Prototyping Dynamic Task Migration on Heterogeneous Reconfigurable Systems

Arief Wicaksana
Univ. Grenoble Alpes, TIMA, CNRS
Grenoble, France
arief.wicaksana@
univ-grenoble-alpes.fr

Alban Bourge
Univ. Grenoble Alpes, TIMA, CNRS
Grenoble, France
alban.bourge@
univ-grenoble-alpes.fr

Olivier Muller
Univ. Grenoble Alpes, TIMA, CNRS
Grenoble, France
olivier.muller@
univ-grenoble-alpes.fr

Arif Sasongko
Institut Teknologi Bandung
Bandung, Indonesia
asasongko@stei.itb.ac.id

Frédéric Rousseau
Univ. Grenoble Alpes, TIMA, CNRS
Grenoble, France
frederic.rousseau@
univ-grenoble-alpes.fr

ABSTRACT

Reconfigurable devices, such as FPGAs, have been known to offer an excellent performance and a high efficiency in computation. Due to their improving capacity and more efficient architecture recently, there are growing interests in using FPGAs as coprocessors in reconfigurable systems. However, FPGAs still lack the support in dynamic scheduling, e.g. to manage multiple tasks or users in a system. Performing runtime task relocation or load distribution is not possible unless the reconfigurable system supports dynamic task migration. Such ability requires the automation of configuration and context management in reconfigurable architecture, which is not available in the existing solutions.

In this paper, we propose a framework for prototyping dynamic task migration between heterogeneous FPGAs. A task running on one FPGA can be suspended and resumed on another FPGA with different architecture. The extraction and restoration of FPGA registers and memory values are possible due to the task-specific extraction mechanism provided by the tasks. The proposed framework exploits a high-performance embedded processor tightly-coupled to an FPGA to automatically manage the configuration and context. It utilizes two popular heterogeneous reconfigurable systems in the implementation, Xilinx Zynq ZC706 and Altera Arria V SoC. Tests are performed using graphical and non-graphical benchmark applications and performance results are presented.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; *System on a chip*; • **Hardware →** *Hardware accelerators*; *Reconfigurable logic applications*;

RSP'17, October 15–20, 2017, Seoul, Republic of Korea

ACM ISBN 978-1-4503-5418-9/17/10…$15.00
https://doi.org/10.1145/3130265.3130316

KEYWORDS

Task Migration, Heterogeneous FPGAs, Task-Specific Extraction

ACM Reference Format:
Arief Wicaksana, Alban Bourge, Olivier Muller, Arif Sasongko, and Frédéric Rousseau. 2017. Prototyping Dynamic Task Migration on Heterogeneous Reconfigurable Systems. In *Proceedings of RSP'17, Seoul, Republic of Korea, October 15–20, 2017,* 7 pages.
https://doi.org/10.1145/3130265.3130316

1 INTRODUCTION

For decades, Field-Programmable Gate Arrays (FPGAs) have been extensively used as prototyping tools for Application-Specific Integrated Circuit (ASIC) development. They allow an early implementation of circuits and due to their reconfigurable properties, FPGAs are suitable for emulating circuits before fabrication. Despite their lower performance compared to ASIC devices, there have always been interests in using FPGAs as coprocessors to accelerate computation. FPGA's attractiveness comes from their programmable characteristic which provides more flexibility in execution. They also offer better performance and low energy consumption compared to most multiprocessors-based systems.

As early as 1960, Estrin [4] demonstrated that a system with multiprocessors coupled to reconfigurable resource provides better performance due to the hardware implementation. Complex and repetitive tasks, which typically occupy substantial resources if launched in a processor, can be assigned on the reconfigurable resource; thus releasing the processor to execute other tasks in parallel. Such concept has been widely adopted in reconfigurable systems which consists of CPUs and FPGAs, from the latest Commercial Off-The-Shelf (COTS) reconfigurable SoCs, e.g. Xilinx Zynq and Altera (now Intel) SoC, to the cutting-edge cloud services, e.g. Microsoft Catapult [14]. However, FPGAs are not conceived to handle dynamic task management, for instance, when there exist multiple tasks or users. As a consequence, standard features in multiprocessors, such as task relocation and resource sharing, are missing from reconfigurable systems. To solve this problem, dynamic task migration should be enabled on FPGAs.

Dynamically migrating hardware tasks between FPGAs will potentially improve the flexibility of reconfigurable systems. For

many years, studies have been conducted on task migration particularly for Multi-Processor System-on-Chip (MPSoC) architectures [1, 3, 16], whereas little attention has been paid on task migration for reconfigurable systems. Lately, the capacity of FPGAs has been increasing considerably and they have become more and more heterogeneous. The difference in size, technology, and architecture between FPGAs increases the complexity of reconfigurable systems. This heterogeneity of FPGAs, in fact, is the most debilitating factor of implementing hardware task migration from one FPGA to another. Several works have been proposed to enable task migration on heterogeneous FPGAs, however the solutions are limited to overlaying the FPGA architecture [10, 12].

This paper addresses the mechanism to support task migration on heterogeneous reconfigurable systems. It consists of interrupting a running task on an FPGA (source) and migrating it to another FPGA (destination) without losing its progress. The information necessary to resume the execution (the context) is extracted from the source FPGA and restored at the destination FPGA. This context extraction mechanism can be integrated into the hardware tasks with little effort due to the task generation using High-Level Synthesis (HLS) tool [2].

In this work, framework for prototyping the dynamic task migration is built using heterogeneous FPGAs. It allows automatic and distributed management of task configuration and context. A high-performance embedded processor is used to manage the FPGA resource as well as to perform control from an operating system. A reconfigurable SoC, which integrates an FPGA and a processor, is chosen due to the rapid communication offered between the integrated elements and the ease of FPGA configuration from the processor. In this work, we use two reconfigurable SoCs with heterogeneous FPGA from different families: Xilinx Zynq and Altera Arria SoC. The communication between SoCs is done through Ethernet network.

We will begin by reviewing some literature related to our work in Section 2. Then, we will introduce the framework as well as our migration protocol in Section 3. The implementation using reconfigurable SoCs and several experimental results will be shown in Section 4. Finally, the conclusion will be drawn in Section 5 with possible future works.

2 RELATED WORKS

Dynamic task migration has been widely implemented in MPSoC architectures [1]. The goal is to provide a more efficient system while maintaining its reliability by migrating task between Processing Elements (PEs). For instance, when a task is in a long waiting state, it can be replaced with another task so the resource can be used more efficiently. In fault-tolerant systems, the dynamic task migration is also necessary to save the task progress in case of system failure [16]. These works have one thing in common: the task migration procedure is implemented in software/operating system.

In reconfigurable systems, dynamic task migration is a feature that still needs to be developed. The ability to perform task migration is not natively supported by FPGAs. A dynamic task allocation solution in a CPU-FPGA architecture was proposed in [9], however, the solution was based on non-preemptive scheduling. Since hardware tasks are executed as state machines and logic operators, they are not made to be interrupted. Migrating a running task from one FPGA to another cannot be done without losing the execution progress. In order to save the current state of the suspended task, the context needs to be extracted from the FPGA.

For years, researchers have been proposing context extraction methods to enable dynamic task migration on FPGAs. According to [6], these methods can be classified into two families: platform-specific and task-specific. In the platform-specific method, the registers and memory values as well as the configuration data is extracted from FPGA configuration port [15]. Since FPGA configuration is specific to the architecture, the task migration can be performed only between identical FPGAs, not to mention the large context size due to the included configuration. The task-specific method allows extracting registers and memory values from FPGAs by instrumenting the task design with additional state machines, connections, etc. This method does not extract the configuration data, resulting in smaller context and platform-independent migration. However it requires significant efforts to inserts the extraction mechanism to the design. In [8], hardware tasks are modified to enable context extraction at certain states (checkpoints) to reduce the overhead and user's effort.

Some works in task migration on heterogeneous FPGAs are built on Coarse-Grained Reconfigurable Array (CGRA) or overlay architecture. Metzner et al. [10] presented an architecture which consists of PEs and interconnection network built on top of Look Up Tables (LUTs) and Flip-Flop (FF) registers. Hardware multithreading is possible by extracting and restoring the values of PE registers, which is also proposed in [12]. The overlay architecture offers more flexibility in task management, however it has a significant drawback in performance. It loses the optimization brought by FPGA tool suite when generating hardware configuration, particularly for placement and routing. Hence, the solution results in a poor overall system performance.

In prior work, Bourge et al. [2] introduced a method that allows inserting extraction mechanism to hardware tasks using High-Level Synthesis (HLS) flow. The respective method is depicted in Figure 1. Different from the method presented in [8], the insertion of extraction mechanism in [2] is done to Hierarchical Task Graph (HTG) inside HLS tool. This method requires very little effort, particularly in selecting the states with the smallest overhead (checkpoints) and inserting the additional scan-chain extraction with the lowest overhead possible. As a result, the hardware tasks generated (in HDL) by the HLS tool integrate the infrastructure for extraction and restoration of FPGA registers and memories contents. The method proposed in [2] is implemented as Checkpoint PinPoint (CP3) plugin in free and open-source HLS tool AUGH [13]. In this work, we use AUGH-generated hardware tasks with the integrated extraction mechanism in our proposed task migration framework.

3 METHODOLOGY

The present paper focuses on enabling dynamic task migration in a system which is constituted by one or more heterogeneous FPGAs. The heterogeneity of the FPGAs includes different architecture, technology, and capacity. The hardware task migration itself consists of saving the task context from the source FPGA and restoring

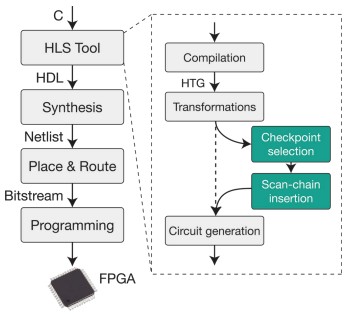

Figure 1: Inserting extraction mechanism to tasks using HLS flow

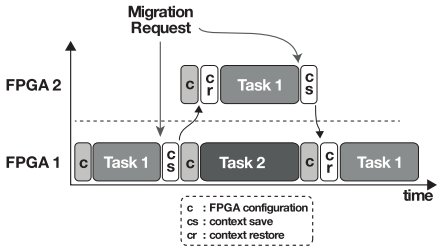

Figure 2: Example scenario of hardware task migration

it to the destination FPGA. The destination FPGA is selected based on the resource availability for the task.

To provide some insights of what we are trying to achieve, we present an example scenario of dynamic task migration illustrated in Figure 2. Two FPGAs with different size and technology are used in the system. FPGA 1 is bigger and has better performance compared to FPGA 2. In the beginning, Task 1 is configured on FPGA 1 to benefit from its architecture. Since our system handles dynamic scheduling, a new task can arrive during runtime. In our case, Task 2 which has higher priority arrives. As Task 2 is more important, we want to use FPGA 1 rather than FPGA 2. A migration request is sent to Task 1 to free FPGA 1. After the context of Task 1 is saved, Task 2 can occupy the resource on FPGA 1. Meanwhile, Task 1 is resumed on FPGA 2 which has sufficient resource for Task 1 while waiting for Task 2 to finish. Task 1 is restored again to FPGA 1 when the resource is available.

To satisfy the explained scenario, the system needs to dynamically manage the tasks between FPGAs. Not only does it need to perform context extraction and restoration, but it also has to manage the configuration and context transfer. The role of a manager in task migration was clearly understood by Morales-Villanueva and Gordon-Ross [11]. They used a soft processor implemented on an FPGA to manage the context. The proposed system requires partial reconfiguration since the processor should be static to control all the process while tasks are being migrated. Meanwhile, Jozwik et al. [6] utilized embedded processor that runs on an FPGA. Another case where a processor is necessary to manage the migration task is also found in [7].

However, controlling the migration flow from inside of FPGAs consumes a considerable amount of resources. A trade-off should be made between the performance of the processor and the resource

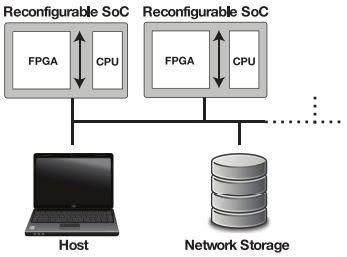

Figure 3: Overview of task migration framework

occupied. Another approach is by using a processor dedicated to dynamic task migration outside of FPGAs. We propose to use reconfigurable SoCs in our framework, which consists of an FPGA and a hard processor. The hard processor in the SoCs will assist the task reconfiguration and context management during task migration. It provides higher performance for the system and it will not occupy the FPGA resource.

3.1 System Overview

In this section, we introduce the framework that enables dynamic task migration between heterogeneous FPGAs. Figure 3 depicts our framework overview. The framework consists of one or more reconfigurable SoCs, with each of them integrating an embedded processor (CPU) and a heterogeneous FPGA. The CPU and the FPGA inside of each reconfigurable SoC communicates using internal high-speed connection, e.g. AXI communication bus. These SoCs are connected to a common Ethernet network with a Host. In our model, the Host assigns hardware tasks to the FPGA accelerators. Note that the scheduling policy is not covered in this work and the Host will implement task migration from arbitrary requests. The detail will be explained in the Implementation Section 4.

In the same network, we also add a Network Storage for two objectives. First, it exists to store the configuration files of each FPGA used in the framework. A set of configuration files are prepared in advance for every FPGA that can run certain tasks. When a task is to be launched on a SoC, the CPU in the same SoC will program the FPGA with the respective configuration. Instead of putting the load on the Host, we propose that each CPU searches the reconfigurable file in the Network Storage for the associated FPGA.

The second objective is to store the task context temporarily when migrating a task from one FPGA to another. With this approach, the Host will decide which FPGAs will perform the migration and which CPUs will store and fetch the context. The task restoration does not have to be done immediately with the context safely stored in a shared storage, thus providing more flexibility in scheduling. The alternative is to transfer the context directly from the source to the destination which requires synchronous transfer between SoCs.

3.2 Task Migration Protocol

The proposed framework implements a task migration protocol between heterogeneous FPGAs. This protocol consists of extracting the task context from one FPGA and restoring it to another FPGA to resume the execution. As our extraction mechanism is based on the task-specific method, the context only includes the registers

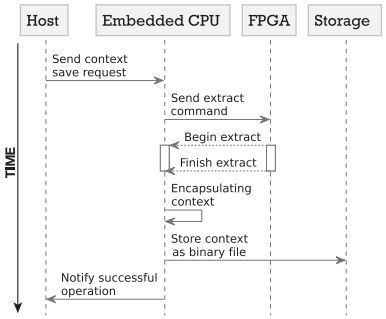

Figure 4: Context Extraction Protocol

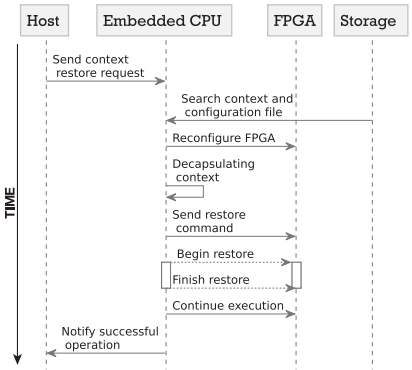

Figure 5: Context Restoration Protocol

and memory values without the FPGA configuration. As a result, the FPGA should be reconfigured before restoring the context. The context extraction and restoration are performed by the CPU from the same SoC as the FPGA and the process is triggered by the Host.

The first part of migration protocol is the context extraction from the source FPGA which is presented in Figure 4. The protocol begins with the Host sending a context save request to the CPU from the same SoC as the source FPGA. The CPU will then send an extract command that interrupts the task on the FPGA and requests the context. The registers and memories values of the task will be extracted by the CPU. To facilitate the destination FPGA in finding the task context, the CPU will encapsulate the context with a short header containing the context size and save it in a binary file with the task identification as the file name. The context file will be stored in the Network Storage. Finally, the CPU will send a notification to the Host with the identification of the task context.

Figure 5 illustrates the context restoration protocol. It begins with the Host sending a restore request to the CPU from the same SoC as the destination FPGA. The request also contains the identification of the extracted task. The CPU then searches the context file based on the identification and its respective configuration from the Network Storage. The FPGA will then be reconfigured by the CPU before the restoration. The context restoration is then triggered by restore command after the header is removed from the context (context decapsulation). After the context is restored, the task will resume its execution on FPGA and a notification will be sent to the Host.

As we have seen in this section, the CPU in reconfigurable SoC holds a very important role in the proposed framework. It controls the extraction and restoration procedure on FPGA, and at the same time, it communicates to the rest of the system. In our framework, the FPGA configuration is also performed by the CPU in reconfigurable SoC as it is facilitated by the internal communication in the SoC.

4 IMPLEMENTATION AND RESULT

This section presents the implementation of the proposed framework. The framework architecture consists of hardware and software built on reconfigurable SoCs. The hardware architecture mainly concerns the hardware tasks and a custom communication IP on the FPGA to manage the execution and the communication with the CPU. The software architecture consists of the drivers and the programs in the CPU to control the task execution and migration.

The proposed framework was built using two COTS reconfigurable SoCs, Xilinx Zynq ZC706 Evaluation Board and Altera Arria V SoC Development Platform. These two SoCs were chosen to represent heterogeneous reconfigurable devices with different architectures: ZC706 Evaluation Board has an FPGA from Zynq-7000 family, whereas Arria V SoC uses an FPGA from Arria family. For the working frequency of both FPGAs, 50 MHz was arbitrarily chosen. Both platforms integrate high-performance embedded processor Dual-Core ARM Cortex-A9. Unlike the former implementation with PowerPC [6], the latest reconfigurable SoCs have a standard high-performance AXI communication bus which does not consume the FPGA resource.

Both SoCs were connected to a Host through Ethernet network with maximum 100 Mbps in data transmission speed (Figure 3). The Host was a PC equipped with a 4-core Intel i3 CPU @ 2.3 GHz, 3 MB L3 Cache and 4 GB RAM. It ran Debian Operating System on Linux 3.16.0 kernel. In our experiment, a private local network was used to avoid interaction with any device outside of the system. And for the same reason, we set up a DHCP server in the Host to minimize the number of devices connected to the network.

We implemented the Network Storage using Network File System (NFS) protocol. A disk partition in the Host was mounted on the network as the NFS disk and it is accessible to all the SoCs connected to the network. The NFS protocol was chosen for the simplicity and rapid integration.

During the framework evaluation, a task migration request was sent arbitrarily from the Host. Using several benchmark programs, both graphical and non-graphical applications, we ran the tasks on FPGAs and migrated them a number of times. The non-graphical benchmarks were IDCT, AES, and GSM from CHStone benchmarks suite [5], while the graphical benchmarks were Sobel Edge Detection filter and Pong game. These programs were compiled in AUGH with CP3 plugin to generate the hardware tasks necessary when using the FPGA accelerator. In the next sections, we will detail the hardware and software architecture of the proposed framework as well as the performed measurements.

4.1 Hardware Architecture

The hardware architecture consists of the hardware task and the custom IPs programmed on the FPGA. It is depicted on Figure 6. In this implementation, we only programmed one hardware task at a given time and as the context is extracted to the outside of

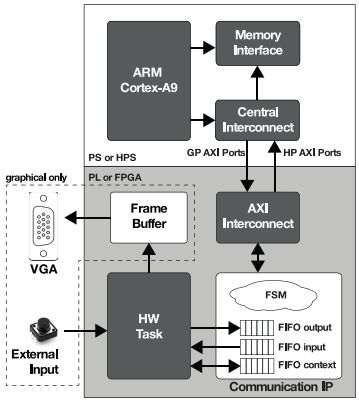

Figure 6: Hardware Architecture of Task Migration Framework

the FPGA, partial reconfiguration is not necessary. In other implementations, the proposed framework is perfectly applicable in a partial reconfigurable system. Figure 6 shows the two parts of the reconfigurable SoC: (1) the part where the CPU, external memory interface, and central interconnect reside; this part is called Processing System (PS) by Xilinx and Hard Processor System (HPS) by Altera, and (2) the part where the reconfigurable logics reside; this part is called Programmable Logic (PL) by Xilinx and FPGA by Altera.

As previously mentioned, we exploited AUGH-generated hardware tasks that offer context extraction mechanism in the design. Using the CP3 plugin of AUGH, a context interface is added automatically to the hardware task for extraction and restoration purpose. This interface uses handshake protocol to perform data transfer. Meanwhile, the execution flow and the communication are controlled by the CPU in the reconfigurable SoC. The CPU and the FPGA are interconnected via AXI communication ports. We dedicated 32-bit High-Performance (HP) AXI Ports for data communication. The data transfers between PL and PS (FPGA and HPS) are done via DMA communication. The CPU manages the communication and the execution flow on the FPGA through 32-bit General Purpose (GP) AXI Ports. To support the communication through AXI bus, we developed a Communication IP which integrates the AXI interface on the PL (or FPGA) side.

The Communication IP supports data transfers between the hardware task and PS (or HPS). It is a register-controlled IP which can receive command from CPU through GP AXI Ports. It can hold the I/O and context data in FIFO buffer while being transferred and performs the conversion between AXI and handshake protocol. The interface of Communication IP which is connected to HP AXI Ports is an AXI master whereas the interface connected to GP AXI Ports is an AXI slave. The Communication IP is also able to manage the I/O data channel between PS (or HPS) and the task. The number of I/O channel in Communication IP depends on the hardware task. In this work, one I/O channel is used by non-graphical benchmarks, whereas the graphical benchmarks do not need the I/O channel since the external inputs and a frame buffer to connect to VGA screen are added.

To provide an idea of the resource overhead caused by the Communication IP, we present its resource utilization in Table 1, both

Table 1: Resource Utilization of Communication IP (Post Place-and-Route)

	ZC706			Arria V SoC	
	LUT	FF	BRAM	ALM	M10K
non-graphic	1092	1147	9	754	34
graphic	750	980	8	548	32

Table 2: Resource Utilization of Hardware Tasks (Post Place-and-Route)

	ZC706				Arria V SoC		
	LUT	FF	BRAM	DSP	ALM	M10K	DSP
non-graphic							
idct	10401	1516	0	0	6810	3	126
aes	3163	1093	2	0	3720	6	3
gsm	4581	1218	0	6	2398	2	8
graphic							
sobel	4713	4932	0	0	3127	0	7
pong	1650	1716	0	0	5692	0	1

for ZC706 and Arria V SoC. Most of the logic utilizations (LUTs) are for the AXI interfaces which take approximately 40% and the rest are for the FSM. Compared to the total FPGA resource in ZC706 and Arria V SoC, the Communication IP only consumes less than 1% in terms of LUTs. The Communication IP for non-graphical benchmarks consumes more LUTs since it requires more complex FSM to manage the I/O transfer flow, whereas the graphical benchmarks manage the I/O directly from external interface. The amount of BRAMs (or M10Ks) used depends on how big are the FIFO buffers allocated to the I/O and context channels. In this work, we allocated 4096×32-bit memories as context channel and 1024×32-bit for I/O channel (for non-graphical version). The resource utilization on ZC706 and Arria V SoC is not comparable since the technology used in both SoCs are different.

Table 2 describes the resource utilization of the hardware tasks on the FPGA of both SoCs. We can see from the table that the Communication IP consumes less LUTs and FFs compared to all the hardware tasks used in our experiments. Even when we compare to a simple application like *pong*, the Communication IP is still smaller, which confirms the efficiency of our implementation.

4.2 Software Architecture

In the software architecture, the ARM processor runs Linux Linaro for its operating system on both SoCs. The kernel version used in Xilinx ZC706 is different from Arria V SoC due to the support provided by each vendor. The software architecture consists of some drivers, functions, and a migration script to manage dynamic task migration. The interaction with NFS protocol is also done in the software architecture.

4.2.1 Drivers. Using C language, we developed some drivers to access the IPs on the FPGA part of SoC. The core algorithms of the drivers are similar for both ZC706 and Arria V SoC. The only difference is the address of AXI interface on both FPGAs. In our framework, there are mainly 3 drivers conceived:

- Write
  We use this driver to send (write) input from CPU to the Communication IP on FPGA via DMA transfers.
- Read
  We use this driver to receive (read) output as the result of the execution from the Communication IP via DMA transfers.
- Command
  This driver is used to manage the execution and task migration from the CPU. Sending a command is done by writing a value to the control register of Communication IP. The command includes start, extract, restore, and continue the execution, which controls the state machine of Communication IP. The CPU can also read the status of the execution by reading the status register.

4.2.2 Functions. Accessing the driver to control the FPGA requires detail parameters, e.g. the memory address, data size, etc. To hide the technical details, we developed some functions written in C which access the drivers:

- Configure
  This function accesses the configuration driver that performs configuration from CPU to the FPGA via Processor Configuration Access Port (PCAP) through the */dev/xdevcfg* port in ZC706 and */dev/fpga0* in Arria V SoC. The configuration driver is supported in Linux by each SoC vendor.
- Start
  The Start function sends a trigger to the hardware task to start the execution by writing a command in the control register of Communication IP.
- Send input
  This function reads input from a binary file and sends it to Communication IP via DMA transfers.
- Receive output
  This function accepts the output trigger from the task, reads the output via DMA transfers, and saves it to a binary file.
- Extract context
  This function sends a command to interrupt the task execution and its context. The context will be transferred to the CPU via DMA and saved as a binary file after encapsulation to allow the destination FPGA to find it later for restoration.
- Restore context
  This function searches the context file from NFS disk, decapsulates it, and sends the context to the Communication IP via DMA transfer. A command to resume the execution is sent to hardware tasks after Communication IP finishes the context restoration.

4.2.3 Migration script. We developed a migration script in bash to send arbitrary migration request from the Host. The scenario, in general, includes an interruption request to a running task and migrating the task to another FPGA. In our test, the script will control the execution and migration of the hardware task between FPGAs by executing related functions.

4.3 Task Migration Time

This section presents the performance evaluation of our proposed task migration framework. The prototyping framework using Xilinx ZC706 and Altera Arria V SoC connected to a screen (for graphical benchmark) is shown in Figure 7. Here, we are interested in measuring the time to perform task migration between the two FPGAs in both SoCs.

Figure 7: Task Migration Framework Built using Reconfigurable SoCs

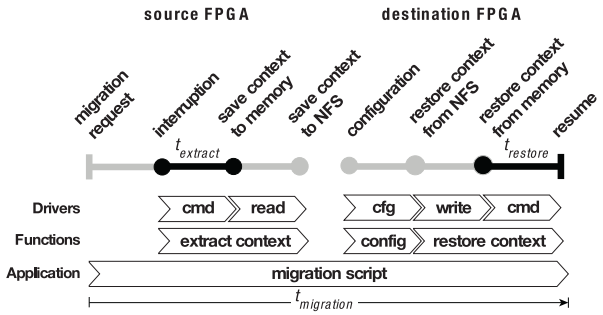

Figure 8: Task Migration Timeline

Figure 8 describes the migration steps in a timeline. The software components related to each step are presented in the figure as well. The entire migration process is controlled by the *Migration script* in the Host. It is started when the function *Extract context* is called to perform the extraction of the task context. In this function, the *Command* driver (cmd) is executed to interrupt the task and start the extraction to the context buffer. The *Read* driver is used to send the context via DMA transfers and the context will be saved in NFS disk. On the destination, the FPGA will be reconfigured for the migrated task. This is done by the function *Configure* (config) with the driver supported by each SoC vendor (cfg). The function *Restore context* calls the *Write* driver to send the context from NFS to the Communication IP. Then, the context restoration will be started using the *Command* driver.

While executing the task migration, the script measures the total migration time ($t_{migration}$). A timer IP was also built specifically to perform cycle-accurate measurements of extraction ($t_{extract}$) and restoration ($t_{restore}$) on FPGA. We launched the migration script 1000 times and the average of migration time for each benchmark applications are described in Table 3. As we can see, the sum of $t_{extract}$ and $t_{restore}$ is negligible compared to $t_{migration}$. At 50 MHz as the FPGA working frequency, the time ratio obtained is around 1:10000. This result shows the high efficiency of the extraction mechanism inserted to the hardware task by CP3 plugin. However, the mechanism does not extract the task configuration like in the platform-specific method. It means there exists non-negligible reconfiguration overhead. The configuration file used in the experiment is approximately 13 MB for ZC706 and 23 MB for

Table 3: Task Migration Time Measurements

	$t_{extract}(cycles)$	$t_{restore}(cycles)$	$t_{migration}(s)$
non-graphic			
idct	471	373	1.88
aes	1517	1511	1.87
gsm	568	550	1.89
graphic			
sobel	114	109	1.87
pong	2765	2757	1.86

Arria V SoC. If we take maximum bandwidth of PCAP (400 MB/s), the reconfiguration will take around 50 ms, which is 500× slower than the extraction and restoration.

For the latency due to the file transfer, we need to consider the transfer of the reconfiguration and context files via NFS disk. We obtained the transfer time for these files of approximately 1.2 seconds for every benchmark program which is significant compared to the total migration time. This is due to the asynchronous transfer using NFS protocol and the Host must observe the disk before triggering the restoration to ensure the context is valid. This can be improved by implementing direct transfer from the migration source and to the destination. A faster Ethernet cable can be used as well to improve the transmission speed. In this work, several technical implementations are chosen for rapid integration so they might not give the best performance. Nevertheless, we have shown the advantage of task migration framework without overlay architecture. Although it is hard to perform a fair comparison, the latest architecture in [12] has 20-30× lower frequency with much more significant resource utilization. We do not have the chance to present a comparison with non-overlay solution since very few literature discusses end-to-end task migration on heterogeneous reconfigurable systems.

As we have presented throughout this paper, the proposed framework is built as generic as possible so that it can be implemented in different reconfigurable SoCs. The Communication IP is written in standard HDL and can be targetted to any FPGA architecture. The software architecture is made to work on any Linux distribution. Some adjustment at the Drivers is necessary to adapt to AXI interface addresses on a different FPGA. Note that we built the framework using SoC that uses AXI interconnection between the CPU and the FPGA. To use the framework in other type of reconfigurable systems, such as CPU-FPGA hybrid with PCIe connection, extensive modifications in Communication IP and the Drivers are expected.

5 CONCLUSION

In this paper, we introduced a complete framework that enables dynamic task migration on heterogeneous reconfigurable systems. A running task on an FPGA can be interrupted and resumed on another FPGA without losing the execution progress. We proposed a generic mechanism that can work on heterogeneous FPGAs with different capacity, technology, and architecture. Using a high-performance embedded processor, the configuration and context can be automatically managed during the migration process. Hardware tasks used in this framework integrate low overhead extraction mechanism inserted automatically using HLS tool AUGH and CP3 plugin. To the knowledge of the authors, this work is the first that demonstrates end-to-end dynamic task migration on heterogeneous FPGAs without using overlay technique.

Despite the encouraging results obtained in this work, there are still issues which have not been solved yet, e.g. in communication. If there exists frequent data transfer during computation, e.g. in a streaming-based task, the communication will be interrupted due to the task migration and there is no guarantee of the system maintaining data integrity. This lack of communication management in reconfigurable systems limits the number of tasks that can be migrated. In future works, we will focus on providing the task migration in reconfigurable systems while maintaining the communication consistency.

REFERENCES

[1] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. 2006. Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study. In *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*. European Design and Automation Association, 15–20.

[2] Alban Bourge, Olivier Muller, and Frédéric Rousseau. 2016. Generating Efficient Context-Switch Capable Circuits through Autonomous Design Flow. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 10, 1 (2016), 9.

[3] Ashaf El-Antably, Olivier Gruber, Frederic Rousseau, and Nicolas Fournel. 2015. Transparent and Portable Agent Based Task Migration for Data-Flow Applications on Multi-Tiled Architectures. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2015 International Conference on*. IEEE, 183–192.

[4] Gerald Estrin. 1960. Organization of Computer Systems: The Fixed Plus Variable Structure Computer. In *Papers presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*. ACM, 33–40.

[5] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. 2008. CHStone: a Benchmark Program Suite for Practical C-based High-Level Synthesis. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*. IEEE, 1192–1195.

[6] Krzysztof Jozwik, Hiroyuki Tomiyama, Masato Edahiro, Shinya Honda, and Hiroaki Takada. 2012. Comparison of Preemption Schemes for Partially Reconfigurable FPGAs. *IEEE Embedded Systems Letters* 4, 2 (2012), 45–48.

[7] Oliver Knodel, Paul R Genssler, and Rainer G Spallek. 2017. Migration of Long-Running Tasks between Reconfigurable Resources using Virtualization. *ACM SIGARCH Computer Architecture News* 44, 4 (2017), 56–61.

[8] Dirk Koch, Christian Haubelt, and Jürgen Teich. 2007. Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays (FPGA)*. ACM, 188–196.

[9] Roman Lysecky, Greg Stitt, and Frank Vahid. 2006. Warp Processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 11, 3 (2006), 659–681.

[10] Michael Metzner, Jesus A Lizarraga, and Christophe Bobda. 2015. Architecture Virtualization for Run-Time Hardware Multithreading on Field Programmable Gate Arrays. In *International Symposium on Applied Reconfigurable Computing*. Springer, 167–178.

[11] Aurelio Morales-Villanueva and Ann Gordon-Ross. 2013. On-Chip Context Save and Restore of Hardware Tasks on Partially Reconfigurable FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. IEEE, 61–64.

[12] Mohamad Najem, Théotime Bollengier, Jean-Christophe Le Lann, and Loïc Lagadec. 2017. Extended Overlay Architectures For Heterogeneous FPGA Cluster Management. *Journal of Systems Architecture* (2017).

[13] Adrien Prost-Boucle, Olivier Muller, and Frédéric Rousseau. 2014. Fast and Standalone Design Space Exploration for High-Level Synthesis under Resource Constraints. *Journal of Systems Architecture* 60, 1 (2014), 79–93.

[14] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, and others. 2015. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro* 35, 3 (2015), 10–22.

[15] Harald Simmler, L Levinson, and Reinhard Männer. 2000. Multitasking on FPGA Coprocessors. In *International Workshop on Field Programmable Logic and Applications (FPL)*. Springer, 121–130.

[16] Kamel Smiri, Safa Bekri, and Habib Smei. 2016. Fault-Tolerant in Embedded Systems (MPSoC): Performance Estimation and Dynamic Migration Tasks. In *Design & Test Symposium (IDT), 2016 11th International*. IEEE, 1–6.