INTEGRATION, the VLSI journal 77 (2021) 180–192

Contents lists available at ScienceDirect

Integration

journal homepage: www.elsevier.com/locate/vlsi

Efficient hardware task migration for heterogeneous FPGA computing using HDL-based checkpointing☆

Hoang-Gia Vu [a,*], Takashi Nakada [b], Yasuhiko Nakashima [b]

[a] Le Quy Don Technical University, Vietnam
[b] Nara Institute of Science and Technology, Japan

ARTICLE INFO

Keywords:
Hardware task migration
FPGA
Checkpointing
Cluster

ABSTRACT

Task migration plays an important role in load balancing and energy savings in data centers. It also challenges service providers to minimize service interruptions during task migration. FPGA computing requires checkpointing as an essential function for hardware task migration. However, the current methods of implementing such a function for FPGAs have a high cost in hardware resources and significant degradation in performance. To overcome these problems, in this paper we propose a system using checkpointing at the hardware description language (HDL) level for hardware task migration. First, we propose a hardware task migration scheme in which checkpointing procedures and context transfer can overlap to reduce the service downtime. Second, we present a new checkpointing architecture for FPGAs that flattens the structure of nested modules at the HDL level. Third, we propose a static analysis of the original HDL source code to reduce the cost of hardware. Fourth, we introduce a Python-based tool to generate the checkpointing architecture at the HDL level. We evaluated our checkpointing architecture and the migration scheme using four application benchmarks running on a heterogeneous FPGA cluster. Our evaluations showed that the migration downtime was minimized at only 1.251 ms in the S-Search benchmark. When compared with a tree-based checkpointing architecture, the proposed architecture with the static analysis can reduce the LUT overhead by up to 50%, on the average. The checkpointing hardware caused small degradation in the maximum clock frequency (1.66% on the average), and consumed small memory footprints. Other comparisons with the previous hardware task migration scheme highlight the advantages of our migration scheme.

1. Introduction

Hyperscale clouds, with millions of servers, provide the ability to run an enormous number of applications and workloads, from Web services through data processing to artificial intelligence (AI) and the Internet of Things. However, the end of Moore's law and the subsequent slowdown in CPU scaling has raised the need for hardware specialization to increase total performance and energy efficiency. Due to high computational capabilities, reconfigurability, power efficiency, and the advantages of customizing hardware for domain-specific applications, Field Programmable Gate Arrays (FPGAs) are now widely deployed in modern data centers [1–5]. Amazon's EC2 F1 [6], Baidu's SDA [7], IBM's FPGA fabric [8], and Novo-G [9] are also FPGA deployments with detailed descriptions. As in CPU-based data centers, FPGA-based data centers are expected to employ task migration to save energy, balance the load, and prepare production servers for maintenance.

Compared with the migration of virtual machines [10–12], hardware task migration also requires copying the memory and processor state, or FPGA context, from the source node to the target node. However, hardware task migration also requires several additional procedures, such as FPGA capture on the source node, along with FPGA configuration and FPGA restoration on the target node. These requirements raise a question regarding how to overlap data transfer, FPGA capture, FPGA configuration, and FPGA restoration to minimize migration duration and migration downtime.

In addition, FPGAs in future data centers must be able to serve a huge number of various tasks. Different tasks are suitable for different FPGA structures. For instance, signal-processing tasks may require more DSP

☆ We would like to thank Prof. Michael Barker for proofreading and many valuable comments in this manuscript. This work is supported in part by Mazda foundation and JSPS KAKENHI Grant Number JP17H00730.

* Corresponding author.
E-mail address: giavh@lqdtu.edu.vn (H.-G. Vu).

https://doi.org/10.1016/j.vlsi.2020.11.011
Received 11 May 2020; Received in revised form 11 September 2020; Accepted 15 November 2020
Available online 3 December 2020
0167-9260/© 2020 Elsevier B.V. All rights reserved.

blocks, while data streaming tasks may require more block RAMs. Tasks with a low-latency constraint need to be implemented on high clock frequency FPGAs, thus requiring FPGAs with the advance in process technology. Other tasks can be mapped onto lower clock frequency or lower process technology FPGAs for more efficiency in cost and energy. Therefore, future data centers should employ various kinds of FPGAs for efficiency in hardware resource allocation, performance, energy, and cost. Those data centers are called heterogeneous FPGA data centers in this paper. Although each task is suitable to be implemented on an FPGA, it can be also implemented on other kinds of FPGAs. In the two following cases, task migration should be performed between different FPGAs. Firstly, a task on an FPGA may share memory bandwidth with other tasks on the host CPU. If there is another FPGA available with higher memory bandwidth, then the task should be migrated to the available FPGA so that it can utilize more memory bandwidth for higher performance. Secondly, if a data center receives a task request (task A), but the suitable FPGA (FPGA 1) is engaged in executing another task (task B), then task A will be executed on another FPGA (FPGA 2). However, when task B finishes, task A should be migrated from FPGA 2 to FPGA 1. In both of the above cases, if the hardware task is not migrated to the other FPGA, it will run slower because of limited memory bandwidth or non-suitable hardware platform. Furthermore, if the entire application is re-run on another FPGA instead of transferring the checkpoint data, the execution time from the beginning will be a waste. Therefore, we believe that hardware task migration should be performed if the execution time of the task is much higher than the migration time. Otherwise, the migration is not required.

The previous approach [13] to hardware task migration uses a readback-of-bitstream method that has several drawbacks which can prevent task migration from being deployed in FPGA-based data centers. First, the readback-of-bitstream method [14–19] cannot guarantee the consistency of the readback bitstream with the snapshots of other components. For example, a bitstream taken while the FPGA is accessing the main memory will not be consistent with the main memory. As a result, the application cannot be resumed correctly. Second, the readback bitstream is not sufficient to resume the normal operation of dedicated blocks, which have outputs delayed compared with inputs [20]. Third, as Kalte and Porrmann reported [14] less than 8% of the data in the bitstream was useful, thus 92% of readback time was wasted. Fourth, snapshots taken by the readback-of-bitstream method cannot be used to resume with FPGAs using different architectures or technologies. For instance, a bitstream read back from a Xilinx Zedboard cannot be used to resume an application on a Zynq UltraScale+. Fifth, this approach requires either sending bitstreams, including the Intellectual Property (IP) configuration, over networks or manipulating bitstream before sending state information [14]. This approach either raises issues concerning FPGA security [21,22] or consumes additional time and additional computing resources for processing bitstream. For security reasons, readback may even be disabled after an FPGA loads an encrypted bitstream [21].

We believe that these five drawbacks can be overcome by using HDL-based checkpointing for hardware task migration. Before discussing the advantages of the HDL-based checkpointing, we define FPGA checkpointing as follows. FPGA checkpointing is to capture the state of FPGA operation so that the state can be used to resume the FPGA operation later. In order to perform FPGA checkpointing in our method, additional circuits are inserted into the application circuit to capture and restore the state of FPGA operation. The insertion is in the HDL level, in which the additional circuits are not inserted randomly but in a specific way that is called checkpointing architecture. Particularly, the HDL source code is parsed into an abstract syntax tree. Then the checkpointing mechanism is inserted before the HDL with checkpointing is generated. In this paper, FPGA checkpointing is employed to migrate hardware tasks from an FPGA to another FPGA. For the advantages of the HDL-based checkpointing, first, channel finite state machines and request throttling circuits can be inserted into the application circuit at the HDL level to manage communication channel states, thus guaranteeing consistent snapshots [20]. Second, HDL-based checkpointing takes into account dedicated blocks by inserting additional registers to store consecutive values of these blocks' input signals [20]. These values are then used to virtualize the output signals of the blocks, allowing their normal operation to be resumed. Third, since there is no readback of a bitstream, there is no wasted time. Fourth, a checkpointing architecture designed in HDL allows hardware context captured on an FPGA to be resumed on another FPGA which has a different technology and/or a different architecture, provided that the two different bitstreams are generated from the same HDL design. In other words, HDL-based checkpointing allows a task to be migrated between different kinds of FPGAs in a heterogeneous FPGA system. Fifth, hardware task migration in our method is more secure than the migration using readback of bitstream because of two reasons. a) With hardware task migration using HDL-based checkpointing, instead of a configuration bitstream, only the register context and RAM context are sent over the networks. It is noted that the values of registers and RAMs in the FPGA context are arranged in a specific way according to the checkpointing architecture. Therefore, without knowledge of the checkpointing architecture and the configuration bitstream, attackers have no information on the task operation. Thus, context encryption is not required. In contrast, hardware task migration using readback of bitstream always send the configuration bitstreams over the networks. With the configuration bitstreams and information about the used FPGAs, attackers may know everything about the task operation. b) Since our migration method does not require readback of the bitstream, this allows FPGAs to be locked after configuration. Therefore, the configuration bitstreams are confidential. In contrast, in the migration method using readback of the bitstream, FPGAs cannot be locked, thus being not confidential to attackers.

Inserting checkpointing circuits in the HDL level offers several advantages over doing so in other levels of the design phase. First, compared to the netlist level [23], the HDL level provides higher programmability for designers to insert checkpointing circuits and explore design space. Also, HDL designs are independent of technology, while designs in the netlist level may be dependent on technology and FPGA architectures. Second, compared to a higher level of abstraction in HLS [25,26], working in the HDL level brings with it more flexibility and full customizability to designs. HDL also is widely used in hardware design. Therefore, we chose HDL-based checkpointing as the basis of our migration scheme. There are two restrictions in the HDL we consider. First, the original HDL source code should be pure, that means the source code consists of HDL statements only. Second, the HDL design should operate with a single clock domain.

It should be noted that inserting checkpointing circuits also brings with it hardware overhead for the application circuit. However, we believe that there is some room to improve checkpointing architectures to minimize the hardware overhead. In many cases, hardware resources utilized for normal operation can also be employed for checkpointing to reduce the total hardware consumption.

To deal with the above issues, we make the following specific contributions:

1) We propose a scheme for migrating a hardware task in heterogeneous FPGA clusters. In this scheme, FPGA capture, FPGA configuration, and FPGA restoration can overlap with data transfer, thus minimizing migration duration and migration downtime.
2) We propose CPRflatten - a new checkpointing architecture for FPGAs based on flattened module structures at the HDL level. The architecture is transparent to applications and portable across hardware platforms.
3) We provide a static analysis of HDL original source code to re-use hardware resources for checkpointing, thus reducing hardware consumption caused by checkpointing.
4) We present a Python-based tool which modifies HDL source code and inserts checkpointing circuits for CPRflatten.

2. Related work

For migration of virtual machines, many algorithms and methods aimed at reducing migration downtime and duration have been proposed based on post-copy and pre-copy [10–12]. While migration is a well-explored topic in CPU-based cluster/cloud computing, there have been few studies of hardware task migration. O. Knodel et al. [13] presented an approach to task migration between reconfigurable resources using the readback-of-bitstream method. However, as discussed in Section 1, this approach has several drawbacks. Apart from those drawbacks, other issues and parameters such as data transfer, migration downtime, and migration duration, were not considered and evaluated in that work. In this section, we will discuss related work on FPGA checkpointing. Along with the readback-of-bitstream method, there have been three other approaches described in the literature.

The first approach is the netlist-based method. In this method described in Ref. [23], a scan-chain structure was employed. Scan multiplexers were inserted as checkpointing infrastructures at the netlist level to connect flip-flops, thus building scan chains. Scan chains were also used in Ref. [24] to observe the state of the full chip and to control internal signals, but not for checkpointing. Although the netlist-based method with scan chains is an attractive option for hardware test and verification, for hardware checkpointing it also has several drawbacks, such as poor programmability and technology dependence. Furthermore, in this scan-chain netlist-based method, only flip-flops were extracted while RAM contents were not considered. Consistent snapshots were not considered in this method. Those drawbacks make the checkpointing method incomplete.

The second approach is the high-level synthesis based (HLS-based) method. Alban Bourge et al. [25,26] presented a high-level synthesis design flow for manipulating the intermediate representation of an HLS tool to insert a scan chain into the initial circuit. The main contribution of this work is the checkpoint selection, which limits checkpointing to only some states of the finite state machine. In the checkpoint selection, the automated 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 of consistent snapshots of FPGAs and other components. Taking this issue into account, the states where checkpointing can be performed should depend on the state of communication channels between the FPGA and other components [20]. In this case, the checkpoint selection may no longer be feasible. Furthermore, the authors limited their application benchmarks to use of a specific HLS tool generating application circuits with a single finite state machine. This constraint may prevent developers from designing complicated applications.

The third approach is the HDL-based method. In Ref. [27], the authors revealed a method for capturing and restoring state-holding elements, such as registers, BRAMs, finite state machines, and FIFOs, by providing a context interface. However, when evaluating the LUT utilization of additional hardware, they only evaluated LUT consumption in the context interface, even though the LUT consumption caused by inserting multiplexers alongside registers for restoring context was significant. Furthermore, they did not propose a checkpointing architecture to deal with the structures of nested HDL modules, even though dealing with these structures is more complicated than checkpointing a single module. Our previous work on the HDL-based method [20,28] introduced three improvements. First, it presented a reduced set of state-holding elements for the definition of FPGA context. This set can be used to checkpoint dedicated blocks on the FPGA. Second, we also introduced a method to guarantee the consistency of snapshots between the FPGA and other components. Third, we proposed CPRtree - a tree-based checkpointing architecture for the FPGA. In CPRtree, the structure of nested modules can be considered as a model of a tree, in which the top module is the foot of the tree while sub-modules are nodes of the tree. Therefore, a checkpointing architecture based on the model of a tree is an approach to deal with the complicated structure of nested modules. Each module has its corresponding checkpoint/restart infrastructure, called CPR node, and the CPR nodes of all modules form a checkpointing tree. However, the hardware overhead in our previous work was still high. Therefore, in this work we propose a new checkpointing architecture to reduce the hardware overhead. The basics of the new architecture has been presented in Ref. [29]. In this paper, we provide more details on the architecture along with a Python-based tool to generate the checkpointing architecture in HDL. We also propose a hardware task migration scheme employing this checkpointing architecture.

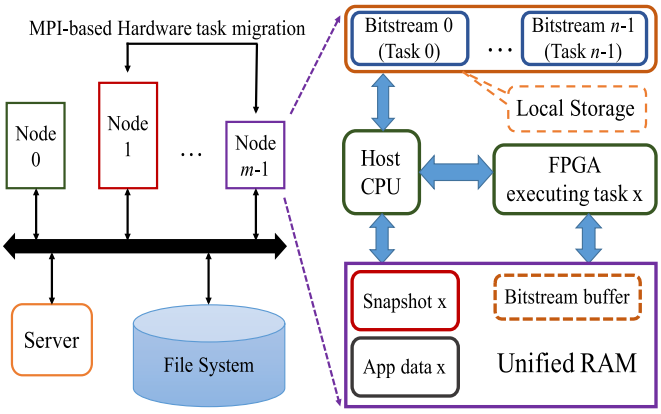

Fig. 1. Hardware task migration in a heterogeneous FPGA cluster.

It is noted that the previous works [23,25,26] performed FPGA checkpointing and task switching, but no hardware task migration. However, we believe that hardware task migration can be performed between different kinds of FPGAs on condition that the corresponding checkpointing insertion is performed in a level of design phase before associated with a specific FPGA fabric. Therefore, the checkpointing solutions in the netlist level [23], the HLS level [25,26], and the HDL level as our approach can be employed to migrate hardware tasks between different kinds of FPGAs.

3. Hardware task migration scheme

In this section, we describe the proposed hardware task migration scheme for heterogeneous FPGA clusters. First, we present the proposed structure of such clusters. Second, we explain the timing diagram of our proposed scheme. Finally, we discuss the memory footprint of our scheme.

3.1. Hardware task migration structure

Fig. 1 shows a heterogeneous FPGA cluster with hardware task migration. 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 main memory. FPGAs in nodes may be different in architecture, technology, and vendor. In each node, bitstreams for potential tasks running on the corresponding FPGA are stored in the corresponding local storage. For example, if it is assumed that task 0, task 1, …, and task $n-1$ can be migrated to a node, then the corresponding bitstream 0, bitstream 1, …, and bitstream $n-1$ must be prepared and stored in the local storage of that node. For hardware task migration between two nodes, HDL-based checkpointing is employed in both the source node and the target node to capture the FPGA context and to restore the context, respectively. The captured context is then written to the main memory at a given physical address as a snapshot of the corresponding task. The snapshot is not saved to the local storage or the file system. Instead, it is sent to the target node using a message passing interface (MPI) [30]. On the target

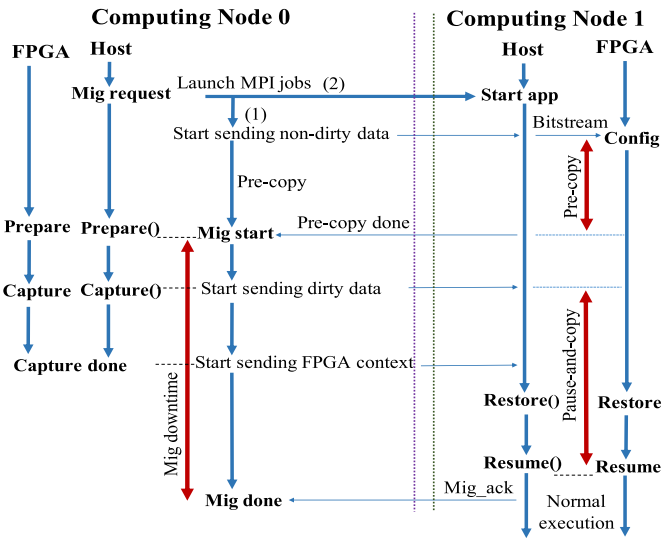

Fig. 2. Timing diagram of hardware task migration.

node, the snapshot is read from the main memory to the FPGA and restored to the state-holding elements before normal operation is resumed. Before the normal operation is resumed, the FPGA on the target node must be configured with the corresponding bitstream. To reduce the configuration time, the *n* potential bitstreams are pre-loaded to a given physical address area in the main memory on the target node, called a bitstream buffer, before being written to the FPGA.

There are several requirements for hardware task migration in an FPGA cluster. First, it requires contiguous memory allocation for both the application data and the snapshot in the main memory. Second, since the task context contains the physical address of the application data, the same physical address in the main memory must be allocated to the application data of the migrated task on both the source node and the target node. These two requirements are easily satisfied in stand-alone systems but more difficult in operating systems (OS) using virtual addresses such as Linux. For the latter, a memory space for the FPGA should be reserved when building the OS so that other software applications will not touch the space.

One of the key advantages of our approach is that it supports hardware task migration between different kinds of FPGAs in a heterogeneous FPGA system. As described in the Introduction, a readback bitstream from one type of FPGA cannot be used to configure different types of FPGAs. Therefore, the readback-of-bitstream method cannot support hardware task migration between different kinds of FPGAs. However, in our method, checkpointing circuits are inserted at 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 can be used to generate different bitstreams for different FPGAs. Instead of transferring the whole bitstream from the source node to the target node, only checkpoints, the values of the state-holding elements, are transferred. On the target node, after configuring the FPGA with the corresponding bitstream, the checkpoints are restored for the hardware task using the state-holding elements. Then the normal operation can be resumed. Therefore, our HDL-based checkpointing method allows task migration between different kinds of FPGAs.

3.2. Hardware task migration timing diagram

Fig. 2 depicts a timing diagram for 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), on the same host, sends the data context in main memory and FPGA context to node 1. The other program (2), on the host of node 1, resumes the task on its FPGA. The two programs communicate via MPI. It is noted that a hardware task in our scheme uses data from the main memory as its input data, and the output data of the task are written back to the main memory. When the task is migrated to another computing node, all the hardware context along with its input data and output data in the main memory are copied to the target node. As soon as the second program (2) is launched and initiated, it configures the FPGA on node 1. At the same time, all the data in the memory area allocated to the task on node 0 are copied to node 1. It is noted that the output data of the task is written back to the main memory, and they are also copied to node 1. The copy includes two phases. In the first phase, node 0 sends the data in its main memory that will not be modified by the FPGA during execution to node 1. These data are called *non-dirty data*, while this phase is called *pre-copy* in this paper. This phase uses the blocking function MPI_Send(). The two computing nodes are synchronized by using the two blocking functions MPI_Send() and MPI_Recv() [30] to send or receive data and synchronous bytes. The second phase sends all the data in the main memory that may be modified during task execution from node 0 to node 1 after the task is paused on node 0. These data are called *dirty data*, while this phase is called *pause-and-copy* in this paper. This phase uses the non-blocking function MPI_Isend() to allow context capture and data transfer to be performed at the same time. It is noted that we have no mechanism on hardware to manage how many data are modified by the FPGA while the hardware task is being executed. Therefore, we define dirty data as the data that may be modified by the FPGA during execution. With these definitions, the size of dirty data and non-dirty data is constant during the task execution. In other words, the migration time has no impact on the size of dirty data and non-dirty data.

In detail, during the second phase, when the pre-copy finishes, to guarantee a consistent snapshot, node 0 calls the Prepare() function before calling the Capture() function to capture the FPGA context [20]. Then, after the FPGA context is captured and saved to the main memory, the host in node 0 sends the FPGA context to node 1. When both the transfer of the FPGA context and the configuration of the FPGA on node 1 finish, the Restore() function is called on node 1 to restore the context to the state-holding elements on the FPGA. After both the restoration and the transfer of dirty data finish, the Resume() function is called on node 1 to resume the task. Then, a migration acknowledgment is sent back to inform node 0 that the migration has been completed.

The advance of this timing scheme is that data transfer and checkpointing procedures can overlap to minimize migration downtime and migration duration. We believe that this advantage can be explained by mathematical expressions. In the timing diagram, we adopted the following parameter and latency definitions:

- T_{config} is the configuration time for FPGA on node 1.
- bw is the bandwidth allocated for transferring data from node 0 to node 1 during the migration.
- $mrbw_i$ is the memory read bandwidth on node *i* allocated to the FPGA during the migration.
- $mwbw_i$ is the memory write bandwidth on node *i* allocated to FPGA during the migration.
- C_{FPGA} is the amount of FPGA context.
- $C_{non\text{-}dirty}$ is the amount of non-dirty data that can be pre-copied.
- C_{dirty} is the amount of dirty data that must be copied after pausing the application.
- C_{data} is the amount of total data context in memory.

$$C_{data} = C_{non-dirty} + C_{dirty} \quad (1)$$

- $T_{pre\text{-}copy}$ is the latency required for copying the non-dirty data from node 0 to node 1.
- T_{prep} is the latency of Prepare() function on node 0.
- T_{cap} is the latency of the capture of the FPGA context on node 0.

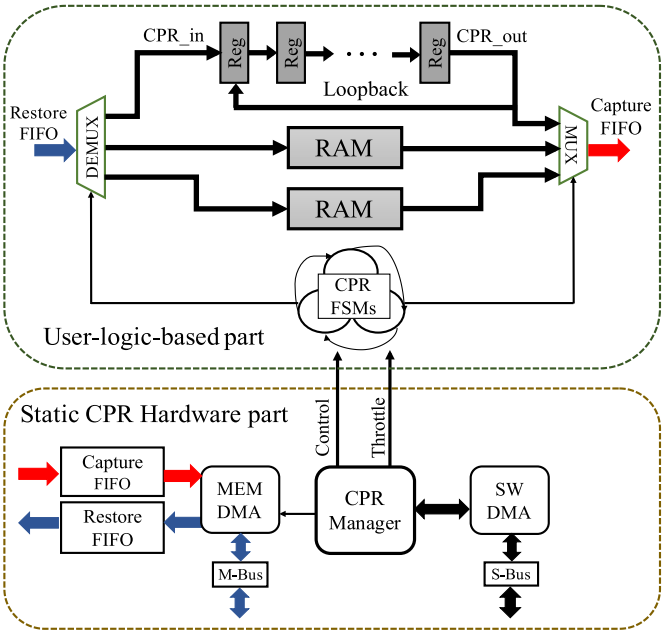

Fig. 3. Ring-based flattened checkpointing architecture.

- T_{transf_FPGA} is the latency required for transferring the FPGA context from node 0 to node 1.
- T_{res} is the latency required for restoring the FPGA context on node 1.
- T_{transf_dirty} is the latency required for transferring the dirty data from node 0 to node 1.
- T_{down} is the downtime of the migrated task, which is measured from when the host on node 0 calls the Prepare() function to when it receives a migration acknowledgment.

It is noted that the FPGA context capture on node 0 and the transfer of dirty data can be performed simultaneously. The restoration of the FPGA context on node 1 can also be performed simultaneously with the receipt of dirty data. Therefore, the downtime can be represented as follows:

$$T_{down} = T_{prep} + T_{transf\_FPGA} + Max\{(T_{cap} + T_{res}), T_{transf\_dirty}\} \quad (2)$$

$$T_{down} = T_{prep} + \frac{C_{FPGA}}{bw} + Max\{\frac{C_{FPGA}}{mwbw_0} + \frac{C_{FPGA}}{mrbw_1}, \frac{C_{dirty}}{bw}\} \quad (3)$$

As can be seen from equation (2), the downtime does not include the readback-of-bitstream latency, which is replaced by T_{cap}, or the configuration latency, both of which are time-consuming. T_{cap} and T_{res} can also be overlapped with T_{transf_dirty}. Therefore, the downtime is expected to be lower than with the readback-of-bitstream scheme.

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

$$T_{mig} = Max\{T_{config} + T_{res}, T_{pre-copy} + T_{down}\} \quad (4)$$

Using equation (3), the migration duration can then be calculated by:

$$T_{mig} = Max\left\{T_{config} + \frac{C_{FPGA}}{mrbw_1}, \frac{C_{non-dirty}}{bw} + T_{prep} + \frac{C_{FPGA}}{bw} + Max\left\{\frac{C_{FPGA}}{mwbw_0} + \frac{C_{FPGA}}{mrbw_1}, \frac{C_{dirty}}{bw}\right\}\right\} \quad (5)$$

3.3. Memory footprint

To reduce the FPGA configuration time, all the bitstreams stored in the local storage are also pre-loaded to the main memory on the target node. We assume that the local storage stores n bitstream files, and the bitstream size is *btr_size*. The memory consumed by the bitstreams then is $n \times btr\_size$. The memory footprint for checkpointing a hardware task is the amount of the FPGA context C_{FPGA}. Therefore, the memory footprint for task migration on the target node is $C_{FPGA} + n \times btr\_size$, while on the source node, it is only C_{FPGA}. In addition to this footprint in the main memory, a size of $n \times btr\_size$ is also required in the local storage of the target node to store the n bitstream files.

4. CPRflatten: A ring-based flattened checkpointing architecture for FPGA

In this section, we present our proposed checkpointing architecture. First, we provide an overview of the architecture. Second, we describe the shifting ring used for register checkpointing. Third, we provide a detailed description of the RAM capture/restore circuit used in this architecture. Fourth, we describe the checkpoint/restart (CPR) finite state machines used to capture and restore the registers and on-chip RAMs.

4.1. Overview of CPRflatten

As shown in Fig. 3, the proposed architecture can be divided into two parts: *static CPR hardware part* and the rest, called *user-logic-based part*. The static part is portable across platforms since it is fixed and does not depend on the user hardware. This part includes 1) SW DMA - direct memory access (DMA) engine for AXI4-Lite protocol to communicate with the software in the host CPU via a 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 a master bus (M-Bus). 5) CPR Manager - a checkpoint/restart (CPR) manager for 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.

The user-logic-based part includes a shifting ring for register checkpointing, capture/restore circuits for RAMs, and two CPR finite state machines (CPR FSMs) for capturing and restoring. We assume that hardware structures are flattened at the HDL level. This means that two objects can be connected at this level without considering complicated structures of nested modules. At first, the hierarchy of HDL modules is ignored to design checkpointing architecture. However, when the checkpointing architecture is implemented, we flatten the HDL designs by passing signals through nested HDL modules. Our checkpointing architecture is based on the idea of removing all connectors between the checkpoint/restart (CPR) levels of CPRtree. In this approach, Mux-based capture/restore circuits and Shift-Reg-based capture/restore circuits cannot be used, so instead a shifting matrix of register bits is employed for register checkpointing. In capture mode, the data in registers are shifted to Capture FIFO via CPR_out and MUX as in Fig. 3. The data in Capture FIFO are then written to the main memory at a given physical address area dedicated to FPGA snapshots. To ensure that the values of registers are kept unchanged after capture, the output of the matrix (CPR_out) is looped back to the input of the matrix. Thus, this forms a shifting ring, and in this paper, this architecture is called the ring-based flattened checkpointing architecture. The novelty of this contribution is the shifting ring of register bits, which is achieved by flattening nested structures of HDL modules then connecting the register bits in a ring

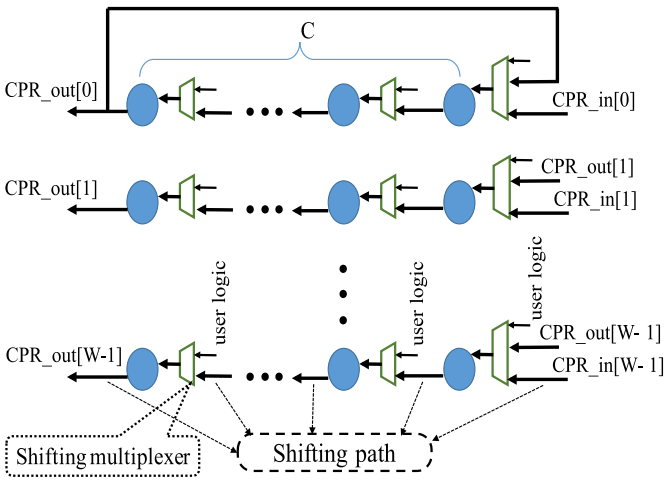

Fig. 4. Shifting ring of register bits.

manner. In this architecture, the reduced set of state-holding elements is divided into two sets: the register set and the RAM set. While capture/restore circuits for the registers are configured as a shifting ring, capture/restore circuits for RAMs are separated from each other and are separated from the shifting ring.

Pausing the operation of the user hardware is to pause all the sequential circuits by preventing the latch of all registers in the user hardware. There are no requirements of the FSMs in the applications. The application can be paused at any stage, provided that all communication channels between the user hardware and other components are idle.

How to pause a hardware task: When the host CPU needs to migrate a hardware task, it sends a command to CPR Manager. After that, CPR Manager throttles communication 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. After all channels become idle, CPR Manager throttles the user logic to pause the operation of the user hardware.

How to resume a hardware task: After checkpointing data are restored to state-holding elements, the host CPU sends a command to CPR Manager. CPR Manager then allows channel requests and the user logic operation.

4.2. Shifting ring

The shifting ring is formed as a shifting matrix of register bits with the outputs looped back to the inputs. As shown in Fig. 4, the shifting matrix of register bits is a W-by-C matrix. Let W be the data width for 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 an integer multiple of W, then a padding register is added to guarantee a multiple of W. The padding register bits are called dummy bits. The number of dummy register bits is less than W. The path used to shift values between two register bits is called a *shifting path* in this paper. As can be seen in the figure, each row of register bits includes $(C + 1)$ shifting paths. Therefore, the shifting matrix includes $W \times (C + 1)$ shifting paths. There are two advantages of using a shifting ring for checkpointing. First, it ensures that the contents of registers are kept unchanged after capture without using any additional buffer registers. Second, the shifting ring can be used for both capture and restoration processes, thus saving hardware resources.

However, adding shifting paths will increase the hardware complexity. In particular, one more input pattern will be added to each register bit, thus one input will be added to the multiplexer in front of each bit. As a result, additional LUTs will be used for such logic

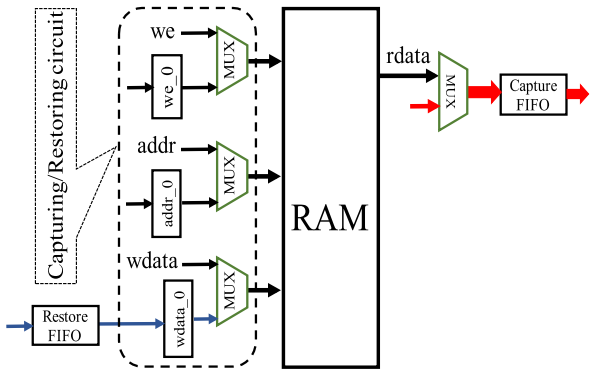

Fig. 5. RAM capture/restore circuit.

functionality. Furthermore, if the multiplexer is re-structured with more levels, this will result in a reduction of the maximum clock frequency due to the increase of the critical path.

It is noted that register ring is not a new idea, and it was applied in BIST and the JTAG protocol. However, we decided to create our register ring in HDL code because of the following reasons. First, our register ring is formed from register bits used in the HDL source code of the application design, not from all register bits available on the FPGA device as in the JTAG protocol. Therefore, the number of register bits in our ring is smaller than the number of register bits in the register ring of the JTAG protocol. As a result, the memory footprint is smaller and the capture time is shorter in our approach. Second, the context captured by our register ring can be used to resume the application on a different FPGA, while the context captured by JTAG cannot. Therefore, the JTAG protocol cannot be used for hardware task migration in heterogeneous FPGA computing.

4.3. RAM capture/restore circuit

On-chip RAM context can be captured and restored by iterating reading and writing through its entire address space. To keep the inputs and outputs of a RAM unchanged after capture to guarantee the user circuit is resumed correctly, multiplexers are employed to separate checkpointing from normal operation. However, instead of writing the context read from a RAM to the next CPR level for capture as in CPRtree [28], in our architecture, as shown in Fig. 5, the context is written directly to the Capture FIFO. Conversely, for restoration, the context is read from the Restore FIFO and then written to the on-chip RAM. This implementation is possible due to flattening the HDL modules for RAM checkpointing.

In Fig. 5, the 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 normal operation and capture, the other signals (inputs) require multiplexers to be shared between the normal operation and the restoration process. Therefore, RAM checkpointing adds three registers: we_0, addr_0, and wdata_0. As can be seen in Fig. 5, the capture/restore circuit requires a 1-bit register we_0, an A-bit register addr_0, and a DW-bit register wdata_0. Let A be the number of address bits of the RAM, and DW be the data width of the RAM. In total, the circuit requires $DW + A + 1$ register bits. Since all the multiplexers are 2-input, $(DW + A + 1)$ 2-input 1-bit multiplexers are required. Note that all these 2-input multiplexers use the same select bit - the throttling signal. Furthermore, an LUT6 can also be configured as two 5-input LUTs (32-bit ROMs) with separate outputs but common logic inputs. Therefore, $(DW + A + 1)$ 2-input 1-bit multiplexers can be mapped onto $(DW + A + 1)/2$ LUT6s. So, if there are k RAMs in the hardware structure with data widths $DW_0, DW_1, \ldots, DW_{k-1}$, and numbers of address bits $A_0, A_1, \ldots, A_{k-1}$, respectively, then the estimated number of utilized slice registers and the estimated number of utilized slice LUTs caused by the RAM capture/restore circuits can be calculated as follows:

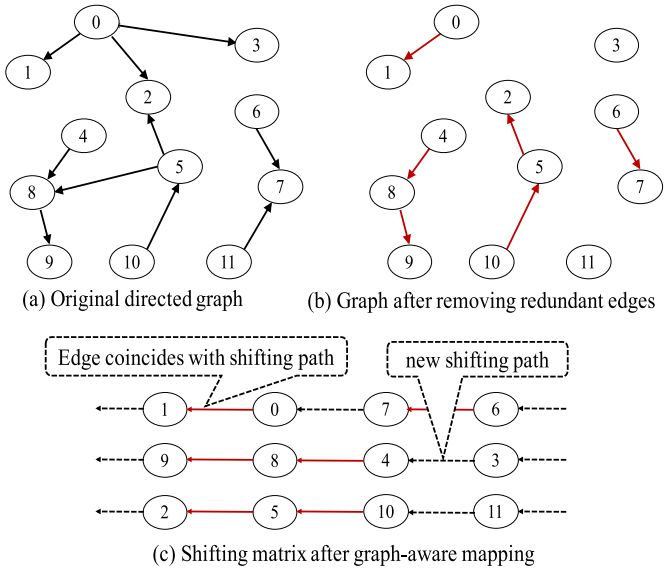

Fig. 6. Static analysis of original HDL source code.

$$F_{\mathrm{RAM}} = \sum_{j=0}^{k-1} (DW_j + A_j + 1) \tag{6}$$

$$L_{\mathrm{RAM}} = \sum_{j=0}^{k-1} (DW_j + A_j + 1)/2 \tag{7}$$

4.4. CPR finite state machines

Checkpoint/restart (CPR) procedures require two finite state machines, one for capture and the other for restoration. The two CPR FSMs are controlled by signals from the CPR manager. In both CPR FSMs, the first state is used to capture/restore registers. The next states are used to capture/restore on-chip RAMs. Therefore, if there are k RAMs in the user design, the number of states in the two CPR FSMs is $k + 1$. In each state, additional registers are used as counters to manage register or on-chip RAM checkpointing processes.

5. Static analysis of original HDL source code

In this section, we describe the basic approach to static analysis of HDL designs and the two algorithms used in the analysis.

5.1. Fundamentals of static analysis

In the shifting ring of register bits shown in Fig. 4, the added shifting paths may bring an additional input to the shifting multiplexer in front of each register bit. This also causes complexity for the combinatorial circuit that generates the select bit of the multiplexer. This input may require more LUTs for the multiplexing functionality, whereas the complexity of the combinatorial circuit may also consume more LUTs for the logic functionality. While the complexity cannot be avoided, the additional input would be not required if the shifting path coincides with one of the inputs from the user logic. Such a 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 time t is a function of F1 at the time t-1, written as F0(t) = f(F1(t-1)). If all the register bits in the reduced set of state-holding elements [28] are considered as vertices of a directed graph, then F0 and F1 are two of the vertices of the 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 the 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 needed to design the shifting matrix. The first step is to analyze the original HDL source code to identify the graph of register bits in the HDL design. 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 are called *redundant edges*. The third step is to map all remaining edges onto the shifting paths of the shifting matrix. In this paper, this mapping method is called *graph-aware mapping*, and these three steps are called *static analysis* of original HDL source code.

Fig. 6 shows an example of these three steps. As can be seen in (a), the original graph has 12 vertices and 10 directed edges. Then in (b), after removing redundant edges, there are only 6 remaining edges. Finally, in (c), after mapping vertices and edges onto the shifting matrix, only 3 additional shifting paths are required. So in this example, the graph-aware mapping eliminated 6 shifting paths. We expect that such elimination will reduce the LUT consumption of the shifting ring.

5.2. Algorithms

Algorithm 1 outlines the Pseudo code to remove redundant edges, and Algorithm 2 has the Pseudo code for how to map register bits onto a shifting matrix. The variables and parameters used in these algorithms are:

- *bitSet* is the set of register bits in the reduced set of state-holding elements.
- *unvisitedBitSet* is the set of register bits that have not been visited.
- *rightBitSet* of register bit B is the set of register bits used to determine B in the next clock cycle. For example, in Fig. 6a, *rightBitSet* of register bit 8 has two elements, bit 4 and bit 5.
- *leftBitSet* of register bit B is the set of register bits that are determined by B in the next clock cycle. For example, in Fig. 6a, *leftBitSet* of register bit 0 has three elements, bit 1, bit 2 and bit 3.
- *preceding* of register bit B is the preceding register bit of B in the shifting path. *preceding* will be None if the shifting path does not coincide with an edge.
- *following* of register bit B is the following register bit of B in the shifting path. *following* will be None if the shifting path does not coincide with an edge. In Fig. 6c, bit 7 is the *following* of bit 6, and bit 6 is the *preceding* of bit 7.
- *noFollowBitSet* is the set of register bits that have no *following* but have *preceding* in the shifting path.
- *noFollowNoPrecBitSet* is the set of register bits that have no *following* and no *preceding* in the shifting path.
- *matrixBitList* is 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.
- *unmappedIndexList* is the list of indexes of bits in *matrixBitList* that has not been mapped onto.
- *Nb* is the number of register bits in the reduced set of state-holding elements.

In Algorithm 1, register bits are consecutively visited to remove redundant edges (line 4). Note that after removing redundant edges, all edges starting from a register bit can be removed, although one of them is expected to be mapped onto a shifting path. To avoid the case that all the edges are removed, register bits having a *rightBitSet* with fewer elements should be visited first as shown in line 3 and line 5. In line 3, *length(B.rightBitSet)* is the number of elements in the set. Then, 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 the *rightBitSets* (line 10, 11).

Algorithm 2 describes the next step, mapping register bits and remaining edges onto the shifting matrix. Since the register bits with edges must be mapped onto the most left-side column first, the *noFollowBitSet* must be visited first as shown 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*). This finally leads to a visit to the bits that have *following* but no *preceding* (line 7, *when B0.preceding* is *None*). After that, the algorithm continues to visit the *noFollowNoPrecBitSet* to cover all bits in the *bitSet* (line 18, 19 and 20).

Algorithm 1
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}
```

Algorithm 2
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:      ifB0.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
```

6. Generating the checkpointing infrastructure in HDL

In this section, we describe the automated tool used to insert checkpointing into HDL designs, and the algorithm that controls the insertion.

6.1. Automated checkpointing insertion

Our proposed design flow adds an insertion tool to the typical hardware design flow before the synthesis as shown in Fig. 7. Our tool, the Checkpointing Inserter, uses Pyverilog [31], a Python-based analysis and synthesis tool for Verilog HDL source code, to generate an abstract

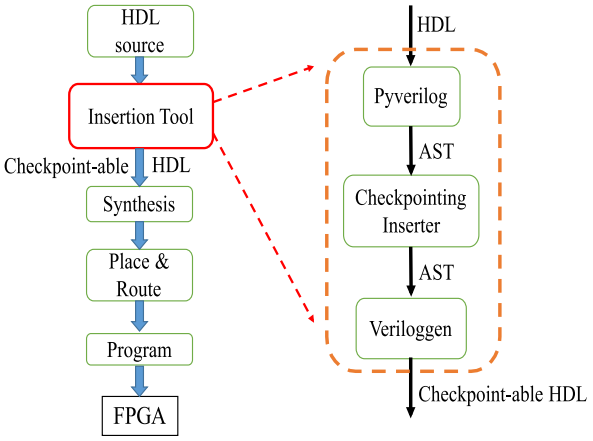

Fig. 7. Proposed design flow.

syntax tree (AST) for the Verilog HDL source code. This tree is in the form of nested class objects in Python. The AST consists of AST nodes containing full information about the Verilog HDL source code, such as module definitions, parameters, ports, instances, and always blocks. Our Checkpointing Inserter then modifies the AST to provide checkpointing functionality without changing the functions of the user circuits. After this modification, the AST is combined with Veriloggen [32], a Python-based hardware description and hardware customization library, to generate the output Verilog HDL source code. Note that the insertion tool has both input and output in HDL and is located before the synthesis process in the proposed design flow. Therefore, our checkpointing methodology is transparent to applications and not dependent on specific tools and technologies. Our previous work [28] introduced a tool for checkpointing insertion according to the tree-based checkpointing architecture. Compared to the previous work, the tool in this work uses the algorithm of checkpointing insertion according to the ring-based flattened checkpointing architecture instead of the tree-based checkpointing architecture. The following subsection describes the algorithm in details.

6.2. Algorithm of checkpointing insertion

Based on CPRflatten, we use Algorithm 3 to insert checkpointing functionality into an HDL design. In this algorithm, if the module is the top module, then the static CPR part will be inserted (line 6). Otherwise, control ports from the CPR manager 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 not matched, the check_instance() returns *normal_inst* and the function insert_checkpointing() is called as a recursive algorithm (line 12). If it is matched, the function returns *RAM_inst* and the function insert_RAM_checkpointing() is called (line 14). Then, the visit_always_block() function (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 the always blocks and registers in the design will be visited. Finally, in the HDL top module, the modify_always_blocks() function is called (line 17) to modify always blocks based on the above static analysis, and a capture FSM and a restoration FSM are also inserted (line 18, 19).

Algorithm 3
Inserting checkpointing functionality.

```
1: m = top module
2: call insert_clieckpointing(m)
3://————————————————————————————————————————————————————————————
4: function insert_clieckpointing(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 notmal_inst then
12:      call insert clieckpointing(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_captoring_FSM(m)
19:    call insert_restoring_FSM(m)
```

7. Evaluation

Table 1 shows the experimental setup we used to evaluate the proposed hardware task migration systems. In this section, we describe the hardware resource utilization, maximum clock frequency degradation, register bits and memory footprint, along with the migration duration and downtime.

7.1. Hardware resource utilization

In this work, we assume that FPGAs are employed in future data centers for computational tasks. The tasks usually use data from main memory as their inputs to compute. The computation results are written back to the main memory. These tasks use AXI4 interfaces to communicate with the main memory and use AXI4-Lite interfaces to communicate with the host CPU. For high performance, the tasks usually use many on-chip RAMs for data buffers or communications. Based on this assumption, we select four benchmarks: pipelined SIMD matrix multiplication (Mat-Mul), Dijkstra graph processing (Dijkstra), 9-point Stencil Computation (Stencil), and String Search (S-Search) to evaluate our proposals. All four benchmarks use AXI4 interfaces and AXI4-Lite interfaces to communicates with the main memory and the host CPU. They all employ many on-chip RAMs in their hardware circuits. Since the static CPR hardware part is fixed and transparent to applications, to evaluate the efficiency of our checkpointing architecture, only the hardware overhead of the user-logic-based part is considered.

In our architecture, the LUT consumption for checkpointing mainly comes from register checkpointing circuits and RAM checkpointing circuits. As shown in Table 2, the register checkpointing circuits of the four benchmarks are shifting rings considered as graphs of register bits. More register bits may seem to require more additional LUTs for register checkpointing. However, as described above, after removing redundant edges, the remaining edges are mapped onto shifting paths to eliminate the need for employing more LUTs. Because of this, the more remaining edges there are, the more LUT overhead can be expected to be reduced. For example, the Stencil benchmark with 5172 remaining edges should show the most significant decrease in LUT consumption. In contrast, the Dijkstra benchmark with only 193 remaining edges should show the smallest decrease in LUT consumption.

On the other hand, the LUT consumption for RAM checkpointing depends on the total data width, the total number of address bits, and the total number of instances of RAMs used in the user circuit according to equation (12). Table 3 provides such numbers for the four benchmarks. As can be seen, Mat-Mul and Stencil have greater total data widths and greater numbers of address bits of RAM. Also, they employ larger numbers of register bits. Therefore, they require more LUTs for checkpointing.

Table 4 shows the results of our synthesis in terms of LUT utilization on the Zedboard evaluation board in our evaluation setup. Note that in this table and also Table 5, Original refers to the hardware design before applying either CPRtree or CPRflatten. In Table 4, the results match the above discussions. In particular, with both CPRtree [28] and CPRflatten, Mat-Mul and Stencil have greater numbers of additional LUTs than Dijkstra and S-Search do. They also have more significant decreases between CPRtree and CPRflatten (2746 and 3512) in the number of additional LUTs needed. However, with the S-Search application, CPRflatten reduces the number of additional LUTs the most (a decline of 67.91%), going from an increase of 92.97% with CPRtree to 29.83% with CPRflatten. The average decline of 49.25% in the number of additional LUTs highlights the efficiency of CPRflatten over CPRtree. The register utilization is provided in Table 5. As can be seen in the table, the register overhead is very high, up to 174.5% in CPRtree for Stencil benchmark. The overhead is reduced significantly in CPRflatten for the

Table 1
Experimental setup.

Evaluation Board	Zedboard
EDA Tool	Vivado 2014.4 and ISE 14.7
FPGA	Xilinx Zynq-7000 XC7z020clg484-1
Clock frequency	100 MHz
Host CPU	ARM Cortex-A9
Operating system	Debian 8.0
MPI version	OpenMPI 3.0.0
Evaluation Board	Zynq Ultrascale+
EDA Tool	Vivado 2017.2
FPGA	XCZU9eg-ffvb1156-2L-e-es1
Clock frequency	100 MHz
Host CPU	ARM Cortex-A53
Operating system	Petalinux 2017.2
MPI version	OpenMPI 3.0.0

Table 2
Graph of register bits.

Apps	Register bits	Total edges	Redundant edges	Remaining edges
Mat-Mul	5723	2430	164	2266
Dijkstra	1637	565	372	193
Stencil	10300	6026	854	5172
S-Search	3590	2177	268	1909

Table 3
Total data width and address bits of RAMs.

Apps	Number of RAM instances	Total data width	Total address bits
Mat-Mul	13	885	61
Dijkstra	8	448	32
Stencil	14	750	77
S-Search	10	430	46

same benchmark (98%).

For hardware overhead, the readback method is the best since it does not require any additional hardware resource. That means the hardware overhead of the readback method is zero. The average LUT overhead in Refs. [23,25,26] are 33% and 23%, respectively, which are lower than our average LUT overhead (44.5%). However, we evaluated on a different set of benchmarks, and some benchmarks used in the previous works have the LUT overhead higher than our average LUT overhead, such as FIR16 (66%) in Ref. [23] and MPEG2 (85%) in Ref. [25,26]. Due to drawbacks presented in Section 2, we believe that the checkpointing solutions in Refs. [23,25,26] are incomplete. Once taking those drawbacks into account, the checkpointing solutions in Refs. [23,25,26] need to add more hardware to the application circuits, thus increasing the hardware overhead. It should be noted that the readback method cannot be used to migrate a task from an FPGA to another kind of FPGA. Furthermore, the readback method and the methods in Ref. [23,25,26] did not consider the consistency of snapshots.

7.2. Maximum clock frequency degradation

Table 6 shows the maximum clock frequencies achieved by the synthesis on the Zedboard evaluation setup. Both CPRtree and CPRflatten have a negligible impact on the maximum clock frequency. The average degradation is only 2.31% and 1.66%, respectively. The most significant degradation is 9.73% (Mat-Mul). The maximum clock frequency of Stencil and Dijkstra increases slightly after inserting checkpointing functionality. This small increase may come from the optimization of the synthesis tool. In particular, no logic element is inserted into the critical paths, while the physical distance of logic elements may be reduced, thus reducing the critical path. These average values are very small compared with the degradation of around 20% reported for the scan-chain netlist-based method in Ref. [24]. The average clock frequency degradation of the scan-chain method in Ref. [23] and in Ref. [25,26] are 3.33% and 2.6%, respectively, which are slightly more than the average degradation in our method.

7.3. Memory footprint

Our checkpointing mechanism uses a reduced set of state-holding elements [28], thus minimizing the context size. As can be seen in Table 7, the context size varies from 1.07 kB (Dijkstra) to 14.32 kB (Stencil). These sizes are much smaller than the context size of the readback-of-bitstream method, which is around 4 MB for the device XC7z020clg484-1 on the Xilinx Zedboard. The context size is also the memory footprint on the source node. However, on the target node, the memory footprints used for task migration include both an FPGA snapshot and a bitstream buffer, as was shown in Fig. 1. The bitstream buffer must be allocated enough memory to buffer the bitstreams of all possible tasks. In our experiments, there were four potential tasks migrated to the Zedboard. Therefore, the memory footprint on the Zedboard was C_{FPGA} + 16 MB.

7.4. Migration duration, downtime and their breakdown

Our experiments migrated tasks from a Zynq Ultrascale + board to a Zedboard in a heterogeneous FPGA cluster connected via an Ethernet network. On these boards, the host CPUs were responsible for creating

Table 4
LUT utilization.

Apps	LUTs (Original)	Additional LUTs (CPRtree)((CPRtree/Original) * 100%)	Additional LUTs (CPRflatten)((CPRflatten/Original) * 100%)	Decline in additional LUTs (CPRtree - CPRflatten) (((CPRtree - CPRflatten)/CPRtree) * 100%)
Mat-Mul	3323	5339 (160.67%)	2593 (78.03%)	2746 (51.43%)
Dijkstra	8126	1461 (17.98%)	1020 (12.55%)	441 (30.18%)
Stencil	6748	7395 (109.59%)	3883 (57.54%)	3512 (47.49%)
S-Search	4056	3771 (92.97%)	1210 (29.83%)	2561 (67.91%)
Mean			(44.5%)	(49.25%)

Table 5
Register utilization.

Apps	Original	CPRtree (Overhead)	CPRflatten (Overhead)
Mat-Mul	3449	8940 (159.2%)	6732 (95.2%)
Dijkstra	8023	10527 (31.2%)	10077 (25.6%)
Stencil	5256	14428 (174.5%)	10408 (98%)
S-Search	3241	6452 (99.1%)	4869 (50.2%)

the connection and for sending/receiving task bitstreams. As shown in Table 8, the four benchmarks have different sizes of non-dirty and dirty data in memory. As can be seen in Table 9, first, the capturing latency (T_{cap}), restoring latency (T_{res}), and the FPGA context transfer latency (T_{tranf_FPGA}) depend linearly on the FPGA context (C_{FPGA}). The pre-copy duration ($T_{pre\text{-}copy}$) and dirty data transfer latency (T_{transf_dirty}) also depend linearly on non-dirty context ($C_{non\text{-}dirty}$) and dirty context (C_{dirty}), respectively. Third, the migration downtime (T_{down}) and the migration duration (T_{mig}) are matched with equations (2) and (4). Therefore, the experimental results confirm the correctness of equations (2) and (4).

However, all four benchmarks show that ($T_{pre\text{-}copy} + T_{down}$) is greater than ($T_{config} + T_{res}$). As a result, in equation (4), Max{$T_{config} + T_{res}$, $T_{pre\text{-}copy} + T_{down}$} is equal to ($T_{pre\text{-}copy} + T_{down}$). Therefore, in this case, we did not observe how T_{mig} depends on T_{config}.

It is noted that the migration downtime (Tdown) and the migration duration (Tmig) do not only depend on FPGA context (CFPGA) but also depend on the dirty context (Cdirty) and the non-dirty context (Cnon-dirty). In our applications, the dirty context and non-dirty context are much bigger than the FPGA context. As a result, the migration downtime (Tdown) and the migration duration (Tmig) depend mostly on the dirty context and the non-dirty context. Therefore, we choose to show how the migration downtime (Tdown) and the migration duration (Tmig) depend on the dirty context and the total context as in Figs. 8 and 9. The benchmark Dijkstra was used again while scaling the number of vertices and the number of edges in the graph of the benchmark to evaluate the migration downtime and migration duration. Table 10 shows how the non-dirty ($C_{non\text{-}dirty}$), dirty (C_{dirty}), and total context (C_{data}) in memory are also scaled. In this scale, T_{transf_dirty} was always greater than ($T_{cap} + T_{res}$), thus ($T_{cap} + T_{res}$) was hidden by T_{transf_dirty} in equation (2). As a result, equation (2) can be reduced to the following equations:

$$T_{down} = T_{prep} + T_{transf\_FPGA} + T_{transf\_dirty} \tag{8}$$

$$T_{down} = T_{prep} + \frac{C_{FPGA}}{bw} + \frac{C_{dirty}}{bw} \tag{9}$$

Therefore, the downtime should increase linearly with the amount of dirty context (C_{dirty}), but independently of the non-dirty context ($C_{non\text{-}dirty}$). Fig. 8 shows the experimental results that confirm this linear increase. In contrast, as shown in equation (5), the migration duration (T_{mig}) depends on both the non-dirty context ($C_{non\text{-}dirty}$) and dirty context (C_{dirty}). Using equations (4), and (9), the migration duration can be reduced to the following equations:

$$T_{mig} = Max\left\{T_{config} + \frac{C_{FPGA}}{mrbw_1}, \frac{C_{non-dirty}}{bw} + T_{prep} + \frac{C_{FPGA}}{bw} + \frac{C_{dirty}}{bw}\right\} \tag{10}$$

$$T_{mig} = Max\left\{T_{config} + \frac{C_{FPGA}}{mrbw_1}, T_{prep} + \frac{C_{FPGA}}{bw} + \frac{C_{data}}{bw}\right\} \tag{11}$$

As can be seen in equation (11), when the number of vertices is small enough (C_{data} is small), ($T_{pre\text{-}copy} + T_{down}$) can be less than ($T_{config} + T_{res}$). As a result, T_{mig} is kept constant at ($T_{config} + T_{res}$) until the total context (C_{data}) increases enough. After that, the migration duration increases linearly with the total context (C_{data}). Fig. 9 shows the experimental results of migration duration while scaling the number of vertices. The horizontal axis is scaled with the logarithm of total data context (C_{data} in kB) to base 10. The migration duration remains constant at around 32,000 us when the total context is small, before increasing. Therefore, the figure confirms equation (11).

The experimental results also confirmed the efficiency of our migration scheme. First, the HDL-based checkpointing employed in our scheme has much smaller capturing latency and restoring latency than in the previous scheme [13]. Specifically, capturing latency with our scheme was 0.018 ms on the average, compared to 953 ms for readback capturing with the previous scheme. Similarly, restoring latency was 0.019 ms on average with our scheme, compared to 52 ms for restoring (configuration) latency with the previous scheme. The previous study did not consider the transfer of FPGA context, non-dirty, and dirty context. Therefore, we cannot compare the migration downtime and migration duration. Second, as shown in equation (2), the capturing (T_{cap}) and restoring (T_{res}) latency can be hidden by the transfer of dirty context. As a result, they have no impact on the migration downtime. Third, the configuration latency (T_{config}) can be hidden in the migration duration by the pre-copy and the transfer of FPGA context and dirty context. This latency also has no impact on the migration downtime. Fourth, equations (3) and (5) shows that the migration downtime and

Table 7
Context size.

Apps	Register Context (kB)	RAM Context (kB)	Total context (kB)
Mat-Mul	0.70	7.31	8.01
Dijkstra	0.20	0.87	1.07
Stencil	1.26	13.06	14.32
S-Search	0.44	2.38	2.82

Table 8
Data context size in memory.

Apps	$C_{non\text{-}dirty}$(kB)	C_{dirty} (kB)	Total data (C_{data}) (kB)
Mat-Mul	8192	4096	12,288
Dijkstra	549.68	12.50	562.18
Stencil	0	2048	2048
S-Search	32,768	8	32,776

Table 6
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%)

Table 9
Migration downtime, migration duration and their breakdown (us).

Apps	T_{config}	$T_{pre\text{-}copy}$	$T_{prepare}$	T_{cap}	T_{transf_FPGA}	T_{res}	T_{dirty}	T_{down}	T_{mig}
Mat-Mul	32,093	142,598	2	21	809	23	63,940	64,807	207,455
Dijkstra	32,402	44,436	1	4	542	5	2252	2607	47,047
Stencil	31,900	0	14	38	1174	39	51,351	52,575	52,646
S-Search	32,033	524,373	7	8	555	10	637	1251	525,627
Mean	32,107	177,852	6	18	770	19	29,545	30,310	208,194

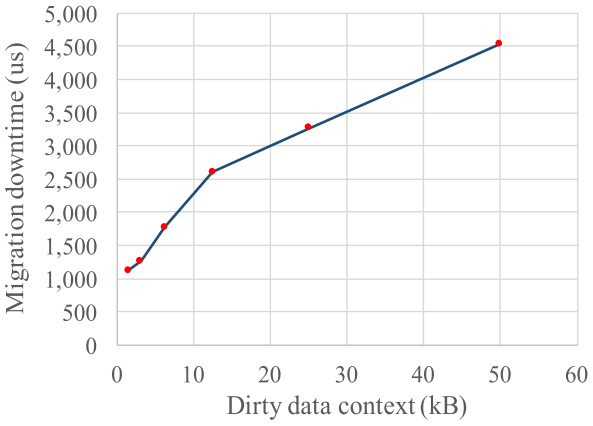

Fig. 8. Migration downtime while scaling the graph size.

duration can be much reduced compared with the readback scheme since the amount of FPGA context (less than 15 kB) is much smaller than the bitstream size (4 MB). Note that the downtime in Ref. [8] must be more than the sum of the readback latency and configuration latency (1005 ms) although the context transfer was not considered. However, in our scheme, the downtime is much smaller. As shown in Table 9, the downtime is only 1.251 ms for the S-Search benchmark and 30.31 ms on the average (less than 3.02% of the downtime in Ref. [13]) for the four benchmarks. Furthermore, the smaller amounts of FPGA context also help to reduce the total network traffic in the cluster/cloud.

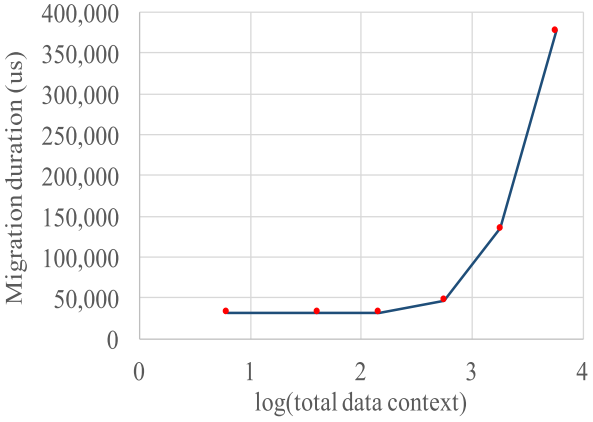

Fig. 9. Migration duration while scaling the graph size.

Table 10
Memory context size while scaling the graph size in Dijkstra benchmark.

Vertices	Edges	$C_{non\text{-}dirty}$ (kB)	C_{dirty} (kB)	C_{data} (kB)
200	1193	4.66	1.56	6.22
400	9545	37.29	3.12	40.41
800	35,283	137.82	6.25	144.07
1600	140,719	549.68	12.50	562.18
3200	461,723	1803.61	25.00	1828.61
6400	1,433,782	5600.71	50.00	5650.71

8. Conclusion

In this paper, we studied the migration of hardware tasks between different kinds of FPGAs in a heterogeneous FPGA system employing an HDL-based checkpointing method. First, we presented a hardware task migration scheme where the checkpointing procedures overlap with the data transfer. The scheme also allows pre-copy and an overlap between the capture on the source node and the FPGA configuration on the target node. Since our scheme uses HDL-based checkpointing with a reduced set of state-holding elements, both the capturing latency (0.018 ms on the average) and the restoring latency (0.019 ms on the average) are much smaller than those of the readback-of-bitstream method. The context size (less than 15 kB) is also much smaller than the bitstream size (4 MB). Therefore, the migration scheme has shorter downtime, less than 3.02% of the downtime with the previous scheme. Other improvements of our scheme over the previous schemes include: 1) it can maintain the consistency of snapshots taken for capture and restoration, and 2) by the use of a reduced set of state-holding elements at the HDL level, our checkpointing mechanism allows hardware tasks, even those designed with dedicated blocks, to be migrated between different kinds of FPGAs in a heterogeneous FPGA system.

Second, we proposed CPRflatten, a ring-based flattened checkpointing architecture for FPGAs that is transparent to applications and portable across hardware platforms. In this architecture, the user hardware resources can also be employed for register checkpointing. Third, we proposed a static analysis of the original HDL source code to realize the user hardware structure and to map registers onto the checkpointing circuit. As a result, the LUT utilization for checkpointing functionality in CPRflatten was reduced nearly 50% on the average compared with that of CPRtree. The synthesis results also showed that CPRflatten had a negligible impact on the maximum clock frequency. The average degradation was only 1.66%. Fourth, we also introduced a Python-based tool to insert checkpointing infrastructures into the original design, thus eliminating an additional task for the designers.

With our approach, the five drawbacks of the previous scheme indicated in Section 1 have been overcome. This enables hardware tasks to be migrated between different kinds of FPGAs in data centers with minimal service downtime and migration duration, while the hardware cost is reduced significantly. As a result, our proposals allow flexible hardware task management in data centers with minimal degradation in performance to save energy, balance the load, and prepare production servers for maintenance.

Besides the above contributions, this work has two limitations. First, our checkpointing technique was not implemented for multiple clock domain applications. Second, our hardware task migration scheme and checkpointing method were not tested against different vendor FPGAs.

As future work, we want to study the migration of multiple hardware tasks between different kinds of FPGAs in a heterogeneous FPGA system. The multiple tasks on a single FPGA should be implemented using partial run-time reconfiguration. In this case, a migration scheduler might be integrated from mathematical models to algorithms and implementation so that checkpointing procedures can best be overlapped with data transfer.

Author statement

Hoang-Gia Vu: Methodology, Algorithm design, Software programming, Hardware design, Data collection, Manuscript writing. **Takashi Nakada:** Supervision. **Yasuhiko Nakashima:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] A. Putnam, et al., A reconfigurable fabric for accelerating large-scale datacenter services, Commun. ACM 59 (11) (Nov. 2016) 114–122.
[2] A. Putnam, FPGAs in the datacenter - combining the worlds of hardware and software development, in: Proceedings of Great Lakes Symposium on VLSI, May 2017, p. 5.
[3] B. Falsafi, et al., FPGAs versus GPUs in Data centers, IEEE Micro 37 (1) (2017) 60–72.
[4] N. Tarafdar, et al., Enabling flexible network FPGA clusters in a heterogeneous cloud data center, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA2017), Feb 2017, pp. 237–246.
[5] A.M. Caulfield, et al., Configurable clouds, IEEE Micro 37 (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 Comput. Pract. Ex. 28 (8) (2016) 2374–2393.
[10] V. Kherbache, E. Madelaine, F. Hermenier, Scheduling live migration of virtual machines, IEEE Trans. Cloud Comput. (99) (2017) (early access).
[11] W. Cerroni, F. Esposito, Optimizing live migration of multiple virtual machines, IEEE Trans. Cloud Comput. (99) (2017) (early access).
[12] M.R. Hines, U. Deshpande, K. Gopalan, Post-copy live migration of virtual machines, ACM SIGOPS - Oper. Syst. Rev. 43 (3) (July 2009) 14–26.
[13] O. Knodel, P.R. Genssler, R.G. Spallek, Migration of long-running tasks between reconfigurable resources using virtualization, Comput. Architect. News 44 (4) (Sep. 2016).
[14] H. Kalte, M. Porrmann, Context saving and restoring for multitasking in reconfigurable systems, in: International Conference on Field Programmable Logic and Applications, 2005, pp. 223–228.
[15] I.H. Simmler, L. Levinson, R. Manner, Multitasking on FPGA coprocessors, in: The 10th International Conference on Field Programmable Logic and Application (FPL'00), 2000, pp. 121–130.
[16] A. Morales-Villanueva, A. Gordon-Ross, On-chip context save and restore of hardware tasks on partially reconfigurable FPGAs, in: FCCM 2013, 2013, pp. 61–64.
[17] W.J. Landaker, M.J. Wirthlin, B.L. Hutchings, Multitasking hardware on the SLAAC1-V reconfigurable computing system, in: 12th International Conference on Field-Programmable Logic and Applications (FPL2002), 2002, pp. 806–815.
[18] L. Levinson, R. Manner, M. Sessler, H. Simmler, Preemptive multitasking on FPGAs, in: FCCM 2000, 2000, pp. 301–302.
[19] M. Happe, A. Traber, A. Keller, Preemptive hardware multitasking in ReconOS, in: ARC 2015, 2015, pp. 79–90.
[20] H.G. Vu, S. Kajkamhaeng, S. Takamaeda-Yamazaki, Y. Nakashima, CPRtree: a tree-based checkpointing architecture for heterogeneous FPGA computing, in: 4th International Symposium on Computing and Networking (CANDAR 2016), Nov 2016.
[21] S.M. Trimberger, J.J. Moore, FPGA security: motivations, features, and applications, Proc. IEEE 102 (8) (Aug 2014) 1248–1265.
[22] B. Badrignans, D. Champagne, R. Elbaz, C. Gebotys, L. Torres, SARFUM: security architecture for remote FPGA update and monitoring, ACM Trans. Reconfigurable Technol. Syst. (TRETS) 3 (2) (May 2010).
[23] D. Koch, C. Haubelt, J. Teich, Efficient hardware checkpointing - Concepts, overhead analysis, and implementation, in: FPGA'07, February 2007, pp. 188–196.
[24] Iakovos Mavroidis, Ioannis Mavroidis, I. Papaefstathiou, Accelerating emulation and providing full chip observability and controllability, IEEE Design & Test of Computers 26 (6) (Dec. 2009) 84–94.
[25] A. Bourge, O. Muller, F. Rousseau, Automatic high-level hardware checkpoint selection for reconfigurable systems, in: FCCM 2015, 2015, pp. 155–158.
[26] A. Bourge, O. Muller, F. Rousseau, Generating efficient context-switch capable circuits through autonomous design flow, ACM Trans. Reconfigurable Technol. Syst. (TRETS) 10 (1) (Dec 2016).
[27] A.G. Schmidt, B. Huang, R. Sass, M. French, Checkpoint/restart and beyond: resilient high performance computing with FPGAs, in: FCCM 2011, 2011, pp. 162–169.
[28] H.G. Vu, S. Takamaeda-Yamazaki, T. Nakada, Y. Nakashima, A tree-based checkpointing architecture for the dependability of FPGA computing, IEICE Trans. Info Syst. E101-D (2) (Feb 2018) 288–302.
[29] H.G. Vu, T. Nakada, Y. Nakashima, Efficient multitasking on FPGA using HDL-based checkpointing, in: ARC 2018, 2018, pp. 590–602.
[30] https://www.open-mpi.org.
[31] S. Takamaeda-Yamazaki, Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL, in: 11th International Symposium on Applied Reconfigurable Computing (ARC 2015), Lecture Notes in Computer Science, 9040/2015, April 2015, pp. 451–460.
[32] M. Watanabe, K. Sano, S. Takamaeda, T. Miyoshi, H. Nakajo, Japanese high-level synthesis tools for FPGA hardware acceleration, IEICE Trans. Commun. J100-B (1) (2016) 1–10 (in Japanese).