Methods and Mechanisms for Hardware Multitasking: Executing and Synchronizing Fully Relocatable Hardware Tasks in Xilinx FPGAs

Xabier Iturbe*†, Khaled Benkrid*, Tughrul Arslan*, Raul Torrego† and Imanol Martinez†
*System Level Integration Group, The University of Edinburgh, Edinburgh EH9 3JL, Scotland, UK
{x.iturbe, k.benkrid, t.arslan}@ed.ac.uk
†Embedded System-on-Chip Group, IKERLAN-IK4 Research Alliance, Mondragón 20500, Basque Country (Spain)
{x.iturbe, rtorrego, imartinez}@ikerlan.es

Abstract—This paper presents the details of a novel technique which allows for the implementation and execution of completely relocatable hardware tasks onto dynamically reconfigurable FPGAs. Our novel technique harnesses the internal configuration access port (ICAP) for inter-task communication and synchronization, leading to very small logic overheads. The advantages of this technique include fault-tolerance, as tasks could be relocated freely on the fabric to circumvent damaged resources, and high performance, due to better exploitation of the logic fabric. We provide an proof-of-concept implementation, namely of cognitive Software Defined Radio (SDR), on a Xilinx Virtex-4 FPGA. The work is part of a larger effort in our group which aims to build a fully operational dynamically reconfigurable computer which would satisfy the often conflicting requirements of high performance, fault-tolerance and high level programming.

I. INTRODUCTION

In the late 90’s, Gordon J. Brebner proposed for the first time the possibility of executing swappable hardware tasks onto Field-Programmable Gate Arrays (FPGAs) [1]. He stated that hardware tasks could be allocated, executed and then replaced by other hardware tasks, leading to a continuous stream of input operands, computation and output results. Over the last decade, Brebner’s original idea has gathered a lot of interest in the research community with many achievements reported in the literature. For instance, it is now possible to relocate partial bitstreams from one position to another within a device’s silicon substrate on-the-fly [2], or even suspend hardware tasks and restart them at another time [3]. However, there are still several challenges to meet for the proliferation of the new hardware tasks based computation. Two of these challenges are inter-task communications and synchronization; e.g. how to deliver input operands and retrieve output results to/from hardware tasks which can be placed everywhere on the device? and, how to ensure that these tasks cooperate together safely to complete a particular computation?

In this paper we deal with these two problems. We propose a novel reconfigurable computing method which covers not only hardware task relocation and inter-task communications, but also synchronization aspects. Our method has been tested on Xilinx Virtex-4 FPGAs and could also be applied to Virtex-5 and Virtex-6 families.

The main contributions of this paper are:
1) A novel wrapper to be attached to hardware modules that transforms them into easy-to-use relocatable hardware tasks with software-like interface (Section III);
2) A novel ICAP-based Inter-task Communication Infrastructure (2CI), which makes hardware tasks accessible regardless of their placement on the device (Section III.A), and,
3) The definition and implementation of a Hardware Semaphore (HWS), which permits to synchronize the execution of the hardware tasks (Section III.B).

The remainder of this paper is organized as follows. An introduction to Xilinx dynamic partial reconfiguration technology is provided in Section II. Then, in Section III we describe our reconfigurable computing method, including our inter-task communication and synchronization mechanisms. Finally, concluding remarks and plans for future work are drawn in Section IV.

II. XILINX DYNAMIC PARTIAL RECONFIGURATION TECHNOLOGY

An FPGA can be modeled as an architecture with two layers: the functional layer, which contains the physical resources used to perform the computation, and the configuration layer, which controls the configuration of the functional layer.

A modern Xilinx FPGA includes several types of resources (e.g. CLBs, DPS48s, IOBs, BRAMs) in its functional layer (See Fig. 1). CLBs are the most abundant and main logic resources. They are organized in a regular array, referred to as the “Sandbox” by some authors when targeting the high-capacity LX family devices [4]. In Virtex-4 FPGAs each CLB is divided into four interconnected slices; and each of these slices includes two 4-input Look-Up Tables (LUTs). The LUTs are used to provide logic, arithmetic, data storage and data shifting functions. In this context, the DSP48s and BRAMs can be seen as the heterogeneous resources embedded in the middle of the homogeneous sandbox.

The functionality assigned to the aforementioned resources as well as their interconnection is defined by a configuration bitstream. Hence, when a hardware task is synthesized, a
partial bitstream is obtained which configures the resources included within a rectangular region of the FPGA. The bitstream is stored into the FPGA’s configuration memory [5], which makes up the configuration layer.

The configuration memory is organized in several configuration frames of 41 words of 32 bits each. The address of the frames is related to the position of the physical resources they configure (See Fig. 2). The device is divided into two halves, top and bottom and each of these parts is also divided into clock regions or rows. In Virtex-4 FPGAs, a clock region includes all the resources within a rectangular region delimited by the whole width of the device and up to 16 CLBs, 4 BRAMs, 32 IOBs or 4 DSP48s height (See Fig. 1). The frames are also distinguished by the type of resources they configure. In fact, each column within a row is configured by a different number of frames according to the type of resources it includes. CLB columns are configured by 22 frames, DSP48 columns require 21 frames and IOB columns need 30 frames. All these resources share the same resource block type identifier. However, BRAM configuration data consist of two separate block types. The first block contains 20 frames of routing information and the second block contains 64 frames of BRAM data. Every frame is thus addressed by a 32-bit address which includes five values referred to the resources it configures: (a) block resource type, (b) top / bottom half, (c) row, (d) major column address, which identifies the column within the row, and (e) minor intra-column address, which identifies the specific frame within the column.

A. Dynamic Partial Reconfiguration

The use of SRAM technology in the configuration memory of Xilinx FPGAs permits to download new configuration data at runtime. This process is called Dynamic Partial runtime Reconfiguration (DPR) and makes it possible to change the functionality assigned to a specific region of a device while the rest of the device is still operating [6].

Modern Xilinx FPGAs ease the access to the configuration memory by including an Internal Configuration Access Port (ICAP), which removes the need for external components that control the reconfiguration process. The ICAP included in Xilinx Virtex-4 FPGAs has a 32 bits data bus width and runs at 100 MHz, allowing a maximum theoretical reconfiguration bandwidth of 400 MB/s [5].

The smallest amount of configuration information that can be accessed in the configuration memory is a frame. Hence, the minimum chip’s logic area modifiable when using DPR is a whole fabric clock region spanning column. Consequently, by using DPR any given piece of circuitry allocated in a dedicated clock region can be individually replaced without interrupting the operation of the rest of the system in a similar way to software tasks are switched in a Von Neumann processor. Following this analogy we call such circuits hardware tasks.

As shown in Fig. 3, a battery of different hardware tasks can be allocated at different times in the same Partially Reconfigurable Region (PRR). On the other hand, the functionality that must remain active at all times (e.g. the reconfiguration controller, which loads partial bitstreams through the ICAP interface) is implemented in the Static Region (SR).

With the objective of making all of the hardware tasks to be placed in the same PRR pin compatible with each other, Xilinx states that all their connections must pass through Bus Macros (BMs). This does not apply to the clock signal, which is delivered to the resources through a dedicated net. BMs are pre-routed CLB-based bridges between the two sides of the PRR, which provide a means of locking the routing and which must be mapped to specific positions at design phase [6]. However, the use of fixed communication interfaces leads to fixed boundaries for the PRR. That is, despite the fact that different hardware tasks can be configured in the PRR on-the-fly, its boundaries are fixed because the position of the BMs is fixed. In the new partition-based partial reconfiguration flow introduced by Xilinx, BMs are replaced by proxy logic that is automatically set by Xilinx design tools [7]. Therefore, the same restrictions apply. As a result, the PRR is turned into
a set of reconfigurable slots whose size and shape are set by the user at design time. However, as this size must be chosen based on the largest task, the rest of the tasks are unnecessarily enlarged within the slot. Consequently, slot-based reconfiguration results in inefficient resource exploitation. Besides this, the number of tasks which can run concurrently is limited by the amount of reconfigurable slots included in the system.

Xilinx reconfiguration flow establishes that the static design and each hardware task must be implemented separately and defines a final merge step to generate the full and partial configuration bitstreams. This step is intended to deal with the static signals that go through the PRR. The partial bitstreams of the hardware tasks are generated in such a way that they use the same routing resources for crossing static signals. The persistency in the configuration data ensures the integrity of these signals even while the hardware tasks are being reconfigured. The main problem of this flow is that generated partial bitstreams are location specific. In contrast, software tasks are fully relocatable as their code can be executed at any address in the program memory.

B. Partial Bitstream Relocation: Slotless Reconfiguration

In order to overcome the aforementioned constraint, and make the hardware tasks location-independent, relocatable bitstreams have been proposed. Based in the relation between the configuration memory map and the configured physical resources, a partial bitstream can be relocated to any arbitrary position within the FPGA by simply “shifting” its configuration data within this memory.

There are only two conditions to meet. First, the target position must be identical to the original position in the type and arrangement of resources as well as communication interfaces. Second, the static signals crossing the target position must be preserved. Unfortunately, Xilinx tools do not support relocation yet. This means that, the merge between static routes and relocated partial bitstreams must be made manually when downloading the bitstream to the target position (See Fig. 4). In the worst-case a task relocation could be prevented if the task needs to use the same routing resources assigned to the crossing static signals in the target position. This is a real problem because it is impossible to direct Xilinx router not to use the routing resources within the PRR.

![Fig. 4: Overview of partial bitstream relocation](image)

In this context, the PRR becomes a nearly homogeneous resource (only constrained by the resource arrangement) to be shared by multiple hardware tasks. The tasks are separately synthesized to fit the exact amount of necessary resources and thus, inter-task boundaries within the PRR can be modified at runtime. Therefore, the number of concurrent tasks is limited only by the amount of resources and BM-based interfaces included in the PRR and not by the amount of reconfigurable slots defined at design phase. However, the problematic online routing capability is necessary for keeping the relocated tasks linked to the fixed communicating interfaces.

In [9] the authors explain the basis for dealing with the heterogeneous resources incorporated in the last generation of FPGAs. They succeeded to allocate a hardware task in a target position where there were different types of resources from the region for which it had been synthesized. However, communication interfaces continue to be based on the inflexible BMs.

In the way of building fully relocatable hardware tasks, we present a new paradigm for reconfigurable systems which replaces position-dependant BM physical components with an ubiquitous virtual interface.

III. OUR DPR COMPUTING APPROACH

We envision a software-driven multitasking scenario, which supports the benefits brought about by reconfigurable hardware without significantly modifying the traditional programming style. These benefits include: (i) true hardware multitasking and (ii) on-demand hardware-based computation. The system itself is able to schedule its own workload and manage its own resources on behalf of the user application and thus, the FPGA acts as a shared computational resource which serves multiple on-demand tasks.

In our approach, hardware tasks are managed as if they were processes or threads in a traditional software multitasking operating system because we think that preserving the software programming style is, after decades of prevalence, the only way to guarantee the success of any new computer platform. Hence, the execution of the reconfigurable application is driven by a software program, which is executed in the main system processor. The program includes task definitions but the body of these tasks is defined in hardware. The user can rely in a hardware abstraction API when programming his/her application. This API offers support for hardware task invocation (e.g. similar to POSIX `pthread_create()` or `fork()`), synchronization (e.g. similar to POSIX `pthread_join()` or `sem_wait()`) and inter-task communication (e.g. equivalent to POSIX `msgsnd()` and `msgrcv()`). Therefore, the user only needs to define the execution flow of the application, specifying the triggering conditions for each task.

This Section describes the methods and mechanisms we have developed to support our reconfigurable computing approach.

A. The Hardware Task and its Communication Interface

In order to promote the concurrency and improve the allocatability of the hardware tasks within the FPGA, we propose a Communication Interface (CIF) to be attached to each hardware module which performs the computation. This module can be designed by the user or it can refer to a third-party IP-core. The merger of a CIF and a “pure
hardware” module gives rise to a relocatable hardware task with a “software look and feel”, which is the basic unit of computation in our model.

The CIF manages the interface of the hardware module by delivering the input operands and receiving the output results computed by it. As a result, the hardware task is a closed computing structure where the clock is the only signal crossing its boundaries. This permits to circumvent the need to use BMs for the interfaces and thus, the allocatability of the tasks is not constrained by the reconfiguration method anymore. Moreover, the number of tasks that can be concurrently executed does not depend on the number of communication interfaces defined at design time.

The CIF includes several elements:

- **Input and Output Data Buffers**: These elements make up the interface of the hardware tasks, as they exchange data through them. The input data is passed to the hardware tasks by writing in the positions of the configuration memory where the physical resources of the input buffers are mapped, while output results are read from the positions where the physical resources of the output buffers are mapped (See Fig. 6). Since the tasks are relocatable, the system needs to keep track of the position of each running task within the FPGA to know which configuration frames must be accessed.

- **Glue Logic**: It is added for adapting both the input and output data from the way it is stored in the buffers to the needs of the hardware module and vice versa (e.g. data rate, bit length, etc.).

- **Synchronization Hardware Semaphore (HWS)**: It is used for synchronizing the execution of the hardware tasks.

We distinguish between two different types of hardware tasks: (i) **Data-stream processing tasks**, to be used in data-intensive applications with regular dependencies, and (ii) **Hardware-acceleration tasks**, which speed-up the execution of portions of computationally intensive software code. The latter are characterized by less inter-task communication demands.

The implementation of the CIF is different for each type of task (See Fig. 5). While the CIF of a data-stream processing task is suited to handle a hardware module which operates in a pipelined fashion, the CIF of a hardware accelerator task is designed to deal with random accesses to different positions in the buffers. In the former CIF implementation, both `wr_addr` and `wr_en` signals are equal to `rd_addr` and `rd_en` signals, respectively, albeit delayed by `N` clock cycles where `N` can be explicitly configured by the user and is equal to the depth of the pipeline. Since access to the data is sequential, a counter is used to generate the addresses. In this scenario, `rd_en`, `wr_en` signals as well as the clock enable of the hardware module (`ce`) are permanently active while the computation takes place, and turn inactive when the last position of the output buffer is written. This is not the case for the CIF of a hardware accelerator. Its implementation is much more open, conceptually limited to the connection of data, address and control ports of the hardware module to the buffers. A special case is when the module includes a processor and the buffers are directly mapped to its program memory or to a separate data memory (Harvard architecture), which acts as a shared memory. Then the communication interface becomes completely transparent, removing the need for attaching a CIF.

The buffers of a data-stream processing task are similar to the FIFO queues commonly inserted in-between pipeline stages, and the buffers of a hardware accelerator act as local caches. The use of high density storage yet location-specific resources (such as BRAMs) is more suitable for implementing the buffers of either data-stream processing tasks or of hardware accelerators which process a high volume of data or whose software interface is implemented as shared memory. On the other hand, it might be preferable to use low density storage resources (LUT-RAM/ROM) for implementing the buffers of hardware accelerators with lower communication demands. In this case, LUT-ROMs, whose initial values are written through the ICAP, are used in the input buffers and LUT-RAMs, whose final values are read through the ICAP, are used in the output buffers.

In order to achieve the highest efficiency in BRAM-based buffers, the input and output buffers are organized based on the way the memories are mapped in the configuration memory. As shown in Fig. 6b, the buffers are distributed along the four BRAMs in the same column, in consecutive content frames. Moreover, each of the buffers is accessed through each port of the dual-port BRAMs. In total, a CIF requires 4 BRAMs, 22 LUTs and (42 + `N`) Flip-Flops.

In order to enable high data rates in LUT-based buffers (e.g. an input data reading / output data writing per clock cycle), each bit of the same data is mapped to a different LUT (See Fig. 7b). Otherwise the data should be read bit by
A. BRAM-based buffer address map

(b) Buffer implementation using BRAMs

Fig. 6: BRAM-based buffer

An inter-task communication service is central to any operating system as the tasks do not work in isolation from each other. Software operating systems include many options for exchanging data between processes (e.g. Pipes, Message Queues, etc.)

In this Section we describe a low-level hardware-oriented communication mechanism upon which some of the aforementioned high-level software communication services could be built. Data is transmitted from a sender hardware task \( A \) to a receiver task \( B \) by simply copying the content of the output buffer of \( A \) to the input buffer of \( B \) (See Fig. 8). In other words, data is relocated from task to task [10]. Hence, while the hardware tasks perform the computation in the functional layer of the FPGA, data transactions are carried out through the configuration layer.

As tasks’ buffers are accessed through the ICAP, we name our solution ICAP-based Inter-task Communication Infrastructure (I2CI). I2CI is a ubiquitous communication solution, as it enables access to the buffers regardless of task position within the FPGA. The main advantages of I2CI include: (i) it does not constrain the allocatability of the tasks, (ii) it introduces no area overheads. Additionally, I2CI permits to advance towards the desirable objective of building a PRR free of crossing static signals, where task relocation would be always feasible.

Despite the fact that BRAM-based buffers lead to a more efficient exploitation of I2CI, with lower information overheads in the configuration frames, the way their memory cells are mapped in the content frames is tricky. Therefore, a coder/decoder is needed for adapting the way raw data is stored in the frames to the format used by the software and vice versa. Both the coder and the decoder consist in four BRAMs each, with direct access to the main processor and located in a fixed position. Data to be passed to the hardware tasks is written in the coder-BRAMs, in the way shown in Fig. 6. Then, the content of these memories is read-back from the configuration layer and copied to the frames where the receiving tasks’ buffers are mapped. Alternatively, the output results of sender hardware tasks are read-back from their output buffers through the configuration layer and written in the content frames of decoder’s BRAMs. Finally, these memories are accessed by the processor in the functional layer.

On the other hand, LUT-RAM/ROM data bits are consecutively stored in the CLB frame. However, as mentioned before, we do not store one data per LUT, instead we store each bit of the same data in a different LUT in order to increase access throughput. Therefore, a data format adaptation is needed. This adaptation can be seen as a vertical to horizontal transformation where the vertical index is the data index within the buffer and the horizontal index is the position of each bit within the data (See Fig. 7a and 7b). In order to make the communication interface with the main processor simpler, we reuse the aforementioned coder-decoder. The BRAMs are used as a temporary storage resource where the processor writes data to or reads data from. Hence, the coder puts together the
same position bit of up to 16 data sets to be written in each LUT. Analogously, the decoder stores the LUT content readback through the ICAP in the suitable order, the 16 LUT bits are written in 16 consecutive memory addresses.

![Fig. 8: Inter-task communications: Data relocation](image)

In order to circumvent the most important communication related problems in reconfigurable computing (e.g. deadlocks and race conditions), computation is carried out atomically; Data exchanges among tasks are performed only prior to task execution start and after computation has finished. In any case, the desired latency of inter-task communications can be set by appropriately choosing the granularity of the atomic computation.

The major limitation of I2CI is that it is unable to conduct concurrent communications, even if several hardware tasks are running in parallel on the chip. Hence, while I2CI can be compared in performance with a traditional bus, it clearly achieves lower bandwidth compared to other approaches with support for concurrent communications (e.g. Network-on-Chip). In any case, the achievable bandwidth is, by no means negligible, being in the range of hundreds of MB/s [11].

C. Hardware Semaphore: Synchronizing the Hardware Tasks

In order to synchronize the computation and communication, each of them occurring in a different layer of the FPGA, a Hardware-oriented Semaphore (HWS) is used.

Our HWS is implemented on a single LUT-RAM as shown in Fig. 9a. On its part, Fig. 9b shows its functioning. The value of State bit is '0' when the task is to start computing or when it has already finished computation and '1' while it is performing active computation. It is responsibility of I2CI to set State value to '1' to start the computation (through the configuration layer) once the whole input data stream has been copied to the input buffer. To speed-up this process the HWS is placed in the bottom-CLB of the last input buffer’s CLB-column. Hence, the State bit is used as the reset signal for all the circuitry in the CIF as well as the hardware module. On its part, the hardware task automatically returns State value to '0' when the output results are ready to be retrieved from its output buffer. In this way, polling can be done on the HWS of the hardware tasks when the execution time of the tasks is not known or cannot be predicted with precision.

![Fig. 9: HWS implementation (a) and functioning (b)](image)

IV. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel technique that allows for the efficient execution of completely relocatable tasks onto Xilinx partially and dynamically reconfigurable FPGAs. We have described two types of wrappers to be attached to hardware tasks; one of them is suitable to be used with high bandwidth communicating tasks while the other is oriented to low bandwidth yet computationally intensive tasks. The wrappers enable the use of the configuration layer of the FPGA for communication purposes, circumventing the need to use fixed communication interfaces which traditionally have constrained the allocatability of the hardware tasks and the exploitation of multitasking capabilities. Finally, we have presented a novel mechanism for synchronizing the execution of hardware tasks.

Future work includes testing our technology on several real world applications with benefits expected in fault tolerance, real-time, high performance and low power computing.

REFERENCES