-based checkpointing is employed to capture FPGA context when a task is swapped out, and to restore FPGA context when another task is swapped in. Captured context will be written to the unified memory as a snapshot of the corresponding task at a given physical address. Therefore, saving snapshots to the local storage is not required in our scheme. In addition, to reduce the configuration time, a bitstream can be pre-loaded to the unified memory at a given physical address area, called a bitstream buffer, before being written to FPGA.

Multiple tasks running on FPGA require fixed contiguous memory allocation at a given physical address space for both snapshots and application data in the unified memory. This is easy on stand-alone systems, but more complicated

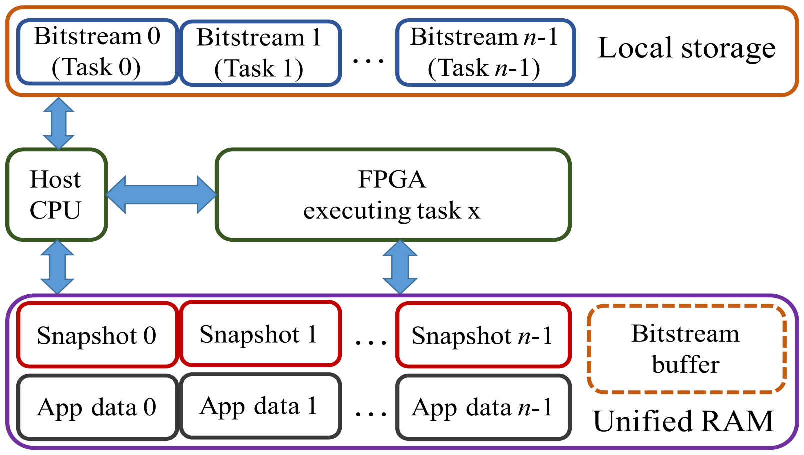


Figure 26. Multitasking on an FPGA-based computing node

on operating systems (OS) using virtual addresses like Linux. For the later, a reserved memory should be created when building the OS so that the kernel and other software applications will not touch the memory space.

6.3 Timing Diagram for Task Execution and Task Switching

Figure 27 shows a timing diagram for three tasks running on FPGA. The tasks can be allocated different time periods (T0, T1, and T2) to run. The cost of multitasking on FPGA consists of performance overhead, hardware overhead, additional footprints on the unified memory, and maximum clock frequency degradation on FPGA when checkpointing functionality is inserted. The performance overhead is the total task switch latency. Since HDL-based checkpointing is employed, no readback of the bitstream as [14] is required. Also, saving snapshots to the local storage as [29] is not required. Therefore, the task switch latency (T_{switch}) is the sum of the context capturing latency (T_{cap}), the bitstream configuration latency (T_{conf}), and the context restoring latency (T_{res}). The restoring latency is not required if the task swapped in runs on FPGA for the first time.

The timing diagram for a switch from task 0 to task 2 includes task 0 swapping-out, bitstream configuration for task 2, and task 2 swapping-in. The host uses API functions *Prepare*() and *Capture*() to swap task 0 out. It also uses *Prepare*(),

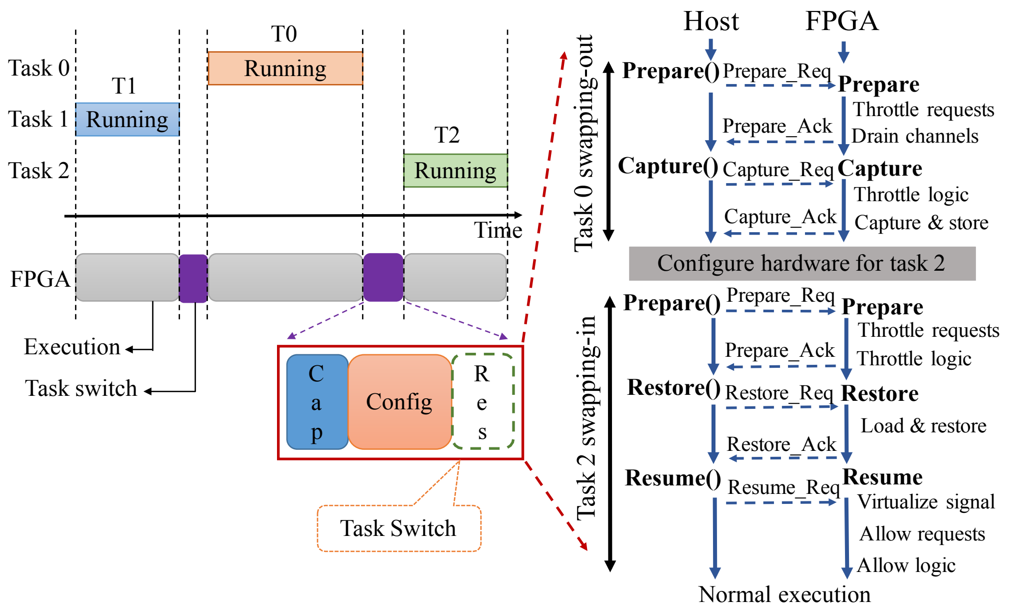


Figure 27. Task execution and task switch timing diagram

Restore(), and *Resume*() to swap task 2 in. The consistency of snapshots between the FPGA and other components, such as the host CPU and the unified memory, is guaranteed by the *Prepare* procedure on FPGA. Checkpointing of dedicated blocks is also guaranteed by the *Resume* procedure with signal virtualization.

Table 11. Context size (data footprint) and task switch latency.

Apps	Register context kbyte	RAM context kbyte	Total context (kbyte)	Capturing latency (ms)	Restoring latency (ms)	Configuration time (ms)
Mat-Mul	0.70	7.31	8.01	0.022	0.023	31.095
Dijkstra	0.20	0.87	1.07	0.005	0.007	31.095
Stencil	1.26	13.06	14.32	0.039	0.04	31.095
S-Search	0.44	2.38	2.82	0.009	0.01	31.095

6.4 Evaluation

Table 11 shows the breakdown of the total context size (memory footprint) and the task switch latency. The memory footprints (less than 15 kbyte) are much lower than the memory footprint in the readback-of-bitstream method (4 Mbyte for Zedboard). The capturing latency and the restoring latency depend linearly on the corresponding context size. The FPGA on Zedboard is configured from host programs in C language using processor configuration access port (PCAP). The measured configuration time is stable at 31.095 ms. The task switch latency caused by switching from Mat-Mul to Stencil is 31.157 ms for example. The improvement of our scheme over the previous schemes based on the readback method in terms of the task switch latency is that the readback latency (tens to hundreds of milliseconds [14, 36]) is removed and replaced by the capturing latency (less than 0.039 ms).

7. Hardware Task Migration in Heterogeneous FPGA computing

7.1 Related Work

Problems of previous works on FPGA checkpointing could prevent hardware task migration from being implemented and evaluated. In [39] the authors presented a task migration method using ICAP for configuration and readback of partial bitstreams. However, their evaluations show that the extraction time (953 ms), the configuration time (52 ms), and the relocation time are high. In addition, the total migration latency was not evaluated. Because of using the readback-of-bitstream method, this work has several additional drawbacks. First, the readback-of-bitstream method cannot ensure that a readback bitstream (a taken snapshot) is consistent with other components. For example, a bitstream, taken while FPGA is accessing off-chip memory, is not consistent with the off-chip memory. Thus, the application cannot be resumed correctly on FPGA. Second, the readback bitstream is not enough to resume the normal operation of dedicated blocks, which have outputs delayed compared with inputs.

7.2 Hardware Task Migration Structure

A heterogeneous FPGA cluster with hardware task migration is organized as in Figure 28. The cluster includes a server for task management, a file system, and m computing nodes integrated with FPGA. The server, file system, and m nodes are connected via an Ethernet network. Each node consists of a host CPU, a local storage, an FPGA, and a unified memory. FPGAs in nodes may be different in architecture, technology, and vendor. Bitstreams for potential tasks running on FPGA is stored in the local storage. For hardware task migration between two nodes, HDL-based checkpointing is employed in the source node to capture FPGA context. The captured context is then written to the unified memory as a snapshot of the corresponding task at a given physical address. In the cluster, the snapshot is not saved to the local storage or the file system, instead, it is sent to the destination node using message passing interface (MPI) [40]. In the destination node, the snapshot is read from unified memory to FPGA and restored to the state-holding elements before being resumed. However, the FPGA on the destination node must be configured before that with the corresponding bitstream. To reduce the configuration time, the corresponding bitstream can be loaded from the local storage or pre-loaded to the unified memory at a given physical address area, called a bitstream buffer, before being written to FPGA.

There are several requirements for hardware task migration in a FPGA cluster. First, it requires contiguous memory allocation for both app data and snapshots in the unified memory. Second, since task context contains the physical address of app data, the app data of a migrated task must be allocated the same physical address in the unified memory of both source node and destination node. These two requirements are easy to satisfy in stand-alone systems, but more complicated in operating systems (OS) using virtual addresses like Linux. For the later, a memory space for FPGA should be reserved when building the OS so that other software applications will not touch the space. It is noted that if the source and the destination FPGA fabrics are different, the corresponding bitstreams must be different but generated from the same HDL source code.

Heterogeneity: It is noted that a readback bitstream from an FPGA fabric cannot be used on different types of FPGA fabrics. Therefore, the readback-of-bitstream method cannot allow hardware task migration on heterogeneous FPGAs. In our method, checkpointing circuits are inserted in the HDL level. They allow a task snapshot to be taken or restored by capturing or restoring the values of the state-holding elements. For each task, the same HDL source code is used to generate different bitstreams for different FPGA fabrics. Instead of transferring the whole bitstream from the source node to the destination node, only checkpoints (the values of the state-holding elements) are transferred. On the destination node, after configuring the FPGA fabric with the corresponding bitstream, the checkpoints are restored to the state-holding elements of the hardware task. Then the normal operation can be resumed. Therefore, our checkpointing method (HDL-based) allows hardware task migration in heterogeneous FPGA computing.

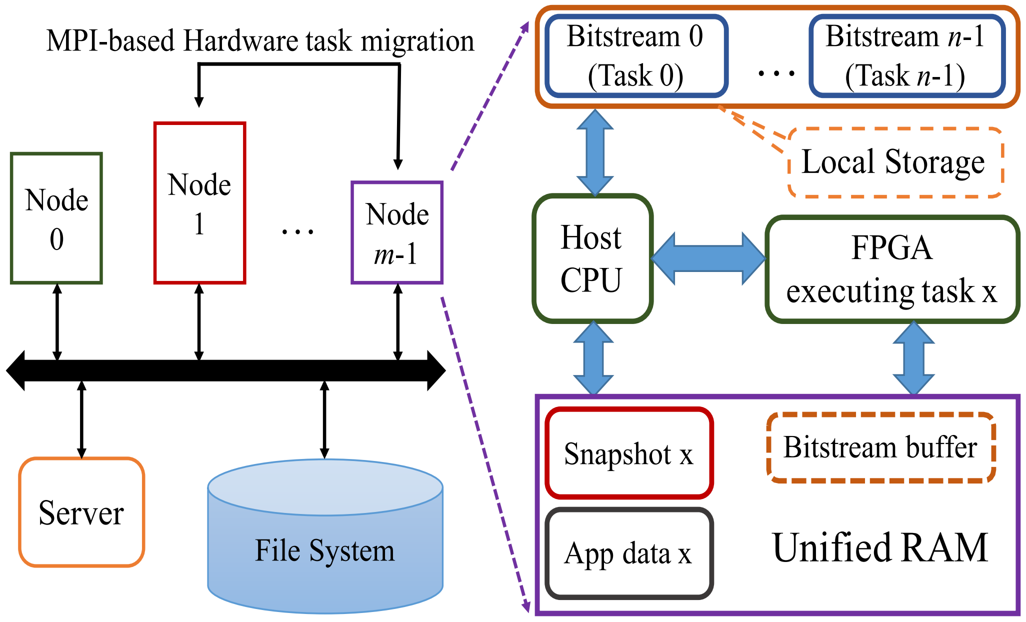


Figure 28. Hardware task migration in a heterogeneous FPGA cluster

7.3 Hardware Task Migration Timing Diagram

Figure 29 depicts a timing diagram of hardware task migration from node 0 to node 1. When node 0 receives a migration request, it launches two MPI jobs. The first program (1) is in the same host to send DRAM context and FPGA context to node 1. The other (2) is in the host of node 1to resume the task on its FPGA. The two programs communicate via MPI. As soon as the program (2) starts, it configures the FPGA on node 1. At the same time, it copies all memory used by the task on node 0. The copy includes two phases. In the first phase, node 0 sends the data (un-dirty RAM context), that will not dirty during the task execution, to node 1. This phrase is called pre-copy in this paper. The second phase is to send all the dirty data from node 0 to node 1 after the task is paused on node 0. This phase uses a non-blocking function MPI_Isend().

When the FPGA configuration and the pre-copy finish, a setup acknowledge is sent to node 0. Then node 0 calls the Prepare() function to guarantee a consistent snapshot before calling the Capture() function to capture FPGA context. After FPGA context is captured and saved to the unified memory, the host in node 0 sends the FPGA context to node 1 using a blocking function MPI_Send(). When

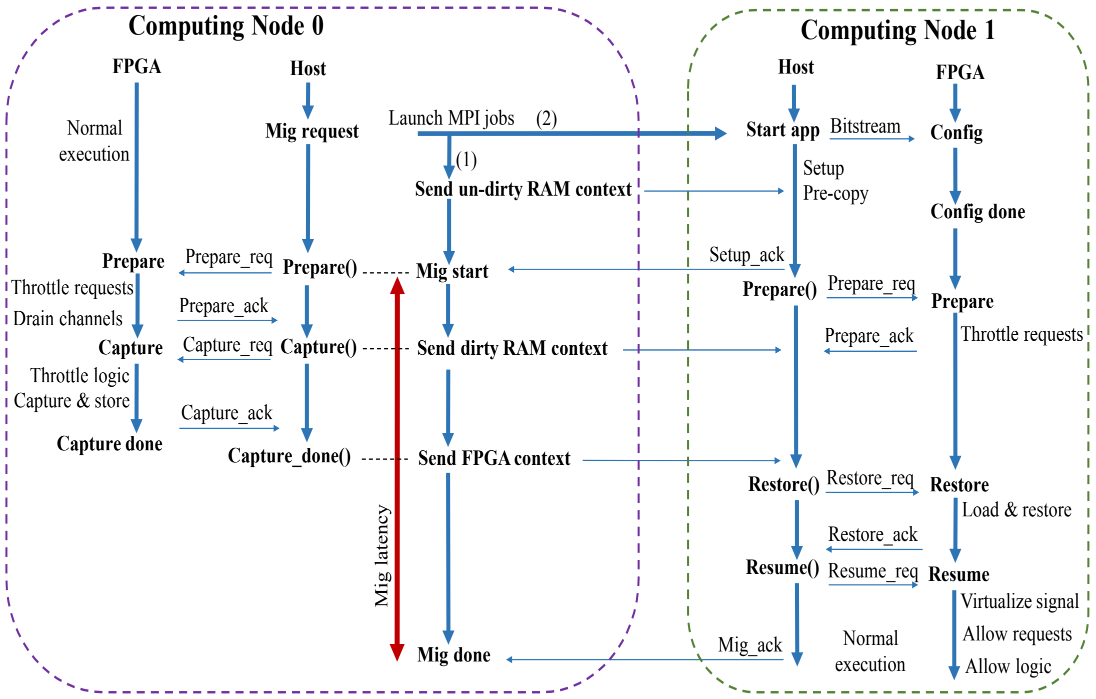


Figure 29. Timing diagram of hardware task migration

node 1 receives all the FPGA context, it calls the Restore() function to restore the context to the state-holding elements on FPGA. After the restoration and receiving all the dirty data, it calls the Resume() function to resume the task. Then, a migration acknowledge signal is sent back to inform node 0 that the migration has already done.

In the timing diagram, we adopt the following parameter and latency definitions:

• T_{config} denotes the configuration time for the FPGA on node 1.

• bw is the bandwidth allocated for transferring data from node 0 to node 1 during the migration.

• mrbw_{i} denotes the memory read bandwidth on node i allocated to FPGA during the migration.

• mwbw_{i} is the memory write bandwidth on node i allocated to FPGA during the migration.

• C_{FPGA} denotes the amount of FPGA context.

• $C_{\text{non-dirty}}$ is the amount of non-dirty data that can be pre-copied.

• $\mathrm{C}_{\text{dirty}}$ denotes the amount of dirty data that must be copied after pausing the application.

• $\mathrm{T}_{\text{pre-copy}}$ denotes the time required copying the non-dirty data from node 0 to node 1.

$$T_{\text{pre-copy}} = \frac{C_{\text{non-dirty}}}{bw} \tag{1}$$

• T_{prep} is the latency of the Prepare() function on node 0.

• T_{cap} denotes the latency of the FPGA context capture on node 0.

$$T_{\text{cap}} = \frac{C_{\text{FPGA}}}{mwbw_0} \tag{2}$$

• $\mathrm{T}_{\text{transf\_FPGA}}$ denotes the time for transferring FPGA context from node 0 to node 1.

$$T_{\text{transf\_FPGA}} = \frac{C_{\text{FPGA}}}{bw} \tag{3}$$

• T_{res} is the latency of FPGA context restoration on node 1.

$$T_{\text{res}} = \frac{C_{\text{FPGA}}}{mrbw_1} \tag{4}$$

• $\mathrm{T}_{\text{transf\_dirty}}$ denotes the time for transferring the dirty data from node 0 to node 1.

$$T_{\text{transf\_dirty}} = \frac{C_{\text{dirty}}}{bw} \tag{5}$$

• T_{down} is the downtime of the migrated task, which is the time period from when the host on node 0 calls the Prepare() function to when it receives a migration acknowledge. It is noted that the FPGA context capture on node 0 and the dirty data transfer can be performed simultaneously. The restoration of FPGA context on node 1 can be also performed simultaneously with receiving dirty data. Therefore, the downtime can be presented as follows:

$$T_{\text{down}} = T_{\text{prep}} + T_{\text{transf\_FPGA}} + Max\{(T_{\text{cap}} + T_{\text{res}}), T_{\text{transf\_dirty}}\} \tag{6}$$

Using equations (2), (3), (4), and (5), the downtime can be determined by:

Table 12. Experimental Setup for Task Migration.

EDA Tool	Vivado 2017.2
FPGA	XCZU9eg-ffvb1156-2L-e-es1
Evaluation Board	Zynq UltraScale
Clock frequency	100 MHz
Host CPU	ARM Cortex-A53
Operating system	Petalinux 2017.2
MPI version	OpenMPI 3.0.0

Table 13. Data context size in memory

Apps	$\mathrm{C}_{\text{non-dirty}}$ (kB)	$\mathrm{C}_{\text{dirty}}$ (kB)	Total data (C_{data}) (kB)
Mat-Mul	8,192	4,096	12,288
Dijkstra	549.68	12.50	562.18
Stencil	0	2,048	2,048
S-Search	32,768	8	32,776

$$T_{\text{down}} = T_{\text{prep}} + \frac{C_{\text{FPGA}}}{bw} + Max\{\frac{C_{\text{FPGA}}}{mwbw_0} + \frac{C_{\text{FPGA}}}{mrbw_1}, \frac{C_{\text{dirty}}}{bw}\} \tag{7}$$

As can be seen from the equation (6), the downime does not include the readback-of-bitstream latency (replaced by T_{cap}) and the configuration latency, which are time-consuming. T_{cap} and T_{res} can be also overlapped by $\mathrm{T}_{\text{transf\_dirty}}$. Therefore, the downtime is expected lower than that of the previous scheme using the readback-of-bitstream method.

• The migration duration T_{mig} is measured from when the pre-copy starts to when node 0 receives a migration acknowledgment.

$$T_{\text{mig}} = Max\{T_{\text{config}} + T_{\text{res}}, T_{\text{pre-copy}} + T_{\text{down}}\} \tag{8}$$

The migration duration can be presented using expressions (1) and (4) as follows:

Table 14. Capturing/restoring latency

Apps	T_{prep} UltraS (ms)	T_{cap} UltraS (ms)	T_{res} Zedboard (ms)	T_{config} Zedboard (ms)
Mat-Mul	0.002	0.021	0.023	32.093
Dijkstra	0.001	0.003	0.007	31.817
Stencil	0.014	0.038	0.040	31.727
S-Search	0.007	0.008	0.010	32.033

Table 15. Migration downtime, migration duration and their breakdown (ms)

Apps	$T_{\text{pre-copy}}$	$T_{\text{transf\_FPGA}}$	T_{dirty}	T_{down}	T_{mig}
Mat-Mul	142.598	0.809	63.940	64.807	207.455
Dijkstra	45.944	0.540	2.287	2.781	48.729
Stencil	0	1.174	51. 351	52.575	52.646
S-Search	524.373	0.555	0.637	1.251	525.627

$$T_{\text{mig}} = Max\{T_{\text{config}} + \frac{C_{\text{FPGA}}}{mrbw_1}, \frac{C_{\text{non-dirty}}}{bw} + T_{\text{down}}\} \tag{9}$$

In expression (9) for the migration duration, the FPGA configuration time can be overlapped by the pre-copy, the transfer of FPGA context, and the copy of the dirty data. Such overlap hightlights the advantages of our scheme.

7.4 Memory Usage

To reduce the FPGA configuration time, all the bitstreams stored in the local storage are pre-loaded to the memory on the destination node. We assume that the local storage stores k bitstreams, and the bitstream size is btr_size. The memory consumed by the bitstreams is k * btr_size. The memory footprint for checkpointing a hardware task is the amount of FPGA context C_{FPGA}. Therefore, the memory usage on the destination node for migrating the task is C_{FPGA} + k * btr_size, while on the source node, it is only C_{FPGA}.

Table 16. Memory context size while scaling the graph size in Dijkstra benchmark

Vertices	Edges	$C_{\text{non-dirty}}$ (kB)	C_{dirty} (kB)	C_{data} (kB)
200	1,193	4.66	1.56	6.22
400	9,545	37.29	3.12	40.41
800	35,283	137.82	6.25	144.07
1,600	140,719	549.68	12.50	562.18
3,200	461,723	1,803.61	25.00	1,828.61
6,400	1,433,782	5,600.71	50.00	5,650.71

7.5 Evaluation

Table 12 shows additional experimental setup for hardware task migration in heterogeneous FPGA computing. In our experiments, tasks are migrated from a Zynq Ultrascale board to a Zedboard in a heterogeneous FPGA cluster connected via an Ethernet network. The four benchmarks have different sizes of non-dirty and dirty data in memory as shown in Table 13. As can be seen from Table 14, first, the capturing latency (T_{cap}), restoring latency (T_{res}), and the $T_{\text{tranf\_FPGA}}$ depend linearly on the FPGA context (C_{FPGA}). Second, $T_{\text{pre-copy}}$ and $T_{\text{transf\_dirty}}$ also depend linearly on $C_{\text{non-dirty}}$ and C_{dirty}, respectively as shown in Table 15. Third, the migration downtime (T_{down}) and the migration duration (T_{mig}) are matched with the expressions (6) and (8). Therefore, the experimental results show the correctness of the expressions (1), (2), (3), (4), (5), (6), and (8).

However, all the four benchmarks show the case that ($T_{\text{pre-copy}} + T_{\text{down}}$) > ($T_{\text{config}} + T_{\text{res}}$). As a result, in the expression (8), Max{ $T_{\text{config}} + T_{\text{res}}$, $T_{\text{pre-copy}} + T_{\text{down}}$} = $T_{\text{pre-copy}} + T_{\text{down}}$. In order to verify the expression more in other cases, the benchmark Dijkstra is used while scaling the number of vertices in the graph. The non-dirty ($C_{\text{non-dirty}}$), dirty (C_{dirty}), and total context (C_{data}) in memory are also scaled as in Table 16. The corresponding migration downtime is depicted in Figure 30. Basically, the downtime depends linearly on the amount of dirty context C_{dirty}. It is also independent of $C_{\text{non-dirty}}$. In contrast, the migration duration T_{mig} depends on both $C_{\text{non-dirty}}$ and C_{dirty} as in the expression (9). However, when

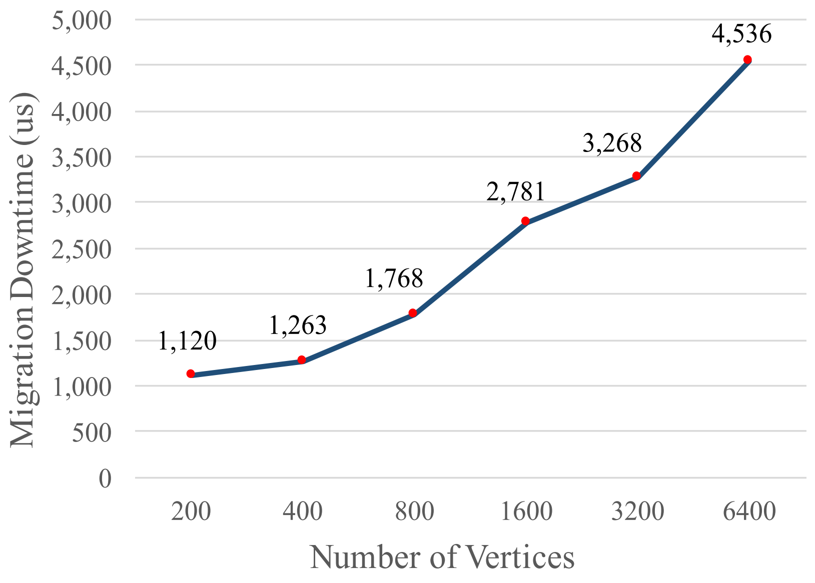


Figure 30. Migration downtime while scaling the graph size.

the number of vertices is small enough ($C_{\text{non-dirty}}$ and C_{dirty} are small), ($T_{\text{pre-copy}} + T_{\text{down}}$) can be less than ($T_{\text{config}} + T_{\text{res}}$). As a result, T_{mig} is kept constant at ($T_{\text{config}} + T_{\text{res}}$) until the number of vertices ($C_{\text{non-dirty}}$ and C_{dirty}) increase enough as shown in Figure 31. After that, the migration duration depends linearly on $C_{\text{non-dirty}}$ and C_{dirty}.

The experimental results also reveal the efficiency of our migration scheme. First, since the HDL-based checkpointing is employed in our scheme, the capturing latency and restoring latency (less than 0.040 ms) are much smaller than the capturing (readback) (953 ms) and restoring (configuration) (52 ms) latency in the readback-of-bitstream-based scheme [39]. This work did not consider the transfer of FPGA context, non-dirty, and dirty context. Therefore, the migration downtime and migration duration were not evaluated. Second, the capturing (T_{cap}) and restoring (T_{res}) latency can be hidden by the transfer of dirty context as in the expression (6). As a result, they have no impact on the migration downtime. Third, the configuration latency (T_{config}) can be hidden by the pre-copy and the transfer of FPGA context and dirty context. The latency also has no impact on the migration downtime. Fourth, the expressions (7) and (9) shows

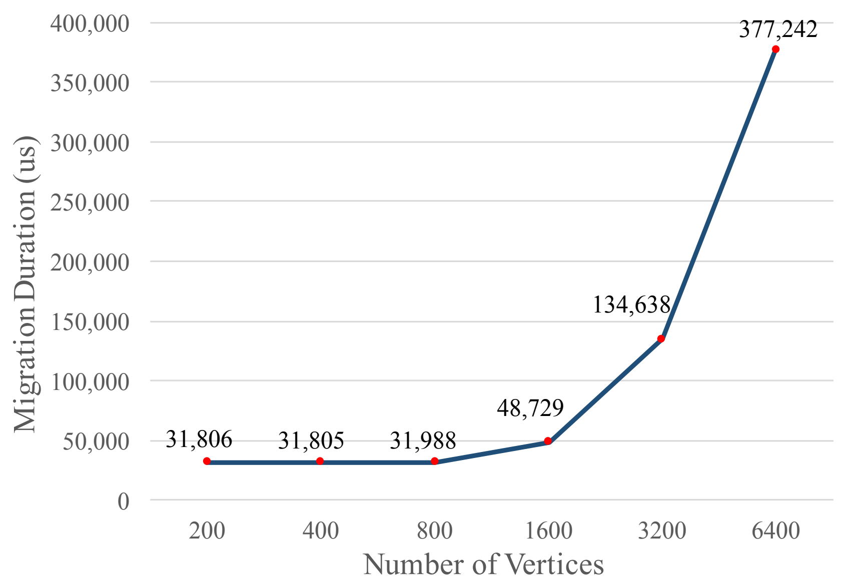


Figure 31. Migration duration while scaling the graph size.

that the downtime can be reduced much compared with the readback scheme since the amount of FPGA context (less than 15 kB) is much smaller than the bitstream size (4 MB). Although the downtime and migration duration were not evaluated in [39], it can be noted that the downtime in [39] must be more than the sum of the readback latency and configuration latency (1005 ms). However, in our scheme the downtime is only 1.251 ms (less than 0.13%) for the S-Search benchmark. Furthermore, small amounts of FPGA context also help to reduce the total network traffic in the cluster/cloud.

8. Conclusion

FPGAs have been playing an important role in high performance computing and large-scale computing, such as datacenters and clusters/clouds due to their high performance, energy efficiency, and flexibility. Dependability and Scalability of FPGA computing have risen as essential issues to achieve higher reliability, performance, and energy efficiency. This thesis studied dependable and scalable FPGA computing based on checkpointing in the HDL level. In this work, we proposed two checkpointing architectures along with a checkpointing mechanism on FPGA, that are transparent to application and portable across hardware platforms with different technology. We optimized the checkpointing design by analysing the original HDL source code to reduce the hardware resource utilization. We also provided two Python-based tools to generate checkpointing infrastructures according to the two architectures. Our evaluation shows that although the checkpointing architectures and mechanism cause significant hardware overhead, they have negligible impact on performance, power consumption, and maximum clock frequency.

We then described how FPGA computing is scalable by proposing a multitasking scheme on FPGA and a hardware task migration scheme in heterogeneous FPGA computing. The two schemes reveal great improvements over previous schemes using the readback-of-bitstream method in terms of latency and memory footprints. In addition, our checkpointing method (HDL-based) allows tasks to be migrated among different FPGA fabrics (heterogeneous). The management of snapshot consistency in our checkpointing mechanism allows the two schemes to be demonstrated on more complicated and realistic application benchmarks.

For future work and potential applications, we will consider the following issues:

- Can FPGA resources be virtualized and shared by multitasking with run-time partial reconfiguration? We will implement and evaluate task switching on partial reconfigurable partitions. Hardware overhead caused by decoupler between static regions and reconfigurable regions will be also evaluated.

- Can hardware tasks be migrated between partial reconfigurable regions on heterogeneous FPGAs?

- Can a hardware application running on multiple FPGA fabrics be resilient using the proposed checkpointing mechanism and management of snapshot consistency?

Acknowledgments

I would like to thank all members in Computing Architecture lab, my family, my friends for supporting me during my doctoral course.

To professor Yasuhiko Nakashima, he supported me a lot from when I started applying MEXT scholarship. He recommended many good papers to me and taught me how to find research ideas. He guided me how to prepare slides and make a good oral presentation. He also gave me advice on how to write a good paper. During my PhD research, he gave me many valuable comments on how to solve problems in research and implementation.

To professor Michiko Inoue, she gave me many valuable comments and interesting questions in mid-term presentation and during my PhD thesis defense. She showed me more motivation and other aspects of my research. Her comments and advice encourage me to expand my research in near future.

To associate professor Takashi Nakada, my advisor, he has supported me a lot since he came to NAIST one and a half year ago. He gave me many valuable comments and advice on my research. He helped me a lot in improving my manuscripts. He always encourages me to submit manuscripts to journals and international conferences.

To assistant professor Renyuan Zhang, he gave me some good comments and suggestions in my PhD thesis defense. He showed me his enthusiasm for doing research.

To assistant professor Tran Thi Hong, she is a good example of women in research. She also encouraged me to write papers.

To assistant professor Shinya Takamaeda, he supported me much in using tools, Linux when I first came here. He showed me his motivation and dream in research.

To other students in my lab, they supported me a lot in Japanese, tools, Linux. We enjoyed student life together.

To Vietnamese Association in NAIST, they helped me a lot in daily life. We shared fun, sadness, difficulty, and happiness together. We played games and traveled a lot together. They cooked a lot of Vietnamese foods and held many parties. Thanks to their support and friendship, I did not get homesick much.

References

[1] Andrew Putnam et al, “A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services,” Communications of the ACM, vol. 59, no. 11, pp. 114-122, Nov. 2016.

[2] Andrew Putnam, “FPGAs in the Datacenter – Combining the Worlds of Hardware and Software Development,” Proceedings of Great Lakes Symposium on VLSI, pp. 5, May. 2017.

[3] Babak Falsafi et al, “FPGAs versus GPUs in Data centers,” IEEE Micro, Vol. 37, No. 1, pp. 60 – 72, 2017.

[4] Naif Tarafdar el al, “Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center,” Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA2017), pp. 237 – 246, Feb 2017.

[5] Adrian M. Caulfield el al, “Configurable Clouds,” IEEE Micro, Vol. 37, No. 3, 2017.

[6] Amazon EC2 F1 Instances (Preview)” 2017; [Online]. Available: http://aws.amazon.com/ec2/instance-types/f1

[7] J. Ouyang , “SDA: Software-Defined Accelerator for Large-Scale DNN Systems,” in Proc. Hot Chips 26 Symposium, 2014.

[8] J. Weerasinghe , “Enabling FPGAs in Hyperscale Data Centers,” in Proc. IEEE 12th Int’l Conf. Ubiquitous Intelligence and Computing, 12th Int’l Conf. Autonomic and Trusted Computing, and 15th Int’l Conf. Scalable Computing and Communications (UIC-ATC-ScalCom), 2015.

[9] A.G. Lawande, A.D. George, H. Lam, ”Novo-G#: A Multidimensional Torus-Based Reconfigurable Cluster for Molecular Dynamics”, Concurrency and Computation: Practice and Experience, vol. 28, no. 8, pp. 2374-2393, 2016.

[10] 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.

[11] 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.

[12] Dirk Koch, Christian Haubelt and Jurgen Teich, “Efficient Hardware Checkpointing - Concepts, Overhead Analysis, and Implementation,” FPGA’07, pp.188-196, February 18–20, 2007, Monterey, California, USA.

[13] 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.

[14] 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.

[15] Hoang Gia Vu, Supasit Kajkamhaeng, Shinya Takamaeda-Yamazaki and Yasuhiko Nakashima, “CPRtree: A Tree-based Checkpointing Architecture for Heterogeneous FPGA Computing,” 4th International Symposium on Computing and Networking (CANDAR 2016), Nov 2016.

[16] Hoang Gia Vu, Shinya Takamaeda-Yamazaki, Takashi Nakada, Yasuhiko Nakashima, ”CPRring: A Structure-aware Ring-based Checkpointing Architecture for FPGA Computing”, The 25th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM2017) (poster), pp. 192, May 2017.

[17] Hoang-Gia VU, Shinya TAKAMAEDA-YAMAZAKI, Takashi NAKADA, and Yasuhiko NAKASHIMA, ”A Tree-based Checkpointing Architecture for the Dependability of FPGA Computing,” IEICE Transactions on Information and systems, Vol.E101-D, No.2, pp.xxx-xxx, Feb. 2018.

[18] Hoang-Gia VU, Takashi NAKADA, and Yasuhiko NAKASHIMA, "Efficient Multitasking on FPGA Using HDL-based Checkpointing," 14th International Symposium on Applied Reconfigurable Computing (ARC2018), Lecture Notes in Computer Science, Springer, Book Title: Applied Reconfigurable Computing. Architectures, Tools, and Applications, Vol. 10824, Chapter. 47, 2018.

[19] 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 2014.

[20] 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.

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

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

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

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

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

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

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

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

[29] Alban Bourge, Olivier Muller and Frederic Rousseau, "Generating Efficient Context-Switch Capable Circuits through Autonomous Design Flow," ACM Transactions on Reconfigurable Technology and Systems, Vol. 10, No. 1, Dec 2016.

[30] 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.

[31] 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).

[32] 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.

[33] Shinya 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.

[34] Minoru Watanabe, Kentaro Sano, Shinya Takamaeda, Take- fumi Miyoshi, Hironori Nakajo: "Japanese High-level Synthesis Tools for FPGA Hardware Acceleration," IEICE Transactions on Communications, Vol. J100-B, No.1, pp.1-10 (in Japanese), 2016.

[35] Shinya Takamaeda-Yamazaki, Kenji Kise and James C.Hoe: "PyCoRAM: Yet Another Implementation of CoRAM Memory Architecture for Modern FPGA-based Computing," Workshop on the Intersections of Computer Architecture and Reconfig- urable Logic (CARL 2013) (Co-located with MICRO-46), De- cember 2013.

[36] Wesley J. Landaker, Michael J. Wirthlin, and Brad L. Hutchings, "Multitasking Hardware on the SLAAC1-V Reconfigurable Computing System," 12th International Conference on Field-Programmable Logic and Applications (FPL2002), pp. 806-815, 2002.

[37] L. Levinson, R. Manner, M. Sessler, and H. Simmler, "Preemptive multitasking on FPGAs," FCCM2000, pp. 301-302, 2000.

[38] M. Happe, A. Traber, and A. Keller, "Preemptive Hardware Multitasking in ReconOS," ARC 2015, pp. 79-90, 2015.

[39] Oliver Knodel, Paul R. Genssler and Rainer G. Spallek, "Migration of long-running Tasks between Reconfigurable Resources using Virtualization," ACM SIGARCH Computer Architecture News, vol. 44, no. 4, Sep. 2016.

[40] https://www.open-mpi.org.