Scuola Superiore Sant'Anna

Towards the heterogeneous, real-time reconfigurable embedded system

A dissertation submitted for the degree of

Doctor of Philosophy

by

Enrico Rossi

Supervisor: Professor Giorgio Buttazzo

Tutor: Mauro Marinoni

March 2017

Towards the heterogeneous, real-time reconfigurable embedded system

Enrico Rossi

Acknowledgements

Good. The last page, last but not least. With these words I wrote my master thesis acknowledgments and I would like starting these in the same way, because this is indeed the most important page.

Many things have changed since I finished my master thesis, really a lot, maybe... too many. Perhaps, though everything happened led me to be what I am today (and I'm happy with it), it would have been better if something had remained as it was.

Despite difficulties such a long time passed so quickly and, for this reason I would like to thank those many people that helped me getting over them, making them less hard. I will thank everyone, trying not to forget anyone, and it will no longer matter who I will mention first, everyone has been important, none less than others.

I would like to thank Luca and Federico. Among Federico's daily moaning and Luca's past-new-year's-eve-cotechino: we built a deep friendship. Between ups and downs, they always helped me and I know they will continue to support me. They were quite patient...I have to admit it! In addition to that, they taught me the importance of verticalising the work on horizontal lines of project which leads to greater efficiency that transversely affects all the working areas. Thanks to Federica and her daily attempts to make me date some of her friends (I would not say really successfully, so far).

Thanks to my new friends, Fabio, Cecche, Barsi, Peda, Chiarina, Orso, Leo, Ali, Caro, Cri, Fede, Pippo and Vale for dinners, holidays on the rocks and roast potatoes (which I just tasted!), for the mid-August pool-parties and barbecues and for beach-volley matches. Thanks for wine, laughs, but especially for raising my humor when I was down. With your positive attitude and your company you helped me moving forward, you made me smile even when I did not think I was able to. I hope I returned you at least part of the laughs you made me do.

Thanks to Ale, Tizi, Lore, Emi, Ste, Renzo and Leo for the nights spent together, the reunions around Europe, the homecomings, the laughs and friendship that we have been carrying out for almost 15 years. Despite we are in different times of our life, we always find ways to feed our friendship. I hope each of you, my friends, though sometimes feel drowning in the sea of problems, does not loose the strength to smile because “we do not have to wait for happiness to smile, but smile to make our own happiness”.

Thanks to Miche who has always been able to listen, always having the right words and being silent when it was needed. He is one of those rare people who actually knows how to help the others. Thank you.

Thanks to Benve, for the friendship and the evenings together that often helped me to see a little of light (and a good cocktail Martini) besides the dark moments.

If, as Aristotle said, ”the antidote against fifty enemies is a friend”, I am ready to face fifty thousand enemies.

Finally, I know I am contradicting myself but a special thank goes to my family, my Father, my Mother and my Sister, who always supported me, taught and encouraged me. Thanks to my Aunts Lidia, Rosa and Uncle Maurizio for the chats, the pleasant meetings, the warmth and the good wine. Eventually, I would like to thank my Grandmother for being the wisest, best person I have ever known.

Good. I reached the end of this experience and it is also your success if I have been able to get here. I wanted to write these acknowledgements because each of you, more or less consciously, has helped me to become what I am today.

So, thank you. Without you this thesis and this person would not have existed.

Ringraziamenti

Bene. L'ultima pagina, last but not least. Ho cominciato così i ringraziamenti della mia tesi magistrale e comincerò anche questi nello stesso modo, perché questa è davvero la pagina più importante.

Sono cambiate molte cose dall'ultima Discussione, davvero tante, forse... troppe. Forse, anche se gli avvenimenti mi hanno portato ad essere quel che sono oggi (e ne sono contento), sarebbe stato meglio se qualcosa fosse rimasto com'era.

Questi tre anni sono passati velocemente nonostante siano stati difficili. Devo ringraziare molte persone per averli resi meno pesanti... più vivibili. Ringrazierò tutti, cercando di non scordare nessuno, e non avrà più importanza chi menzionerò per primo, ognuno è stato importante, nessuno meno di altri.

Voglio ringraziare Luca e Federico con cui ho stretto una grande amicizia. Tra i giornalieri, mille lamenti di Federico ed i cotechini-del-capodanno-passato di Luca abbiamo messo in piedi un bella amicizia. Tra alti e bassi, mi hanno sempre aiutato e sono convinto che continueranno a farlo. Sono stati piuttosto pazienti... devo ammetterlo! Inoltre, non meno importante, mi hanno insegnato che verticalizzare il lavoro sulle linee di progetto orizzontali porta ad avere un'efficienza maggiore che trasversalmente influisce su tutti gli ambiti aziendali. Grazie anche a Federica ed ai suoi continui tentativi di appiopparmi qualche amica (per ora, non direi proprio riusciti).

Grazie ai miei nuovi amici, Fabio, Cecche, Barsi, Peda, Chiarina, Orso, Leo, Ali, Caro, Cri, Fede, Pippo e Vale per le cene, le vacanze sugli scogli appuntiti e le patate arrosto (che ho solo assaggiato!), per i ferragosto in piscina e le braciate, le partite a beach-volley, per il vino e le risate, ma sopratutto per avermi aiutato e risollevato il morale quando non era giornata. Con il vostro spirito positivo e la vostra compagnia mi avete aiutato ad andare avanti, mi avete fatto sorridere anche quando non pensavo

di averne la capacità. Spero di avervi restituito almeno una parte delle risate che mi avete regalato.

Voglio ringraziare anche Ale, Tizi, Lore, Emi, Ste, Renzo e Leo per le uscite, le rimpatriate in giro per l'Europa, la compagnia, le risate e l'amicizia che portiamo avanti da quasi 15 anni. Pur essendo ognuno in momenti diversi della propria vita, troviamo sempre il modo di coltivare l'amicizia. Spero che ognuno di voi, amici miei, anche se sommerso dalle difficoltà e dai problemi, non perda la capacità di sorridere perché "non bisogna aspettare di essere felici per sorridere, ma sorridere per essere felici".

Grazie a Miche in cui trovo sempre una persona capace di ascoltare, che ha sempre le parole giuste e conosce il valore del silenzio. È una delle poche persone che conosco che sappia aiutare l'altro. Grazie.

Grazie a Benve, per l'amicizia e le serate in compagnia che mi hanno spesso aiutato a vedere un po' di luce (ed un buon cocktail Martini) a lato dei momenti bui.

Se, come diceva Aristotele, "l'antidoto contro cinquanta nemici è un amico", io sono pronto ad affrontarne cinquantamila.

Adesso, in effetti, mi contraddirò, perché un grazie particolare va alla mia famiglia, Babbo, Mamma ed Ila, che mi hanno sempre supportato, insegnato ed incoraggiato. Grazie ai miei Zii Lidia, Rosa e Maurizio per le chiacchierate, i piacevoli confronti, la compagnia ed il buon vino e grazie a mia Nonna per essere la persona più saggia e buona che conosca.

Bene. Siamo arrivati alla fine anche di questo capitolo ed é anche grazie a tutti voi che mi siete vicino che sono potuto arrivare fin qui. Questi ringraziamenti non sono dovuti, bensì voluti perché ognuno di voi, più o meno consapevolmente, mi ha aiutato a diventare quello che sono oggi.

Quindi, grazie. Senza di voi questa tesi e questa persona non esisterebbero.

There are no problems, only challenges.

List of Publications

Enrico Rossi

1. F. Civerchia, **E. Rossi**, L. Maggiani , S. Bocchino, C. Salvadori and M. Petracca, "Lightweight Error Correction Technique in Industrial IEEE802.15.4 Networks", *42nd Annual Conference of IEEE Industrial Electronics Society (IECON)*, Firenze, Italy, (2016)
2. A. Biondi, A. Balsini, M. Pagani, **E. Rossi**, M. Marinoni and G. Buttazzo, "A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs", *37th IEEE Real-Time Systems Symposium (RTSS)*, Porto, Portugal, (2016)
3. F. Civerchia, S. Bocchino, C. Salvadori, **E. Rossi**, L. Maggiani and M. Petracca, "Industrial internet of things monitoring solution for advanced predictive maintenance applications", *Journal of Industrial Information Integration*, (2017)
4. **E. Rossi**, M. Damschen, L. Bauer, G. Buttazzo and J. Henkel, "Preemptive Partial Reconfiguration to Enable Real-Time Computing with FPGAs", *Transactions on Reconfigurable Technology and Systems*, (2018)

Abstract

Towards the heterogeneous, real-time reconfigurable embedded system

by

Enrico Rossi

To improve the computing performance in real-time applications, modern embedded platforms comprise hardware accelerators that speed up the tasks' most compute-intensive parts. A recent trend in the design of real-time embedded systems is to integrate field-programmable gate arrays (FPGA) that are reconfigured with different accelerators at runtime, to cope with dynamic workloads that are subject to timing constraints, like in signal processing or computer vision applications.

One of the major limitations when dealing with partial FPGA reconfiguration in real-time systems is that the reconfiguration port can only perform one reconfiguration at a time: if a high-priority task issues a reconfiguration request while the reconfiguration port is already occupied by a lower-priority task, the high-priority task has to wait until the current reconfiguration is completed (a phenomenon known as *priority inversion*), unless the current reconfiguration is aborted (introducing unbounded delays in low-priority tasks, a phenomenon known as *starvation*).

Moreover, hardware accelerators reconfigured at runtime inside the FPGA usually require minimum interaction with the software side and perform massive computations on data which have to be read from the main memory or written to it. Therefore, In case of high-throughput hardware accelerators may happen that the communication medium shared between main memory and hardware and software sides is not able to accept more requests jeopardizing the functioning of the whole system. Furthermore, as the software can not control each bus transaction of an hardware accelerator, mis-

designed accelerators could perform illegal memory accesses corrupting the main memory.

This thesis shows how priority inversion and starvation can be solved by making the reconfiguration process *preemptive*, i.e., allowing it to be interrupted at any time and resumed at a later time without restarting it from scratch. Such a feature is crucial for the design of runtime reconfigurable real-time systems, but not yet available in today's platforms. Furthermore, the trade-off of achieving a guaranteed bound on the reconfiguration delay for low-priority tasks and the maximum delay induced for high-priority tasks when preempting an ongoing reconfiguration has been identified and analyzed.

Besides, this work addresses the problems of memory protection and bus predictability by showing a solution to prevent hardware accelerators from choking the communication bus or performing illegal memory accesses, making the communication more predictable and allowing for more precise analysis. A custom memory protection and budgeting unit (MPBU) has been developed for this purpose.

Experimental evaluation on the Xilinx Zynq-7000 platform have been realized for preemptive reconfiguration and MPBU. Results show that the proposed implementation of preemptive reconfiguration introduces a low runtime overhead, thus effectively solving priority inversion and starvation. Moreover, experimental results show that *memory corruption* and *bus chocking* problems can be avoided and the communication over a shared bus can be made more predictable allowing to have less stringent timing constraints in the analysis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Heterogeneous, Real-Time Computing Era

Real-time systems are ubiquitous in our everyday life, e.g., in safety-critical domains such as automotive, avionics, or the rapidly growing domain of smart/autonomous machines (e.g., robotics or automated driving). In contrast to general-purpose computing systems, the correctness of a real-time system depends not only on the results of its computations, but also on the *time* at which outputs are produced. Delivering a result after a predetermined deadline may lead to malfunctions that can jeopardize the entire system. Therefore, for many safety-critical systems it is essential to *guarantee* that all time-sensitive computations are able to complete their execution within their deadlines.

To improve the performance of real-time systems, modern embedded platforms comprise hardware accelerators that speed up the tasks' most compute-intensive parts.

Current computer architectures are evolving towards heterogeneous platforms consisting of hybrid computational devices that may include processors of different types and field programmable gate arrays (FPGAs). In particular, the reprogrammable capa-

bilities of FPGAs, their increasing capacity, and their suitability for signal processing have made them attractive in several application domains as alternatives to application specific integrated circuits (ASICs) [1]. Xilinx [2] provided an analysis of recent progress in field programmable logic, highlighting that FPGAs have become bigger (comprising several million gates and up to a million bits of on-chip memory), faster (allowing system clock rates up to 200 MHz and I/O speed of up to 800 Mbits/second), more versatile (featuring dedicated carry structures to support adders, accumulators and counters), and cheaper, in terms of cost per logic gate.

A recent trend in the design of real-time embedded systems is to integrate FPGAs that are reconfigured with different accelerators at runtime, to cope with dynamic workloads, like in signal processing or computer vision applications [3, 4, 5, 6, 7]. For instance, platforms like the Xilinx Zynq or Altera SoC combine general-purpose CPUs with an FPGA on a single chip to enable the development of application-specific accelerators that can run on the FPGA in parallel with the software executing on the CPU. Low-latency channels enable communication between accelerators and software.

In addition, the possibility of reconfiguring specific portions of the FPGA at speeds of 400 MB/s enables the adoption of runtime virtualization techniques to share the FPGA among multiple tasks in different time windows, so extending the number of functions that can be accelerated. Such a virtualization technique for the FPGA has been proven to be effective to achieve a significant speedup in real-time applications [8, 6]. Virtualization is generally realized using *partial* FPGA reconfiguration, where parts of the FPGA are reconfigured while the remaining configuration remains fully functional.

1.2 FPGAs and Dynamic Partial Reconfiguration

FPGAs are in demand for their inherent rapid-prototyping and reconfiguration capabilities. The reconfigurable computing [9] is an interesting alternative to ASICs and general-purpose processors for implementing embedded systems, since it offers the flexibility of software processors and the efficiency and throughput of hardware co-processors.

Modern FPGA chips allow dynamic partial reconfiguration capabilities, enabling the user to reconfigure a portion of the FPGA dynamically (at runtime), while the remainder of the device continues to operate [10]. This is especially valuable in mission-critical systems that cannot be disrupted while some subsystems are being redefined. In this context, mission-critical functions could continue to meet external interface requirements while other reconfiguration regions are reprogrammed to provide different functionality.

Partial reconfiguration is also useful in systems where multiple functions share the same FPGA resources. In such systems, one section of the FPGA continues to operate, while other sections are reconfigured to provide new functionality. Such an interesting capability opens a new scheduling dimension for applications running on heterogeneous platforms. As in multitasking, where multiple applications share the processors by switching contexts between software processes, dynamic partial reconfiguration enables the possibility of interleaving multiple functions implemented as programmable logic on an FPGA recurrently shared by different processing components.

Partially reconfigurable FPGAs have many advantages over its non-reconfigurable counterparts. To list a few, a partially reconfigurable system can be re-synthesized after careful considerations resulting in the decrease of design time and consequently reducing the time-to-market. In the same way, parts of such systems can be altered

without having to shut down or even halt the system in its entirety, which undoubtedly enhances the system customizability and maintainability. Reconfigurable systems also allow the implementation of adaptive embedded systems incorporating self-optimization, self-organization of the system setup and self-adaptation to unpredictable changes in the environment [11]. In addition, the possibility of reconfiguring specific portions of the FPGA enables the adoption of runtime virtualization techniques to share the FPGA among multiple tasks in different time windows, so extending the number of functions that can be accelerated thus virtually extending the dimension of the FPGA. Such a virtualization technique for the FPGA has been proven to be effective to achieve a significant speedup in real-time applications [8, 6].

Although the partial reconfigurability of FPGAs offer new possibilities from the application perspectives, yet they pose many design challenges for today's demanding applications. Of the few challenges currently being researched on in this field, some are related to reconfiguration time [12], dynamic allocation and placement of hardware tasks in reconfigurable regions [13], design and development of an on-line scheduler responsible for scheduling real-time tasks to reconfigurable regions [14] [15].

One of the major limitations when dealing with partial FPGA reconfiguration is that the reconfiguration port can only perform one reconfiguration at a time. This means that when multiple tasks issue simultaneous requests to reconfigure different portions of the FPGA fabric, such requests must be serialized according to a given scheduling algorithm. In current implementations, the reconfiguration port of the FPGA does not allow preemptions; that is, once a reconfiguration process is started, it can either be completed or aborted. In other words, if a high-priority task issues a reconfiguration request while the reconfiguration port is already occupied by a lower-priority task, the high-priority task has to wait until the current reconfiguration is completed, unless the current reconfiguration is aborted to be restarted later from the beginning (see Chap-

ter 3). Unfortunately, considering the relatively long reconfiguration delays of current platforms[1], e.g., compared to context switching, both such solutions are not suitable for real-time applications. In fact, while non-preemptive reconfigurations introduce long delays in high-priority tasks (a phenomenon known as *priority inversion* [16]), aborting them may introduce unbounded delays in low-priority tasks (a phenomenon known as *starvation*). Both problems can be avoided by making the reconfiguration process *preemptive*, allowing it to be interrupted at any time and *resumed* at a later time from the point of interruption. Such a feature is not yet available in today's platforms.

Despite this limitation, there is a clear evolution trend showing that reconfiguration times are progressively decreasing. Liu et al. [17] designed a smart reconfiguration peripheral interface, based on the Xilinx internal configuration access port (ICAP) port [18], that is able to approach a throughput of 400 MB/s. Also, Duhem et al. [19] designed a fast reconfiguration interface by over-clocking the ICAP port up to 200 MHz, corresponding to a throughput of 800 MB/s. An overview of the trend of reconfiguration times (obtained by comparing the theoretical maximum throughput calculated from platforms' datasheets) is shown in Figure 1.1, based on the study conducted by Pagani et al. [20]. For this reason, it is plausible to expect that such a trend will continue in the upcoming years, thus making dynamic partial reconfiguration a relevant direction to be explored.

Although reconfiguration times are not negligible, FPGAs allow hardware acceleration of a wide class of algorithms with a significant speedup factor [21, 22] over the corresponding sequential software implementation. In the case study analyzed in the work by Biondi et al. [6], a speedup factor up to 15x has been measured for an image

[1]For example: on the Xilinx Zynq-7010 platform which currently is the lower-end in terms of reconfigurable resources, a maximum reconfiguration bandwidth of 400 MB/s is supported, however, reconfiguring 25% of the total resources still takes more than 2 milliseconds.

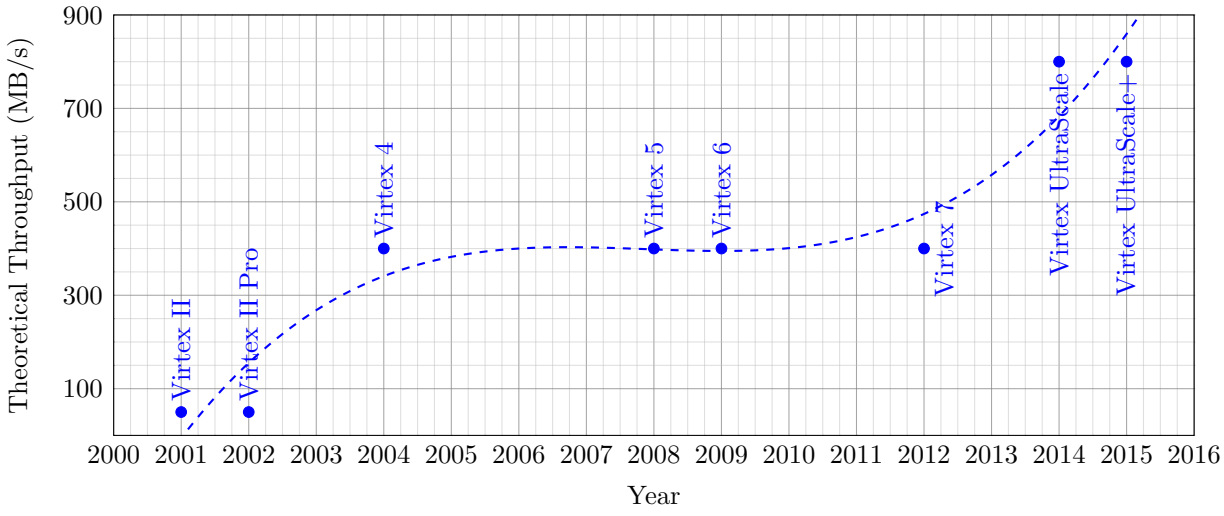


Figure 1.1: Trend of reconfiguration throughput.

processing filter implemented on the Zynq-7010 platform, which can reach a throughput of 145 MB/s for the dynamic partial reconfiguration, allowing to reconfigure an FPGA area containing about 25% of the total resources in less than 3 milliseconds.

1.3 Memory Protection and Bus Predictability

Other challenges of integrating partially reconfigurable FPGAs in real-time systems, reside in the system bus unpredictability. Real-time systems must be correct not only in the correctness and accuracy of calculations but also in the execution time. In particular, all the operation should be finished before deadlines [23].

Real-time systems can be classified into hard real-time systems and soft real-time systems according to the type of constraints applied on tasks. Tasks with a hard-deadline belongs to hard real-time systems where tasks are required to finish their execution strictly before the deadline or fatal mistakes would happen. On the other side, the violation of timing constraints in soft real-time system just leads to system

delay [24, 25].

Real-time tasks are required to fulfill two main requirements: timeliness of responses which demand tasks to quickly respond a request, and predictability of response time which requires the task computation time to be computable [26].

Therefore, real-time systems which integrate FPGAs as accelerators have to comply with those requirements too. Software tasks could require hardware acceleration to speed-up their computation thus reducing the computation time and meeting their deadlines. In particular, systems integrating system-on-chips[2] (SoCs) or simple FPGAs where custom accelerators are loaded at run-time inside the FPGA, may violate real-time requirements due to the unpredictability of the communication medium between hardware and software sides.

In fact, usually in those systems hardware accelerators require minimum interaction with the software side and perform massive computations on data which have to be read from the main memory or written to it. Moreover, as the main memory and the hardware and software sides connect on the same communication medium, a *shared memory* approach is the most common and straightforward method to provide communication between them: the software side shares with the hardware, part of the memory where both have full access.

In case of hardware accelerators that produce high traffic on the communication bus may happen that the bus is not able to accept more requests making other software tasks, which required hardware acceleration, miss their deadlines. Moreover, as the software can not control each bus transaction of an hardware accelerator, otherwise the computational speed-up will drastically decrease, mis-designed accelerators could perform illegal memory accesses corrupting the main memory.

[2]SoC FPGAs consist of processor, peripherals, and memory interfaces with the FPGA fabric using a high-bandwidth interconnect backbone.

The same problems regarding the enhancement of application predictability and memory protection, have been addressed in pure software environments by the work of Cucinotta et al. [27]. They tackle the problem of providing Quality of Service guarantees to virtualized applications, focusing on computing and networking guarantees. They propose a mechanism for providing temporal isolation based on a CPU real time scheduling strategy allowing not only to have control over the individual virtual machine throughput, but also on the activation latency and response-time by which virtualized software components react to external events.

Moreover, the work by Liang et al. [26] shows a high-speed and time-predictable bus architecture called RTBus, where high-performance AXI protocol is employed. They developed a real-time bus arbitration algorithm to accurately calculate the bus access time for master devices.

This thesis presents the first solution for achieving preemptive partial reconfiguration of hardware accelerator's reconfiguration onto an FPGA, preventing problems caused by a non-preemptable reconfiguration port shared among multiple tasks needing hardware acceleration. Moreover, it addresses the problems of memory protection and bus predictability by showing a solution to prevent hardware accelerators from choking the communication bus or performing illegal memory accesses, making the communication more predictable and allowing for more precise analysis.

Chapter 2 gives the required background on FPGAs. In particular, it present a general overview regarding FPGA's architecture and features, focusing on the most commonly used communication bus (the advanced extensible interface - AXI) and deeply explaining the dynamic partial reconfiguration feature.

Chapter 3 shows the problems that could arise using a non-preemptable shared

resource, i.e., the reconfiguration port, and a possible solution using preemptable partial reconfiguration. Furthermore, possible communication problems between hardware and software sides are presented. Eventually, the main contributions of this work are shown.

Chapter 4 presents a real implementation of preemptive partial reconfiguration, focusing on finding valid resumption points for preempted reconfigurations inside Xilinx's partial bitstreams. Moreover, the same chapter describes the implementation of a memory protection and bus budgeting unit that aims at avoiding illegal memory transactions and performs bus' bandwidth budgeting for hardware accelerators. It also describes the software interface that has been realized to correctly use the developed hardware IPs for preemptive partial reconfiguration, memory protection and bus budgeting.

Furthermore, **Chapter 5** determines worst-case bounds on the latency overhead that a higher-priority task experiences, when preempting a lower-priority task. It also determines an upper bound on the reconfiguration delay for reconfiguration requests from the lower-priority task for a given worst-case interval of reconfiguration preemptions and discuss under which circumstances preemption guarantees a minimum progress for these reconfigurations. Chapter 5 ends explaining what would be required to realize a worst-case analysis of a typical communication chain between CPU and FPGA and shows the limitations due to the lack of a detailed, public documentation of ARM and Xilinx IPs.

Eventually, **Chapter 6** shows the performed evaluation of preemptive partial reconfiguration and bus's bandwith budgeting on a real hardware platform integrating a Xilinx Zynq-7000 SoC.

Chapter 7 concludes this thesis and provides open future perspectives.

Chapter 2

Background

In order to fully understand what has been done in this work, it is first necessary to present tools and technological solutions that have been used explaining their advantages and disadvantages. Therefore, this chapter introduces field-programmable gate arrays (FPGAs) (see Section 2.1) describing their general architecture and showing how nowadays systems can benefit from integrating an FPGA in their solutions.

Moreover, Section 2.2 describes the Advanced eXtensible Interface (AXI) which is the most common communication medium used by FPGAs.

Eventually, the chapter ends with the description of a particular feature of Xilinx's FPGAs: dynamic partial reconfiguration. In particular, Section 2.3 introduce the partial reconfiguration feature describing its functioning and showing how the FPGA configuration file can be decoded for preemptive partial reconfiguration.

2.1 Field Programmable Gate Arrays

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by the user after manufacturing. Previously, as it was for application specific

integrated circuits (ASICs), FPGAs were configured using circuit diagrams but this method is increasingly rare as it is inconvenient and error prone for large designs. Nowadays the FPGA is generally configured using a hardware description language (HDL) that is a specialized computer language used to describe the structure and behavior of electronic circuits, and most commonly, digital logic circuits. HDLs were created to implement register-transfer level (RTL) abstraction, a model of the data flow and timing of a circuit [28], and enable a precise, formal description of an electronic circuit that allows for its automated analysis and simulation.

Architecture FPGAs have become one of the key digital circuit implementation media over the last decade. They are heterogeneous compute platforms that include RAMs, DSPs, look-up tables (LUTs) and an array of configurable logic blocks (CLBs) that are connected to each other through programmable routing interconnections (PRIs). A CLB is the basic component of an FPGA which provides the basic functionalities, logic and storage capabilities. Commercial vendors like Xilinx and Altera use LUT-based CLBs to provide basic resources and functionality. Figure 2.1 shows a generalized structure of an FPGA where CLBs are arranged in a two-dimensional grid and are interconnected by a PRI blocks. PRIs enable parallelism and pipelining of applications across the entire platform as all of these compute resources can be used simultaneously.

Contemporary FPGAs have large resources of logic gates and RAM blocks to implement complex digital computations. Some FPGAs have analog features in addition to digital functions. The most common analog feature is programmable slew rate on each output pin, allowing the user to set low rates on lightly loaded pins that would otherwise have unwanted oscillation, and to set higher rates on heavily loaded pins on high-speed channels that would otherwise run too slowly. Also common are quartz-crystal oscillators, on-chip resistance-capacitance oscillators, and phase-locked

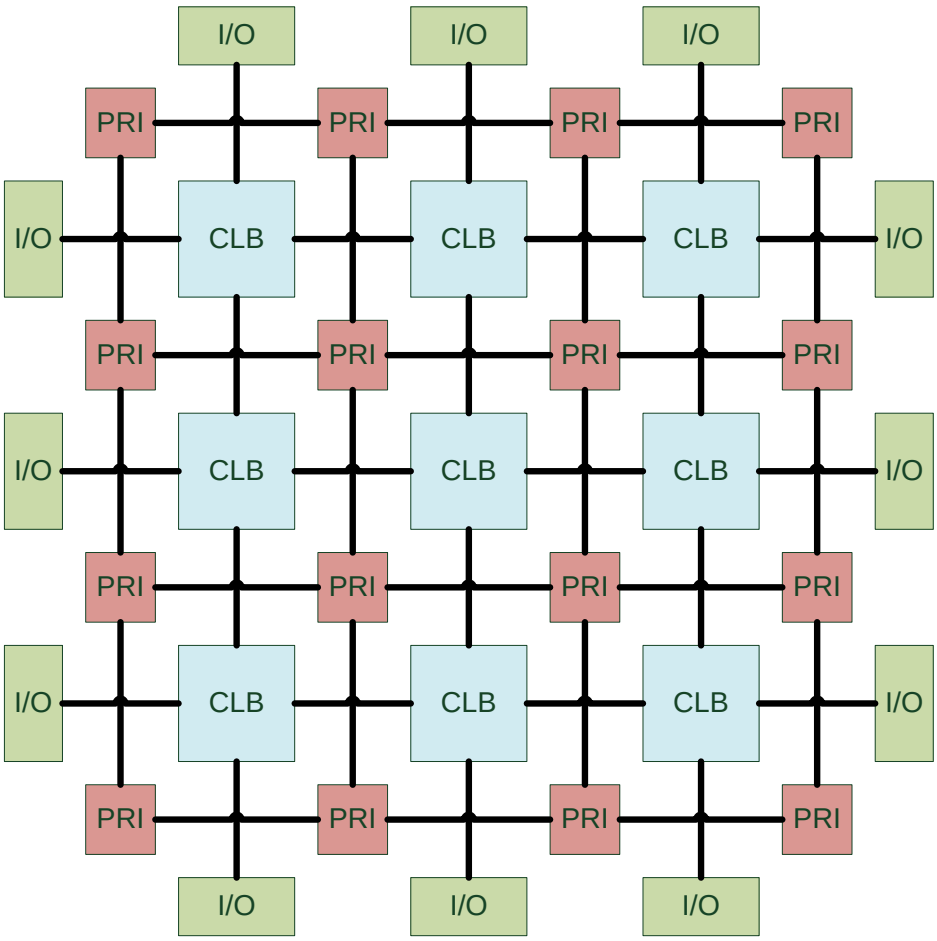


Figure 2.1: Generalized example of an FPGA architecture where configurable logic blocks (CLBs) are arranged in a two-dimensional grid and are interconnected by programmable routing interconnection (PRIs). PRIs also connect input/output (I/O) blocks with the internal configurable logic.

loops with embedded voltage-controlled oscillators used for clock generation and management and for high-speed communication interfaces.

A crucial part of FPGA's creation lies in their architecture, which governs the nature of their programmable logic functionality and their programmable interconnect. Therefore, the architecture and especially the interconnect network has a dramatic effect on the quality of the final device's speed performance, area efficiency, and power consumption.

The routing interconnect consists of programmable switches and wires which are used to build the required connection. As FPGAs claim to be a candidate to implement any type of digital circuits, their routing interconnect has to be very powerful and flexible in order to accommodate different circuits and their routing demands.

Different digital circuits require different routing networks but it is possible to define two types of routing: local routing and global routing. These routing methods connect logic blocks through the entire chip (global routing) or sections of it (local routing) with small propagation delay. Moreover, modern FPGA have specific routing resources to correctly route clock signals through the whole FPGA with the smallest possible propagation delay [29].

Advantages and Disadvantages The main advantage of FPGAs, i.e., flexibility, is also the major cause of its draw back. Flexible nature of FPGAs makes them significantly larger, slower, and more power consuming than their ASIC counterparts. These disadvantages arise largely because of the programmable routing interconnect of FPGAs which comprises of almost 90% of total area of FPGAs.

Despite these disadvantages, FPGAs present a compelling alternative for digital system implementation due to their less time to market and low volume cost.

In general, computing systems based on FPGAs provide many advantages over

conventional ASIC implementations [30]:

- *Easy upgrade*. In contrast to traditional computer chips, FPGAs are completely configurable and functionalities can be upgraded at any time.
- *Long-term maintenance*. FPGAs enable independency from component manufacturers and distributors, since FPGA chips can be reprogrammed to include new functionalities or upgrade the existing ones. Moreover, newer FPGAs support dynamic partial reconfiguration allowing to reconfigure only a portion of them while the rest of the logic continues to operate.
- *Short time to market*. The development of hardware prototypes is significantly shorter, since ideas and concepts can be verified in hardware without going through the long fabrication process of custom ASIC design. Moreover, the growing availability of high-level software tools decreases the learning curve with layers of abstraction and often offers valuable pre-built functions for advanced control and signal processing.
- *Efficiency*. Systems can be customized for the designated task.
- *Cost*. The silicon programmability removes fabrication costs and lead times for assembly. Because system requirements often change over time, the cost of making incremental changes to FPGA designs is negligible compared to the one of redesigning an ASIC.
- *High performance*. Taking advantage of hardware parallelism, FPGAs exceed the computing power of digital signal processors (DSPs) by accomplishing more operations per clock cycle. Also, controlling inputs and outputs at the hardware level provides faster response times and specialized functionality to closely match application requirements.

- *Real-time applications*. In contrast to software activities running in real-time operating systems, FPGAs provide a more deterministic behavior, minimizing reliability concerns with true parallel execution and deterministic dedicated hardware.

2.1.1 Softcores and System-on-Chip

Traditionally, FPGAs have been used only for application specific hardware designs. However, as their capacity and complexity increased with the advance in VSLI[1] technology, more complicated circuits and systems became possible. Vendors as Xilinx and Altera started providing general purpose *softcores* (also called softprocessors) solutions which enable to synthesize a programmable processor using the FPGA logic along with other IP blocks allowing users to develop more sophisticated systems.

General purpose softcores allows for quick prototyping of architectures where the advantages of both software and hardware design methodologies are mixed together. They offer designers tremendous flexibility during the design process, allowing full configuration of the processor to meet system's constraints [31]. Moreover, a softcore-based design outperforms the traditional hardware-based design in terms of rapid prototyping, debugging capabilities and development time and cost while, compared to a purely software-based design is much more flexible for a hardware extension that replaces the critical kernel of the application.

It is well known that designs based on general purpose softcores lack in terms of performances and power consumption but the previously mentioned advantages, flexibility and scalability, allow it to be competitive against traditional methodologies [32].

In the early 1990s, the term *ASIP* has emerged denoting processors with an application specific instruction set [2] and as FPGAs provide resources to build custom

[1] Very Large Scale Integration

[2] A general overview of benefits and challenges of ASIPs is given by Henkel in [33]

hardware modules, it is possible to build custom softcores, similar to ASIPs to fulfill application-specific requirements. Unfortunately, ASIP approach assumes that customizations are undertaken during design-time with little or no adaptation possible during run-time [34]. It is indeed hard or even impossible to predict the performance or other design criteria accurately during design time. Consequently, the more critical design decisions are fixed during design time, the less flexible an embedded processor can react to non predictable application behaviors [35].

Reconfigurable computing [36, 37, 38] i.e., FPGAs featuring dynamic partial reconfiguration, may address this problem by enabling dynamic adaptivity through partial reconfiguration of the FPGA [35, 39, 40]. In fact, Bauer et al. in [34] combine the paradigms of extensible processor design and dynamic re-configuration in order to address dynamic changes of application's characteristics due to switching to different operation modes or changes in design constraints (e.g., systems runs out of energy). Besides the approach of targeting full computational tasks in reconfigurable hardware [39] and the research for CPU-attached reconfigurable systems mainly focused on design-time predefined reconfiguration decisions. This is not suitable when computational requirements/constraints change during run time and are unpredictable during design time. Vassiliadis et al. present the Molen Processor which couples reconfigurable hardware to a base processor via a dual-port register file and an arbiter for shared memory [41]. In Vassiliadis et al. approach, the run-time reconfiguration is explicitly predetermined by additional instructions. The OneChip98 project [42] uses a Reconfigurable Functional Unit (RFU) that is coupled to the host processor and that obtains its speedup mainly from streaming applications. Eventually, Hauck et al. show that separating reconfigurable logic from the host processor, current custom computing systems suffer from a significant communication bottleneck. Therefore they designed Chimaera [43], a system that overcomes the communication bottleneck by integrating

reconfigurable logic into the host processor itself.

Besides general purpose and custom softcores, newer FPGAs exploit the System-on-Chip (SoC) technology which consists of the integration of multiple silicon die inside the same package. Those SoC FPGAs embed a more powerful, single or multi-core hard-processor which primarily differs from a softcore in the integration method: the hard-processor is directly integrated on a silicon die and connected to the FPGA die through an high-performance communication bridge.

Therefore, SoC FPGAs consist of a processor, peripherals, and memory interfaces with the FPGA fabric using a high-bandwidth interconnect backbone. It combines the performance and power savings of hard intellectual property with the flexibility of programmable logic. In particular, SoC FPGAs allow to reduce system power, cost, and board size by integrating discrete processors and digital signal processing (DSP) functions into a single FPGA and improving system performance via high-bandwidth interconnect between the processor and the programmable fabric.

Using a SoC FPGA and exploiting hardware acceleration allows meeting real-time constraints in applications where even high-end PCs usually fail. He et al. [44] present an FPGA-based SoC implementation of an efficient and robust face detection algorithm [45] that uses a cascaded Artificial Neural Network classification scheme based on AdaBoost-trained Haar features [46, 47]. They designed a face detection system which can detect faces at speeds of roughly two orders of magnitude (100×) higher than the corresponding software implementation running on a 2.4GHz CPU. Moreover, the work by Oetken et al. [48] proposes an FPGA-based SoC architecture with support for dynamic runtime reconfiguration in a *smart camera* case study. They implemented a reconfigurable design with support for free module placement and an enhanced memory access method for high-speed communication with an external memory.

2.2 Advanced eXtensible Interface (AXI)

Together with the introduction of soft-cores and SoCs and their increase in computational power, the communication efficiency between cores and the FPGA has become a botleneck which limits the performance of systems based on those solutions [49][50]. Thanks to the flexibility of FPGAs, it is possible to implement with the internal logic a communication medium which allows the communication between soft/hard cores and hardware IPs integrated in the configurable fabric. Therefore, a standard bus architecture has been introduced in such devices in order to realize an efficient, flexible and powerful communication bus.

The Advanced eXtensible Interface (AXI) protocol is used by many SoCs and FPGAs today and is part of the ARM Advanced Microcontroller Bus Architecture (AMBA) specification [51]. The AMBA 4 AXI protocol builds on many benefits of the AMBA 3.0 AHB standard by greatly extending the performance and flexibility of the on-chip bus [52][53][54]. The AXI protocol provides the communication rules that different modules on a chip needs to abide by to communicate with each other. It is based on a handshake-like procedure before all transmissions and allows provides an effective medium for transfers of data between the existing components on the chip.

The main specifications of the protocol are summarized below:

- Before transmission of any control signals, addresses or data, both master and slave must extend their "hand" for a handshake via *ready* and *valid* signals.
- Separate phases exist for transmission of control signals, addresses and data.
- Separate channels exist for transmission of control signals, addresses and data.
- Burst type communication allows for continuous transfer of data.

In particular, the interface works by establishing communication between master and slave devices. Between these two devices, five separate channels exist: Read Address, Write Address, Read Data, Write Data, and Write Response. Each channel has its own unique signals as well as similar signals existing among all five. The valid and ready signals exist for each channel as they allow for the handshake process to occur for each channel. For transmitting any signal (control signals, addresses or data) the relevant channel source provides an active valid signal and the same channel's destination must provide an active ready signal. After both signals are active, transmission may occur on that channel. As stated above, the transmission of control signals, addresses or data, are done in separate phases, therefore an address must always be transferred before the handshake process can occur for the corresponding data transfer. In the case of writing information, the response channel is used at the completion of the data transfer.

Furthermore, there are additional options that the protocol provides which increase its complexity, such as burst transfer, quality of services (QoS) and protections. These options are simply extra signals existing on the different channels that allow for additional functionality.

2.2.1 AXI4 Interface

The AXI bus is especially prevalent in Xilinx's Zynq devices, providing the interface between the processing system and programmable logic sections of the chip. In particular, Xilinx adopted the AXI4 version included in the AMBA 4.0 release. There are three types of AXI4 interfaces:

- *AXI4* for high-performance memory-mapped requirements. It allows bursts of up to 256 data transfer cycles with just a single address phase.

- *AXI4-Lite* a subset of AXI4 for simple, low-throughput memory-mapped communication. It is a single transaction memory mapped interface, has a small logic footprint and is a simple interface to work with.

- *AXI4-Stream* for high-speed streaming data. It removes the requirement for an address phase altogether and allows unlimited data burst size. AXI4-Stream interfaces and transfers do not have address phases and are therefore not considered to be memory-mapped.

Xilinx adopted the Advanced eXtensible Interface (AXI) protocol for Intellectual Property (IP) interconnection and as a communication medium between cores and FPGA inside SoCs. AXI4 provides improvements and benefits to productivity, flexibility, and availability. Indeed improves productivity by standardizing on the AXI interface, enhances flexibility providing the right protocol for the specific application and increase availability by moving to an industry-standard.

AXI4 The AXI specifications describe an interface between a single AXI master and a single AXI slave, representing IP cores that exchange information with each other. Memory mapped AXI masters and slaves can be connected together using a structure called an Interconnect block. The Xilinx AXI Interconnect IP contains AXI-compliant master and slave interfaces, and can be used to route transactions between one or more AXI masters and slaves [51].

Both AXI4 and AXI4-Lite interfaces consist of five different channels: Read/Write Address Channel, Read/Write Data Channel and a Write Response Channel. Data can move in both directions between the master and slave simultaneously, and data transfer sizes can vary. As shown in Figure 2.2, AXI4 provides separate data and address connections for reads and writes, which allows simultaneous, bidirectional

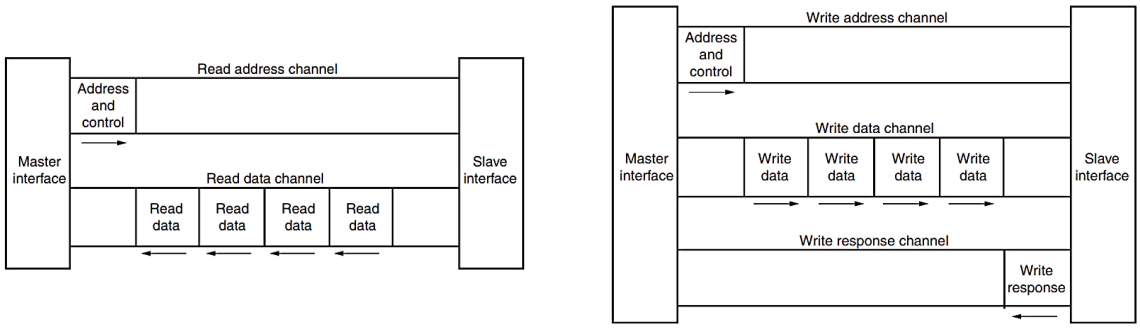


Figure 2.2: AXI channel architecture. The organization of the read channel on the left side and of the write channel on right side.

data transfer. AXI4 requires a single address and then bursts up to 256 words of data. The AXI4 protocol describes a variety of options that allow AXI4-compliant systems to achieve very high data throughput. Some of these features, in addition to bursting, are: data upsizing and downsizing, multiple outstanding addresses, and out-of-order transaction processing.

AXI4-Stream The AXI4-Stream protocol defines a single channel for transmission of streaming data. The AXI4-Stream channel is modeled after the write data channel of the AXI4. Unlike AXI4, AXI4-Stream interfaces can burst an unlimited amount of data and transfers cannot be reordered.

2.3 Dynamic Partial Reconfiguration

Reconfiguring an FPGA is more complex than a simple transfer of the configuration data from main memory to the configuration memory of the FPGA. Especially when reconfiguring an FPGA partially, it needs to be ensured that the FPGA remains in a consistent state and the not-reconfigured parts remain functional all the time. A

partial bitstream contains all the information for the area that should be reconfigured, e.g., the address in the configuration memory that corresponds to the area of the FPGA where the design should be placed. The state of the FPGA is controlled by a finite state machine (FSM) that is part of the reconfiguration port. This FSM executes *operations* that are part of the bitstream besides the configuration data. Section 2.3.1 reports the information about the reconfiguration port and bitstreams that are openly available for Xilinx FPGAs [55] while Section 2.3.2 presents an approach to decode partial bitstreams and define a common structure that can be referred to when detailing preemptive reconfiguration. Such a structure is not openly available for Xiling FPGAs, hence it was part of this work to analyze the bitstream format in order to enable reconfiguration preemption and determine at what points reconfiguration of a bitstream can be preempted and resumed, as well as additional steps necessary to achieve this.

2.3.1 Xilinx Bitstreams and Reconfiguration Port

One of the contributions of this work, focuses on the reconfiguration port of Xilinx FPGAs, called Internal Configuration Access Port (ICAP). The information contained in the bitstreams for these FPGAs can be divided into two categories: operations executed by the reconfiguration port FSM and the actual configuration data that is transferred to the FPGA configuration memory. The smallest addressable segments of the configuration memory are called *frames*, and all operations act upon one or more frames. In Xilinx 7 Series FPGAs, each frame consists of 101 32-bit words [55]. The relevant operations for preemptive reconfiguration that are executed by the reconfiguration port FSM are summarized in the following.

Figure 2.3 shows the chronological sequence of the most relevant operations performed by partial bitstreams (a more detailed description can be found in Section 2.3.2).

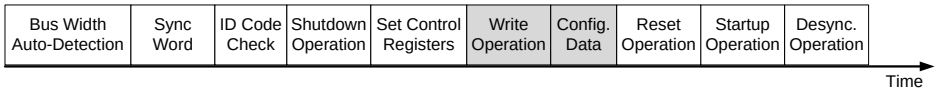


Figure 2.3: Generic sequence of operations performed in a partial bitstream. White blocks are operations common to all partial bitstreams; gray blocks are specific operations that depend both on the size and physical position of the reconfigurable area inside the FPGA and on the used FPGA device. These gray blocks constitute the largest part of the partial bitstream.

Partial bitstreams start with a *Bus Width Auto-Detection* operation that is used to automatically detect the word width that is sent to the reconfiguration port (1, 2 or 4 bytes are possible values). After that, the *Synchronization Word* initializes the reconfiguration port to accept configuration data, followed by the *ID Code Check*, which ensures that the bitstream target device matches the FPGA that is being reconfigured. The *Shutdown Operation* safely shuts the area that is going to be reconfigured down and the *Set Control Register Operation* configures the available reconfiguration features. *Write Operations* transfer the actual configuration data, specifying the starting frame address and the amount of words (multiples of whole frames) to write.

After all configuration data has been loaded into the FPGA configuration memory, a *Reset Operation* is performed to initialize the logic inside the reconfigured region. Partial bitstreams end with a *Startup Operation*, where the device activates I/Os and the logic belonging to the reconfigured area, and a *Desynchronization Operation* that de-synchronizes the reconfiguration port (inverse to the Synchronization Word). After de-synchronization, the reconfiguration port ignores any following data on its inputs until the next synchronization.

Two additional operations are used in partial bitstreams:

- *No-Operation*. Decoded by the state machine without producing any actions. Since each operation is executed by the reconfiguration port FSM, each operation has a defined latency. This operation is used, when required, to introduce clock-cycles delays in order to wait for the completion of the running operation.

- *Null Operation*. This is a write operation that writes zeros to a specific FPGA register. Some operation as the *Shutdown Operation* and the *Startup Operation* need to be activated after being issued to the reconfiguration port FSM. The activation is done by a Null operation.

Each operation can be sent to the reconfiguration port in packets of two possible formats: 'Type 1' or 'Type 2'. Type 1 packets are used when small amount of data words (to be read or written) are required by the operation. Type 2 packets are used to write long segments of data into the FPGA configuration memory. The address within the configuration memory needs to be supplied by a preceding Type 1 packet [55].

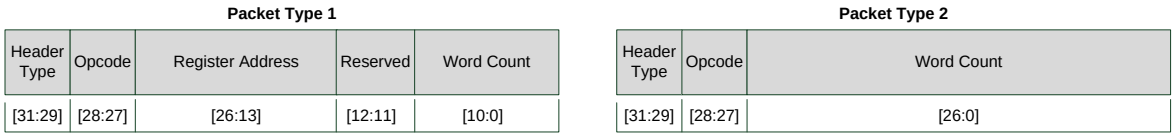


Figure 2.4: The FPGA bitstream consists of two packet types: Type 1 and Type 2. The Type 1 packet is used for register reads and writes; the Type 2 packet, which must follow a Type 1 packet, is used to write long blocks. No address is presented here because it uses the previous Type 1 packet address.

As Figure 2.4 shows, both packet types have a *Word Count* field that contains the exact number of data words following the actual operation. It instructs the reconfiguration port FSM to directly write that amount of words to the configuration registers or memory instead of decoding them.

Two FPGA-internal registers are central for the purpose of preemptive reconfiguration:

- Frame Address Register (FAR). It contains the address of the next frame in configuration memory to be written.
- Frame Data Input Register (FDRI). It contains the number of data frames that have to be written to the FPGA configuration memory, starting from the frame address specified by the FAR register.

These registers are the reconfiguration state that needs to be restored when a reconfiguration is resumed.

The information summarized in this section has been gathered from openly-available Xilinx documentation [55], while the information in the following section were gathered by manually decoding partial and full bitstreams.

2.3.2 Decoding Partial Bitstreams

As mentioned in Section 2.3, there is no openly-documented structure of partial bitstreams provided by Xilinx. Such a structure is required for preemptive partial reconfiguration to determine points in the bitstream at which reconfiguration can safely be preempted and resumed. Therefore, a general structure for partial bitstreams is defined in the following, based on information gathered from decoding numerous bitstreams. There are two types of bitstreams that have been utilized to obtain the required information: *standard* bitstreams and *debug* bitstreams. Standard bitstreams configure multiple frames after a single write to the FAR and the FDRI registers, which increment automatically at the end of each frame. Debug bitstreams configure each frame individually, writing the FAR and the FDRI after each frame, and thus providing information about FPGA-specific configuration memory addressing.

With information gathered by decoding both bitstream types, it has been possible to group operations in partial bitstreams into *sequences* that fulfill specific purposes. Each operation sequence can optionally contain configuration data organized in *data chunks*. Moreover, sequences in the bitstream have been grouped into *sections* (consisting of one or more sequences). Configuration data within a sequence of operations contains the description of the hardware that will be reconfigured inside the pre-defined reconfigurable area, called *reconfigurable slot*, inside the FPGA. The resulting general structure of sections for partial bitstreams in Xilinx 7 series FPGA is shown in Figure 2.5.

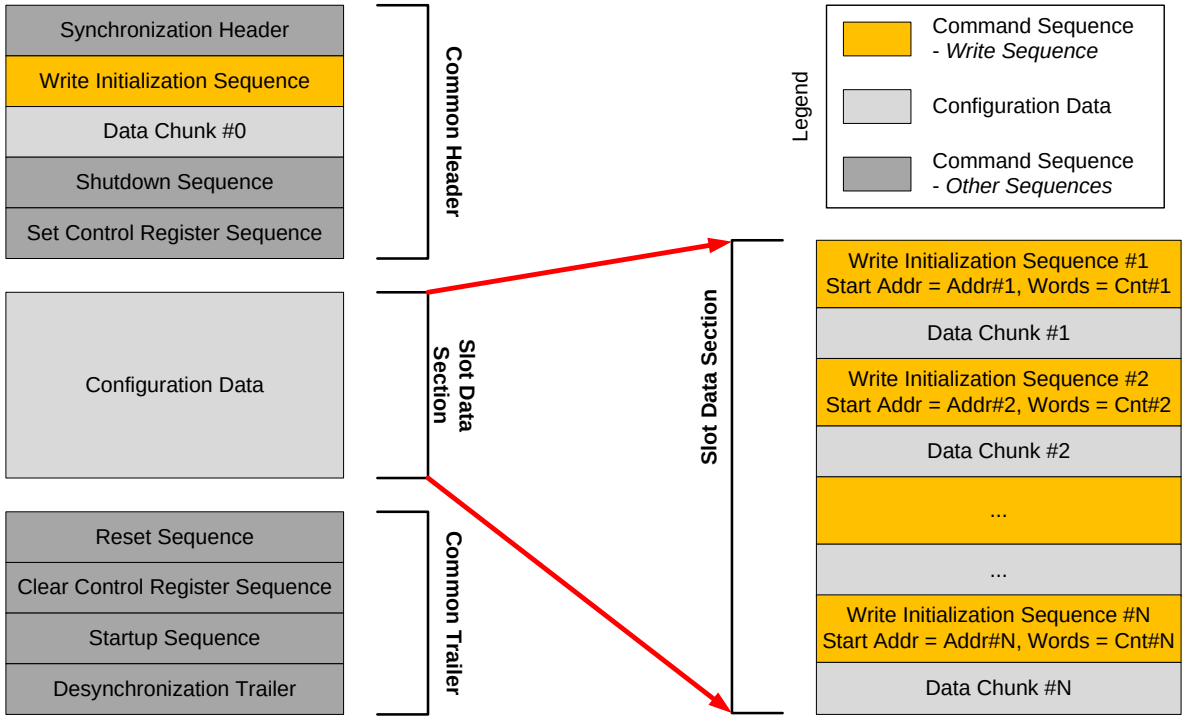


Figure 2.5: Structure of a partial bitstream. The left hand side shows a partial bitstream where three main sections are identified: Common Header, Reconfigurable Slot Data Section, and Common Trailer. Darker sections are common to all partial bitstreams, while the Configuration Data section depends on the area that is reconfigured. The right hand side shows the structure of the Reconfigurable Slot Data Section.

A partial bitstream can be structured into three main sections: Common Header, Reconfigurable Slot Data Section, and Common Trailer.

Common Header This first section is common to all partial bitstreams targeting the same FPGA and it is the only device-specific section. Its main function is to synchronize the reconfiguration port and prepare the device to receive the bitstream. It consists of the following four sequences: *Synchronization Header*, *Write Initialization* followed by its data chunk (which contains the data used to initialize special configurable blocks inside the FPGA), *Shutdown* and *Set Control Register*. These sequences are made of operations that initialize the reconfiguration port to receive data (Synchronization Word), set its bus width (Bus Width Auto-Detection), and perform the ID code check (ID Code Check). Furthermore, during the Write Initialization sequence, an FPGA-specific number of frames is sent to the configuration memory to configure particular resources called CFG_CLB used to define which part of the FPGA itself will be reset or reconfigured [56]. This section ends with a Shutdown sequence, that safely disables the area that is going to be reconfigured, and a Set Control Register sequence which configures the FPGA device [55].

Reconfigurable Slot Data Section This section of the bitstream depends on the slot that is reconfigured. It contains the slot's configuration data which describe the user-logic that will be written to the FPGA configuration memory. Configuration data is divided into an even number of data chunks consisting of numerous frames to be written into the configuration memory.

Common Trailer The last section is common to all partial bitstreams targeting the same FPGA. The function of this section is to reset the programmed logic and to de-

synchronize the reconfiguration port. Four main sequences can be identified: *Reset*, *Clear Control Register*, *Startup* and *De-Synchronization Trailer*. These sequences consist of operations used to initialize and activate the reconfigured logic and safely de-synchronize the reconfiguration port.

As explained in Section 2.3, partial bitstreams contain information for the area to be reconfigured and operations that control the reconfiguration FSM. Therefore, to enable preemptable reconfiguration, the knowledge of partial bitstream structures is essential, because a reconfiguration can be aborted at any time but can only be resumed from specific points in the bitstream. Section 4.1.1 explains which are the point in the bitstream where reconfigurations can be safely resumed.

This chapter presented all basic information necessary to fully understand this thesis. The following chapter explains the problems that have been considered as motivations of this work and the contribution to solve those problems.

Chapter 3

Motivations and Contributions

When exploiting FPGAs with Dynamic Partial Reconfiguration in real-time embedded systems, four main issues arise:

- provide worst-case response time bounds of computations consisting of software tasks and hardware accelerated functions;
- provide a method to avoid scheduling problems due to the use of a single, non-preemptive reconfiguration port;
- protect the system from malicious hardware accelerators that may possibly disrupt the whole application;
- enhance the bus predictability in order to provide more precise worst-case response time bounds.

Although several works have been done to analyze the timing behavior of real-time applications using FPGAs, most of them did not consider Dynamic Partial Reconfiguration capabilities at a job level. Only the work by Biondi et Al. [6] addresses the problem

of providing worst-case response time bounds and proposes a new computing framework for enabling a timing analysis of real-time activities that make use of hardware accelerators developed through programmable FPGAs with Dynamic Partial Reconfiguration capabilities. They use the developed framework to derive a response-time analysis and verify the schedulability of a real-time task set under given constraints and assumptions. Although the analysis is based on a generic model, the proposed framework has been conceived to account for several real-world constraints present on today's platforms and has been practically validated on a real FPGA platform, showing that it can actually be supported by state-of-the-art technologies.

Furthermore, as explained in Chapter 1, Dynamic Partial Reconfiguration enables modifying parts of the logic configured on the FPGA while the remaining logic remains functional and it uses a single dedicated and non-preemptive *reconfiguration port* to write configuration data into the FPGA's configuration memory. Therefore, only one reconfiguration at a time can be performed thus in a multi-tasking system where multiple tasks can request reconfigurations, the reconfiguration port is a contended resource.

Eventually, as hardware accelerators aim at unburdening the software side from compute-intensive tasks, there is the need to make hardware and software communicate in a fast, safe and reliable way.

Therefore, the purpose of this thesis is to provide the tools to make FPGAs integration in real-time systems safer, avoiding malicious hardware accelerators to compromise the the whole application, more predictable, allowing for a more precise analysis of worst-case execution bounds, and more performing exploiting Dynamic Partial Reconfiguration and enabling reconfiguration preemptions.

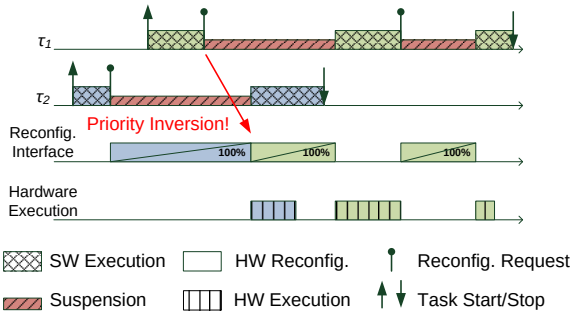
3.1 Scheduling Problems

When a task issues a reconfiguration request (in this thesis is considered the model used and detailed in [6]), it self-suspends to wait for the reconfiguration to be completed, allowing the other tasks to execute on the CPU. Consider, for instance, a single-core system running two tasks, τ_1 and τ_2, with τ_1 having higher priority than τ_2. The following three approaches can be adopted for managing reconfiguration requests:

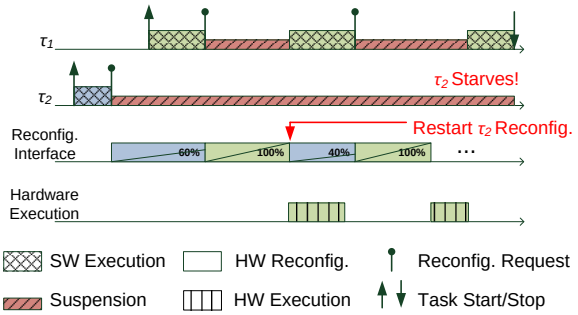
a) **Configure-to-completion**: an active reconfiguration can not be preempted or aborted once started, but occupies the reconfiguration port until completed. This case is shown in Figure 3.1a, where τ_2 starts executing and takes control over the reconfiguration port. When τ_1 starts executing and requests a reconfiguration, it must wait for τ_2 to complete its reconfiguration, i.e., tasks are not executed according to their priority order (*priority inversion*).

b) **Abort**: the reconfiguration process can not be suspended and resumed, but it can be aborted. Every time a reconfiguration was aborted, it needs to be restarted from the beginning. While this policy avoids priority inversion, it can lead to starvation of τ_2, as shown in Figure 3.1b. τ_2 starts executing and takes control over the reconfiguration port, but its reconfiguration is aborted once τ_1 requests a reconfiguration. When higher-priority reconfiguration requests abort τ_2's reconfiguration frequently, then τ_2 suffers from starvation.

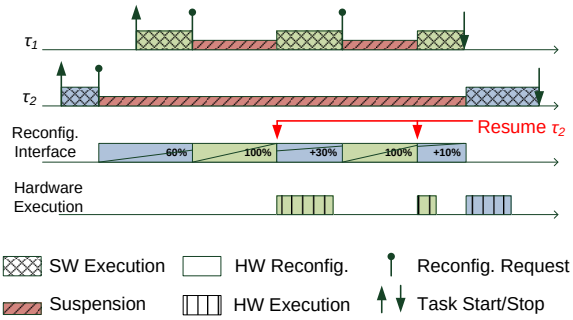
c) **Preemptive reconfiguration**: the reconfiguration requested by a task can be preempted in favor of another higher-priority reconfiguration. When the higher-priority reconfiguration is completed, the suspended reconfiguration is resumed from the last valid point (where it is safe to resume the reconfiguration) already passed at the time it was suspended. As shown in Figure 3.1c, τ_2 starts executing



(a) Configure-to-completion leads to priority inversion: When an active reconfiguration can not be stopped, then τ_1 has to wait for τ_2 to complete its reconfiguration even if τ_1 has a higher priority.



(b) Abort leads to starvation: τ_1 repeatedly requests reconfigurations, aborting τ_2's reconfiguration (that needs to be restarted from the beginning).



(c) Preemptive reconfiguration solves priority inversion and starvation: Lower-priority reconfigurations can be preempted by a higher-priority task. The preempted reconfiguration can be resumed afterwards.

Figure 3.1: Multi-tasking systems can experience priority inversion and starvation problems due to a non-preemptive shared resource. Those problems are solved in this work for reconfigurations, by making the shared reconfiguration port preemptive.

and takes control over the reconfiguration port, its reconfiguration is preempted once τ_1 requests a reconfiguration. Despite the frequent reconfigurations of τ_1, τ_2 will complete its reconfiguration eventually, because the progress made by τ_2 during reconfiguration is kept each time it is preempted.

More realistic scenarios involving multiple tasks and large and more complex hardware accelerators (that require larger reconfiguration time) would lead to a more complex contention of the reconfiguration port, increasing delays and making the problems of priority inversion and starvation even more severe. To solve those problems, a preemptive reconfiguration method has been developed and implemented (see Chapter 4). Using the proposed approach, the reconfiguration delay for lower-priority tasks is bounded, while higher-priority tasks do not experience priority inversion, thus allowing the development of multi-priority and mixed-criticality systems that benefit from runtime reconfiguration.

3.2 Safety and Predictability Challenges

As hardware accelerators aim at unburdening the software side from compute-intensive tasks, there is the need to make hardware and software communicate in a fast, safe and reliable way.

As showed in Figure 3.2, assume to have a system with two hardware accelerator and a CPU, all connected to the system memory through a bus in a multi-master configuration.

Accelerators can be software enabled and perform computations on CPU's data. One of the most simple and effective communication techniques relies on a shared memory approach: the software shares one or more memory buffers, with each hardware accelerator, where both have full access. The CPU loads the hardware accelerator's

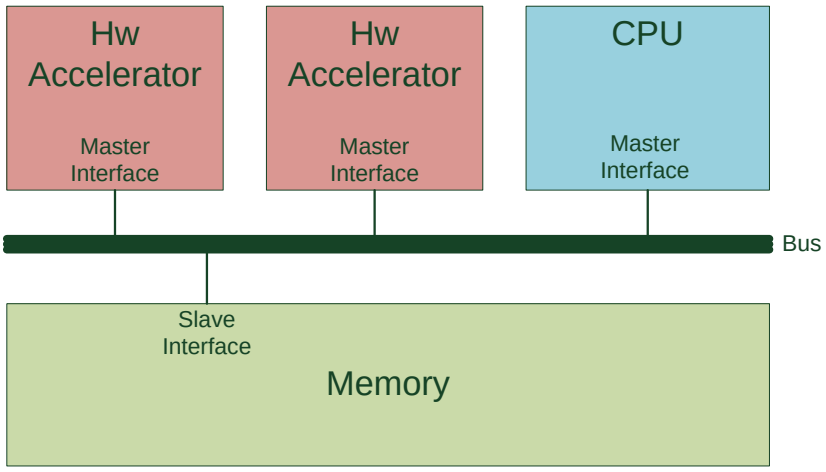


Figure 3.2: Multi-master configuration of a communication bus. Two hardware accelerators and one CPU would exchange information through the main memory.

input data inside the shared buffer and sends to the accelerator the buffer size and base address. Eventually, the CPU activates the accelerator which saves its output data inside another shared buffer.

This approach allows a fast and straightforward communication between hardware and software, but has some drawbacks:

- *Memory Corruption*: the hardware accelerator has full access to the CPU memory, therefore a bugged/malicious hardware could perform illegal memory access leading to memory contamination.

- *Bus Choking*: the hardware accelerator has no transactions limits, so that a bugged/malicious accelerator could continuously perform read/write transactions to the system memory preventing the system bus from accepting other transactions, thus stalling the entire system.

These drawbacks may lead to performance reduction or, even worse, can make a real-time application fail, jeopardizing the entire system. In particular, the Bus

Chocking effect also prevents the possibility to find a precise worst-case bound of the communication time. Furthermore, Memory Corruption and Bus Choking can not be solved using software techniques because the hardware accelerator behaves as a master on the bus and the software side does not have any control over the single bus transaction it performs.

A possible non-optimal approach could integrate the protection logic to prevent Memory Corruption and Bus Choking directly inside the reconfigurable hardware accelerator. However, in partially reconfigurable FPGA designs, the reconfiguration time of accelerators depends on the area that needs to be reconfigured: the bigger the area, the higher the reconfiguration time. Therefore, integrating the protection logic inside the accelerator would results in higher reconfiguration times and loss of performance and does not guarantee that a malicious accelerator correctly integrates it or keeps it enabled.

Eventually, as described in Section 2.1 the integration of FPGAs in real-time, embedded systems not only improves the computational performances but allows to enhance flexibility and power consumption. Having a partially reconfigurable FPGA allows to schedule in time the FPGA area thus obtaining an FPGA with area virtually infinite. Moreover, the use of reconfiguration preemption combined with safety mechanisms and more bus predictability can enhance the overall flexibility and performance of the system.

As benefits and drawbacks in having preemption have already been extensively discussed by the real-time, this work aims at demonstrating the feasibility and verify the performance of the designed hardware and software. Moreover, disadvantages have been analyzed and specifically addressed by the real-time community since ever.

However, those solutions could probably not be fully ported to the FPGA environment but this analysis would require another thesis.

In particular, the novel contributions of this thesis can be summarized as follows:

- To the best of our knowledge, this work provides the first realization of a preemptive FPGA reconfiguration.
- Using the proposed preemptive reconfiguration mechanism, the reconfiguration delay is bounded analytically for low-priority tasks that are subject to preemptions.
- A trade-off analysis is presented to balance the reconfiguration delay experienced by the low-priority tasks and the maximum delay induced in high-priority tasks when preempting an ongoing reconfiguration.
- To the best of our knowledge, this thesis presents the first implementation of a dedicated hardware that provides memory protection and budgeting of the bus bandwidth in systems exploiting FPGAs.
- Using the proposed hardware for memory protection and bus bandwidth's budgeting, the bus predictability is enhanced and a fair scheduling of transactions over it is guaranteed.

The following chapter presents the designed hardware and how the hardware/software integration has been performed.

Chapter 4

Hardware and Software Design

In order to successfully integrate FPGAs in real-time embedded systems and provide performance, safety and predictability, it is necessary to design software-driven hardware IPs to integrate in FPGAs or SoCs. A crucial step of the development of those IPs is the hardware/software co-design, thus the meeting of system-level objectives by exploiting the trade-offs between hardware and software in a system, through their concurrent design. Hardware/software co-design implies hardware/software partitioning, hence the process of deciding, for each subsystem, whether the required functionality is more advantageously implemented in hardware or software. Its goal is achieving a partition that satisfies the required performance within the overall system requirements (i.e., size, weight, power, cost, etc.).

This chapter shows how hardware and software have been partitioned describing the implementation of the needed hardware IPs and their software drivers.

4.1 Preemptive Partial Reconfiguration

A main challenge in realizing a preemptive reconfiguration is to obtain valid *resumption points*, each consisting of a *resumption offset* in the bitstream and its associated Frame Address Register (FAR) and Frame Data Input Register (FDRI) (see Section 2.3.1) from which a preempted reconfiguration can safely be resumed. This section describes different types of resumption points and how they are obtained from a partial bitstream and details how they are utilized at runtime to resume preempted reconfigurations.

4.1.1 Finding Valid Resumption Points

As detailed in Section 2.3, configuration data are transferred into the FPGA configuration memory by data chunks that consist of numerous frames. A frame is the smallest transferable amount of configuration data. It is possible to preempt, but not start or resume, a reconfiguration in the middle of a frame. Furthermore, it would be unsafe to resume a reconfiguration in the middle of an operation sequence that instructs the reconfiguration port FSM to write FPGA-internal configuration registers, since the registers that had been written before the preemption could have been changed by the preempting configuration.

Depending on the offset where a reconfiguration is preempted, different procedures are necessary to resume it afterwards. Therefore, three types of resumption points that require different sequences of operations to resume a preempted reconfiguration are defied:

1. *Trivial resumption point*: beginning of a bitstream.

2. *Simple resumption point*: the resumption offset points to the end of a data chunk in the bitstream. In this case, reconfiguration can be resumed at the offset after

sending the Synchronization Header sequence that is part of the Common Header (see Section 2.3.2).

3. *Per-frame resumption point*: the resumption offset points to the end of a frame within a data chunk. Resuming a reconfiguration from per-frame resumption points requires determining the correct FAR and FDRI values as well as sending the correct Synchronization Header and Write Initialization sequences based on these values.

Figure 4.1 shows the placement of different resumption points within the partial bitstream. Resumption points are fixed in the bitstream thus it is not possible to modify their position by bitstream manipulation in order to improve the timing of the reconfiguration resumption. In the following, it is detailed how all types of resumption points can be found within a bitstream.

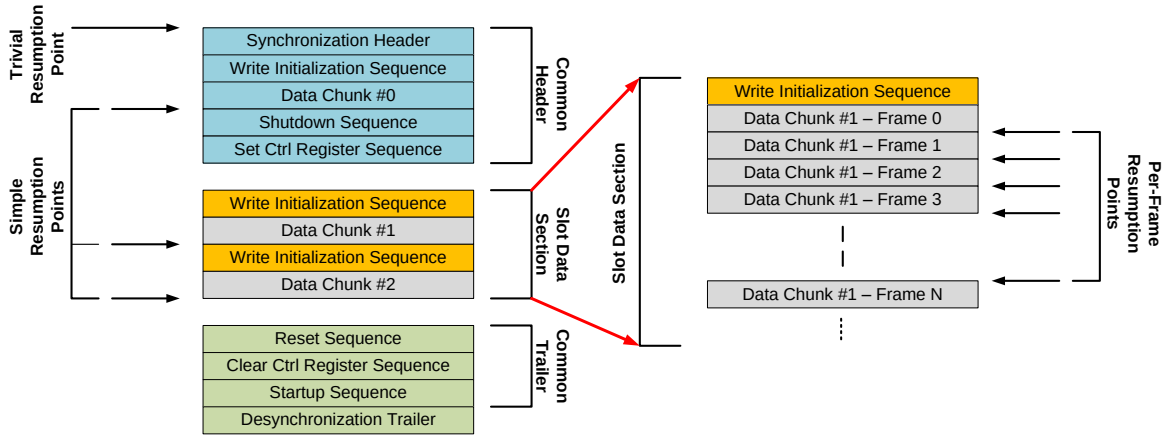


Figure 4.1: Position of Simple resumption points and Per-Frame resumption points inside a partial bitstream. A Per-Frame resumption point is fixed at the end of each frame within a data chunk. The resumption point related to the last frame, is a Simple resumption point.

Simple Resumption Points This type of resumption point can be found inside a partial bitstream by searching for FDRI operations that are part of write initialization sequences as shown in Figure 4.2. When skipping the following configuration data chunk using the word count of the operation (see Section 2.3.1), the resulting offset points to a Simple resumption point.

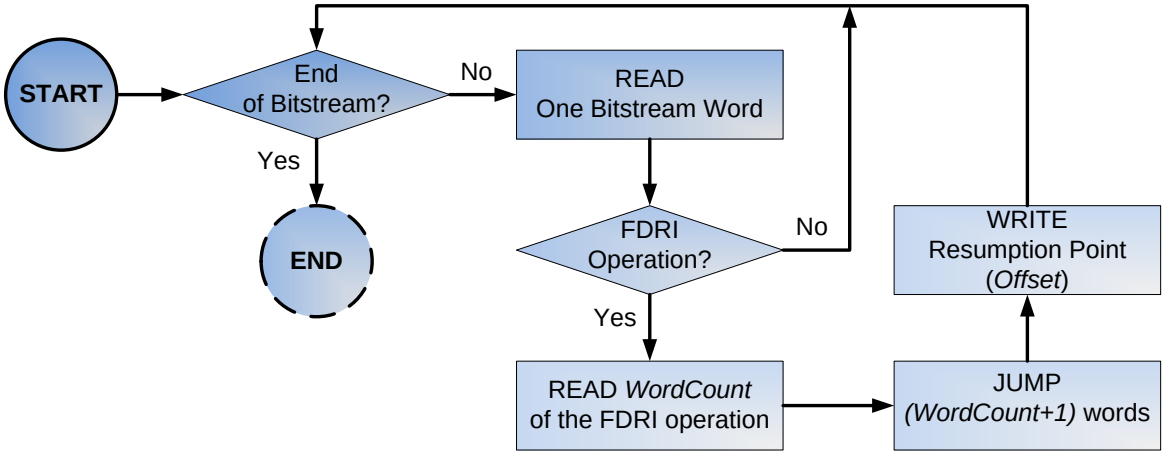


Figure 4.2: Flow-chart diagram to find Simple resumption points inside partial bitstreams.

When a reconfiguration is preempted, it needs to be resumed at the previous resumption point. Therefore, the amount of data between two resumption points directly contributes to the additional delay for the preempted reconfiguration. To guarantee a minimum progress for the preempted lower-priority task (in Section 5.1.2), this overhead should be as low as possible. Simple resumption points impair the performance of the system, because a great amount of progress can be lost for the lower-priority tasks. The distance between Simple resumption points depends on the size of the area that is reconfigured: in a minimum-sized slot on a Zynq 7z010 device (800 LUTs), the average distance between Simple resumption points is 6640 words (ca. 25% of the bitstream) and the biggest is 11744 words (about 44% of the bitstream, i.e.,

the worst case that needs to be considered in the analysis in Section 5.1.2). Therefore, a more fine-granular distribution of resumption points in the bitstream is beneficial.

Per-Frame Resumption Points within the Slot Data Section The Slot Data section is by far the biggest section of a partial bitstream (multiple ten thousands of words). Therefore, precedence has been given on finding Per-Frame resumption points in that section first (without considering the data chunk within the Common Header). In standard bitstreams, the FAR is initialized in the write initialization sequence and then incremented automatically by the reconfiguration port FSM after each frame, while writing the data chunk. Searching for Per-Frame resumption points requires knowing the FAR value increment after each frame, in order to determine the FAR value (FPGA-internal configuration memory address) that corresponds to a certain resumption point (offset within the bitstream). This information can be taken from debug bitstreams: inside the Slot Data section (but not in the data chunk within the Common Header), a FAR value increment of 0x01 (*frame increment*) is applied after each frame.

Figure 4.3 shows how to find Per-Frame resumption points. The first step is to find the beginning of each data chunk (FDRI-write operation inside the Write Initialization sequence) and its starting FAR value. Then, a resumption point can be found at every frame of configuration data, associating its offset with the calculated FAR value and an FDRI value. The correct FAR value for per-frame resumption points within the slot data section is calculated by adding a frame increment for each frame to the FAR starting value. The FDRI value is fixed to the number of words in a frame (101).

Using Per-Frame resumption points in addition to Simple resumption points can reduce the distance between resumption points drastically. However, the worst-case distance between resumption points (that needs to be considered for real-time analyses in Section 5.1.2) is now determined by Data Chunk #0 within the Common Header. This

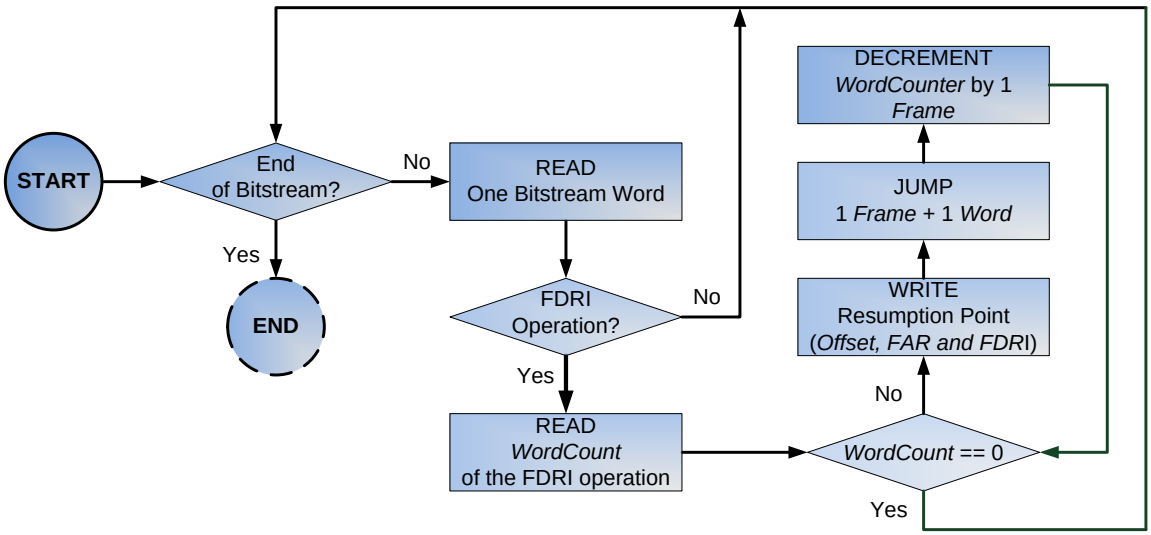


Figure 4.3: Flow-chart diagram that shows how to find Per-Frame resumption points inside the Slot Data section of a partial bitstream. The FAR increment in this section is 0x01.

data chunk requires additional steps to determine the FAR increments, as detailed in the following. As mentioned in Section 2.3.2, the size of Data Chunk #0 is device-dependent and grows proportionally with the size of the FPGA.

Per-Frame Resumption Points within the Common Header Searching for Per-Frame resumption points inside Data Chunk #0 within the Common Header requires the knowledge of its internal device-dependent structure. In contrast to all other data chunks, Data Chunk #0 is divided into smaller *sub-chunks* and each sub-chunk has starting and ending FAR values that are not observable in a standard bitstream. The number of sub-chunks and their starting and ending FAR values depend on the targeted FPGA only, but not on the resources and dimensions of the reconfigurable slot that should be configured.

The structure, FAR values, and FAR increments of Data Chunk #0 can be obtained

by analyzing a debug bitstream for the targeted device. As discussed in Section 2.3.2, the debug bitstream loads each frame individually, writing the FAR and the FDRI value after each frame. Therefore, the internal partitioning of Data Chunk #0 into several sub-chunks can be inferred by inspecting the FAR values in the debug bitstream: a discontinuity in FAR values identifies the beginning of a new sub-chunk. Within the same sub-chunk, FAR values are contiguous. The first FAR write identifies the beginning of the first sub-chunk and every FAR discontinuity provides the ending value of the current sub-chunk and the starting value of the next sub-chunk. Figure 4.4 shows the structure of Data Chunk #0 for a partial bitstream targeting the Xilinx Zynq-7010 FPGA.

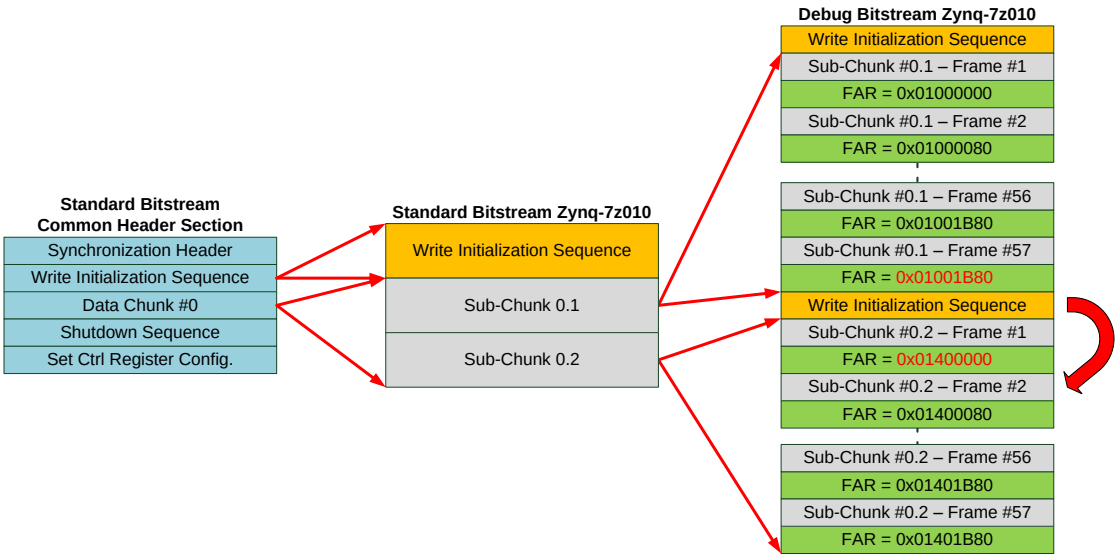


Figure 4.4: Detailed structure of Data Chunk #0. The FAR increment is 0x80 (column increment) and each sub-chunk has its starting and ending FAR value. Highlighted in red the discontinuity of the FAR value between two different sub-chunks.

The process for finding Per-Frame resumption points in Data Chunk #0 is similar to finding per-frame resumption points in the slot data section (see Section 4.1.1 and Figure 4.3). However, the FAR increment after each frame is a *column increment* (0x80)

and not a single-frame increment (0x01). The information about the end of each sub-chunk and the correct FAR values are read from the debug bitstream.

By introducing Per-Frame resumption points in Data Chunk #0, the maximum gap between two resumption points can be reduced to a single frame (101 words), hence at most the configuration progress of a single frame is lost when a reconfiguration is preempted.

In the following, the software interface, the designed custom reconfiguration controller, and the way they interact to realize preemptive reconfiguration under real-time constraints by utilizing resumption points (as detailed in the previous section) are described.

4.1.2 Software Interface for Preemptive Reconfiguration

Each task that wants to reconfigure a reconfigurable slot sends a request to the reconfiguration driver, which provides a unified software interface for runtime reconfiguration for all tasks. Then, the driver handles the reconfiguration request by sending *commands* to the *reconfiguration controller* that translates these commands into signals for the reconfiguration port, and initiates the transfer of configuration data from main memory (or controller-internal SRAM) to the reconfiguration port. The purpose of the reconfiguration controller is to alleviate the CPU from managing the reconfiguration port and keeping track of ongoing reconfiguration requests. A reconfiguration request to the driver specifies the requesting task priority, ID and bitstream's memory address (i.e., the address in memory where the bitstream has been stored). The task priority is used by the driver to determine whether the request has priority over a running reconfiguration (if any). The driver compares the task priority with the priority of the current reconfiguration (stored in a hardware register of the reconfiguration controller)

to decide whether to preempt or not. If the requesting task has the same priority of the current reconfiguration, the driver sends the reconfiguration request only if the requested slot is free. In case the requested slot is busy, the driver returns an error to the user task which can decide to proceed in software or ask again for hardware acceleration. The task ID from the reconfiguration request is used to resume the respective task when the reconfiguration has finished. As soon as a task sends a reconfiguration request, it self-suspends. Additional information can be provided to execute the reconfiguration request, e.g., the reconfiguration controller can be configured to send an interrupt to the CPU in case of a reconfiguration error.

Tasks are unaware of reconfiguration requests from other tasks. The driver translates reconfiguration requests into commands to the reconfiguration controller, while possibly preempting and resuming reconfigurations such that the reconfiguration request with the highest priority is being processed at each point in time. This leads to the following cases that the driver has to handle:

1. *No ongoing reconfiguration*. In case no reconfiguration is currently being performed, the driver translates the reconfiguration request into commands that are sent to the reconfiguration controller and then waits for a new request.

2. *Ongoing reconfiguration*. In case there is a reconfiguration currently being performed, the driver compares the priority of the running reconfiguration request and the new reconfiguration request to determine whether to abort the current reconfiguration to start processing the new request, or to enqueue the new request after the running reconfiguration request. When a task of higher priority requests a reconfiguration while a task of lower priority currently occupies the reconfiguration port, the lower-priority reconfiguration is preempted. To preempt a reconfiguration, the driver aborts the current reconfiguration and determines

how far the lower-priority reconfiguration has proceeded (as an offset in the bitstream). After that, the resumption point for the lower-priority reconfiguration request that is closest to the determined offset (and that was already passed during reconfiguration) is searched in a list of all resumption points for the respective bitstream. Then, the higher-priority request is enqueued. Afterwards, depending on the resumption point, the correct synchronization (see Section 4.1.1) and the lower-priority reconfiguration are enqueued to proceed from the resumption point. Once the higher-priority reconfiguration has finished, the reconfiguration controller automatically resumes the lower-priority reconfiguration (assuming no other higher-priority task requests a reconfiguration), as it is detailed in the following section.

4.1.3 Preemptive Reconfiguration Controller

The reconfiguration driver handles the tasks' reconfiguration requests by translating them into *commands* that are sent to the custom reconfiguration controller over the system bus (ARM AMBA AXI on the Xilinx Zynq devices). The reconfiguration controller processes commands internally using an FSM, based on the command-based reconfiguration queue (CoRQ) [57] that guarantees a fixed latency for each command, and was extended by commands that enable the reconfiguration preemption. The only command which does not guarantee a fixed latency is the one that triggers the bitstream transfer from the main memory which is connected on a multi-master bus. The goal of the designed reconfiguration controller is to provide a high-level interface to perform preemptive reconfigurations for tasks with different priorities.

Figure 4.5 shows an overview of the reconfiguration controller. It communicates to the system using AXI interfaces: one master interface is used by the controller to

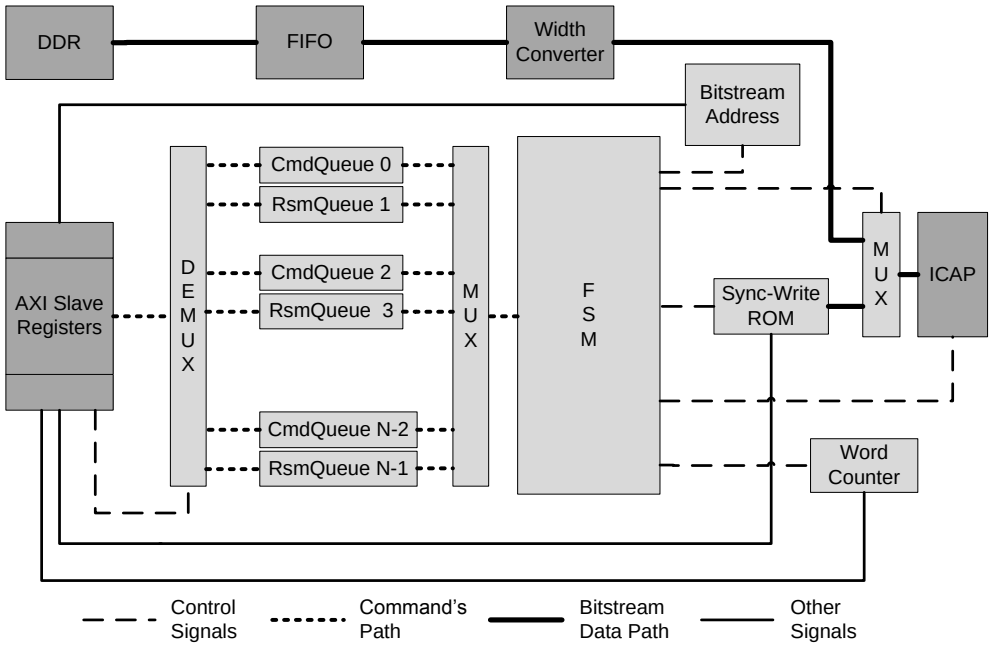


Figure 4.5: Detailed block diagram of the custom reconfiguration controller. Darker blocks are Xilinx generated IP (AXI interface, FIFO and Width Converter) or Xilinx hard IP (ICAP and DDR).

fetch bitstream data from main memory and one slave interface is used to connect the controller to an AXI bus from which the controller can accept commands, e.g., sent by the ARM CPU on Zynq devices. In particular, the AXI slave interface allows to control the input demux to write into a single queue, while the internal logic controls the output mux in order to let the FSM process commands from the highest priority, non-empty queue. It integrates the Internal Configuration Access Port (ICAP) as the reconfiguration port on Xilinx devices.

In addition to the FSM, the reconfiguration controller provides a configurable number of FIFO queues. For each task priority supported by the system, two queues are instantiated: (i) a *command queue* that accepts any FSM commands from tasks of the respective priority and (ii) a *resume queue* that stores only FSM commands to resume reconfigurations from resumption points (see Section 4.1.1). The priority of the queues is strictly ordered: First by the task priorities, and within a single task priority the resume queue has a higher priority than the command queue (i.e., there are two queue priorities for each task priority). In other words, preempted reconfigurations have a higher priority than the following reconfiguration requests from tasks of the same task priority. The reconfiguration controller is able to manage 2 billion couples of queues (32-bit addressing) which reflects on the software side with an equal number of priority levels.

The FSM is the core of the reconfiguration controller and it fetches, decodes and executes commands from the highest-priority non-empty queue (either command or resume queue). Based on the commands from the queue, the FSM controls the FPGA reconfiguration port. The FSM processes 32-bit commands of three different formats as shown in Figure 4.6. Similar to instructions in a CPU, these formats enable commands with different amounts of *data* and optionally bit fields for *settings*.

Commands are either executed immediately or enqueued into one of the internal

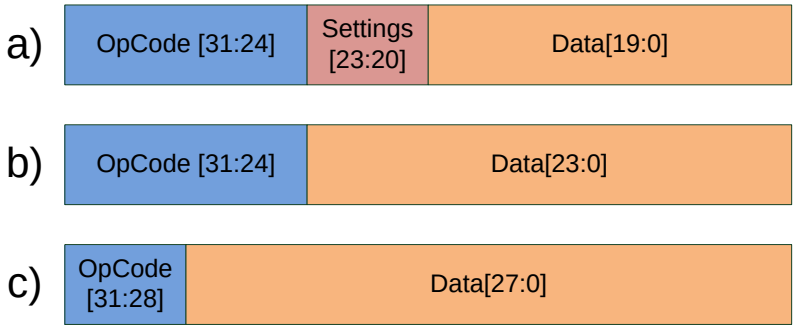


Figure 4.6: Commands for the reconfiguration controller can have three different formats depending on the amount of data and settings required by commands.

FIFO queues (denoted as *immediate* or *queueable* commands). Immediate commands are used to control the FSM itself (e.g., pause/resume processing enqueued commands) and abort a running reconfiguration. Queueable commands relieve the CPU from managing reconfigurations, i.e., they configure bitstreams (from internal or external memory) and notify the CPU once reconfiguration have finished. The commands that are utilized for preemptive reconfiguration are listed in the following and Table 4.1 shows their latencies (in clock cycles).

- `setBaseAddress`: Commands can have, at most, 27-bit for data, but some of them require 32-bit addresses, i.e., `configureBitstream`, `setFAR` and `setFDRI` (detailed below). `setBaseAddress` is used to set an offset that is combined with the address bits supplied by other commands to obtain 32-bit addresses.

- `configureBitstreamInt` / `configureBitstreamExt`: Starts transfer of the bitstream to the reconfiguration port from the memory address specified by 20 bits inside the command combined with 12 bit set prior via the `setBaseAddress` command. It is possible to decide the storage source of the bitstream: it can be fetched from the controller-internal memory, which guarantees a fixed bandwidth, or

from the main memory.

- `abortReconfiguration`: This command will abort the ongoing reconfiguration, stopping the data transfer and resetting the reconfiguration port. A flag allows to choose whether processing of further commands should be paused afterwards or not.

- `setFAR`: Updates an operation sequence with a new FAR value that reflects a previously selected resumption point. The sequence to update is stored in the reconfiguration controller and sent to the reconfiguration port to resume a preempted reconfiguration (Synchronization Header and a Write Initialization sequence, see Section 2.3.2) This command must be preceded by `setBaseAddress`.

- `setFDRI`: Similar to `setFAR`, this command updates the operation sequence that is sent to the reconfiguration port to resume a preempted reconfiguration with a new FDRI value. This command must be preceded by `setBaseAddress`.

- `sendSyncronization`: This command sends the operation sequence to resume a preempted reconfiguration. A flag is used to determine whether the whole sequence should be sent (for resuming per-frame resumption points, see Section 4.1.1) or the Synchronization Header only (for resuming simple resumption points). Before issuing this command, FAR and FDRI must have been set using `setFAR` and `setFDRI`.

- `resumeFSM`: If the FSM was paused (e.g., by `abortReconfiguration`), this command resumes it.

The reconfiguration controller records the state of an ongoing reconfiguration. *Bitstream Address* and *Word Counter* are registers that are used to store the base address of the

Table 4.1: Reconfiguration Controller Commands.

Command	**Im**mediate/**Qu**eueable	Latency
`configureBitstreamExt`	Qu	—
`configureBitstreamInt`	Qu	$9 + \lceil B/4 \rceil$
`setBaseAddress`	Qu	3
`abortReconfiguration`	Im	7
`setFAR`	Qu	3
`setFDRI`	Qu	3
`sendSyncronization`	Qu	470
`resumeFSM`	Im	3

B - size of bitstream [byte]

currently reconfigured bitstream and determine the offset of the currently transferred bitstream word. These two values are used during preemption to determine the resumption point. In particular, the bitstream base address is used as a bitstream ID, to target the right set of resumption points; while the word counter value is used as an offset and compared with all pre-calculated resumption offsets to find the closest accomplished resumption point (details in Section 4.1.4).

As explained in Chapter 2, resuming a reconfiguration means resume the writing to the portion of the FPGA configuration memory associated with a specific reconfigurable slot and does not implies the run-time modification of the bitstream. Therefore, each time a reconfiguration should be resumed from a specific resumption point, the reconfiguration port needs to be synchronized first using the Synchronization Header (and possibly the Write Initialization). The reconfiguration controller includes a *Sync-Write ROM*, a memory that contains the Synchronization Header and Write Initialization

sequences. Two words inside the Write Initialization sequence are programmable: the FAR and FDRI values that define a specific resumption point (set using `setFAR` and `setFDRI`). Depending on the type of resumption point, only the Synchronization Header (simple resumption points) or both Synchronization Header and Write Initialization sequences (per-frame resumption points) are sent to the reconfiguration interface.

Table 4.2 shows the register map of the reconfiguration controller and the description of each register follows.

Register Offset	**Register Name**	**Size**	**Read/Write**
0x00	Status Register	64bit	R
0x08	Word Counter	32bit	R
0x0C	Bitstream Address	32bit	R
0x10	Command's Length Register	32bit	R
0x14	Time-stamp Register	64bit	R
0x1C	Input Demux Control Register	32bit	R/W
0x20	Command Register	32bit	R/W
0x24	Current/Last-ran Command Register	32bit	R

Table 4.2: Memory mapping of all registers accessible by the AXI interface of the reconfiguration controller.

Status Register This 64-bit register records the state of each components inside the reconfiguration controller. In particular it is possible to monitor the state of the ICAP port and check whether it is busy or free. Moreover, the status register records the id of the currently active queue (which reflects the priority supported by the system) and saves its status. Finally, it reports the state of the reconfiguration controller's FSM.

Word Counter During reconfigurations, it holds the number of bitstream's words already sent to the ICAP.

Bitstream Address This register stores the memory address where the currently reconfiguring is located. This address helps to uniquely identify the bitstream.

Command's Length Register Duration, in terms of clock cycles, of the last-ran command. This register has profiling and characterization purposes only.

Time-stamp Register The reconfiguration controller contains a 64-bit, free-running timer. Each time a command is executed by the FSM, this register is updated with the timer's value. This register has profiling and characterization purposes only.

Input Demux Control Register This register allows to control the input demux thus addressing a specific queue to write to.

Command Register After the Input Demux Control Register has been configured, the right queue has been chosen and it is possible to write a command in the Command Register in order to fill the selected queue.

Current/Last-ran Command Register Every time a new command is executed by the state machine, the same command is loaded inside this register. As this register is accessible by the software side, the software driver can always be aware of the currently running command. This feature is particularly useful when checking if a reconfiguration is currently being performed.

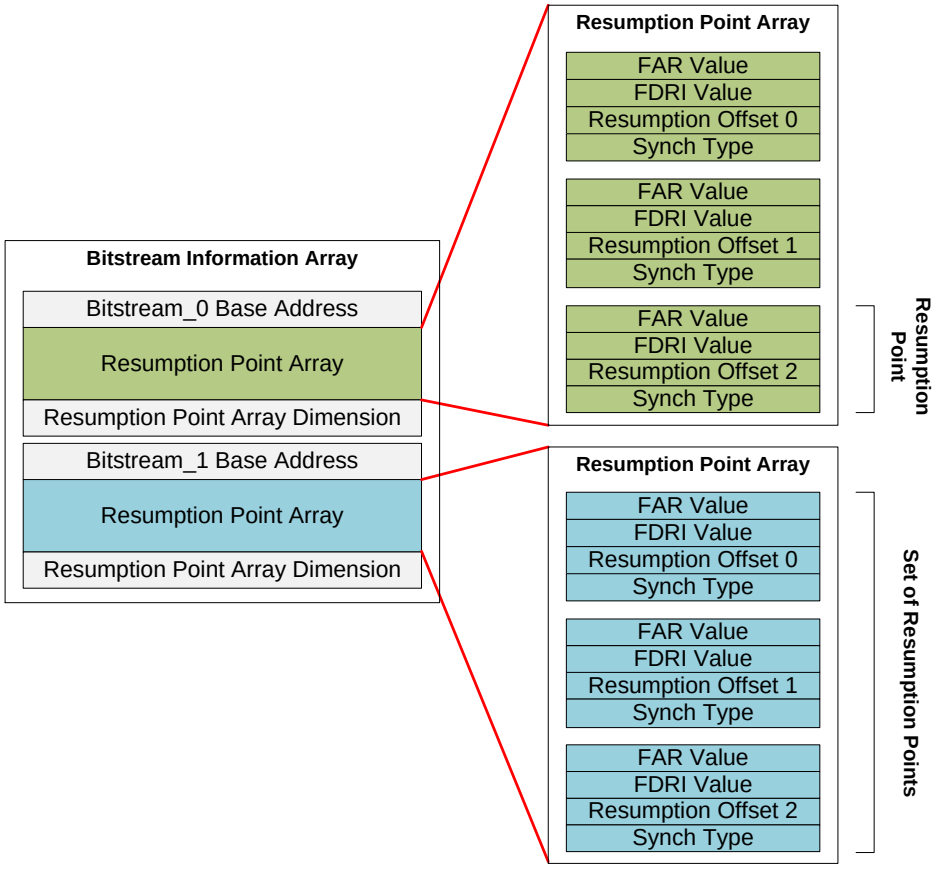


Figure 4.7: Data structure for two bitstreams with three resumption points each. The *Bitstream Information Array* is indexed using the Bitstream Address value read from the hardware, while the Word Counter value is used to identify the correct resumption point within the *Resumption Point Array*. The *Synch Type* field specifies the type of resumption point.

4.1.4 Hardware/Software Integration

To trigger a reconfiguration, the reconfiguration driver enqueues a `setBaseAddress` and a `configureBitstream` command into the command queue that is determined by the task priority. As soon as one of the queues signals that it is non-empty, the reconfiguration controller FSM starts fetching and executing its commands.

To preempt an ongoing reconfiguration, the reconfiguration driver sends an (immediate) `abortReconfiguration` command and reads the state of the aborted reconfiguration to find the correct resumption point. In particular, Word Counter and Bitstream Address values are read from the hardware controller. Since the application could have multiple reconfigurable slots and partial bitstreams, a method to address the respective set of resumption points and the specific resumption point of the preempted reconfiguration is required. Figure 4.7 shows the data structure used to select the correct resumption point. The address from which the bitstream was fetched in the preempted reconfiguration (pointing to main memory or controller-internal memory) is used as an identification method to target the set of resumption points related to the preempted reconfiguration. Moreover, the Word Counter value is used as an offset and compared with all resumption offsets within each resumption point, in order to find the nearest and already passed one. Finally, resuming a reconfiguration requires different commands, depending on the type of resumption point (Figure 4.8):

- Trivial resumption points: In this case, the reconfiguration driver enqueues a `setBaseAddress` and a `configureBitstream` command. There is no need for the reconfiguration controller to send a Synchronization Header sequence, as the reconfiguration simply resumes from the beginning of the bitstream.

- Simple resumption points: The reconfiguration driver enqueues a `sendSynchronization` command specifying to send a Synchronization Header sequence only,

as this type of resumption points already contain a Write Initialization sequence. Then, a `setBaseAddress` and a `configureBitstream` command are enqueued to resume the reconfiguration.

- Per-Frame resumption points: In addition to the Synchronization Header sequence, this type of resumption point needs a Write Initialization sequence. The reconfiguration driver enqueues a `setFAR` and a `setFDRI` command (both have to be preceded by a `setBaseAddress`) to set the FAR and FDRI value taken from the selected resumption point. Then a `sendSyncronization` command (that specifies to send both Synchronization Header and Write Initialization sequence), a `setBaseAddress` and a `configureBitstream` command are enqueued.

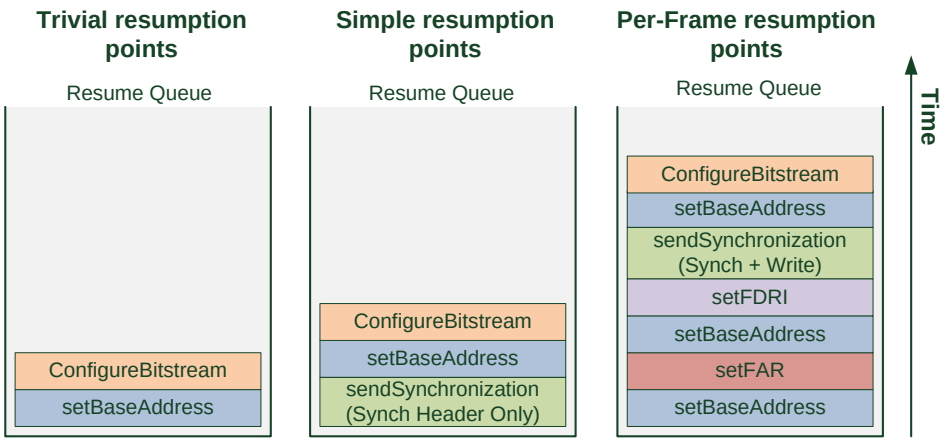


Figure 4.8: Command sequence to resume a reconfiguration from Trivial, Simple and Per-Frame resumption points.

4.2 Memory Protection and Budgeting Unit

As introduced in Chapter 1, partial reconfiugration allows the user to reconfigure a portion of the FPGA at runtime, while the remainder of the device continues to

operate. Therefore, it is possible to combine partial reconfiguration with hardware acceleration, reconfiguring portions of the FPGA with custom hardware accelerators while the application is running.

Those accelerators need to communicate with the CPU in order to exchange data and information and this communication is usually performed through a shared-memory approach. As already detailed in Section 3.2, the shared memory approach has some drawbacks like Memory Corruption and Bus Chocking.

To protect the main memory from illegal transactions and preventing hardware accelerators from choking the system bus, approaches which are usually exploited by software systems to solve similar problems have been used. In particular, a Memory Protection and Budgeting Unit (MPBU) has been developed in order to solves Memory Corruption and Bus Chocking problems, allowing to set a precise bandwidth for each hadrware module that needs to access the main memory. The developed MPBU does not affect the communication performance and the reconfiguration time.

In particular, this section describes the implementation of a custom, user-programmable MPBU to be placed between the hardware accelerator and the main memory as Figure 4.9 shows.

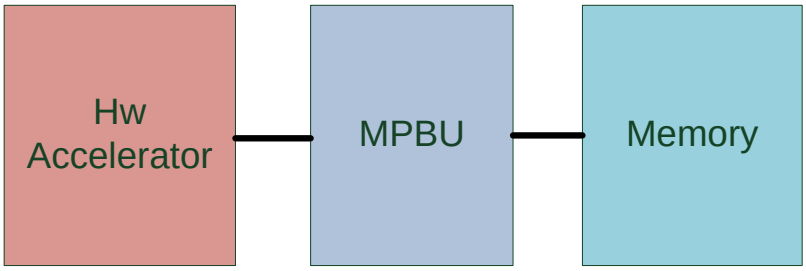


Figure 4.9: Memory Protection and Budgeting Unit placement. The MPBU must be placed between the hardware accelerator and the memory as it controls the data traffic to the memory.

4.2.1 Hardware Design and Functionalities

The MPBU provides memory protection and budgeting control for AXI4 and AXI4-Lite peripheral. As Figure 4.10 shows, the MPBU has one AXI master, one AXI slave interface and a custom configuration interface. The AXI slave port connects to the external master and is internally connected, through a decoupler, to the AXI master port. Therefore it is possible to snoop the state of all bus' signals and act consequently.

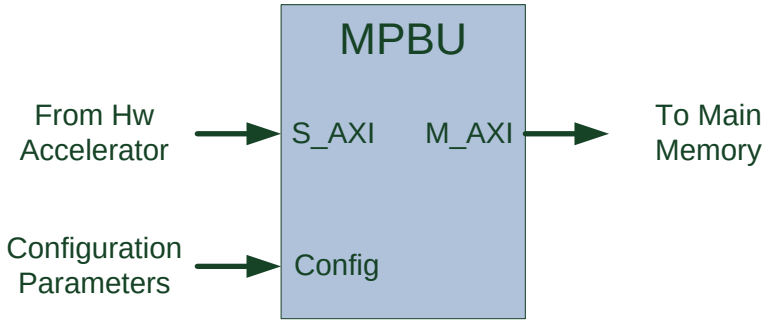


Figure 4.10: MPBU interface organization. Each MPBU has one AXI master and one AXI slave interface and a custom configuration interface.

The custom configuration interface allows to enable and disable the MPBU and send it in configuration mode where it is possible to set transaction budget, budget refreshing period and memory limits. Moreover, it allows to enable the MPBU or send it in configuration mode, thus allowing to configure the MPBU. At compiling time it is possible to define the number of memory buffers that need to be protected and, through the configuration interface, set the limits of each buffer individually.

As the configuration interface is not compliant with the AXI standard, another IP called *MPBU Controller* has been designed to act as a bridge and allowing to configure the MPBU through the AXI bus.

It has been preferred to separate the MPBU from its controller in order to overcome

a limit of the development tool which restricts the maximum number of AXI interfaces, for one each custom IP, to 16. Therefore, this approach allows to connect up to 16 different MPBUs to a single MPBU controller. If more MPBUs are necessary, it is possible to use multiple MPBU Controllers.

The MPBU has to be placed between the master (e.g., a custom hardware accelerator) and the slave (e.g. the system memory) and acts as a gate, decoupling the master from the slave if necessary.

In particular, as the master is allowed to make a number of transactions equal to its transaction budget, if it expires its quota or in case of an illegal master transaction, the MPBU gates all the *valid* and *ready* signals of the master's bus. Therefore, the master is decoupled from the slave and no transactions can occur until the transaction budget has been refreshed or the illegal memory transaction has been removed from the master bus.

In order to be more general, it is assumed that an AXI4-Lite read/write can be seen as an AXI4 read/write burst of one data word. In the following will be shown the implemented solutions used to achieve those functionalities.

- *Memory Protection*. It consists of an address check. The MPBU constantly compares the actual value on the Read/Write Address Channel with two user-configurable limits: *base address* and *maximum size* which respectively represent the starting memory address of the shared buffer and its size. In particular, checks if the master's transaction fits into the shared buffer limits and, in case an illegal transaction is detected, it gates the communication and sends an interrupt request. The MPBU can safely handle eigth different shared buffers.

- *Budgeting Control*. The MPBU features a transaction counter which counts each read/write request and decouples the master from the slave if the first one

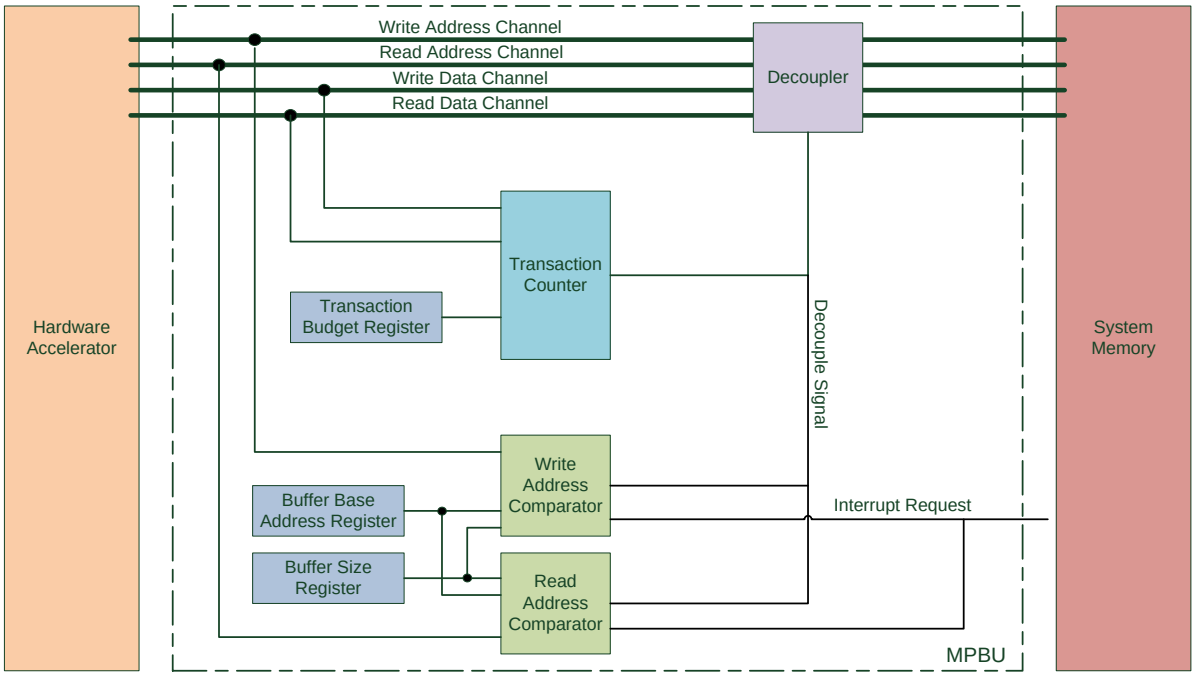


Figure 4.11: Internal block diagram of the MPBU. Black lines on top of the represent the AXI bus and its address and data channels. The communication which happens through the bus is gated according to the output of transaction counter, write address comparator and read address comparator.

exceeds its transaction budget. The budget will be automatically replaced after a user-configurable *budget period*.

Table 4.3 shows the register map of the MPBU Controller and the description of each register follows.

Register Offset	Register Name	Size	Read/Write
0x00	Control Register	32bit	R/W
0x08	Budget Register	32bit	R/W
0x0C	Budget Period Register	32bit	R/W
0x10	Priority Register	32bit	R/W
0x14	Memory Buffer Base Address Register	32bit	R/W
0x18	Memory Buffer Offset Register	32bit	R/W
0x1C	Memory Buffer Mux Register	32bit	R/W
0x28	Status Register	32bit	R
0x2C $\rightarrow$ 0x68	Budget Counter Register	32bit	R

Table 4.3: Memory mapping of all registers accessible by the AXI interface of the MPBU Controller.

Control Register The higher 16 bits of this register, control the possibility to enable or disable one specific MPBU. For example, writing 0x00010000 to this register enables the MPBU connected to channel 0. Lower 16 bits are used to put the related MPBU in configuration mode. Note that the MPBU will accept configuration data only if its configuration bit is set and its enable bit is cleared.

Budget Register When the MPBU is in configuration mode it will take the value held by this register as transaction budget.

Budget Period Register When the MPBU is in configuration mode it will take the value held by this register as budget refresh period (in FPGA clock cycles).

Priority Register When the MPBU is in configuration mode it will take the value held by this register as its priority level over the AXI bus.

Memory Buffer Base Address Register When the MPBU is in configuration mode it will take the value held by this register as the base address of the memory buffer selected by the Memory Buffer Mux Register. Therefore, it is possible to set a different base address for each memory buffer that will be shared between the CPU and the FPGA.

Memory Buffer Offset Register When the MPBU is in configuration mode it will take the value held by this register as the size of the memory buffer selected by the Memory Buffer Mux Register. Therefore, it is possible to set a different size for each memory buffer that will be shared between the CPU and the FPGA.

Memory Buffer Mux Register When the MPBU is in configuration mode the value held by this register will be used by the MPBU to associate the values written in the Memory Buffer Offset Register and Memory Buffer Base Address Register with the correct memory buffer.

Status Register The higher 16 bits of this register, hold the decoupling state of each MPBU. Each bit is related to one specific MPBU connected to the MPBU Controller. The lower 16 bits reflect the interrupt request status of each MPBU. If one MPBU detects an illegal memory transaction, the relative flag in the status register will be set and will stay set until the illegal transaction is removed from the bus.

Budget Counter Register There are 16 Budget Counter Register, one for each MPBU that can be connected to the controller. These registers hold the number of transactions done by the master. The maximum value that can be found in these registers equals the transaction budget.

4.2.2 Software Interface

In order to use the hardware from the Zynq's ARM cores (or another AXI-compatible processor), a low-level software layer has been designed. This software layer drives the MPBU Controller which consequently passes configuration information to all the MPBUs.

Through the control register of the MPBU Controller, each MPBU can be individually configured and enabled. In particular, the MPBU that needs to be reconfigured must be disabled (clearing the corresponding bit in the Control Register) otherwise the sent configuration will be discarded. When one MPBU is disabled it behaves like a pass-through, without affecting the communication.

The following code snippet shows a simple example to configure and enable the MPBU:

```
/*
 * MPBU Controller Initialization.
 *
 * This function initializes the MPBU Controller and
 * its software structure.
 */
MpbuCtrl_Initialize(&MpbuCtrl, MPBU_CONTROLLER_ID);

/*
 * Slot initialization.
```

```
 *
 * Initialization of each specific MPBU with all the required
 * configurations. This functions sets, for a specific MPBU
 * identified by MPBUId[i], the budget quota and refreshing
 * period, the MPBU priority and the associated memory buffer.
 */
for(i = 0; i < NUMBER_OF_MPBU_DEVICES; i++)
{
  MpbuCtrl_ConfigSlot(&MpbuCtrl, MPBUId[i],
                BudgetQuota[i],
                BudgetPeriod[i],
                MPBUPriority[i],
                BufferOffset[i],
                BufferAddress[i]);
}

/*
 * Slot enable.
 *
 * After configuration, the MPBU can be enabled.
 */
MpbuCtrl_EnableAllSlots(&MpbuCtrl);
```

Listing 4.1: Example code that uses the low-level driver to configure and enable the MPBUs.

These sections complete the description of how preemptive reconfiguration, memory protection and bus budgeting were realized. In the following chapter, preemptive partial reconfiguration and bus budgeting have been analyzed under real-time constraints.

Chapter 5

Analysis

5.1 Worst-Case Latency Analysis of Preemptive Reconfiguration

The realization of preemptive reconfiguration is based on CoRQ [57], a reconfiguration controller that guarantees worst-case latency bounds on the commands it processes, even on the reconfiguration itself when a memory bandwidth is guaranteed, e.g., when using FPGA-internal SRAM. Commands that have been implemented for preempting and resuming reconfigurations have guaranteed latencies, as listed in Table 4.1.

This allows to apply standard response time analysis techniques to determine the finish time of real-time tasks [58] and preemptive reconfigurations under real-time constraints. The focus of this chapter is to define worst-case bounds on the latency overhead that a higher-priority task experiences, when preempting a lower-priority task (WCET_{hp}). Furthermore, it has been determined an upper bound on the reconfiguration delay for lower-priority reconfiguration requests from the lower-priority task for a given worst-case interval of reconfiguration preemptions (rdelay_{lp}), and discuss under

which circumstances preemption enables us to guarantee a minimum progress for these reconfigurations.

5.1.1 Overhead for Preempting an Ongoing Reconfiguration

When a higher-priority task sends a reconfiguration request to the reconfiguration driver, the driver needs to preempt the ongoing reconfiguration (if any, as detailed in Section 4.1.4). Preemption entails operations performed by the driver itself (in software), as well as commands that are sent to the reconfiguration controller. The first command that is sent to the reconfiguration controller aborts the ongoing reconfiguration (while keeping information about its progress) and suspends processing of further commands (enqueued commands are kept in the queues). This way, operations performed by the driver and commands sent to the reconfiguration controller to perform the preemption are executed in sequence and therefore analyzed separately.

After `abortReconfiguration`, the driver determines the queue containing the commands that were being processed before the abort (the highest-priority non-empty queue) and then reads the last state of the aborted reconfiguration. Based on this state, the correct resumption point for the preempted reconfiguration is determined using binary search on the table of resumption points available for the respective bitstream. The corresponding commands to resume from this resumption point are then enqueued into the resume queue for the lower-priority task in the reconfiguration controller (see Section 4.1.4). Afterwards, the `configureBitstream` command for the higher-priority task is enqueued into its standard queue which it was empty before, otherwise it would not be possible that the reconfiguration of a lower-priority task is preempted. Therefore, the reconfiguration request of the higher-priority task is immediately processed, when the driver finally resumes processing of commands by the reconfiguration controller

FSM using `resumeFSM`. The reconfiguration driver has been implemented as a bare metal binary (without task handling, a full-featured implementation based on FreeRTOS is evaluated in Chapter 6) to get an estimate on the worst-case execution time (WCET) of the driver's operations when preempting an ongoing reconfiguration. The WCET estimate is obtained using the commercial WCET analyzer AbsInt aiT[1], based on an ARM Cortex R5F real-time CPU running at 600 MHz that is available in the recent Xilinx Zynq UltraScale+ generation. While the presented approach is not tied to a specific CPU, it is not possible to apply WCET analyzers to the ARM Cortex A9 that is available on the Xilinx Zynq-7000, because of its out-of-order pipeline. Eq. (5.1) shows the obtained result for the operations that the driver performs in software in WCET cycles (of the CPU).

$$\mathrm{WCET}_{hp\_\mathrm{driver}}(N_{\mathrm{rsp}}) = 18229 + \lfloor \log_2(N_{\mathrm{rsp}}) + 1 \rfloor \cdot 233 \tag{5.1}$$

Due to the binary search of the resumption point, the WCET depends on N_{rsp} (the number of resumption points that were determined statically), resulting in a "parametric" WCET bound [59].

In the following, is analyzed the worst-case latency of the commands processed by the reconfiguration controller to perform the preemption (as sent by the reconfiguration driver). These are: `abortReconfiguration` (suspending processing further commands) and `resumeFSM` (after enqueueing the higher-priority reconfiguration). Processing these commands takes $t_{\texttt{abortReconfiguration}} = 7$ and $t_{\texttt{resumeFSM}} = 3$ cycles respectively on the reconfigurable fabric [57]. Furthermore, the frequency factor between CPU and reconfigurable fabric c_{freq} needs to be accounted for, i.e., the CPU runs at a frequency c_{freq} times higher than the reconfiguration controller. Therefore, the WCET for processing

[1] `https://www.absint.com/`

the commands for preemption in CPU cycles is:

$$\mathrm{WCET}_{hp\_\mathrm{cmds}} = c_{\mathrm{freq}} \cdot (t_{\texttt{abortReconfiguration}} + t_{\texttt{resumeFSM}}) = c_{\mathrm{freq}} \cdot 10 \tag{5.2}$$

In the presented case, the reconfiguration controller runs at 100 MHz, while the ARM Cortex R5F runs at 600 MHz on the Xilinx Zynq UltraScale+, i.e., $c_{\mathrm{freq}} = 6$. In total, the following WCET in CPU cycles that the higher-priority task experiences as an overhead for its reconfiguration requests, in case it needs to preempt an ongoing reconfiguration from a lower-priority task, is obtained:

$$\mathrm{WCET}_{hp}(N_{\mathrm{rsp}}) = \mathrm{WCET}_{hp\_\mathrm{driver}}(N_{\mathrm{rsp}}) + \mathrm{WCET}_{hp\_\mathrm{cmds}} = 18289 + \lfloor \log_2(N_{\mathrm{rsp}}) + 1 \rfloor \cdot 233 \tag{5.3}$$

This is the worst-case overhead that needs to be considered for reconfiguration requests in response-time analyzes like presented in [6], when reconfiguration preemption should be allowed. Figure 5.1 shows how the overhead grows, depending on the number of resumption points available in the bitstream that is preempted. Note that the overhead is *guaranteed* to remain below 21100 cycles when providing resumption points for every second frame (every 202 words) for a bitstream that is smaller than 2.5 MiB (ca. the size of a full bitstream for the Xilinx Zynq-7010). This overhead corresponds to $\simeq$ 8% of the *observed* worst-case scheduling overhead in an ARM-based Linux system with real-time extensions [60]. Even when reconfiguring 25 % of the total resources of the Zynq-7010 at maximum reconfiguration bandwidth, the designed preemption approach would increase the delay by only 1.75 % in the worst case.

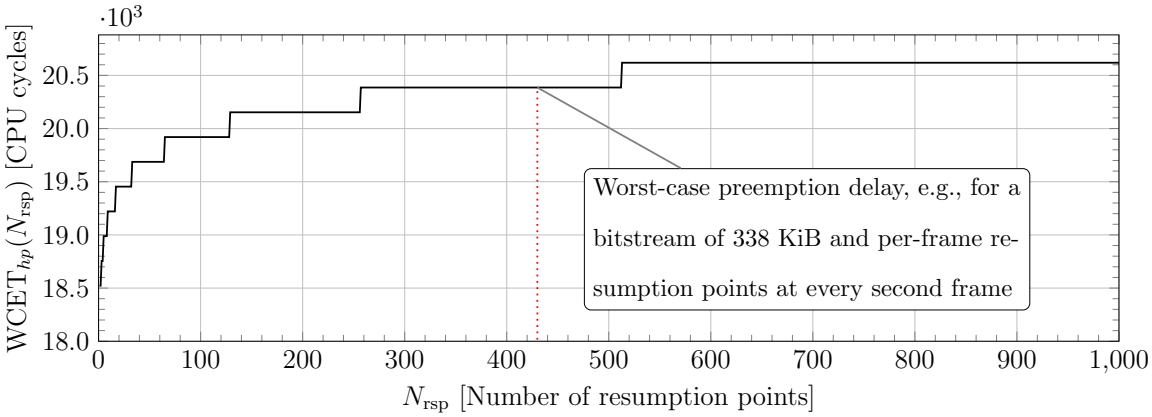


Figure 5.1: Worst-case execution time for preempting a reconfiguration ($\mathrm{WCET}_{hp}(N_{\mathrm{rsp}})$), depending on the number of resumption points (N_{rsp}) available in the bitstream that is preempted. E.g., a bitstream size of 338 KiB has been reported for several image filter accelerators (each) on the Zynq-7010 in [6], resulting in $N_{\mathrm{rsp}} \approx 430$ with per-frame resumption points at every second frame.

5.1.2 Reconfiguration Delay of Preempted Reconfigurations

This section focuses on providing an upper bound for the reconfiguration delay of lower-priority reconfiguration requests from a lower-priority task under reconfiguration preemptions (rdelay_{lp}). Tasks in the lower priority level encounter preemptions from higher priority levels only, therefore in the following is reasoned about preemptions encountered by a single lower-priority task. Using `configureBitstreamInt`, the reconfiguration controller sets up a memory transfer from FPGA-internal SRAM to the reconfiguration port (taking exactly 9 cycles), and then transfers one word (4 byte) of bitstream in each cycle (full utilization of the reconfiguration port). Reconfiguration controller, reconfiguration port and the FPGA-internal SRAM can be clocked at up to

100 MHz, resulting in a maximum reconfiguration bandwidth of 400 MB/s[2]. Given a bitstream of size B in bytes, a reconfiguration without any preemptions takes exactly $\text{rdelay}(B) = 9 + \lceil B/4 \rceil$ cycles on the reconfigurable fabric using the designed reconfiguration controller (details in [57]).

To determine the worst-case reconfiguration delay under preemptions, it is necessary to consider the number of preemptions that can occur during the reconfiguration. Each preemption creates overhead that prolongs the reconfiguration delay, creating the possibility for additional preemptions. To bound this effect, is applied an iterative process from response time analysis [58] as follows. Assuming the minimum distance between preemptions (due to reconfiguration requests from higher-priority tasks) is t_{preempt}, the following recursive sequence has been obtained:

$$\text{rdelay}_{lp}^{n+1}(B) = \text{rdelay}(B) + \underbrace{\left\lceil \frac{\text{rdelay}_{lp}^{n}(B)}{t_{\text{preempt}}} \right\rceil}_{\text{number of preemptions}} \cdot \underbrace{(t_{\text{rsp}} + t_{\text{sync}})}_{\text{overhead of being preempted}} , \tag{5.4}$$

The process starts with $\text{rdelay}_{lp}^{0}(B) = \text{rdelay}(B)$ and terminates when $\text{rdelay}_{lp}^{n+1}(B) = \text{rdelay}_{lp}^{n}(B)$. t_{rsp} is the maximum time that it takes to configure the difference between two resumption points (the maximum progress that is lost when a reconfiguration is preempted), e.g., $t_{\text{rsp}} = 101$ cycles using per-frame resumption points (see Section 4.1.1). $t_{\text{sync}} = 470$ cycles is the latency of the `sendSyncronization` command when processed by the reconfiguration controller. Then, is obtained the reconfiguration delay of a reconfiguration request from a lower-priority task under reconfiguration preemptions as

$$\text{rdelay}_{lp}(B) = \text{rdelay}_{lp}^{n}(B) + \left\lceil \frac{\text{rdelay}_{lp}^{n}(B)}{t_{\text{preempt}}} \right\rceil \cdot \text{waiting}_{hp} \tag{5.5}$$

where waiting_{hp} is the maximum time that the reconfiguration port is occupied by

[2]Considering $1MB = 1000000Byte$ and $1MiB = 1024 * 1024Byte$, results $(4 \cdot 10^8[Byte/s])/(1024^2[Byte/MiB]) = 381.4697265625MiB/s$

higher-priority tasks during a preemption. Note that t_{preempt} and waiting_{hp} need to be bounds for tasks from all higher priority levels (than the priority level of the task that is currently preempted), i.e., for more than two priority levels, this simple analysis will be imprecise. However, if more than two priority level are provided, it is possible to easily extend this analysis and integrate it into the response time analysis of the whole task set. In the following it is shown that a minimum progress for the lower-priority task can be guaranteed.

Guaranteeing Minimum Reconfiguration Progress under Preemptions

Intuitively a reconfiguration (that is subject to preemptions) advances, when the time windows of unpreempted reconfiguration are bigger than the latency overhead of being preempted. More formally, it is shown that minimum reconfiguration progress can be guaranteed when $t_{\text{preempt}} > t_{\text{rsp}} + t_{\text{sync}}$ (see Section 5.1.2). Minimum progress during reconfiguration is equivalent to a bound on the reconfiguration delay (see Eq. (5.5)). Assuming waiting_{hp} is bounded, it is therefore necessary to show that the iterative process of Eq. (5.4) converges.

Proposition 1 $t_{preempt} > t_{rsp} + t_{sync} \Rightarrow \exists n \in \mathbb{N} : rdelay_{lp}^{n+1}(B) = rdelay_{lp}^{n}(B)$

Proof: By induction, showing that $\text{rdelay}_{lp}^{n+1}(B) \leq (t_{\text{rsp}} + t_{\text{sync}} + 1) \cdot \text{rdelay}(B)$ and that the sequence is monotonically increasing. ■

Convergence of Eq. (5.4) immediately follows from proposition 1. Therefore, it is possible to guarantee a minimum progress for a reconfiguration that is subject to preemptions, when the time windows of unpreempted reconfiguration (t_{preempt}) are bigger than the latency overhead of being preempted ($t_{\text{rsp}} + t_{\text{sync}}$).

Figure 5.2 shows how $\text{rdelay}_{lp}(B)$ evolves for different values of t_{preempt} for a bitstream of $B = 338$ KiB and different granularities of placing resumption points.

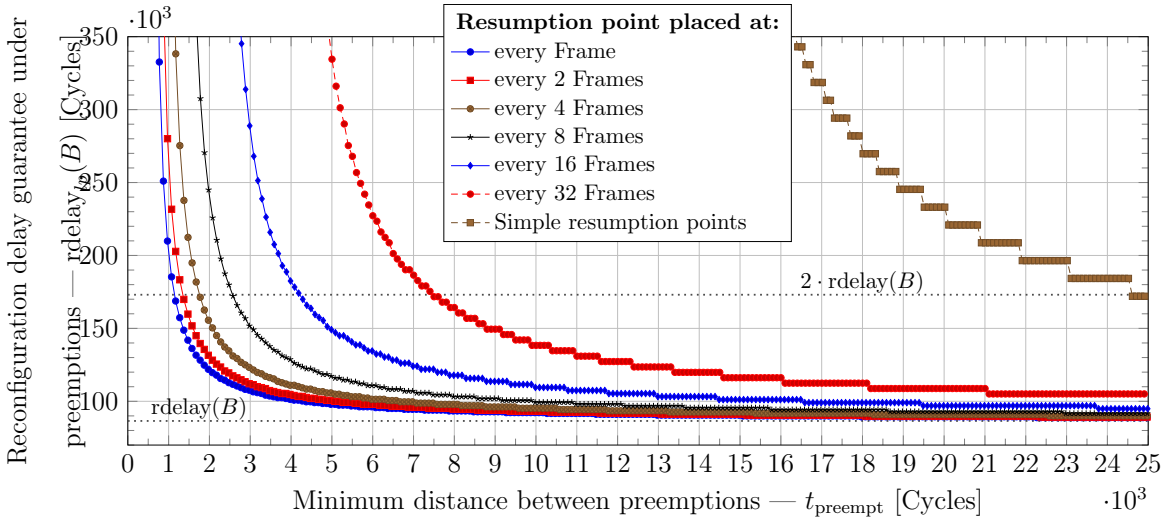


Figure 5.2: Reconfiguration delay guarantee ($\text{rdelay}_{lp}(338\,\text{KiB})$, with $\text{waiting}_{hp} = 0$) for different time windows of unpreempted reconfiguration and different resumption point types (resulting in different latency overheads of being preempted due to lost progress in reconfiguration (t_{rsp})).

waiting$_{hp}$ is set to 0, because the bound of how long a higher-priority task occupies the reconfiguration port has no impact on convergence of Eq. (5.4), and to focus on the overhead of preemption. It is observable that t_{preempt} as well as the granularity at which resumption points are provided for a certain bitstream have a considerable impact on whether the reconfiguration delay can be bounded as well as on the size of the obtainable bounds. Note that t_{preempt} is measured in cycles of the reconfiguration controller (running at 100 MHz in the presented case). When using Simple resumption points only, higher-priority tasks can preempt the lower-priority reconfiguration at most every 73,290 CPU Cycles ($t_{\text{preempt}} = 12,215$ cycles of the reconfiguration controller) on a 600 MHz CPU so that a minimum progress can be guaranteed. This bound is more than one magnitude lower for per-frame resumption points. Combining the results shown in Figure 5.1 and Figure 5.2, it can further be seen that there is a tradeoff between the minimum guaranteed bandwidth for reconfigurations of the lower-priority task and the WCET bound on performing preemptions for the higher-priority tasks. Reasonable tradeoffs can be determined for specific use cases by choosing a suitable granularity of placing resumption points. This was achieved by introducing per-frame resumption points in Section 4.1.1.

5.2 Worst-Case Analysis of the Budgeting Approach

As described in Section 4.2, the MPBU has been developed to realize Memory Protection and Bus Budgeting allowing for a safe and more predictable communication between the CPU and reconfigurable hardware modules.

Unfortunately, realizing a worst-case analysis which gives worst-case guarantees regarding the communication between the CPU and the FPGA (as has been done for preemptive reconfiguration), would be a quite hard task. The main reason resides in the

lack of documentation describing the deep functioning of Xilinx and ARM intellectual properties, i.e., the Xilinx interconnect IP and the ARM memory controller.

For example, Figure 5.3 shows a typical communication chain starting from the hardware module to the main memory. In the showed chain, it is possible to model the communication behavior and its timing until the interconnect IP, but beyond that limit there is no documentation that explains how the interconnect and the memory controller behave and handle conflicts. In particular, no information are available regarding the scheduling policy of transactions in case of interferences on the communication bus, neither for the interconnect IP nor for the memory controller.

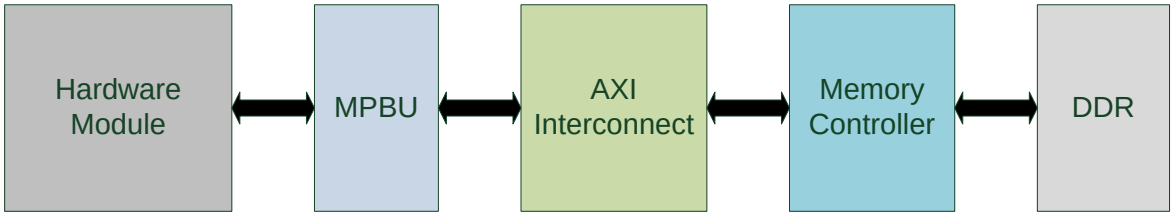


Figure 5.3

The documentation of the interconnect IP states that, with specific settings of the interconnect instance [61] its behavior is *fair*, thus all hardware modules connected to it have the same amount of bus bandwidth (this statement has been verified by performed experiments in Section 6.2.1). However, the way the memory controller handles conflicts on its input bus is still missing and it is not possible to gather worst-case parameters neither for the interconnect IP nor for the memory controller.

Those problems lead to the impossibility of realizing even a simple analysis because the most important information is missing, i.e., the worst-case amount of time for both the interconnect and the memory controller to complete a bus transaction. There is no documentation that provides such information.

Even if it is possible, through experiments, to empirically measure the longest time of a transaction over the bus, the analysis that results from it would be imprecise and not suitable for a real-time system.

Section 6.2 shows that, thanks to the MPBU modules, it is possible to guarantee at the interconnect port a fixed and upper-bounded bandwidth for each hardware module, and varying their average execution time.

Chapter 6

Experimental Evaluation

This Chapter shows the results of preemptive partial reconfiguration and MPBU evaluation giving the possibility to analyze an appraise their performance. They have been developed with the purpose of providing more performance, predictability, and a better analysis of real-time systems involving FPGAs. As already explained at the end of Chapter 3, this thesis gives the first, real hardware/software implementation to enable preemptive partial reconfiguration, memory protection, and bus budgeting. Therefore, as no results regarding similar architectures are available from the state-of-the-art, there are no means of comparison. Hence, both solutions have been separately evaluated to test and analyze their performance.

Both experimental evaluations of this work have been done on a Xilinx Zynq-7z010, featuring a dual-core ARM Cortex-A9, running at 600 MHz, and an Artix-7 FPGA on the same SoC [62]. This SoC uses the standard AXI bus interface as the communication medium between the FPGA and the processing system (ARM cores). The real-time operating system FreeRTOS[1] has been extended to support preemptive reconfiguration and MPBU: a reconfiguration driver has been added for the custom reconfiguration

[1] `http://www.freertos.org/`

controller, and the task scheduler is aware of pending reconfiguration requests to wakeup the respective tasks for completed reconfigurations. Moreover, the MPBU support has been integrated inside FreeRTOS in order to ease the use and configuration of each MPBU instantiated in the design.

6.1 Evaluation of Preemptive Partial Reconfiguration

The evaluation of preemptive reconfiguration utilizes two reconfigurable slots with different dimension and incorporating different type of resources: the bigger slot has 2,400 LUTs, 10 BlockRAM and 20 DSPs, its bitstream's size is $364KB$ and has a reconfiguration time of $932.35\mu s$; the smaller has 800 LUTs, 10 BlockRAM and no DSPs, its bitstream's size is $103KB$ and has a reconfiguration time of $265.59\mu s$. Since preempting a reconfiguration could leave part of the reconfigurable slot in an undefined state, each reconfigurable slot has a hardware decoupler that allows to decouple the slot during reconfiguration thus preventing spurious signals to affect the rest of the hardware. As explained in Section 4.1.3, it is possible to define at design time, the number of command and resume queues, and their size within the reconfiguration controller. In this evaluation, the reconfiguration controller contains 8 command and resume queues each, one (pair) for each FreeRTOS priority level (see Section 4.1.3). Each queue can store up to 128 pending reconfiguration commands. The reconfiguration controller's interrupt line and interrupt lines for each reconfigurable slot have been connected to the Zynq system. The reconfiguration controller's interrupt line is triggered as soon as a reconfiguration is completed, allowing the reconfiguration driver to resume the related task. Moreover, the interrupt line of each reconfigurable

slot is connected to the Zynq processing system allowing the software to be notified when the hardware execution of each accelerator configured in the slot ends. Figure 6.1 shows the block design of the evaluation system.

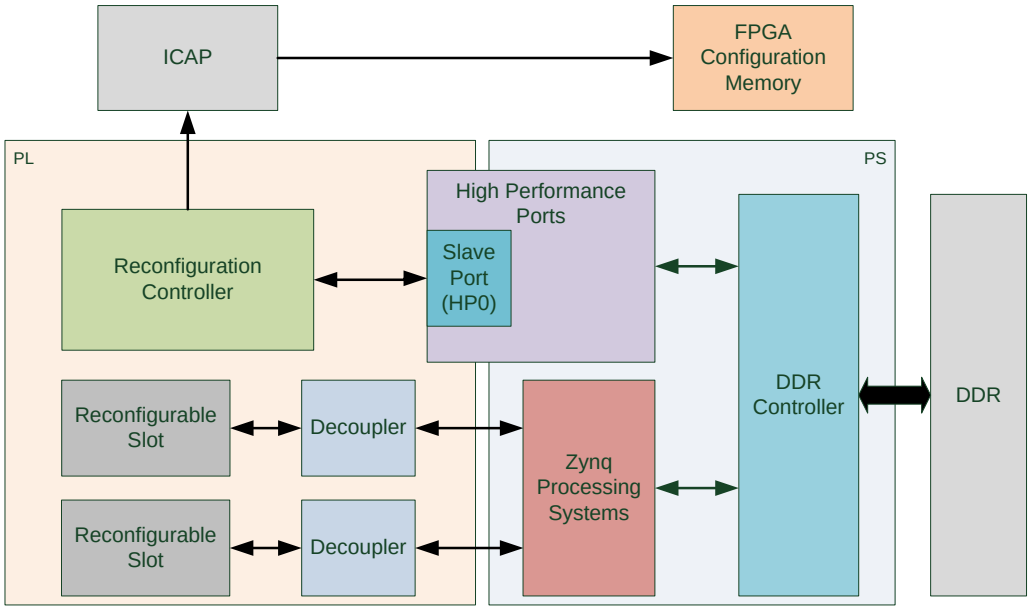


Figure 6.1: Block design of the evaluation system for preemptive reconfiguration. The left block named "PL" represent the Programmable Logic which contains reconfigurable slots, decouplers and the reconfiguration controller. The right block namend "PS" represent the Processing System which contains the dual-core ARM A9 and the DDR controller. PL and PS are able to exchange information through high performance ports.

The test application consists of two periodic *user tasks* and one *driver task* that handles all reconfiguration requests. Each user task has its own private reconfigurable slot that reconfigures multiple time during its execution. After a reconfiguration request, the user tasks self-suspend.

The driver task is the highest priority task in the system and the only task allowed

to communicate with the reconfiguration controller. It executes immediately after a reconfiguration request and checks whether the reconfiguration controller is idle or a reconfiguration is ongoing. If a lower-priority reconfiguration is ongoing, the driver handles the whole process of preempting the reconfiguration (see Section 4.1.4). In the performed experiments, only Simple resumption points have been used as they are enough to guarantee a forward progress for preempted FPGA reconfiguration.

Every time a reconfiguration is completed, the reconfiguration controller triggers an interrupt and the interrupt handler resumes the user task that requested the reconfiguration. In order to keep track of all reconfiguration requests and the awakening order of their related user task, a specific software structure of queues has been built. In particular, there are 8 software queues (one for each FreeRTOS priority level) storing the ID of the user task that asked for a reconfiguration. Every time a reconfiguration is completed, the reconfiguration controller triggers its interrupt and the related software routine scans the software queue structure to find the highest priority pending request and resume the related user task. After the reconfiguration of its reconfigurable slot, the user task feeds the data to the accelerator and self-suspends waiting for the accelerator to complete its execution.

6.1.1 Resource Utilization

Table 6.2a shows the resource utilization of the entire system while Table 6.2b shows the resource utilization of the reconfiguration controller. Results in Table 6.2a do not account for any logic inside the reconfigurable slots, i.e., configured accelerators would add to the resource utilization.

Utilization rates of the FPGA shown in Tables 6.2a and 6.2b refer to a Zynq-7z010 device which is the smallest Zynq-7000 device. E.g., on the next-bigger Zynq-7z015,

Resource	Utilized	Available	% Utilized
LUT	7252	17600	41.20%
LUTRAM	729	6000	12.15%
FF	8449	35200	24.00%
BRAM	12.50	60	20.83%
BUFG	10	32	31.25%

(a)

Resource	Utilized	Available	% Utilized
LUT	2006	17600	11.40%
LUTRAM	193	6000	3.22%
FF	2261	35200	6.42%
BRAM	5	60	8.33%
BUFG	8	32	25.00%

(b)

Table 6.1: a) Resource utilization of the whole hardware design. It consists of two reconfigurable slots and the designed reconfiguration controller that are connected to the ARM cores on the Xilinx Zynq. b) Resource utilization of the reconfiguration controller that enables preemptive reconfiguration. It supports 8 priority levels and up to 128 pending commands for each level. In both tables, percentages refer to a Zynq 7z010 chip.

only 15.7% of LUTs and 13.9% of BRAMs would be utilized by the whole hardware design. On the biggest Zynq-7000, the Zynq-7z100 less than 3% of the resources would be utilized, respectively. Despite its constrained resources, it is showed that preemptive reconfiguration can be realized on the low-end Zynq-7z010.

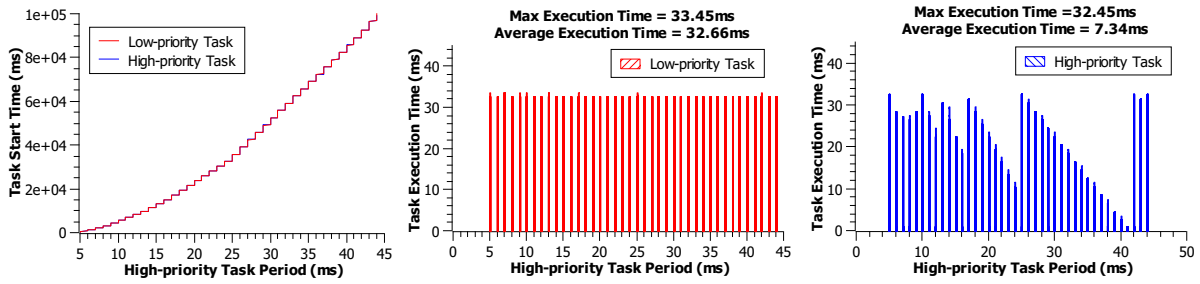
6.1.2 Maximum Observed Execution Time

Three experiments were performed that reflect the motivational examples explained in Section 3.1 and show the benefits of preemptive reconfiguration. Due to the used FPGA device, both reconfigurable slots are quite small and have small reconfiguration time, so small that would have been difficult, with only two tasks in the system, to clearly show problems of non-preemptive and benefits of preemptive reconfiguration. For this reason the reconfiguration time of each slot has been software-extended, associating one reconfiguration request with multiple physical reconfiguration.

In all the experiments, the low-priority task has a fixed period of $50ms$ and a reconfiguration time of $32.63ms$ while the high-priority task period is varied from $5ms$ to $44ms$ with a fixed reconfiguration time of $0.79ms$.

Response time analysis in real-time systems needs to provide guarantees for the worst-case, i.e., that reconfiguration requests from the low-priority and high-priority tasks are in conflict and that the higher-priority task needs to preempt the lower-priority reconfiguration. To provoke this case, the tasks perform frequent reconfigurations, but no additional computations.

Figure 6.2 shows configure-to-completion (Chapter 3, item a)), where the reconfiguration interface cannot be preempted and the ongoing reconfiguration has to be completed. In this scenario, both tasks can run concurrently, as Figure 6.2a shows, but the high-priority task is delayed due to priority inversion. As Figure 6.2c shows, the



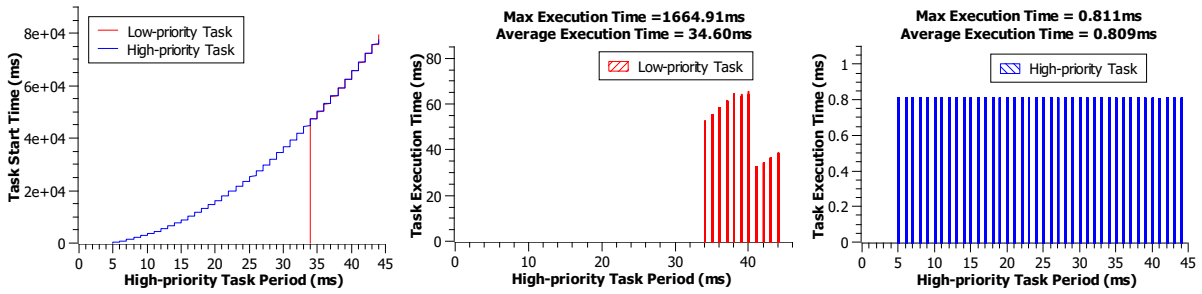
(a) (b) (c)

Figure 6.2: Priority inversion experiment. The ongoing reconfiguration can not be stopped and must be completed before starting another reconfiguration. (a) shows the start time of both tasks while the period of the high-priority task varies. (b) and (c) show the distribution of the maximum observed execution time of user tasks for each value of the high-priority task period.

maximum execution time of the high-priority task that has been measured is 32.45*ms*.

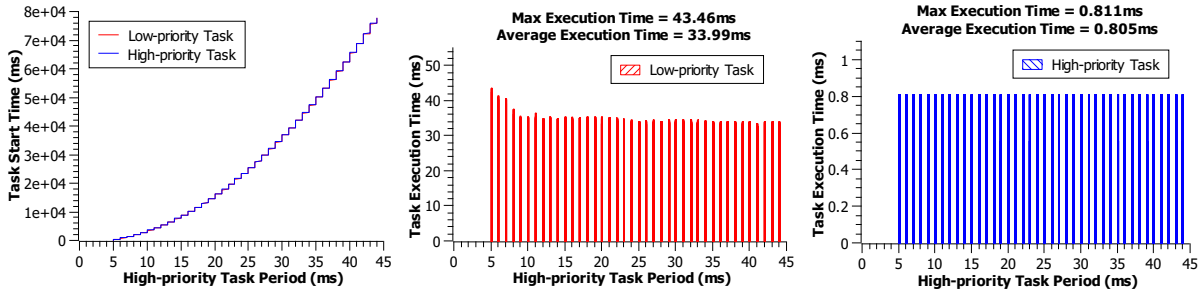
Figure 6.3 shows the abort approach (Chapter 3, item b)). This approach allows to abort the ongoing reconfiguration and restart it later from the beginning. Figure 6.3a shows the start time of both user tasks while the period of the high priority task varies. Both user tasks are periodic and the start time is the time where a task starts its routine from the beginning, and it is clearly visible that the low-priority task starves until the high-priority task period reaches 34*ms*. Then, the period of the high-priority task is large enough to allow the low-priority one to reconfigure the FPGA.

Figure 6.4 shows the preemption approach, i.e., it is possible to preempt and later resume the ongoing reconfiguration (Chapter 3, item c)) while keeping the reconfiguration progress. Both tasks benefit from preemptable reconfiguration because the low-priority task does not suffer from starvation while the high-priority task does not experience priority inversion. Therefore, as Figure 6.4a shows, both task can run



(a) (b) (c)

Figure 6.3: Abort experiment. The ongoing reconfiguration can be aborted and later restarted from the beginning. (a) shows the start time of both tasks while the period of the high-priority task varies. (b) and (c) show the distribution of the maximum observed execution time of user tasks for each value of the high-priority task period.



(a) (b) (c)

Figure 6.4: Preemption experiment. The ongoing reconfiguration can be preempted to serve an higher priority reconfiguration request and later resumed. (a) shows the start time of both tasks while the period of the high-priority task varies. (b) and (c) show the distribution of the maximum observed execution time of user tasks for each value of the high-priority task period.

	Configure-to-completion		Abort		Preemption	
	LP Task	HP Task	LP Task	HP Task	LP Task	HP Task
MOET	$33,46ms$	$2,45ms$	$1664,12ms$	$0,811ms$	$42,34ms$	$0,81ms$
AOET	$32,68ms$	$0,805ms$	$33,16ms$	$0,808ms$	$32,68ms$	$0,805ms$

Table 6.2: Summary of the Maximum Observed Execution Time (MOET) and Average Observed Execution Time (AOET) of both tasks.

concurrently on the system, reconfiguring the FPGA.

As all the experiments show, priority inversion increased the execution time of the high-priority tasks by more than 3 times and starvation prevented the low-priority task to complete its execution. Preemptable reconfiguration solved these problems by avoiding priority inversion and preventing the low-priority task to starve, allowing it to improve the utilization of the system.

Table 6.2 compares the most significant results from the experiments.

6.2 Evaluation of Memory Protection and Budgeting Unit

To extensively evaluate and analyze the effectiveness of the reservation mechanism enforced by the MPBU, it is necessary to have hardware module continuously performing transactions on the main bus.

This experimental evaluation, models a scenario where multiple high-performance hardware accelerators access a bus to share the same memory controller. Each accelerator acts like an active component or co-processor in the system being able to autonomously generate memory transactions to the main memory.

For the purpose of this evaluation, hardware accelerators have been implemented as custom hardware modules that act like a DMA[2] (i.e., copying a memory buffer from a location to another) because, from a bus and memory contention perspective, the only visible effect of the accelerators' activity are bus transactions being generated. Therefore, a DMA module can be considered as an "upper bound" of the behavior of an active accelerator since its only function is to move data at the maximum possible rate thus stressing the communication bus without being slowed by any possible internal processing.

Such modules can be programmed and controlled by the processor through a set of control registers where it is possible to specify the source and destination addresses. In this evaluation, all the hardware modules are programmed to move a fixed amount of data of $2MiB$ (they read and write $1MiB$).

Hardware System Description The evaluation of Memory Protection and Budgeting Unit utilizes four hardware modules connected, through a standard *AXI interconnect* IP [61], to one *high performance port* of the processing system. The interconnect IP is used in *performance mode* in order to maximize the performance of the infrastructure IP cores used within the interconnect instance. Specifically, the IP is configured in Shared-Address/Multiple-Data (SAMD) mode, packet-mode FIFOs are enabled on all the slave interfaces, and multiple outstanding transactions for each connected master and slave are allowed [61].

Between each module and the interconnect, an MPBU module is placed performing memory protection and bus budgeting. All the MPBU modules are managed by the MPBU controller which, in turn, is software programmable (see Section 4.2.1). As described in Section 4.2.1 the only programmable parameter of the MPBU controller,

[2]Direct Memory Access

which must be decided at compiling time, specifies the number of MPBUs that the user would like to connect to it: in this evaluation it has been configured to manage 4 MPBU.

From the PL-PS interconnection perspective, this hardware system represents a sort of "worst-case" scenario in which only a single Zynq's high performance port is used. This choice has been made to simulate a more realistic scenario which considers a heterogeneous SoC with a large FPGA where many hardware accelerators share a limited number of memory access ports.

Moreover, all the interrupt lines belonging to each MPBU have been connected to the Zynq system. When a hardware module performs an illegal memory access, the corresponding MPBU's interrupt line is set and the Zynq system can asynchronously handle the problem, removing the illegal condition from the system bus. Figure 6.5 shows the block diagram of the evaluation system used for MPBUs.

Resource Utilization Table 6.3 shows the resource utilization of both MPBU and MPBU controller individually and of the entire MPBU evaluation system.

Utilization rates of the FPGA shown in Table 6.3 refer to a Zynq-7z010 device which is the smallest Zynq-7000 device. Moreover, despite the design only requires one MPBU controller, each hardware module would need a different MPBU. Therefore the resource utilization increases proportionally with the number of module and MPBUs that are instantiated.

Note that addressing greater devices embedding more resources, the utilization rates decrease drastically.

Software Description As described in Chapter 6, the software relies on FreeRTOS and performs MPBU configuration and module enabling.

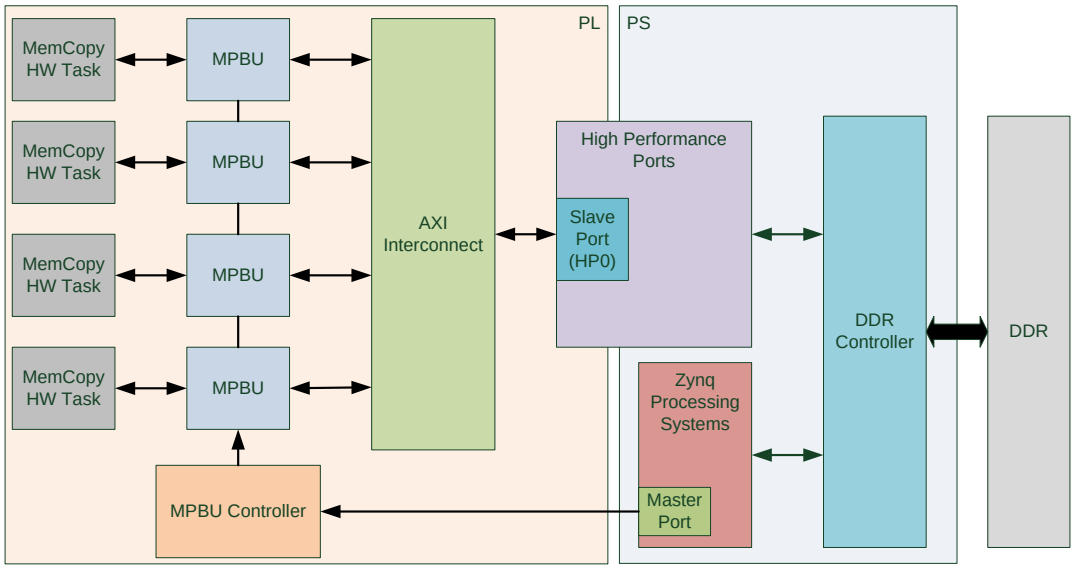


Figure 6.5: Block diagram of the system used to evaluate the MPBUs. Each hardware task can access the main memory through an MPBU unit which provides memory protection and transaction budgeting. MPBUs are connected to the main memory through an AXI Interconnect IP which arbitrates the multi-master access to the memory. The MPBU controller acts as a bridge between the Zynq's AXI interface and the custom interface used to configure the MPBUs.

		Evaluation System	MPBU	MPBU Controller
Resource	**Available**	**Utilized (%)**	**Utilized (%)**	**Utilized (%)**
LUT	17600	12530 (71.19%)	1211 (6.88%)	360 (2.88%)
LUTRAM	6000	1652 (27.53%)	0 (0%)	0 (0%)
FF	35200	18870 (53.61%)	777 (2.21%)	561 (1.59%)
BRAM	60	28 (46.67%)	0 (0%)	0 (0%)
BUFG	32	1 (3.13%)	1 (3.13%)	1 (3.13%)

Table 6.3: The third column reports the resource utilization of the whole hardware design used for the MPBU evaluation. It consists of four hardware task connected, through a standard AXI interconnect IP, to one high performance port of the processing system. Last two columns report the resource utilization of the MPBU controller and one MPBU unit both enabling memory protection and bus budgeting. In both tables, percentages refer to a Zynq 7z010 chip.

The four hardware modules are managed by four periodic tasks running on top of the operating system and each task controls a specific module. All tasks in all experiments are equivalent: they have the same priority and a fixed period of 250ms in order to generate the maximum possible contention on the memory bus.

During its execution, a task programs and launches its correspondent hardware module and self-suspends waiting for the hardware to complete its memory transfer. Once the hardware module completes its memory transfer, sends an interrupt to the processor and the corresponding interrupt service routine provides wakes up the correct task.

6.2.1 Evaluation's Results

A set of experiments have been created in order to extensively test and evaluate the MPBU.

Baseline Performance Experiments In particular, the first set of experiments aim at calculating the baseline bandwidth performance of hardware modules and bus. In this experiments, the MPBU modules are disabled (i.e., transparent) to avoid any interference and it has been measured the real throughput of one hardware module when one, two, three or four modules compete for the memory.

Table 6.4 shows the the downstream bandwidth of each hardware module and the execution time of the related software task. In particular, it is possible to notice that a single, standalone, hardware module has a downstream bandwidth of $\simeq 755 MiB/s$ while having multiple modules running in parallel leads to a progressive reduction of the bandwidth due to the contention. By multiplying the effective single-module downstream bandwidth times the number of active modules (or tasks), it is possible to estimate the total available bandwidth the interconnect and memory controller can

Exp	Task	Observed exec time		Avg task demand [MiB/s]	Avg Int+Mem supply [MiB/s]
		Avg [ms]	Worst [ms]		
0	0	2.6483	2.6502	754.6457	754.6457
1	0	2.8077	2.8170	712.3147	1424.6046
	1	2.8078	2.8172	712.2899	
2	0	4.2311	4.3309	472.6855	1418.0366
	1	4.2311	4.3300	472.6896	
	2	4.2313	4.3293	472.6615	
3	0	5.5109	5.5506	362.9160	1451.1384
	1	5.5135	5.5535	362.7400	
	2	5.5139	5.5535	362.7187	
	3	5.5132	5.5528	362.7637	

Table 6.4: A single, standalone hardware module has a downstream bandwidth of $\simeq 755MiB/s$. However, as the number of concurrently active hardware modules increases, the throughput of a single module decreases due to contentions of the communication bus. The table shows that the bandwidth the memory controller can handle saturates at an average value of $\simeq 1430MiB/s$.

handle. The maximum bandwidth estimation is $\simeq 1430MiB/s$.

This set of experiments show that the whole infrastructure, including the interconnect and the memory controller, is *fair* thus the available bandwidth is shared equally among the contenders. This validates the initial assumption (see Section 5.2) and allows to implement a reservation mechanisms using the MPBU on top of a fair infrastructure. Figure 6.6 shows the results in term of software task execution time.

Reservation Experiments A second set of experiments has been carried out to evaluate the effectiveness of the bandwidth reservation mechanism enforced by the MPBU. In these experiments, the hardware modules are supervised by the MPBU thus the

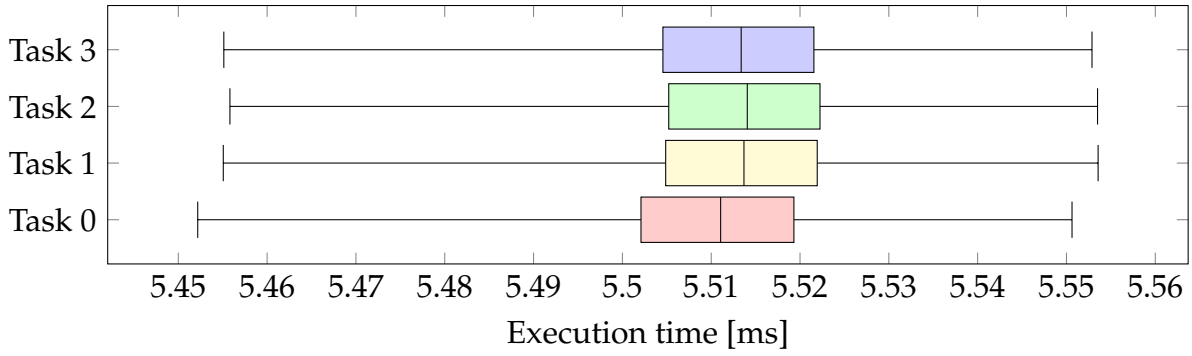


Figure 6.6: Baseline bandwidth of hardware modules and AXI bus in terms of execution time. In this experiment each hardware module moves 2 MiB (reads and writes 1MiB of data) and each MPBU is disabled in order not to affect the module's performance. The left and right edges of colored boxes are the first and third quartiles while the band inside it represent the median (second quartile). Black, vertical lines at the ends of each row are the minimum and maximum of all of the data.

amount of transactions that a hardware module can perform, during each MPBU period, is upper bounded by its budget.

In this set of experiments, all four hardware modules are periodically activated by a set of four tasks running with the same period of 250ms. Moreover, one MPBU have its transaction budget fixed to 48 while other two to 32. The transaction budget of the last MPBU increases, from the first to the fourth experiments, from 64 to 128. The budget refresh period is 128 clock cycles for all MPBUs. Table 6.5 summarizes MPBUs' parameters across all the experiments.

The results of all four experiments are summarized in Figure 6.7. By observing the execution time of software tasks it is possible to evaluate the effectiveness of the reservation mechanism. Each hardware module (and its related task) can access only a predefined fraction of the total available bandwidth and it is upper-bounded by the MPBU. Moreover, increasing the transaction budget of one module, within the available

MPBU	Budget (transactions)				Refreshing Period (clock cycles)
	Exp. 1	Exp. 1	Exp. 1	Exp. 1	
MPBU0	64	80	96	128	128
MPBU1	48	48	48	48	128
MPBU2	32	32	32	32	128
MPBU3	32	32	32	32	128

Table 6.5: Summary of MPBUs' settings for reservation experiments. Three MPBUs have fixed parameters while the transaction budget of MPBU0 increases from the first to the fourth experiment.

bandwidth limit, does not affect the throughput of the other modules and consequently, the execution time of their software tasks. Since all the hardware modules tries to access the memory at the maximum rate they are capable of, it is worth to notice that this propriety is enforced by the MPBU independently from the behavior of the hardware module.

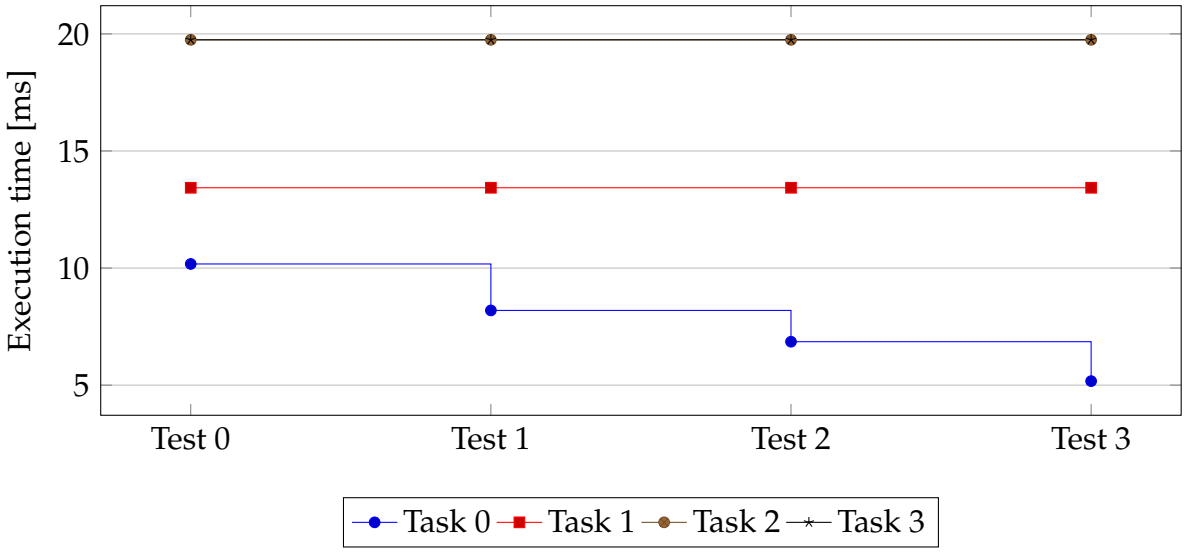


Figure 6.7: Summary of tasks' execution time across all the reservation experiments. The MPBU guarantees that each hardware module accesses only a predefined fraction of the total available bandwidth. Therefore, increasing the transaction budget of one module does not affect the throughput of the other modules and, consequently, the execution time of their software tasks.

Chapter 7

Conclusions

This work has been developed with the purpose of providing more performance, predictability, and a better analysis of real-time systems involving FPGAs in order to improve and ease the integration of FPGAs in such systems.

In particular, this thesis presented the first production of preemptive reconfiguration for Xilinx 7 series FPGAs and SoCs which uses the ICAP and solved the problems of priority inversion and starvation caused by contention on the reconfiguration port. To achieve preemptive reconfiguration, *resumption points* have been defined, i.e., parts of a bitstream from which a preempted reconfiguration can safely be resumed. Furthermore, methods to determine all possible and safe resumption points have been presented. Resumption points have been leveraged by a combination of a custom command-based reconfiguration controller and a reconfiguration driver that manage reconfiguration requests and handle reconfiguration preemption and resumption.

Worst-case bounds on the latency overhead that a higher-priority task experiences when preempting a lower-priority task and upper bounds on the reconfiguration delay – experienced by the lower-priority task for a given worst-case interval of reconfiguration preemptions – have been analytically determined and revealed the trade-off between

the minimum guaranteed bandwidth for reconfigurations of lower-priority tasks and the WCET bound on performing preemptions for higher-priority tasks. Experimental results showed that priority inversion and starvation caused by contentions on the reconfiguration port are effectively solved using the proposed realization of preemptive reconfiguration, when integrated into the industry-grade real-time operating system FreeRTOS. As this work presented the first, real approach to enable preemptive partial reconfiguration, comparison with state-of-the-art methods providing the same functionality on FPGAs has not been possible.

Preemptive partial reconfiguration has been achieved even on a low-end (in terms of resources) reconfigurable device, the Xilinx Zynq-7z010. In total, it has been shown that the overhead for achieving preemptive reconfiguration is considerably outweighed by the benefits it provides. In particular, while the average measured execution time of both tasks is similar across all the experiments, the measured worst-case execution time have been significantly improved. Using preemptive reconfiguration, the delay experienced by the high-priority task has been reduced by 3 times with respect to configure-to-completion experiment where priority inversion arises. Moreover, the measured worst-case execution time of the low-priority task has been reduced by 39 times compared to abort experiment where starvation affects the low-priority tasks preventing it to proceed with its execution.

Preemptive resources are fundamental to fulfill real-time constraints and this work enables multi-priority real-time systems (including mixed-criticality systems) to benefit from runtime-reconfigurable hardware accelerators with preemptive reconfiguration capabilities.

Moreover, the first, custom approach to solve *memory corruption* and *bus chocking* problems in FPGA-accelerated real-time systems has been presented. Those problems could arise in a reconfigurable FPGA system where misbehaving custom hardware

accelerators are deployed inside reconfigurable slots at runtime. In those systems, where the FPGA is used as a hardware co-processor to support a standard CPU in speeding up compute-intensive functions, the accelerators have full access to the main memory thus a bugged/malicious accelerator could jeopardize the functionality of the whole system.

A custom Memory Protection and Budgeting Unit, which does not affect the AXI bus performance and throughput, have been designed and realized in order to avoid memory corruption and bus chocking problems. Moreover, the MPBU alleviate the unpredictability of the communication bus used in an FPGA-based system for real-time applications, allowing to have a more predictable system with less timing constraints thanks to the bus bandwidth budgeting.

Experiments with MPBUs have been performed to verify their functioning, performance and the real available bandwidth of the Zynq memory controller. Those experiments show that the total bandwidth available from the interconnect and the memory controller can be estimated in the order of $\simeq 1430 MiB/s$. Moreover, it has been verified that the proposed budgeting approach, implemented by the MPBU, guarantees a programmable bandwidth reservation for each task and does not affect other tasks performance. Eventually, it has been showed that memory corruption and bus chocking problems can be avoided and the communication over a shared bus can be made more predictable allowing to have less stringent timing constraints in the analysis. Results on MPBU's performance have not been compared to state-of-the-art results as there is no similar hardware in literature providing the same functionalities.

Details on the hardware and software designs have been reported, describing the method used to implement preemptive reconfiguration, memory protection and bandwidth budgeting and explaining the software functionalities to configure and use the IPs.

Bibliography

[1] Sandi Habinc. Technical report: Suitability of reprogrammable FPGAs in space applications. Technical report, Gaisler Research, September 2002.

[2] P. Alfke. Recent progress in field programmable logic. In *Proceedings of the 6th Workshop on Electronics for LHC Experiments*, 2000.

[3] Klaus Danne and Marco Platzner. An edf schedulability test for periodic tasks on reconfigurable hardware devices. *ACM SIGPLAN Notices*, 41(7):93–102, 2006.

[4] Rodolfo Pellizzoni and Marco Caccamo. Real-time management of hardware and software tasks for FPGA-based embedded systems. *IEEE Transactions on Computers*, 56(12):1666–1680, 2007.

[5] Russell Tessier, Kenneth Pocek, and Andre DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):332–354, 2015.

[6] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. A framework for supporting real-time applications on dynamic reconfigurable FPGAs. In *Real-Time Systems Symposium (RTSS)*, pages 1–12, 2016.

[7] Marvin Damschen, Lars Bauer, and Jörg Henkel. Timing Analysis of Tasks on Runtime Reconfigurable Processors. *IEEE Trans. on Very Large Scale Integration Syst.*, 25(1):294–307, June 2016.

[8] Sangeet Saha, Arnab Sarkar, and Amlan Chakrabarti. Scheduling dynamic hard real-time task sets on fully and partially reconfigurable platforms. *IEEE Embedded Systems Letters*, 7(1):23–26, 2015.

[9] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '01, pages 642–649, Piscataway, NJ, USA, 2001. IEEE Press.

[10] Mark Goosman, Nij Dorairaj, and Eric Shiflet. How to take advantage of partial reconfiguration in FPGA designs, 2006.

[11] Stefan Wildermann, Gregor Walla, Tobias Ziermann, and Jürgen Teich. Self-organizing multi-cue fusion for fpga-based embedded imaging. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 132–137. IEEE, 2009.

[12] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. Performance of partial reconfiguration in fpga systems: A survey and a cost model. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(4):36, 2011.

[13] Shobharani Tatineni. *Dynamic Scheduling, Allocation, and Compaction Scheme for Real-Time Tasks on FPGAs*. PhD thesis, Citeseer, 2002.

[14] Yi Lu, Thomas Marconi, Koen Bertels, and Georgi Gaydadjiev. A communication aware online task scheduling algorithm for fpga-based partially reconfigurable systems. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 65–68. IEEE, 2010.

[15] Christoph Steiger, Herbert Walder, Marco Platzner, and Lothar Thiele. Online scheduling and placement of real-time tasks to partially reconfigurable devices. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 224–225. IEEE, 2003.

[16] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[17] Shaoshan Liu, Richard Neil Pittman, and Alessandro Forin. Minimizing partial reconfiguration overhead with fully streaming dma engines and intelligent ICAP controller. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, February 2010.

[18] Xilinx. *Partial Reconfiguration User Guide*, 2012. v14.1.

[19] François Duhem, Fabrice Muller, and Philippe Lorenzini. Farm: Fast reconfiguration manager for reducing reconfiguration time overhead on FPGA. In *Proceedings of the 7th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, November 2011.

[20] Marco Pagani, Mauro Marinoni, Alessandro Biondi, Alessio Balsini, and Giorgio Buttazzo. Towards real-time operating systems for heterogeneous reconfigurable platforms. In *Proc. of the 12th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2016)*, 2016.

[21] Stephen D. Scott, Ashok Samal, and Shared Seth. Hga: A hardware-based genetic algorithm. In *Proceedings of the ACM Third International Symposium on Field-programmable Gate Arrays*, February 1995.

[22] Wang Chen, Panos Kosmas, Miriam Leeser, and Carey Rappaport. An FPGA implementation of the two-dimensional finite-difference time-domain (fdtd) algorithm. In *Proceedings of the ACM/SIGDA 12th international symposium on Field programmable gate arrays*, February 2004.

[23] Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009(1):758480, 2009.

[24] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[25] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[26] DC Liang. Hard real-time bus architecture and arbitration algorithm based on amba. 2015.

[27] Tommaso Cucinotta, Dhaval Giani, Dario Faggioli, and Fabio Checconi. Providing performance guarantees to virtual machines using real-time scheduling. In *Euro-Par Workshops*, pages 657–664. Springer, 2010.

[28] Michael D Ciletti. *Advanced digital design with the Verilog HDL*, volume 1. Prentice Hall Upper Saddle River, 2003.

[29] Umer Farooq, Zied Marrakchi, and Habib Mehrez. Fpga architectures: An overview. *Tree-based Heterogeneous FPGA Architectures*, pages 7–48, 2012.

[30] National_Instruments. Introduction to FPGA technology: Top 5 benefits, 2012.

[31] Roman Lysecky and Frank Vahid. A study of the speedups and competitiveness of fpga soft processor cores using dynamic hardware/software partitioning. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 18–23. IEEE, 2005.

[32] Kisun You, Hyunjin Lim, and Wonyong Sung. Architectural design and implementation of an fpga softcore based speech recognition system. In *System-on-Chip for Real-Time Applications, The 6th International Workshop on*, pages 50–55. IEEE, 2006.

[33] Jörg Henkel. Closing the soc design gap. *Computer*, 36(9):119–121, 2003.

[34] Lars Bauer, Muhammad Shafique, Simon Kramer, and Jörg Henkel. Rispp: Rotating instruction set processing platform. In *Proceedings of the 44th annual Design Automation Conference*, pages 791–796. ACM, 2007.

[35] Lars Bauer, Muhammad Shafique, Stephanie Kreutz, and Jörg Henkel. Run-time system for an extensible embedded processor with dynamic instruction set. In *Proceedings of the conference on Design, automation and test in Europe*, pages 752–757. ACM, 2008.

[36] Reiner Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the conference on Design, automation and test in Europe*, pages 642–649. IEEE Press, 2001.

[37] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[38] Francisco Barat and Rudy Lauwereins. Reconfigurable instruction set processors: a survey. In *Rapid System Prototyping, 2000. RSP 2000. Proceedings. 11th International Workshop on*, pages 168–173. IEEE, 2000.

[39] Michael Ullmann, Michael Hübner, Björn Grimm, and Jürgen Becker. On-demand fpga run-time system for dynamical reconfiguration with adaptive priorities. In *International Conference on Field Programmable Logic and Applications*, pages 454–463. Springer, 2004.

[40] Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev. Architectural exploration of the adres coarse-grained reconfigurable array. In *International Workshop on Applied Reconfigurable Computing*, pages 1–13. Springer, 2007.

[41] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The molen polymorphic processor. *IEEE transactions on computers*, 53(11):1363–1375, 2004.

[42] Partha Biswas, Vinay Choudhary, Kubilay Atasu, Laura Pozzi, Paolo Ienne, and Nikil Dutt. Introduction of local memory elements in instruction set extensions. In *Proceedings of the 41st annual Design Automation Conference*, pages 729–734. ACM, 2004.

[43] Scott Hauck, Thomas W Fry, Matthew M Hosler, and Jeffrey P Kao. The chimaera reconfigurable functional unit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2):206–217, 2004.

[44] Chun He, Alexandros Papakonstantinou, and Deming Chen. A novel soc architecture on fpga for ultra fast face detection. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, pages 412–418. IEEE, 2009.

[45] Paul Viola and Michael J Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004.

[46] Henry A Rowley, Shumeet Baluja, and Takeo Kanade. Neural network-based face detection. *IEEE Transactions on pattern analysis and machine intelligence*, 20(1):23–38, 1998.

[47] Chin-Chuan Han, Hong-Yuan Mark Liao, Gwo-Jong Yu, and Liang-Hua Chen. Fast face detection via morphology-based pre-processing. *Pattern Recognition*, 33(10):1701–1712, 2000.

[48] Andreas Oetken, Stefan Wildermann, Jurgen Teich, and Dirk Koch. A bus-based soc architecture for flexible module placement on reconfigurable fpgas. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 234–239. IEEE, 2010.

[49] Xinping Zhu and Sharad Malik. A hierarchical modeling framework for on-chip communication architectures of multiprocessing socs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(1):6, 2007.

[50] Terry Tao Ye, Luca Benini, and Giovanni De Micheli. Packetized on-chip interconnect communication analysis for mpsoc. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 344–349. IEEE, 2003.

[51] Xilinx Inc. *AXI Reference Guide*, 1 2012. UG761.

[52] Anurag Shrivastav, GS Tomar, and Ashutosh Kumar Singh. Performance comparison of amba bus-based system-on-chip communication protocol. In *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*, pages 449–454. IEEE, 2011.

[53] ARM. *AMBA® AXI™ and ACE™ Protocol Specification*. ARM IHI 0022D.

[54] ARM. *AMBA™ Specification*. ARM IHI 0011A.

[55] Xilinx Inc. *7 Series FPGAs Configuration - User Guide*, 9 2016. UG470.

[56] Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C Diniz. *Applied Reconfigurable Computing: 11th International Symposium, ARC 2015, Bochum, Germany, April 13-17, 2015, Proceedings*, volume 9040. Springer, 2015.

[57] Marvin Damschen, Lars Bauer, and Jörg Henkel. CoRQ: Enabling Runtime Reconfiguration under WCET Guarantees for Real-Time Systems. *IEEE Embedded Systems Letters*, 2017. to appear.

[58] Alan Burns. *Preemptive priority based scheduling: An appropriate engineering approach*. University of York, Department of Computer Science, 1993.

[59] Emilio Vivancos, Christopher Healy, Frank Mueller, and David Whalley. Parametric timing analysis. *ACM SIGPLAN Notices*, 36(8):88–93, 2001.

[60] Gilles Chanteperdrix and Richard Cochran. The ARM fast context switch extension for linux. In *Real Time Linux Workshop*, pages 255–262, 2009.

[61] Xilinx Inc. *AXI Interconnect v2.1*, 4 2017. PG059.

[62] Xilinx Inc. *Zynq-7000 All Programmable SoC - Overview*, 9 2016. DS190.