Extended overlay architectures for heterogeneous FPGA cluster management

Mohamad Najem, Théotime Bollengier, Jean-Christophe Le Lann, Loïc Lagadec

Extended overlay architectures for heterogeneous FPGA cluster management

Mohamad Najem [a,*], Théotime Bollengier [a,b], Jean-Christophe Le Lann [a], Loïc Lagadec [a]

[a] *Lab-STICC UMR 6532, ENSTA Bretagne, Brest, France*
[b] *B<>com, Research Institute of Technology, Brest, France*

1. Introduction

Nowadays, hardware architectures, especially the ones dedicated for signal and image processing, must offer high performance computation, design flexibility, and upgrade capabilities. Reconfigurable chips, such as Field Programmable Gate Arrays (FPGAs), have been addressed as a reasonable solution in this area, combining flexibility, re-programmability, power efficiency, and low development cost [1,2].

Moreover, a trend which has recently emerged is the remote-use of commodity off-the-shelf (COTS) FPGAs as hardware acceleration units in a heterogeneous computing cluster [3,4]. Over the lifetime of the infrastructure, components of such a cluster are gradually updated and replaced to follow the technology evolution over time, and FPGAs sales and trends. This results in FPGAs with different characteristics and from different vendors being used at the same time. However, a bitstream generated for a given FPGA cannot be loaded into an FPGA of a different model. It is therefore not possible to blindly dispatch hardware applications in such a FPGA cluster. For this purpose, this paper proposes an FPGA resources virtualization approach based on overlay architectures. The idea is to deploy an intermediate layer of reconfigurable resources, the overlay-based virtual FPGA (vFPGA), to hide the realm of the infrastructure and offer a unified view of resources. An application design targeting the vFPGA is no longer tied to a limited set of FPGAs from the cluster, and can run on any node implementing the vFPGA.

Moreover, the sharing of reconfigurable resources among different applications is a key requirement that gives rise to a higher hardware utilization. For this purpose, applications have to be efficiently instrumented, hence offer a simple management of such a cluster. In this work, we propose to extend the classical overlay architectures by adding new features to freely snapshot the state of the running application, and enable restoring the circuit back to any previously saved state. Hardware task preemptive scheduling on a node, application live migration between nodes can then be deployed; our complete system can perform load balancing or consolidation, manages application priorities and provides fault tolerance. Moreover, to support early performances estimation, we propose some accurate cost models for scheduling and live migration time.

The remainder of this paper is organized as follows. Section 2 presents the limitation of existing works from the literature, while in Section 3, the proposed FPGA virtualization

* Corresponding author.
E-mail addresses: mohamad.najem@ensta-bretagne.fr (M. Najem), theotime.bollengier@ensta-bretagne.fr (T. Bollengier), jean-christophe.Le_Lann@ensta-bretagne.fr (J.-C. Le Lann), loic.lagadec@ensta-bretagne.fr (L. Lagadec).

approach is discussed and the classical overlay architectures are presented. Section 4 discusses Zeff, the deployment architecture, while Section 5 presents our novel overlay features. Section 6 describes the complete system supporting live migration and offering scheduling of hardware tasks on overlays. Section 7 draws conclusions and discusses future lines of research.

2. Related works

The main goal of this paper is to propose an FPGA virtualization solution for an efficient remote-use of FPGAs for general purpose computing, which has recently become a new area of interest for many researchers. A useful analysis of the difficulty to bringing FPGAs as shareable resources is presented in [5], where four requirements are identified, including the binary compatibility among FPGAs and resource sharing capabilities. The authors propose to expose FPGAs to the cloud stack as a resources pool, which can be dynamically managed by a global manager. They clearly highlight the need of a virtualization support for an efficient sharing of FPGA resources.

The virtualization of FPGA resources provides more flexibility and portability for user applications. Several frameworks are proposed in the literature, which abstract the physical resources by a system layer (or services) providing a virtualized environment to access them. In [6], a paravirtualized Xen Virtual Machine (VM) environment provides multiple-user services to access FPGA accelerators. Moreover, BORPH [7] is also a well known project working on UNIX interfaces for creating hardware-based drivers and providing FPGA hardware abstractions and management, while the VirtualRC platform [8] proposes a uniform hardware/software interface to access FPGAs. Moreover in [9], authors have proposed the Object-Oriented Communication Engine (OOCE), a system-level middleware for FPGA-SoC heterogeneous architecture. They abstract FPGA synthesis flow and acceleration by a set of OS calls. These frameworks move a step forward in the virtualization of FPGAs. However, such services virtualize (or abstract) the access to the FPGA not the resource itself. In other words, FPGAs are not completely virtualized in a generic way, and remains an open question how to provide an efficient sharing algorithm between heterogeneous FPGAs.

Most of hardware virtualization schemes are based on the dynamic reconfiguration of the FPGA [4,9–11]. The key idea is to decompose the physical FPGA into several reconfigurable regions using the Dynamic Partial Reconfiguration (DPR). Each region is considered as a single virtualized FPGA resource (vFPGA) making the FPGA a multi-tenant device. Additionally, a static region is used to connect all vFPGAs in each physical FPGA to the system manager. Authors in [10], have proposed an architecture of four vFPGAs with NetFPGA-10G infrastructure in the static region, while in [11] the control unit in the static region communicates with the external host via a PCI-express interface. Moreover, in [4], authors have proposed a prototype framework (RC3E) for integrating vFPGAs in the cloud using also a DPR technique. The virtualization of FPGA resources based on the DPR technique seems very promising, but remains dependent on the FPGA; it is a service provided by FPGA vendors, and applications must be synthesized and programmed for a specific FPGA and using vendors tools. Also, DPR-based vFPGAs do not meet application management requirements, especially when migration and scheduling techniques are envisioned to increase the usability of FPGAs [5,11].

Furthermore, sharing of FPGA resources is a natural requirement and a promising technique to reuse resources by different circuits (configurations). In [12], authors propose to switch between signal processing DVB-T2 tasks in a time multiplexed fashion using DPR. Another technique is proposed in [13] to share circuit-specific DSP blocks from Xilinx. Instead of reconfiguring a set of DSP blocks to implement all operations, authors use multiple sets of DSP blocks controlled by state machines to ensure that each set achieves a high initiation interval. A hardware context-switch mechanism is introduced in [14], where user tasks, designed by HLS tools, are modified at design-time: several checkpoints are inserted inside the code and a scan-chain structure is included to extract/restore the state of the circuit. A useful approach based on FaRM (Fast Reconfigurable Manner) controller for an advanced scheduling of hardware tasks on a DPR-based resource is introduced in [15]. Authors present an architecture with on-chip controllers and FSMs enabling the pre-loading of the partial bitstream in order to reduce PR configuration time overhead.

From the variety of approaches in the literature, it is clear that most works have focused on the virtualization of FPGAs as a software layer. Few efforts have been reported on the hardware architecture of the vFPGA, which is crucial for effective management of FPGA resources. A spatial-sharing of FPGAs is provided through the DPR technique, while the time-sharing is addressed for some application-specific and circuit-specific contexts. DPR-based solutions can hardly be generalized for whichever task or FPGA. Also, the compatibility requirement among DPRs of different FPGAs cannot be entirely fulfilled. Despite the effort in reducing resource reconfiguration time, DPRs might suffer from a significant reconfiguration latency, as they use relatively slow communication interfaces for the configuration [16]. In this paper, we propose a novel approach for the hardware virtualization of FPGAs. vFPGAs are overlay-based architectures that can be implemented on any FPGA device. Our generic solution homogenizes any cluster of heterogeneous FPGAs. Several overlays architectural features are introduced to extend the flexibility providing a cost-effective configuration, thanks to our bitstream pre-loading feature, and a lightweight management for running applications.

3. FPGA virtualization: overlay approach

Virtualizing FPGAs aims at designing, implementing and running hardware applications independently from the FPGA technology and tools. This section presents our proposed method, highlights the advantages of overlay architectures, and introduces most of existing architectures from the literature.

3.1. Overlay-based approach

We investigate overlay architectures to virtualize heterogeneous FPGAs. Overlays have been reported to offer many advantages against FPGAs:

- Bitstream compatibility: Overlays are portable over any physical FPGA. Hence, circuits targeting the overlay can be implemented on any physical FPGA supporting the virtual layer. All physical FPGAs, which have almost different characteristics and from different vendors, are seen as homogeneous re-configurable overlays for front-end applications.
- Open-source usage: The ability to deploy a vFPGA, with a fully accessible architecture description, allows the developers to use any open-source tool targeting such architectures. This alleviates the need for FPGA vendor tools.
- Flexibility: As a general rule, being independent from the underlay, overlays implementations can integrate advanced architectural features to provide new capabilities, such as: application monitoring, clock management, bitstream pre-loading, *etc*.

Fig. 1 illustrates the proposed approach. It consists on deploying a layer between the application and the physical FPGA, the overlay. Overlays are re-configurable architectures mapped on top of the COTS FPGAs. The proposed overlay-based virtual FPGA is composed of three layers (as shown in the Fig. 1):

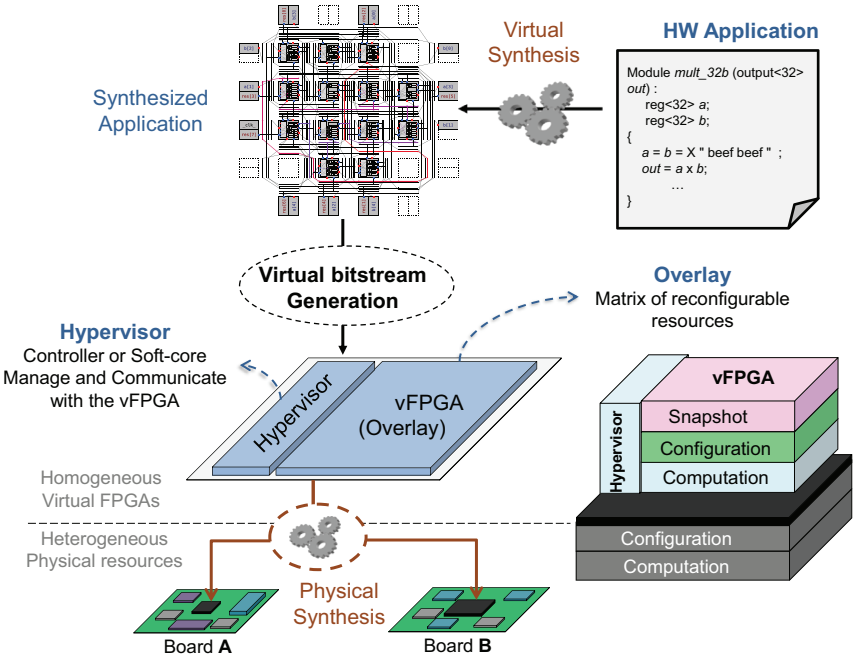

Fig. 1. Overview of the proposed FPGA virtualization approach.

- The *computation layer*, which is the set of reconfigurable elements available to the application.
- The *configuration layer*, which configures the computation layer.
- The *snapshot layer*, which snapshots the state of the computation layer, hence the state of the running application.

From the application interface, the virtual FPGA corresponds to a set of re-configurable elements available to the application which targets the vFPGA (*functional architecture*), while the interface with the physical FPGA focuses on how the functional architecture is implemented and synthesized to the host FPGA. So, the functional architecture of the vFPGA and the implementation are independent, hence the concept of FPGA virtualization.

3.2. Overlay architectures

Overlays consist of a regular arrangement of Reconfigurable Elements (REs), connected by routing channels of an interconnection network. In the literature, overlays have been developed for various applications, and are either *fine-grained* or *coarse-grained* architectures. Fig. 2 shows a conceptual view for the computation layer of island-style overlay architectures.

3.2.1. Fine-grained overlays

Fine-grained overlays are FPGA-like architectures [17,19–21], where REs are composed of fine-grain reconfigurable elements, such as Configurable Logic Blocks (CLBs). Fig. 2a) presents a generic LUT-based architecture compatible with standard architectures used in the academic Versatile Place and Route (VPR) tool [22]. The basic architecture has *Height* × *Width* CLBs, each of which has I bits inputs and N bits outputs. A CLB is composed of N Basic Logic Elements (BLEs). A BLE has one LUT with K inputs and one register that can be bypassed (also called the virtual application register). Inputs of BLEs are derived from a global crossbar that has $I + N$ inputs (the I CLB inputs plus N feedback signals from the BLEs outputs). Each Switch Box (SB) has W unidirectional wires, connected to other wires from adjacent SB in a configurable way. Each element of such fine-grained architecture is configured by one or more bits from the configuration register.

Mapping a fine-grained reconfigurable architecture on top of the fine-grained FPGA might suffer from a significant virtualization overhead. However due to large commercial FPGA capacities this approach makes sense for some applications be portability as important as resources utilization. Despite, several works from the literature focus on optimizing the implementation of such a vFPGA on the physical FPGA. Brant and Lemieux have proposed to use target specific dynamically re-programmable LUTs available in some host architectures (called LUTRAMs) to optimize the implementation of their fine-grained overlay ZUMA [17], getting a ratio down to 40 physical Look-Up-Table (LUT) per virtual LUT. Dirk Koch et al. [19] integrated a fine-grained overlay in the datapath of a MIPS processor to get a portable custom instruction set. They also propose an optimization of the direct mapping of overlay interconnection network into the switch fabric of the host FPGA. Besides, there also exist research efforts that focus on overlay integration and implementation. Wiersema et al. [21] propose to embed a ZUMA-based vFPGA architecture into their configurable system-on-chip ReconOS. Moreover, in [20], authors designed a fine grained overlay with extra routing resources to ease the task of their Just-In-Time synthesizer.

3.2.2. Coarse-grained overlays

The key attraction of coarse-grained overlays is the computational, energy, and software-like engineering efficiency. Fig. 2b) shows the basic architecture of coarse-grained overlays; REs consist of an array of Functional Units (FUs), or Processing Element (PEs). FUs can execute common word-level operations, including addition, subtraction, and multiplication, compared to the single-bit BLE operation in fine-grained architectures. Some architecture may contain in-tile memories, such as Register files, to hold temporary values and instructions. The majority of coarse-grained overlays can be restricted to just two classes, using the classification in [16]: spatially-configured, and time-multiplexed overlays. The largest group consists of spatially-configured overlays [16,23–25], where FU executes a single arithmetic operation and data is transferred over a dedicated point-to-point links between FUs. Both the FU and the interconnect remain unchanged while an application is executing, thus supporting the execution of pipelined

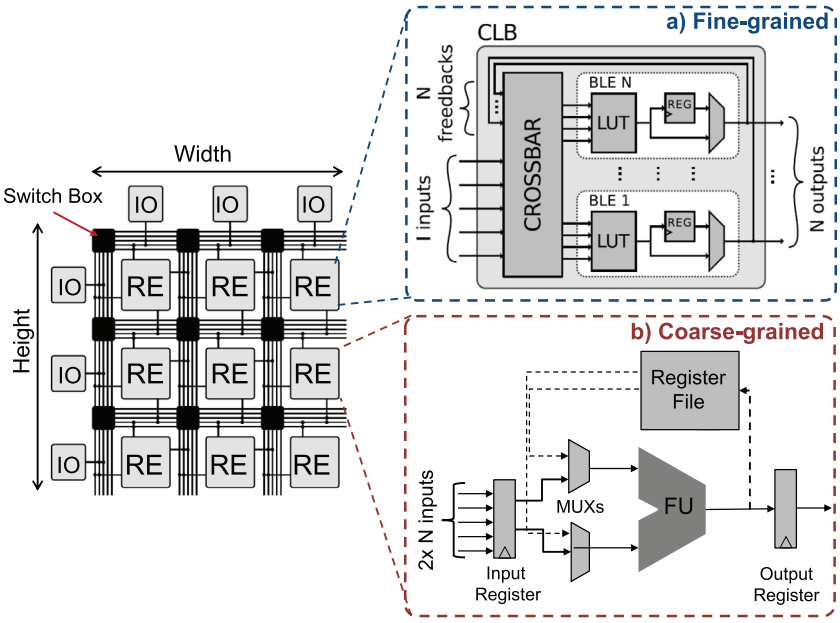

Fig. 2. Basic overlay architecture: a) LUT-based fine-grained VPR compatible architecture similar to ZUMA [17], and b) ALU-based coarse-grained architecture [18].

Data Flow Graph (DFG) applications. The configuration layer is therefore similar to the fine-grained overlay, where a specific register is used to configure the entire architecture for implementing a given DFG function. Furthermore, the time-multiplexed coarse-grained overlays [26] behave similarly to multi-core architectures; each tile (FU) contains a specific instructions memory, that is time-multiplexed among multiple operations.

3.2.3. Summary

To summarize, the overlay architecture can be seen as a set of reconfigurable elements. Fine-grained architectures allow the synthesis of a large range spectrum of applications, but suffer from a significant overhead. Coarse-grained are more cost-effective, but it is hard to define a particular configuration that suits a sufficient range of applications for the approach to be viable as a stand-alone solution. In this paper, we focus on exploitation of overlay architectures for virtualizing FPGAs, and to extend their functionalities in order to provide an efficient management of applications in a heterogeneous cluster. For this purpose, the target overlay, which serves as a case study, is a fine-grain architecture similar to ZUMA, the most advanced open-source architecture freely available today. Furthermore, our work provides a baseline for future virtual FPGA approaches that may reduce the performance or area overhead through various means.

3.3. Virtual and physical synthesis

The overlay HDL description is automatically generated from a specification of the computation layer. This specification expresses the available resources and their interconnections; the configuration layer is then automatically derived and eventually added to the model. Finally, a model transformation generates VHDL textual description of the architecture, allowing the overlay module to be instantiated from a user design. The generated RTL code is portable, simulation friendly, and synthesizable. The synthesis of this RTL model for the physical FPGA corresponds to the *physical synthesis*, and is a step required once each time new FPGA is introduced to the infrastructure.

The top part of the Fig. 3 shows the synthesis flow from the overlay generation to its physical implementation on the FPGA. Synthesizing an application design to the overlay architecture is done in different steps. First an RTL synthesizer transforms the application description into a netlist composed of latches and arbitrary logic gates. This netlist is then transformed, optimized and mapped to the overlay resources. Next, it is placed and routed on the overlay. Finally, the virtual bitstream (vBitstream) is generated by extracting the configuration of each one of the overlay's resources according to the placed and routed netlist. These four synthesis steps are gathered in one step called "synthesis targeting the overlay" at the bottom of the Fig. 3.

3.4. Vfpga architecture evolution

Overlays offer a clear separation of concerns between architecture (the vFPGA) and implementation (the FPGA) points of view. As previously mentioned, this promotes stability over time of the architecture, whichever physical FPGA acts as the host platform. Obviously, this does not necessarily mean that no new overlay templates should be made available. The questions are: when does change happen? and How can we ensure that overlays do support changes?

Change may be application driven, with the intent to offer a tailored overlay to new application needs. Yet, developing new overlays, in order to maximize performances for a class of applications, raises no portability issue as the applications are novel with no legacy to preserve. Change may also come from relaxing the implementation constraints, hence reflecting some improvements in the host platform up to the overlay. When the overlays evolve while the applications do not, portability is a critical issue. Two directions have to be considered for overlays scaling.

First, a new FPGA, with more abundant resources can host several overlays, appearing as nodes within a hierarchical architecture. This leads to update the architecture supervisor, but programming tools are kept the same. Second, scaling can be absorbed by reshaping the same template, but then, binary compatibility is lost.

However, as our framework offers a full control over the design phase and operation of overlays, we ensure backward compatibility is preserved. This is illustrated in Fig. 4, in which an A-architecture bitstream is implemented over a B-Architecture. A bitstream is first read back to produce a netlist, which is placed and routed over

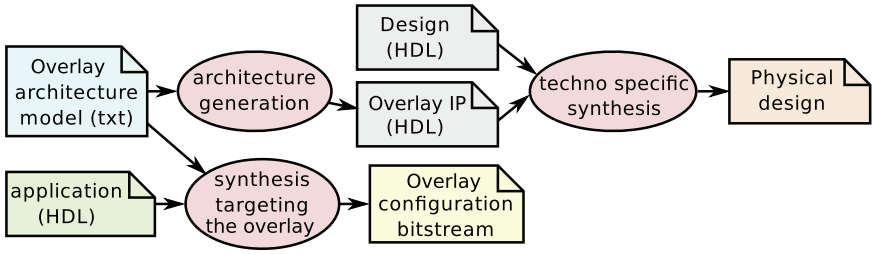

Fig. 3. Two flows: overlay synthesis on the FPGA, and application synthesis on the overlay.

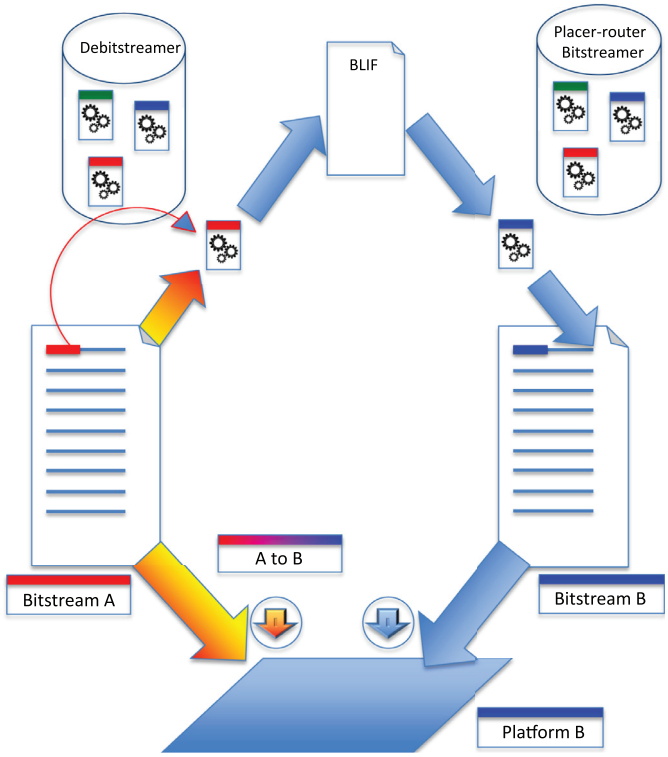

Fig. 4. Ensuring backward bitstream compatibility.

the new overlay, prior to bitstream generation. From a functional point of view, this appears as a model-to-model transformation. This translator can be automatically generated, assuming a tag is inserted within the bitstream to identify the target overlay.

Algorithm 1 illustrates how the netlist is extracted from the bitstream. Note that the bitstream can be extremely noisy, only the information that makes sense is considered (paths ignore any route that does not start/end by LUT, register or IO). Depending on the new overlay, the netlist is post-processed (technology mapping, LUT/flip-flop packing, etc.).

Algorithm 1: Generate a netlist

```
Data: Nodes, Nets, Sources
Result: A netlist
initialize Nets and Nodes as Empty Collections;
initialize Sources as the set of Used LUTs plus Used IOs;
while Sources is not empty do
    remove s from Sources;
    register s as Node;
    build p the path from the output pin of s;
    forall the destinations of p do
        add to Sources;
    end
    register p as Net;
end
```

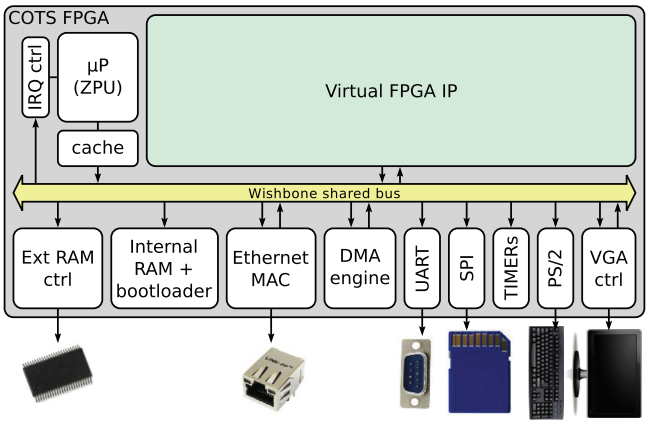

Fig. 5. ZeFF platform overview.

4. Zeff deployment platform

As previously discussed, designing of a virtual FPGA (vFPGA) on top of COTS FPGAs allows to homogenize any heterogeneous cluster. However, a natural requirement at this stage is to be able to communicate with the vFPGA layer: load a bitstream to the configuration register, extract the state of the snapshot, stop and run the clock, *etc.* For this purpose, this section introduces the integration of the vFPGA in a complete System on Chip (SoC), called Zeff.

4.1. Hardware platform

A vFPGA is seen as an element, integrated along with a configuration controller, some memories and communication devices to form a SoC. To this end, Lagadec et al. [27] introduced ZeFF, a vFPGA host platform synthesized in the physical FPGA, along with its attached virtual FPGA. ZeFF offers monitoring and management facilities (guest configurations and data streams, remote access through a standard Ethernet interface and TCP/IP protocol). As shown in Fig. 5, the SoC platform embeds, among others: a processor, some memory controllers (*e.g.*. external RAMs, flash memories, and SD cards), and some communication peripherals (*e.g.* Ethernet, UART).

In this architecture, the vFPGA is wrapped as a peripheral device, connected to the system bus. The wrapper has a Wishbone interface, which map in memory slave registers: *i*) to receive from external interfaces the value of the configuration register decoded from the vBitstream, *ii*) to extract and restore the state of the application, and *iii*) run/stop the application clock. Moreover, the virtual inputs/outputs (vIOs) are also mapped in a specific memory

area. The access to these vIOs is done through a dedicated DMA (stream in and out).

ZeFF SoC is minimalist, in order to let most of the host FPGA resources available to the vFPGA. The soft-core processor is a ZPU, known to be the smallest 32-bit processor supported by gcc. For example, the Zeff platform without vFPGA occupies only 11% of the total LUTs and 10% of the total flip-flops on the Xilinx Artix7, and 7% of the combinatorial functions and 2% of the total registers on the Altera Cyclone-IV. Even with its low resource and portability constraints, the SoC can run the real time operating system FreeRTOS, the FatFs FAT file system module, and the lwIP TCP/IP stack.

The embedded software allows to manage the platform and the vFPGA through an API and services such as a TFTP server to exchange the vFPGA's configuration and data files, a minimal web server to easily browse the file-system and a local or TELNET shell to issue commands. As a result, the vFPGA management API can be used from the embedded software, from shell commands for human interactions, and through a network protocol on top of TCP for remote machine management.

When porting the whole platform from one FPGA board to another, some parts of the SoC may have to be changed to adapt to different external devices, such as memories or transceivers, which can have different IO interfaces between boards. Therefore, the SoC is organized around a wishbone shared bus, associated to a dedicated generator, easing the addition and removal of peripherals, thus making the platform more flexible.

4.2. Software platform

The vFPGA architecture that we integrated into ZeFF allows to have full control of the vFPGA clock and to save the execution state of the virtual fabric. These features help to abstract applications away from the bare metal FPGA, providing introspection capabilities and flexibility over execution on the fabric. These control mechanisms are orchestrated through ZeFF's processor, giving the software flexibility to manage the virtual hardware execution flow. Software management of the vFPGA can be done at different levels, bringing the following four use cases:

(1) The vFPGA and application management can be entirely done by the embedded operating systems running on ZeFF. Applications are treated on the vFPGA the same way common software processes are executed on processors. The OS manages which application runs and when, and also manages data processed by the vFPGA.
(2) The vFPGA management can also be done through an API. In this case, the application's developer has to partition his application targeting the vFPGA. He must provide a script making use of the API to control the execution of its application segments. This is similar to a software application segmented into threads, and in which the developer is in charge of synchronizing those threads.
(3) Another classical use of this API is software / hardware co-design, where the software part has a more important computation load and only delegates some parts of the processing to the vFPGA. However, ZeFF's softcore processor (the ZPU) is not suited for computation loads. Running software / hardware applications on ZeFF requires a more powerful processor to be added to the platform via the system bus. It would only process computational tasks, letting the ZPU orchestrate the whole platform.
(4) ZeFF can also serve as an intermediate between the vFPGA and a remote machine, receiving data, virtual configurations and vFPGA management orders from the network. A cluster of ZeFF platforms could then be connected to and managed by a single host computer.

5. New overlay features

The perfect mastery of the virtual layer architecture allows to integrate features into the virtual fabric that are considered missing in the host, increasing the FPGA capabilities, and offering new features: Controllability and Introspection. In this section, we propose new overlay functionalities enhancing the capacity of applications management of overlay architectures.

5.1. Pre-configuration

Configuration latency is a challenge in reconfigurable computing, especially for frequent reconfigurations, as it can offset the performance improvement achieved by hardware acceleration. The configuration layer in the overlay can be implemented as a multiple-chain shift register to speed-up the dynamic reconfiguration. This register contains the application vBitstream, which is a contiguous sequence of bits that corresponds to the adequate configuration of overlay resources (LUTs, CLBs, FU operations, *etc.*) to implement a given circuit. However, such improvement is not sufficient, as the reconfiguration time depends also on the communication interface and the size of the vBitstream to be shifted. In order to achieve a higher efficiency, we propose to add a pre-loading feature to the overlay architecture by duplicating the configuration register, as shown in green in Fig. 6. The idea is to start the transfer of the new configuration file (vBitstream), without affecting the configuration of the executed application. Once the transfer is completed, the overlay commutes from the old configuration to the new one in one clock cycle (*shift config* signal) on demand. In this way, the latency of the configuration is neglected enabling the implementation of cost-effective live migration and scheduling algorithms.

5.2. The snapshot

In general, applications running on reconfigurable architectures can be represented by the resources configuration and the state of the application. Authors in [28] report two ways for accessing the state of a task that executes on a FPGA:

- By using the Internal Configuration Access Port (ICAP), which is mostly used for the Dynamic Partial Reconfiguration (DPR). This solution remains technology and vendor dependent. Additionally the state is read back along with configuration bits, which leads to a slow extraction process. However the mechanism is transparent to the application.
- By decorating applications with some access facilities to state bits. This solution is portable, and state extraction is efficient. However, every application has to be reworked, and both area and frequency are impacted.

In an overlay context, configuring the vBitstream on the vFPGA happens as presented in the previous sub-section, while the state of the application is hold on memory elements in the computation layer. These memories correspond in this architecture to the virtual application registers integrated in each reconfigurable element. In order to enable the state extraction and restoring of applications running on overlays, a novel feature extends the proposed overlay architecture: the *state* layer. Associating one snapshot for each virtual application register (as shown in red in the Fig. 6 does this. Two global signals control the copy of the application register values to their associated snapshot registers (save), or to force the snapshot register values to the application registers (restore). All snapshot registers are connected to form one or more shift registers, similarly to the configuration register, allowing the extraction or loading of the overlay state without affecting the execution. Extracting or loading a state snapshot requires several clock cycles

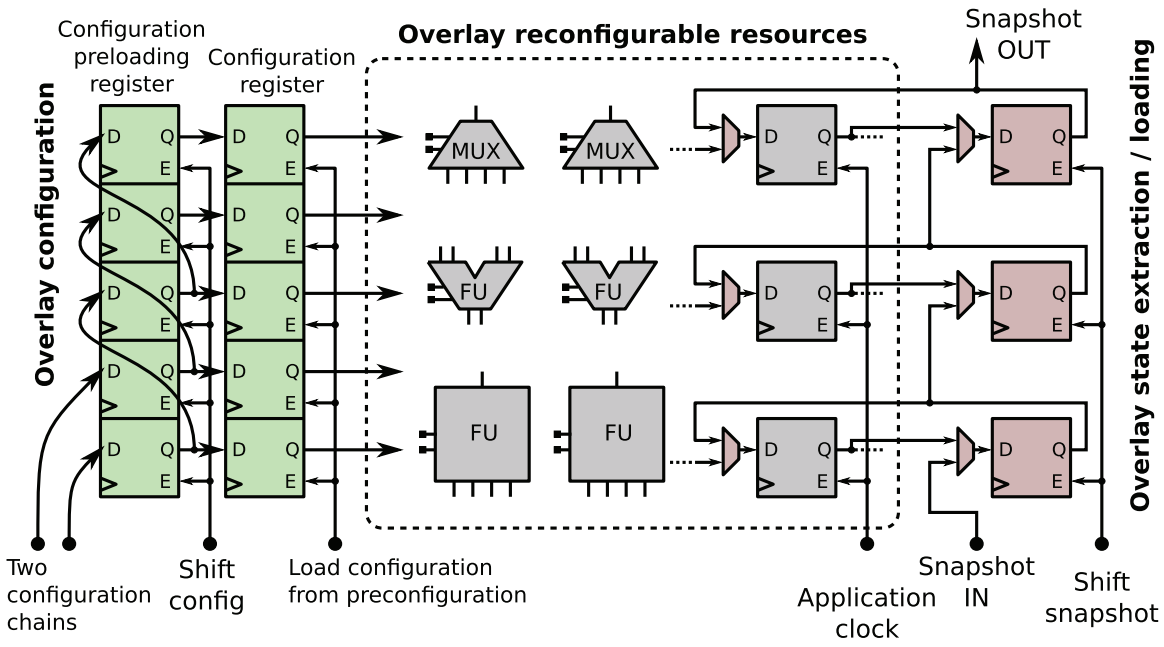

Fig. 6. Configuration double-chain shift register, reconfigurable elements and application registers, snapshot mechanism.

(depending on the number of memory elements). However, saving or restoring a snapshot (from the snapshot registers to the application registers) only takes one clock cycle. A dedicated controller ensures the communication with the state layer through both the overlay *state in* and *state out* signals.

5.3. Clock management

In order to stop/run the application clock, a modified version of the architecture application register is proposed. We integrate an enable control signal to ensure the clock gating by: *i*) allowing the propagation of the information (application clock is running), or *ii*) blocking the information (application clock is stopped).

5.4. Hardware cost evaluation

The previously presented extra features of the overlay architecture allow the easy management of application on any FPGA device. In this section, we evaluate the cost of these features for two FPGAs: the Xilinx Artix-7 X7A100T-1CSG324C (nexys4 board), and the Altera Cyclone-V 5CGXFC9A7U19C8 (APF6-SP board). To this end, we have considered a case study the ZUMA-like fine-grain overlay (see Section 3.2.1), with two architectures: (*1*) a 10 × 10 CLBs with 4-input LUTs ($K = 4$) and 4 LUTs per CLB ($N = 4$) that fit in both FPGAs, and (*2*) a 20 × 20 CLBs with 4-input LUTs ($K = 4$) and 4 LUTs per CLB ($N = 4$) that allocates more Cyclone-V FPGA resources. The only variable parameter in this experiment is the number of wires (W) in each routing channel. For a given architecture, W has a direct impact on the routability of the overlay and the timing performance of applications; the more wires, the less congestion in routing channels. The hardware cost is computed here as the percentage of the additional occupied resources on each host FPGA.

In this experiment, several syntheses of the overlay, W ranges from 8 to 24, have been carried for both FPGAs. The synthesis of a reconfigurable architecture on top of a physical FPGA, makes difficult for classical tools to determine the maximum circuit frequency as all paths depends on the application that has to be later configured. For this purpose, we have used the concept of Virtual Time Propagation Registers (VTPRs), proposed in [29], to break down physical logic chains into short segments, and prevent any combinatorial loop from appearing on the physical FPGA whatever the virtual configuration is. Figs. 7–9 plot the host FPGA resources occupied by the overlay: (*i*) without any feature (blue), (*ii*) with snap-

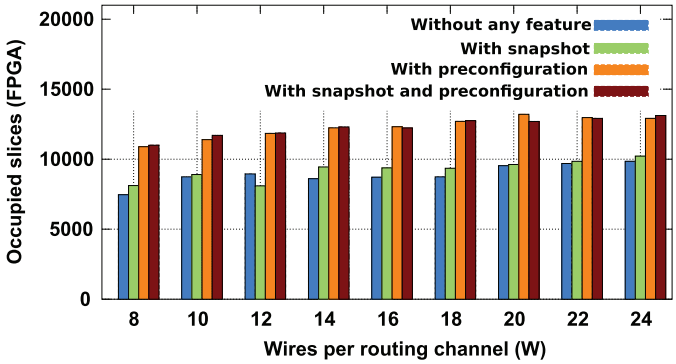

Fig. 7. Occupied resources in terms of total used slices for the synthesis of a 10x10 CLBs overlay on Xilinx Artix-7 varying W.

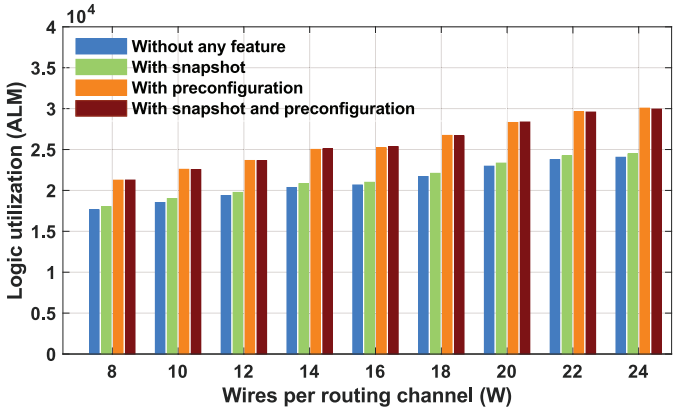

Fig. 8. Occupied resources in terms of Adaptive Logic Module (ALM) for the synthesis of a 10x10 CLBs overlay on Altera Cyclone-V varying W.

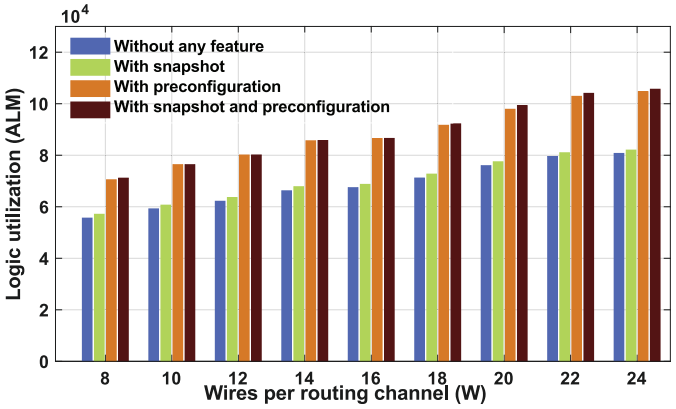

Fig. 9. Occupied resources in terms of Adaptive Logic Module (ALM) for the synthesis of a 20x20 CLBs overlay on Altera Cyclone-V varying *W*.

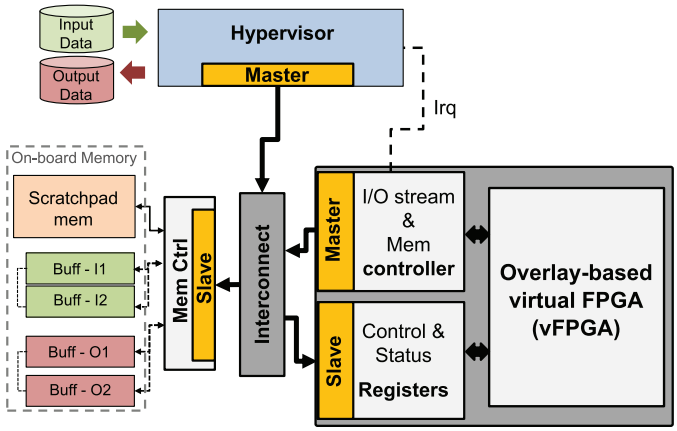

Fig. 10. Overview of the proposed framework.

shot register (green), (*iii*) with pre-configuration register (orange), and (*v*) with both features (brown). As shown in these figures, the snapshot register has a low-cost compared to the naive implementation (blue). The average overhead is about 3.52% for the Artix-7 and 2% for the Cyclone-V. This cost is due to the additional single-bit snapshot shift register added in each reconfigurable element, and does not depend on the parameter *W*. However, it is important to remark that the overhead of the pre-configuration register is much higher; the average overhead is about 37.89% for the Artix-7 and about 20–30% for the Cyclone-V. In fact, this cost is due to the high number of bits required to configure the experimented fine-grained overlay. For instance, the vBitstream for this overlay with $W = 16$ has 22,296 bits: 30.5% for the logic elements and 69.5% for the routing elements. By increasing the number of wires in SB channels, the number of required bits for the configuration increases. In case of a coarser architecture, the snapshot mechanism would have a higher overhead, while the pre-configuration would have less overhead, as in proportions the state is bigger and the configuration is smaller. In fact, the execution state of a coarse grained overlay is the output words of each FU, while it is composed of only one bit per BLE in a fine grained architecture. Also, the configuration selects from few operators per FUs instead of a full LUT content, and routing is done word-wise instead of bit-wise.

6. Hardware task management: scheduling and live migration

As reconfigurable architectures are increasingly integrated in hardware designs, sharing of these resources is a key requirement that gives rise to a higher hardware utilization. In this section, we investigate a possible solution to truly share programmable logic in a generic framework, enabling an efficient and cost-effective scheduling and live migration of hardware applications in a cluster of heterogeneous FPGAs.

6.1. Proposed framework

A generic and complete system is proposed and shown in the Fig. 10. It is composed of: (*i*) a hardware layer based on the vFPGA architecture previously presented in Section 3 integrating the proposed features from the Section 5, (*ii*) a hypervisor to control the entire system (Section 4) and, (*iii*) the system memory. The proposed architecture is built upon a smart interface, with master and slave interfaces that can be plugged to a wrapper. The slave interface deals with the control and status registers, while the master interface is responsible for accessing the memory. I/O streams are stored in double ping-pong buffers inside the on-board memory. A dedicated I/O stream controller is responsible for the management of these buffers. An interrupt is generated each time a complete input buffer ('Buff-I') is consumed, or one output buffer ('Buff-O') is filled. This IRQ informs the hypervisor to update the buffers in the memory. We also allocate an application scratchpad memory ('Mem') that corresponds to the on-chip 'BRAMs' in classical FPGA designs.

6.2. Tasks scheduling

Fig. 11a) illustrates a typical finite state machine for task execution states: *(i) Running*, the application is executing on the vFPGA, *(ii) Ready*, the application is ready to run and the vFPGA is busy, *(iii) Blocked*, the application is waiting for an event to resume the execution on the vFPGA (*e.g.* waiting for input data to be provided). At this stage, we focus on the infrastructure itself, and therefore on the main actions required to set up such a scheduling scenario.

First, we define the structure 'Job' (shown in Fig. 11b) to represent the state of the entire system for each application. It includes the following elements:

- Initialization: It refers to the vBitstream file, and to the status registers.
- State: The state at a given time: *(i)* the value of the snapshot register (circuit state), *(ii)* the execution time, and *(iii)* the scratchpad memory.
- Data: This part contains pointers to the remaining input data to manipulate, and to the produced output data.

This structure is manipulated by a set of functions (the hypervisor), which ensures the management and communication with the vFPGA and the initialization of the memory. It can be executed on the core, usually combined with FPGA in modern SoC+FPGA devices, or can be synthesized as a lightweight soft-core IP on the FPGA (Zeff core previously presented in Section 4. As shown in the Fig. 11a, two actions are required in order to set up a scheduling algorithm:

(1) Save Job: is called at each change from the *Running* state to *Blocked* or *Ready*. Fig. 11c) plots the sequence diagram, illustrating the set of actions and commands of the hypervisor to save the running Job. First, it sends a stop request to the vFPGA controller to freeze the application clock, and then starts extracting the value of the snapshot register. The next request is dedicated to the system memory in order to read the Job scratchpad memory and to flush the output buffers (Buff-O1 and Buff-O2). Collected data serve to update the Job structure and the input and output files.

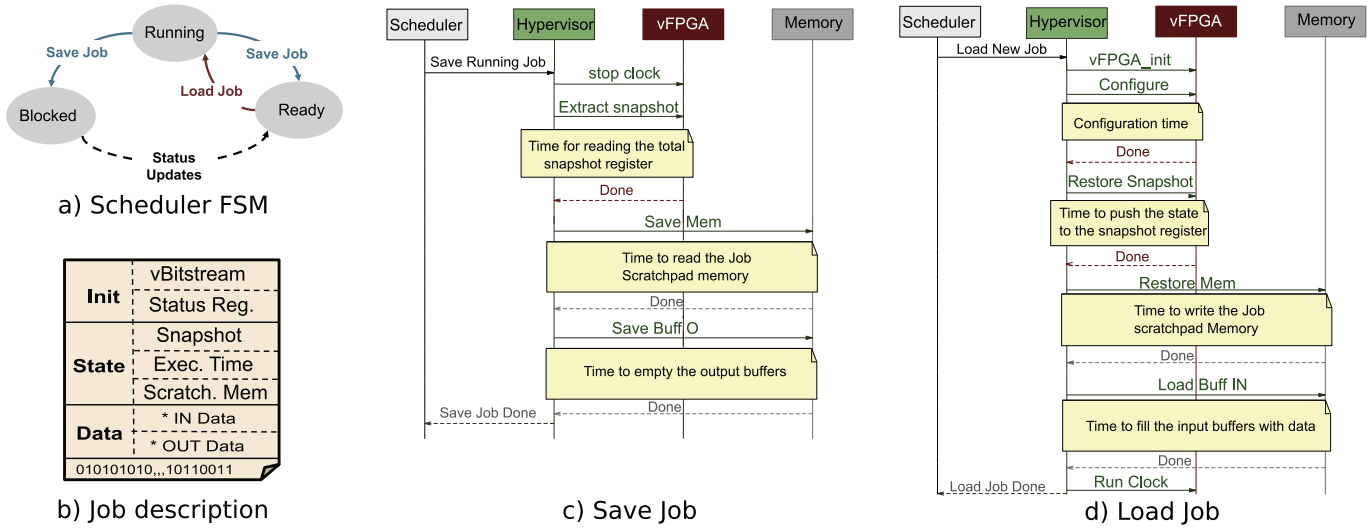

Fig. 11. The entire software layer and functions: a) remind of the classical scheduler FSM, b) the proposed Job structure, c) the sequence diagram of the save job action, and d) the one for the load job action.

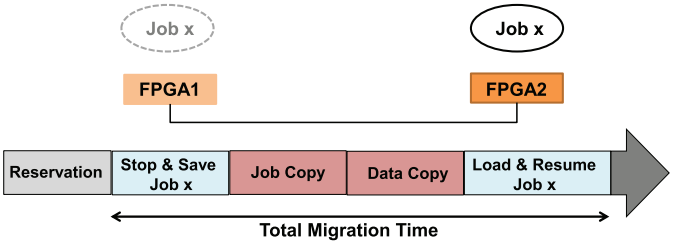

Fig. 12. A Total-copy live migration process.

(2) Load Job: is requested by the scheduler each time a new job is elected to be executed (change to *Running* state). The sequence diagram of this action is shown in the Fig. 11d). The hypervisor initializes the controller registers with a new initialization information, such as the number of clock cycles to be executed. The application bitstream is then sent to this controller, which in turn pushes the value into the configuration register of vFPGA. This step can be done upstream by sending it to the preconfiguration register before saving the Job, which results in reducing the scheduling overhead. The scratchpad memory is next restored and the double input buffers are filled. Lately, a request is sent to the controller to start the clock again.

6.3. Live migration

Applications live migration is a technique widely used in datacenters to move a virtual machine VM from one physical host to another. The mechanism is same as for the previously presented scheduling, except that the application is restored on a different fabric that the one it was previously executing on. The most frequently used migration algorithm is shown in the Fig. 12: a Total-Copy process transfers the entire state and the data before the process execution resumed on the destination FPGA [30].

The following actions are performed successively to migrate a Job from FPGA1 to FPGA2:

(1) Reservation: The cluster manager first searches for the host with the lowest workload, and sends a request to reserve the resource.
(2) Stop and Save the Job: The manager realizes the same *Save Job* action, previously specified, in order to stop and update the Job structure.
(3) Job Copy: The entire Job is copied from the host FPGA1 to FPGA2. The time required to complete this action depends on: *i*) the size of the Job, which is a function of the vFPGA specification, and *ii*) the cluster network bandwidth.
(4) Data Copy: The data manipulated by the Job are then copied to the FPGA2. The time required to complete this action also depends on the size of the data file, and on the network bandwidth.
(5) Load and Resume Job: At this stage, the manager requests a *Load Job* action to initialize the vFPGA with the new job and to resume the execution on the new host. This action is also detailed in the previous sub-section.

6.4. Demonstration: task migration

This system has been demonstrated in the international conference on Design & Architectures for Signal & Image Processing [31], in order to illustrate how to offer:

- an homogeneous view of a heterogeneous set of FPGAs;
- the live migration of a hardware application between two nodes;
- fault tolerance of an overlay cluster.

The setup includes two FPGAs from two vendors (Xilinx and Altera) as compute nodes, and a host PC as a controller. Fig. 13 shows the experimental setup. The two FPGAs are connected to the host PC through Ethernet, each one is attached to an ARM processor running a local hypervisor, which transfers the host management requests to the FPGA. For this demonstration, the processors are also used to display the image being processed, so that the audience can visualize the progress of the job execution.

The experimented vFPGA is a 14× 13 CLBs ZUMA-like overlay architecture, and has 728 4-inputs LUTs and FFs. This architecture takes: (*i*) 64% of the total LUTs (40935) and 41% of the total FFs (52258) on the Xilinx nexys-4 X7A100T-1CSG324C, and (*ii*) 30% of the total Logic elements (34352) and 55% of the total FFs (63140) on the Cyclone V 5CGXFC9A7U19C8. The size of the vBitstream and the snapshot for this vFPGA are 5116 Bytes and 92 Bytes, respectively. The time required to configure, save and restore the vFPGA depends on several parameters and is finely studied in the next section. In this demonstration, three image processing applications were synthesized, placed and routed for the vFPGA (synthesis result are shown in the Table 1): (*i*) Sobel creates an image, em-

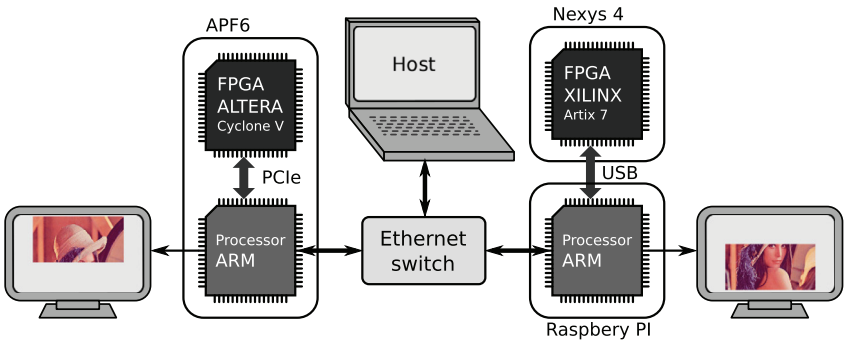

Fig. 13. Experimental setup of the DASIP demonstration.

Table 1
Image processing applications synthesized for a demonstration on a 14× 13 fine-grained vFPGA.

Application	Virtual Synthesis			F_{max}	
	vLUTs	vFFs	pairs vLUT-FF	X7A100T-1CSG324C	5CGXFC9A7U19C8
Sobel	567 (77.9%)	195 (26.8%)	195 (34.4%)	2.7 MHz	1.8 MHz
Smoothing	340 (46.7%)	116 (15.9%)	116 (34.1%)	3.33 MHz	2.16 MHz
harpening	376 (51.6%)	244 (64.9%)	132 (35.1%)	3.22 MHz	2.09 MHz

phasizing edges based on the Sobel-Feldman operator, (*ii*) Smoothing, which uses a matrix to smooth the image data set, and (*iii*) a Sharpening operation of the image.

The first goal is to show binary compatibility among both FPGAs. To this end, the same vBitstream of one of the applications is dispatched by the host PC on both FPGAs. The result of the image filtering starts then appearing on both screens at the same time. This compatibility is ensured, thanks to the vFPGA overlay architecture deployed on both physical FPGAs.

The live migration is then demonstrated by running the filter application on the first node, halting the application execution, capturing the execution state of the node's overlay, and then restoring the state of the vFPGA on the second node. The application resumes filtering the image on the second node, as shown in the Fig. 13.

Finally, distributing computations over a set of nodes offer speedup the execution or guarantee fault tolerant. The Fault tolerance at the cluster level is illustrated by running one application on one node. The host controller periodically backups the execution state of the running node (every second in this demonstration). Then the power of the running node is shut down. When the host controller notices that the running node have disappeared, the node does not respond to heartbeat pings anymore, the host sends the Job of the interrupted application along with the last execution state backup to the second node. The execution resumes on the second node at the last backup.

6.5. Timing models

No efficient scheduling nor meaningful task migration can be gained without an early knowledge of the reconfiguration time overhead. To this end, we have developed the cost model in terms of the required time for each component to perform both actions *save* and *load* job as previously presented. We define first the following variables:

- S_{snap} and S_{config}: the size of the snapshot register and the vBitstream (in Bytes), which are functions of the overlay granularity and parameters.
- $S_{scratch}$ and S_{Buff}: the size in bytes of the scratchpad memory, and the input buffers ('Buff-I1' and 'Buff-I2' in the previous Fig. 10.
- $S_{d_{out}}$: the size in bytes of the available data in the output buffer ('Buff-O1' or 'Buff-O2').

6.5.1. Save job

As can be conducted from Fig. 11c), the time taken to save the Job is a function of the needed time to extract the snapshot register (T_{s_snap}), to save the application scratchpad memory ($T_{s\_scratch}$) and to flush out the output buffer (T_{s_BuffO}). This function is described in the equation (1), where T_s^0 is the constant part, and corresponds to the time required by the hypervisor (or core) to execute the request verification code (*e.g.* checking the compatibility between the bitstream of the application and the available vFPGA architecture, *etc.*). The cost model of the snapshot extraction is linearly approximated in terms of S_{snap} as shown in this equation, where, L^r_{vFPGA} is the latency for a read operation of one word from the hardware layer. $T_{s\_scratch}$ and T_{s_BuffO} are also linear functions of the latency for a read operation from the on-board memory L^r_{MEM}.

$$T_s = T_s^0 + L^r_{vFPGA} \times S_{snap} + L^r_{MEM} \times S_{scratch} + L^r_{MEM} \times S_{d_{out}} \quad (1)$$

6.5.2. Load job

The time required to complete the load job action can also be conducted from Fig. 11d). The equation (2) describes it as a function of the needed time to: (*i*) configure the vFPGA T_{config}, (*ii*) restore the snapshot (T_{rest_snap}), (*iii*) load the application scratchpad memory ($T_{l\_scratch}$) and (*iv*) the time to fill both input buffers (T_{l_Buffl}). T_l^0 is the constant part, and also correspond to the time of the job verification. L^w_{vFPGA} and L^w_{Mem} are the latency for a write operation of one word to the hardware architecture and to the memory respectively.

$$T_l = T_l^0 + L^w_{vFPGA} \times S_{config} + L^w_{vFPGA} \times S_{snap} + L^w_{Mem} \times S_{scratch} + L^w_{Mem} \times Size(2 \times S_{Buff}) \quad (2)$$

6.6. Reconfiguration time estimation

This section reports our experiments when evaluating the accuracy of the proposed cost models, and explores the overhead of the implementation of scheduling algorithms in the proposed framework. The setup used in our experiments is the APF6-SP SoC+FPGA platform from Armadeus, which is composed on the i.MX6 Cortex-A9 processor and the 5CGXFC9A7U19C8 Cyclone V

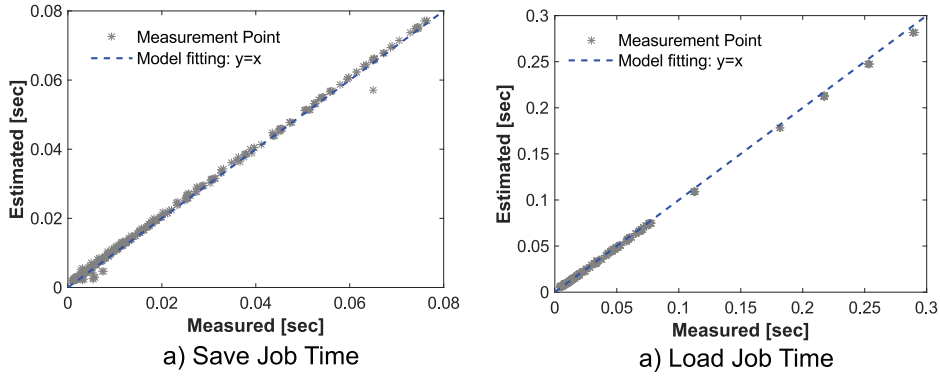

Fig. 14. Model fitting for both: a) Save job time, and b) Load job time.

Table 2
Estimated parameters for the APF6 SP target SoC+FPGA.

Parameters	Value [μ s/32-bit]	Constants	Value [μ sec]
L^r_{vFPGA}	2.37	T^0_s	1300
L^w_{vFPGA}	0.96	T^0_l	3700
L^r_{Mem}	2.34	–	–
L^w_{Mem}	0.26	–	–

GX FPGA from Altera. I.MX6 implements our proposed hypervisor, while the hardware architecture is synthesized for the Cyclone V FPGA. PCI-express Gen1 is used for the communication between the hypervisor and vFPGA, and the on-board DDR3 is used to implement the system memories.

6.6.1. Parameters estimation and models accuracy

The first experiment aims at estimating the model parameters. To this end, we implement the entire system on the APF6-SP platform, and we use the CLOCK_MONOTONIC to measure each timing component. Several system configurations are also considered in order to find the model parameters: S_{Buff} from 1KB to 1MB, $S_{scratch}$ from 64 Bytes to 124KB, S_{config} from 1KB to 18KB, and S_{snap} from 20 Bytes to 340 Bytes. The Table 2 illustrates the estimated parameters by the linear regression from Matlab for more than 10,000 measurements. L^w_{vFPGA} is lower than L^r_{vFPGA}, due to the fact that the PCI write burst is generated by the SoC, while the read burst was not supported in the experimented architecture. Similarly, L^r_{mem} and L^w_{Mem} are estimated from the access to the on-board DDR.

The model's accuracy is evaluated based on the following two metrics: (*i*) the square of the correlation between the measured and estimated values R^2 (closer to 1 indicates the model better fits), and (*ii*) Mean Absolute Error (MAE), which is the average of the absolute error of the regression (closer to 0 is better). Fig. 14a) and b) plot the estimated time compared to the measured time for the save Job and load Job, respectively. As can be noticed, the proposed cost models accurately estimate the time required to load and save jobs: R^2 is higher than 0.99 with a MAE about 0.85 ms and 1.36 ms for both T_s and T_l respectively.

6.6.2. Equal time round robin scenarios

At this stage, we aim to evaluate the re-configuration time overhead. To this end, we implement classical scheduling algorithms, for a 14x13 CLBs ZUMA-like overlay on the APF6 SoC + Cyclone V C9 FPGA, with 4 BLEs in each CLB and 4 inputs LUTs. The pre-loading feature is first disabled. In our system, the nature of the application to be mapped on the overlay is defined by its bandwidth, or the frequency of producing data F_{out}. In this experiment, we choose different applications, as shown in the Table 3, corresponding to different bandwidths.

We first implement the Equal Time Round Robin (ETRR) scheduling, which is a cyclic executive process without priority. It allocates a unique time-slice to each running Job; this is called the Quantum (denoted Q). The execution profile of these jobs for an ETRR with $Q = 250$ ms is shown in Fig. 15. As it can be noticed in this figure, the Job 3, corresponding to the highest F_{out}, fills 5 times the output buffer, activating an interrupt signal to fill the input buffer (DMA IN update event) and to empty the output buffer (DMA OUT update event). Moreover, the measured time between two contexts switch is variable, depending on: (*i*) the time needed to empty the output buffer, where the size of the produced data is a function of F_{out}, (*ii*) the size of the old application scratchpad memory to save, (*iii*) the size of the new application scratchpad memory to load, and (*iv*) the time needed to fill the input buffers. The measured value total context switch time is between 11 ms to 50 ms, corresponding to 4%–20% of performance overhead.

Moreover, we aim at comparing different configurations of ETRR to the basic First Come First Serve scheduling (FCFS) algorithm. The normalized total execution time is shown in Fig. 16, which is the time taken by all jobs to complete their computation on 4GB of input data. FCFS requires 499.5 s to sequentially execute all Jobs, while the total time for ETRRs is between 504 and 560 s depending on the configuration of the system. From this experiment, we conclude that the size of both inputs and output buffers and the choice of the Quantum Q have significant impacts on the system performance, up to 12% of reconfiguration time overhead.

6.6.3. Overlay reconfiguration time

In this paper, we have experimented our virtualization approach using a fine-grained vFPGA architecture, which is a naive and a generic overlay allowing the synthesis of a large spectrum of applications. However, the experimented architecture was not optimized to efficiently exploit physical FPGA resources. In this section, we use the previous timing models to extrapolate the impact of the complexity of overlay architectures on both: the total time to load a job (T_l) and the total time to save a job (T_s). For this purpose, both the size of the input/output buffers and the scratchpad memory are constant in this experiment. We recall that overlay-based architectures are characterized by: (*1*) the size of their configuration register (S_{config}), and (*2*) the size of the snapshot register (S_{snap}).

Fig. 17 plots each component of the total time for a load job, considering different sizes of overlays; we have increased the matrix size of resource elements on ZUMA-like architecture. As can be seen in this figure, the time to configure the overlay (T_{config}) is the most significant part (more than 90%) of the total time to load the job. This motivates the need of a bitstream pre-loading mechanism in order to tackle this challenge. The proposed non-preemptive pre-loading feature support reducing up to 99% of the

Table 3
Applications for a 14× 13 ZUMA-like fine-grained overlay architecture with: 4 BLEs per CLB and LUT4.

Job id	App. name	F_{out} (KB/s)	$S_{scratch}$ (Bytes)	Description
0	cordic	15.2	0	Trigonometric function, 16 bits operations.
1	IIR filter	11.9	64	IIR filter 4th order, 9 16 bits signed multiplication and addition, 20 bits accumulation.
2	mult	120	0	Combinatorial multiplication, 16bits operands and 32 bits result.
3	smult	1031.7	0	Pipelined multiplication, 8 bits operands and 16 bits result.
4	sobel	83.96	4096	Sobel filter for image processing, 3× 3 pixels operations and 16 bits per pixel.

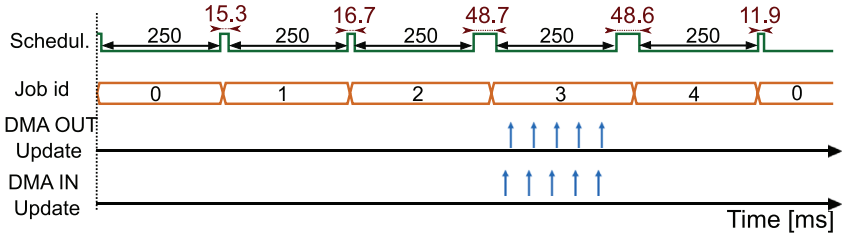

Fig. 15. Execution of the jobs for an ETRR scheduling with Q = 250 ms.

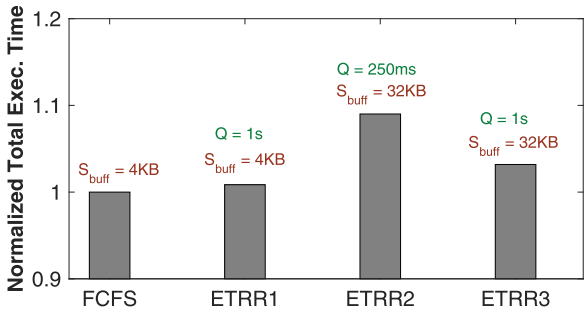

Fig. 16. The total execution time of different configuration of ETRR compared to the FCFS scheduling.

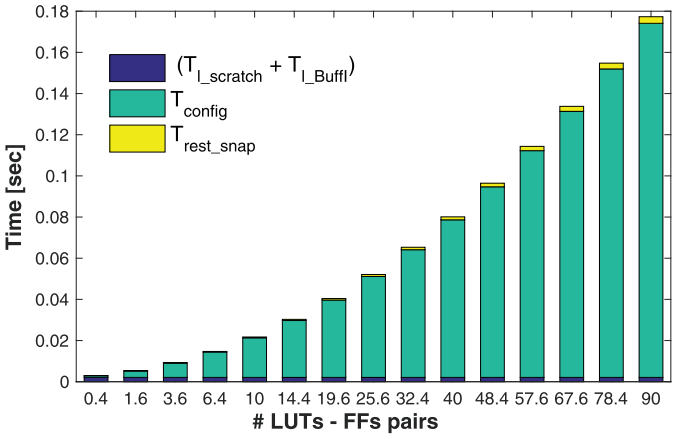

Fig. 17. The time for a load job: the architecture configuration time (T_{config}), the time to load both scratchpad and input buffers, and the time to restore the snapshot(T_{rest_snap}).

total reconfiguration time overhead, as the bitstream configuration is done upstream. Moreover, the time to restore the snapshot (T_{rest_snap}) modestly increases with the increase in the vFPGA matrix size: T_{rest_snap} is estimated to 9.89 ms for the biggest architecture with 90k LUTs and FFs.

Moreover, the overlay architecture has only impact on the time to save the snapshot for the saving job time. Fig. 18 shows both component of the total saving job time, considering different sizes of overlay architecture. Indeed, T_{s_snap} becomes significant for

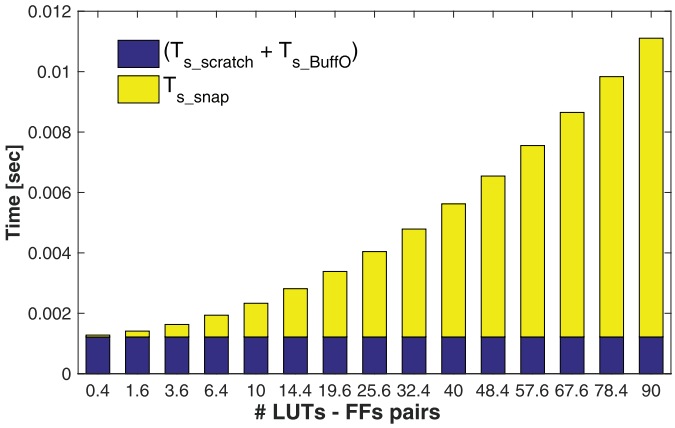

Fig. 18. The time for a save job: the time to save the scratchpad memory and to empty the output buffer, and the time to save the snapshot (T_{s_snap}).

biggest overlay architectures, but however this is negligible compared to the total time of a load job for the same architecture.

7. Conclusion

This paper has presented a new hardware virtualization approach for heterogeneous FPGA cluster. When FPGA are gradually updated and replaced to follow the technology evolution over the lifetime of the infrastructure, overlays have demonstrated to improve portability, speed up reconfiguration, and promote resources abstraction. The proposed idea is to map a second layer of reconfigurable resources on top of the commercial-of-the-shelf (COTS) FPGAs in order to homogenize the infrastructure. This work has also demonstrated how slightly extending the overlay architecture can bring novel features for sake of improved management of applications. The proposed platform was capable of node-to-node application migration. We also presented accurate linear models for the estimation of the reconfiguration time overhead. Designing of efficient scheduling and live migration algorithms and systems for any FPGA platform will use the presented models for the overhead estimation: few measurements are required to adapt models' parameters. In future work, we wish to develop power models for the dynamic reconfiguration. We will also investigate the development of efficient scheduling and load balancing, taking into account both performance and power as targets for optimizations.

References

[1] S. Jayakumar, S. Sumathi, High speed vedic multiplier for image processing using fpga, in: 2016 10th International Conference on Intelligent Systems and Control (ISCO), 2016, pp. 1–4, doi:10.1109/ISCO.2016.7727059.

[2] G. Bieszczad, Soc-fpga embedded system for real-time thermal image processing, in: 2016 MIXDES - 23rd International Conference Mixed Design of Integrated Circuits and Systems, 2016, pp. 469–473.

[3] Y. Toyoda, N. Koike, Y. Li, An fpga-based remote laboratory: implementing semi-automatic experiments in the hybrid cloud, in: 2016 13th International Conference on Remote Engineering and Virtual Instrumentation (REV), 2016, pp. 24–29, doi:10.1109/REV.2016.7444435.

[4] S.A. Fahmy, K. Vipin, S. Shreejith, Virtualized fpga accelerators for efficient cloud computing, in: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), 2015, pp. 430–435, doi:10.1109/CloudCom.2015.60.

[5] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, K. Wang, Enabling fpgas in the cloud, in: Proceedings of the 11th ACM Conference on Computing Frontiers, in: CF '14, ACM, New York, NY, USA, 2014, pp. 3:1–3:10, doi:10.1145/2597917.2597929.

[6] W. Wang, M. Bolic, J. Parri, pvfpga: accessing an fpga-based hardware accelerator in a paravirtualized environment, in: 2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013, pp. 1–9, doi:10.1109/CODES-ISSS.2013.6658997.

[7] R. Brodersen, A. Tkachenko, H.K.H. So, A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph, in: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06), 2006, pp. 259–264, doi:10.1145/1176254.1176316.

[8] R. Kirchgessner, G. Stitt, A. George, H. Lam, Virtualrc: a virtual fpga platform for applications and tools portability, in: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, in: FPGA '12, ACM, New York, NY, USA, 2012, pp. 205–208, doi:10.1145/2145694.2145728.

[9] J.D. Dondo, J. Barba, F. Rincn, F. Moya, J.C. Lpez, Dynamic objects: supporting fast and easy run-time reconfiguration in {FPGAs}, J. Syst. Archit. 59 (1) (2013) 1–15, doi:10.1016/j.sysarc.2012.09.001.

[10] S. Byma, J.G. Steffan, H. Bannazadeh, A.L. Garcia, P. Chow, Fpgas in the cloud: Booting virtualized hardware accelerators with openstack, in: 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines, 2014, pp. 109–116, doi:10.1109/FCCM.2014.42.

[11] O. Knodel, R.G. Spallek, Computing framework for dynamic integration of reconfigurable resources in a cloud, in: 2015 Euromicro Conference on Digital System Design, 2015, pp. 337–344, doi:10.1109/DSD.2015.37.

[12] M. Feilen, M. Ihmig, C. Schwarzbauer, W. Stechele, Efficient dvb-t2 decoding accelerator design by time-multiplexing fpga resources, in: 22nd International Conference on Field Programmable Logic and Applications (FPL), 2012, pp. 75–82.

[13] B. Ronak, S.A. Fahmy, Improved resource sharing for fpga dsp blocks, in: 2016 26th International Conference on Field Programmable Logic and Applications (FPL), 2016, pp. 1–4, doi:10.1109/FPL.2016.7577373.

[14] A. Bourge, O. Muller, F. Rousseau, Automatic high-level hardware checkpoint selection for reconfigurable systems, in: 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, 2015, pp. 155–158, doi:10.1109/FCCM.2015.8.

[15] F. Duhem, F. Muller, P. Lorenzini, Reconfiguration time overhead on field programmable gate arrays: reduction and cost model, IET Comput. Digital Tech. 6 (2) (2012) 105–113, doi:10.1049/iet-cdt.2011.0033.

[16] A.K. Jain, D.L. Maskell, S.A. Fahmy, Are coarse-grained overlays ready for general purpose application acceleration on fpgas? in: Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2016 IEEE 14th Intl C, IEEE, 2016, pp. 586–593.

[17] A. Brant, G. Lemieux, Zuma: an open fpga overlay architecture, in: 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, 2012, pp. 93–96, doi:10.1109/FCCM.2012.25.

[18] A.K. Jain, X. Li, P. Singhai, D.L. Maskell, S.A. Fahmy, Deco: a dsp block based fpga accelerator overlay with low overhead interconnect, in: Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on, IEEE, 2016, pp. 1–8.

[19] D. Koch, C. Beckhoff, G.G.F. Lemieux, An efficient fpga overlay for portable custom instruction set extensions, in: 2013 23rd International Conference on Field programmable Logic and Applications, 2013, pp. 1–8, doi:10.1109/FPL.2013.6645517.

[20] R. Lysecky, K. Miller, F. Vahid, K. Vissers, Firm-core virtual fpga for just-in-time fpga compilation (abstract only), in: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays, in: FPGA '05, ACM, New York, NY, USA, 2005, p. 271, doi:10.1145/1046192.1046247.

[21] T. Wiersema, A. Bockhorn, M. Platzner, Embedding fpga overlays into configurable systems-on-chip: reconos meets zuma, in: ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on, IEEE, 2014, pp. 1–6.

[22] V. Betz, J. Rose, Vpr: a new packing, placement and routing tool for fpga research, in: International Workshop on Field Programmable Logic and Applications, Springer, 1997, pp. 213–222.

[23] G. Stitt, J. Coole, Intermediate fabrics: virtual architectures for near-instant fpga compilation, IEEE Embed. Syst. Lett. 3 (3) (2011) 81–84, doi:10.1109/LES.2011.2167713.

[24] D. Capalija, T.S. Abdelrahman, A high-performance overlay architecture for pipelined execution of data flow graphs, in: Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, IEEE, 2013, pp. 1–8.

[25] J. Benson, R. Cofell, C. Frericks, C.H. Ho, V. Govindaraju, T. Nowatzki, K. Sankaralingam, Design, integration and implementation of the dyser hardware accelerator into opensparc, in: IEEE International Symposium on High-Performance Comp Architecture, 2012, pp. 1–12, doi:10.1109/HPCA.2012.6168949.

[26] C. Liu, H.C. Ng, H.K.H. So, Quickdough: a rapid fpga loop accelerator design framework using soft cgra overlay, in: 2015 International Conference on Field Programmable Technology (FPT), 2015, pp. 56–63, doi:10.1109/FPT.2015.7393130.

[27] L. Lagadec, J.-C. Le Lann, T. Bollengier, A prototyping platform for virtual reconfigurable units, in: 2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2014, pp. 1–7, doi:10.1109/ReCoSoC.2014.6860689.

[28] K. Jozwik, H. Tomiyama, M. Edahiro, S. Honda, H. Takada, Comparison of preemption schemes for partially reconfigurable fpgas, IEEE Embed. Syst. Lett. 4 (2) (2012) 45–48, doi:10.1109/LES.2012.2193660.

[29] T. Bollengier, L. Lagadec, M. Najem, J.-C. Le Lann, P. Guilloux, Soft Timing Closure for Soft Programmable Logic Cores: The ARGen Approach, Springer International Publishing, pp. 93–105.

[30] A. Zarrabi, A generic process migration algorithm, Int. J. Distrib. Parallel Syst. 3 (5) (2012) 29.

[31] T. Bollengier, M. Najem, J.C.L. Lann, L. Lagadec, Demo: overlay architectures for heterogeneous fpga cluster management, in: 2016 Conference on Design and Architectures for Signal and Image Processing (DASIP), 2016, pp. 239–240, doi:10.1109/DASIP.2016.7853832.

Mohamad Najem obtained the B.S. and M.S. degrees in electronic and informatics systems from the University of Pierre and Marie Curie, Paris, France, in 2008 and 2012 respectively, and the PhD in Automatic and Microelectronic systems from the University of Montpellier, Montpellier, France, in 2015. He is currently a Post-Doctoral Fellow at the ENSTA-Bretagne engineering school, and Lab-STICC research institute, Brest, France. His current research interests include the monitoring of self-adaptive systems, statistical analysis and the virtualization of FPGAs.

Théotime Bollengier received the engineering degree from ENSEIRB-MATMECA, Bordeaux, France, in 2013. He is currently a PhD student at IRT bcom and at Lab-STICC laboratory (CNRS), ENSTA Bretagne, Brest. His interests include embedded systems, hardware design, reconfigurable computing architectures. His PhD topic is about reconfigurable architecture hardware virtualization.

Jean-Christophe Le Lann teaches embedded system design at ENSTA Bretagne, Brest and is a research member of LabSTICC, a major laboratory in France. During 9 years (1999,2008) he was HW/SW engineer with Thomson R&D France, developping SoC chips in the field of multimedia and video compression. He was also founder and CTO of Modae Technologies, a startup that developped a toolchain for the synthesis at system level. His main domain of interest are compiler design, HLS, and Domain Specific Languages for Embedded Systems, ranging from early behavioral captures to formal verification.

Loïc Lagadec received the Ph.D. degree in computer science from the University of Rennes 1, Rennes, France, in 2000, and the Habilitation degree from the University of Brest, Brest, France, in 2009. He is a Full Professor at the Laboratory-STICC (CNRS), ENSTA Bretagne, Brest, where he is a Research Head of the IT Department. His current research interests include software tools for reconfigurable computing, cyber security, and interpreted languages. Prof. Lagadec was a Guest Editor for several special issues of scientific journals.