Reconguration of Computation and Communication Resources in Multi-Core Real-Time Embedded Systems

Pezzarossa, Luca

Publication date:
2018

Document Version
Publisher's PDF, also known as Version of record

Reconfiguration of Computation and Communication Resources in Multi-Core Real-Time Embedded Systems

Luca Pezzarossa

Kongens Lyngby 2018
PhD-2018-469

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, Building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk
PhD-2018-469

Abstract (English)

Reconfigurable computing allows application programmers to significantly increase the speed of software algorithms by implementing computationally-demanding tasks in hardware while maintaining a certain degree of flexibility. This can be achieved by using FPGAs to implement hardware accelerators that can be reconfigured when no longer needed, enabling the re-use of the resources of the FPGAs to realise new functionalities. For multi-core platforms, reconfiguration can be extended to the infrastructure supporting inter-core communication and used to dynamically modify the characteristics of the communication channels between the tasks that are affected by the reconfiguration.

This thesis investigates the use of reconfiguration in the context of multi-core real-time systems targeting embedded applications. We address the reconfiguration of both the computation and the communication resources of a multi-core platform. Our approach is to associate reconfiguration with operational mode changes where the system, during normal operation, changes a subset of the executing tasks to adapt its behaviour to new conditions. Reconfiguration is therefore used during a mode change to modify the real-time guaranteed services provided by the hardware platform to fit the requirements of the current mode.

The reconfiguration of the computation resources consists of altering the hardware implementation of selected resources, such as accelerators, and it is achieved by using the dynamic partial reconfiguration feature offered by FPGAs. With regards to this, we also present a lightweight reconfiguration controller, named RT-ICAP, specially developed to support time-predictable dynamic partial reconfiguration. The reconfiguration of the communication resources consists of setting up and tearing down the end-to-end channels offered by the communication fabric

between the cores of the platform. To support this, we present a new network-on-chip architecture, named Argo 2, that allows instantaneous and time-predictable reconfiguration of the communication channels. Our reconfiguration-capable architecture is prototyped using the existing time-predictable multi-processor platform T-CREST. The thesis also presents low-level reconfiguration time analysis for these architectures.

The evaluation of the proposed approach and the developed architectures is carried out using synthetic benchmarks and hardware accelerators generated by high-level synthesis tools. For the reconfiguration of computation resources, the results show that the use of accelerators in combination with dynamic partial reconfiguration leads to better utilisation of the FPGA resources and tighter worst-case execution time bounds than a pure software solution. Moreover, the results show that using a reconfigurable solution delivers a worst-case performance comparable with that of a non-reconfigurable solution. For the reconfiguration of communication resources, the results show that the worst-case reconfiguration time ranges from hundreds to thousands of clock cycles, making our solution considerably faster than other functionally equivalent networks-on-chips.

In addition to the evaluation based on synthetic benchmarks, we also present a proof-of-concept case study based on a multi-core audio digital signal-processing application that combines reconfiguration of both the computation and communication resources. The case study shows that the presented approaches for reconfiguration can be effectively used in a real-world application and can lead to a reduction of the overall hardware size and better use of the platform resources while maintaining comparable computation performance with respect to a non-reconfigurable approach.

Resumé (Dansk)

Re-konfigurerbar hardware gør det muligt for applikationsudviklere at øge hastigheden af softwarealgoritmer betydeligt ved at implementere beregningsmæssigt krævende opgaver i hardware, samtidig med at der opretholdes en vis grad af fleksibilitet. Dette kan opnås ved at bruge FPGA'er til at implementere acceleratorer, der kan re-konfigureres når de ikke længere er nødvendige, hvilket gør det muligt at genbruge FPGA'ernes ressourcer til at realisere ny funktionalitet. For multikerne platforme kan re-konfigurationen udvides til den infrastruktur der understøtter kommunikation imellem kernerne, og bruges til dynamisk at ændre karakteristika for kommunikationskanalerne mellem de opgaver der berøres af re-konfigurationen.

Denne afhandling undersøger brugen af re-konfiguration i forbindelse med realtids systemer med flere kerner rettet mod indlejrede applikationer. Vi undersøger re-konfiguration af både beregnings- og kommunikationsressourcer på en multikerne platform. Vores tilgang er at bruge re-konfiguration sammen med ændring af driftstilstanden, hvor systemet under normal drift ændrer en delmængde af de eksekverende processer for at tilpasse sin funktion til nye forhold. Re-konfiguration anvendes derfor i forbindelse med en tilstandsændring for at tilpasse de realtidsgaranterede ydelser der leveres af hardwareplatformen til den aktuelle tilstand.

Re-konfigurationen af beregningsressourcerne består i at ændre hardware implementeringen af udvalgte ressourcer, såsom acceleratorer. Dette opnås ved at bruge den delvise re-konfigurationsfunktion der tilbydes af FPGA'er. Til udførelse af dette præsenterer vi en minimal konfigurationskontroller betegnet RT-ICAP, der er specielt udviklet til at understøtte realtidsforudsigelig delvis

re-konfiguration. Re-konfigurationen af kommunikationsressourcerne består i at nedlægge og oprette kanal endepunkterne der tilbydes af kommunikationsnettet mellem platformens kerner. For at understøtte dette præsenterer vi en ny netværk-på-chip arkitektur betegnet Argo 2, som tillader øjeblikkelig og tidsforudsigelig re-konfiguration af kommunikationskanalerne. Vores re-konfigurationsfunktionelle arkitektur er udviklet og testet på den tidsforudsigelige multikerne platform T-CREST. Afhandlingen præsenterer også hardware nær tidsanalyse af re-konfigurationerne for disse arkitekturer.

Evalueringen af den foreslåede tilgang og de udviklede arkitekturer udføres ved hjælp af syntetiske benchmarks og hardware acceleratorer, der genereres af højniveaus syntesеværktøjer. For re-konfigurationen af beregningsressourcerne viser resultaterne, at brugen af hardwareacceleratorer i kombination med delvis re-konfigurering fører til bedre udnyttelse af FPGA-ressourcerne og mindre udsving af tidsbegrænsningerne end en software-baseret løsning. Endvidere viser resultaterne, at ved anvendelse af en re-konfigurerbar løsning opnås en worst-case præstation tilsvarende en statisk løsning. For re-konfigurationen af kommunikationsressourcer viser resultaterne, at den værste re-konfigurationstid varierer fra hundrede til tusindvis af klokkecykler, hvilket gør vores løsning betydeligt hurtigere end andre funktionelt tilsvarende netværk-på-chips.

Udover evalueringen baseret på syntetiske benchmarks præsenterer vi også et proof-of-concept casestudy baseret på en multikerne applikation som behandler digitale lydsignaler, der kombinerer re-konfigurering af både beregnings- og kommunikationsressourcerne. Casestudy viser, at de præsenterede strategier til re-konfiguration kan anvendes effektivt i en reel applikation og kan føre til en reduktion af den samlede hardwarestørrelse, samtidig med at man opretholder en beregningsydelse tilsvarende en statisk platform.

Preface

The work presented in this thesis was conducted at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark (DTU Compute) in fulfilment of the requirements of the PhD program.

The work was supervised by Professor Jens Sparsø and co-supervised by Associate Professor Martin Schoeberl. The thesis explores the use of reconfiguration of computation and communication resources in multi-core real-time embedded systems. The thesis is a monograph and consists of eight chapters.

The thesis does not contain any material that has been accepted for the award of any other degree or diploma in my name, in any university or other institution and, to the best of my knowledge, does not contain any material previously published by another person, except where due reference is made in the thesis.

Kongens Lyngby, 15-March-2018

Luca Pezzarossa

Acknowledgements

At first, I would like to thank my supervisor Professor Jens Sparsø and my co-supervisor Associate Professor Martin Schoeberl for guiding, supporting, and helping me throughout my PhD.

I would also like to thank all the members of the section for Embedded Systems Engineering at the Technical University of Denmark. Especially the PhD colleagues and the MSc and BSc students with whom I collaborated or shared part of the last three years: Evangelia, Rasmus, Wolfgang, Daniel, Andreas, Tórur, Oktay, Eleftherios, and all the others.

Special thanks go to Ioannis, who shared with me times of enthusiasm and discouragement, helped me every time I needed, and proofread this thesis.

Finally, I would like to thank my family, Mauro, Daniela and Marco who, even though they are far away, have always supported me in everything.

Contents

Abstract (English) **i**

Resumé (Dansk) **iii**

Preface **v**

Acknowledgements **vii**

List of Acronyms **xiii**

List of Publications **xv**

1 Introduction **1**
1.1 Research Field and Motivation 1
1.2 Thesis Overview 4
1.3 List of Contributions 5
1.4 Source Access 6
1.5 Thesis Structure and Outline 6

2 Background **9**
2.1 Reconfigurable Computing 9
2.1.1 Overview 10
2.1.2 Evolution and Technology 10
2.1.3 High-Level Synthesis 12
2.2 Dynamic Partial Reconfiguration 13
2.2.1 Overview 13
2.2.2 ICAP Interface 14
2.2.3 Design Flow and Requirements 17
2.3 Real-Time Systems 18

2.3.1 Overview and Classification . . . 19
2.3.2 Timing-Analysis . . . 20
2.4 The T-CREST Platform . . . 22
2.4.1 Overview . . . 22
2.4.2 Patmos Processor . . . 22
2.4.3 Support Tools . . . 24
2.4.4 Memory Access NoC . . . 25
2.5 Argo Message-Passing NoC . . . 26
2.5.1 Overview . . . 26
2.5.2 TDM-Schedule . . . 27
2.5.3 NoC Architecture . . . 28

3 Related Work **33**
3.1 Reconfiguration of Computation Resources . . . 33
3.1.1 Methods and Tools . . . 34
3.1.2 Reconfiguration Controllers . . . 36
3.2 Reconfiguration of Communication Resources . . . 38
3.2.1 NoCs Based on Flow Control . . . 39
3.2.2 NoCs Based on TDM . . . 40
3.3 Other Related Topics . . . 42

4 Approach to Reconfiguration **45**
4.1 Definition of Communication and Computation Resources . . . 45
4.2 Reconfiguration at Mode Changes . . . 46
4.3 Extraction of Guaranteed Service Requirements . . . 48
4.4 Model of the Reconfiguration Process . . . 50
4.5 Expected Outcomes and Evaluation Metrics . . . 51

5 Reconfiguration of Computation Resources **55**
5.1 A Multi-Core Platform Supporting DPR . . . 55
5.2 RT-ICAP Controller Architecture . . . 57
5.3 Bit-Stream Compression . . . 61
5.4 Tool Support . . . 63
5.5 Reconfiguration Time Analysis . . . 64
5.6 Single-Core Application Example . . . 66

6 Reconfiguration of Communication Resources **69**
6.1 Overview . . . 69
6.2 Argo 2 NI Architecture . . . 70
6.2.1 Packet Format and Schedule Representation . . . 70
6.2.2 Transmit Module . . . 72
6.2.3 Receive Module . . . 74
6.2.4 Remote Initialization . . . 75
6.3 Support for Reconfiguration . . . 76

6.3.1 Key Ideas and Observations 76
6.3.2 Reconfiguration Process 77
6.4 Reconfiguration Time Analysis 79

7 Evaluation and Discussion **81**
7.1 Reconfiguration of Computation Resources 81
7.1.1 RT-ICAP Controller Characterization 82
7.1.2 Bit-Stream Compression and Reconfiguration Time 84
7.1.3 Synthetic Benchmarks Experiments 87
7.2 Reconfiguration of Communication Resources 97
7.2.1 Argo 2 Characterization 97
7.2.2 Synthetic Traffic Experiments 99
7.3 Audio DSP Application 103
7.3.1 Overview 103
7.3.2 Hardware Platform 105
7.3.3 Effects and Modes of Operation 108
7.3.4 Observations and Results 110

8 Conclusion **117**
8.1 Summary and Final Remarks 117
8.2 Future Work 119

Bibliography **123**

List of Acronyms

ADC	Analog-to-Digital Converter
ASIC	Application-Specific Integrated Circuit
BCET	Best Case Execution Time
BRAM	Block RAM
CC	Clock Cycle
CPU	Central Processing Unit
D$	Data Cache
DAC	Digital-to-Analog Converter
DMA	Direct Memory Access
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processing
eFPGA	embedded Field Programmable Gate Array
FF	Flip-Flop
FIFO	First In First Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
GS	Guaranteed Service
HDL	Hardware Description Language
HLS	High-Level Synthesis
HPU	Header Parsing Unit
HwA	Hardware Accelerator
ICAP	Internal Configuration Access Port
I^2C	Inter-Integrated Circuit
I^2S	Inter-Integrated circuit Sound
I/O	Input/Output

IRQ	Interrupt (ReQuest)
LUT	Look-Up Table
M$	Method Cache
NI	Network Interface
NoC	Network-on-Chip
OCP	Open Core Protocol
PL	Packet Length
RAM	Random-Access Memory
RISC	Reduced Instruction Set Computer
RLE	Run-Length Encoding
S$	Stack Cache
SDRAM	Synchronous Dynamic RAM
SoC	Systems-on-Chip
SPM	Scratch-Pad Memory
T2N	Time-to-Next
TDM	Time-Division Multiplexing
TTA	Time-Triggered Architecture
VC	Virtual Circuit
WCET	Worst-Case Execution Time
XML	eXtensible Markup Language

List of Publications

Journal Publications

[J1] L. Pezzarossa, A. T. Kristensen, M. Schoeberl, and J. Sparsø. Using Dynamic Partial Reconfiguration of FPGAs in Real-Time Systems. Accepted for publication in *Microprocessors and Microsystems: Embedded Hardware Design*, Elsevier, 2018.

[J2] M. Schoeberl, L. Pezzarossa, and J. Sparsø. A Multicore Processor for Time-Critical Applications. In *Journal of Design and Test*, volume PP, number 99, pages 1–1, IEEE, 2017.

[J3] R. B. Sørensen, L. Pezzarossa, M. Schoeberl, and J. Sparsø. A resource-efficient network interface supporting low latency reconfiguration of virtual circuits in time-division multiplexing networks-on-chip. In *Journal of Systems Architecture*, volume 74, pages 1–13, Elsevier, 2017.

Conference Publications

[C1] L. Pezzarossa, A. T. Kristensen, M. Schoeberl, and J. Sparsø. Can Real-Time Systems Benefit from Dynamic Partial Reconfiguration? In *Proceedings of the 3rd Nordic Circuits and Systems Conference (NorCAS)*. IEEE, 2017.

[C2] A. T. Kristensen, L. Pezzarossa, and J. Sparsø. High-Level Synthesis for Reduction of WCET in Real-Time Systems. In *Proceedings of the 3rd Nordic Circuits and Systems Conference (NorCAS)*. IEEE, 2017.

[C3] D. S. Ausin, L. Pezzarossa, and M. Schoeberl. Real-time audio processing on the T-CREST multicore platform. In *Proceedings of the 11th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2017.

[C4] L. Pezzarossa, M. Schoeberl, and J. Sparsø. A Controller for Dynamic Partial Reconfiguration in FPGA-Based Real-Time Systems. In *Proceedings of the 20th International Symposium on Real-Time Computing (ISORC)*. IEEE, 2017.

[C5] R. B. Sørensen, L. Pezzarossa, M. Schoeberl, and J. Sparsø. An Area-Efficient TDM NoC Supporting Reconfiguration for Mode Changes. In *Proceedings of the 10th International Symposium on Networks-on-Chip (NOCS)*. IEEE/ACM, 2016.

[C6] L. Pezzarossa, M. Schoeberl, and J. Sparsø. Reconfiguration in FPGA-Based Multi-Core Platforms for Hard Real-Time Applications. In *Proceedings of the 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE, 2016.

[C7] L. Pezzarossa, R. B. Sørensen, M. Schoeberl, and J. Sparsø. Interfacing Hardware Accelerators to a Time-Division Multiplexing Network-on-Chip. In *Proceedings of the 1st Nordic Circuits and Systems Conference (NORCAS)*. IEEE, 2015.

Other Works

[O1] L. Pezzarossa. Dynamic Reconfiguration in Multi-Core Hard Real-Time Platforms. Abstract and poster presented at the *12th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*. HiPEAC, 2016.

[O2] L. Pezzarossa, M. Schoeberl, and J. Sparsø. Towards Utilizing Reconfigurable Shared Resources in Multi-Core Hard Real-Time Systems Non-publishing article presented at the *9th Junior Researcher Workshop on Real-Time Computing (JRWRTC)*. 2015.

[O3] L. Pezzarossa. Dynamic Partial Reconfiguration in the T-CREST Multi-core Platform. Abstract and poster presented at the *11th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*. HiPEAC, 2015.

Chapter 1

Introduction

This thesis explores the challenges and the potential benefits of reconfigurable computing in the context of multi-core real-time systems targeting embedded applications.

This chapter introduces the research work presented in this thesis. At first, we describe the research fields to which this study relates to, and we present the motivation and hypotheses of this work. This is followed by an overview of the thesis content and by the list of contributions. Finally, we provide the information regarding the access to the source code associated with this work and present the structure and outline of the thesis.

1.1 Research Field and Motivation

In recent years, we have observed a transition from single-core towards multi-core architectures, as well as an increased use of specialised hardware accelerators optimised for specific tasks [1]. Moreover, packet switched networks-on-chip (NoCs) are increasingly used in large multi-core chips to support communication between the cores and between cores and shared memory, instead of standard bus architectures [2, 3]. This shift was possible by evolution in chip fabrication

technology and further driven by increasing market demand for high computational capabilities, higher levels of integration, and low power consumption. Examples of commercially-available multi-core platforms targeting embedded systems include LEON4 by Gaislerflex/ESA [4], MPPA-256 by Kalray [5], and Epiphany by Adapteva [6, 7].

Even more recently, advancements in field programmable gate array (FPGA) technology have enabled reconfigurable computing to become viable and be used in end-products. The combination of FPGA technology and reconfigurable computing empowers application developers to design and use their own hardware accelerators to significantly increase the speed of software algorithms by utilising reconfigurable hardware [8]. The major FPGA vendors are beginning to produce hybrid systems-on-chip (SoC) combining a hard processor with a reconfigurable fabric, such as the Xilinx Zynq SoC [9], the Microsemi SmartFusion2 SoC [10], and the Intel Stratix 10, Arria V and 10, and Cyclone V SoC [11]. Other vendors offer FPGA intellectual property cores to embed into an application-specific integrated circuit (ASIC) design in order to increase the flexibility, differentiation, and market lifespan of a chip. These intellectual property cores are known as embedded-FPGA (eFPGA). Commercially-available solutions are provided by Menta-eFPGA [12] and Achronix [13]. In addition, most currently available FPGA devices allow portions of the chip to be reconfigured at run-time, while the rest of the device continues to operate without interruption [14, 15, 16]. This feature, called dynamic partial reconfiguration (DPR), is an active area of research for general-purpose reconfigurable computing.

The use of FPGAs, eFPGAs, and hybrid SoCs to enable a reconfiguration-oriented approach in the design of computing systems can lead to an increase in flexibility, performance, and more efficient utilisation of chip resources compared to a static solution. From an economic point of view, the non-recurring engineering costs of an ASIC implementation makes this technology prohibitively expensive for those applications in which these costs cannot be amortised over a very large production volume. Low volume professional or high-end real-time applications, such as control systems, medical devices, and flight electronics belong in this category. For these applications, reconfigurable computing in the form of FPGAs, eFPGAs, and hybrid SoCs can mitigate the production costs.

This thesis combines the presented research fields and aims to explore the use of reconfigurable computing in multi-core real-time systems. Real-time systems are a class of computing systems characterised by strict constraints on the execution time of tasks [17, 18]. Therefore, the temporal behaviour of a hardware platform supporting real-time applications must be predictable and analysable. Specifically, the possibility to calculate the worst-case execution time (WCET) of tasks is of fundamental importance. Hence, the entire system must be designed taking into account the specifications of time-predictability and time-analysability [19].

The real-time industry is typically rather conservative towards new architectures and approaches since these systems are often used in safety-critical applications where a failure to respond in time may lead to severe consequences. Nevertheless, the use of multi-core architectures and reconfigurable computing in real-time systems are active topics of research [20, 21].

In a multi-core platform which supports real-time applications, the requirement for time-predictable behaviour goes beyond the individual processors. The entire platform must offer guaranteed services in terms of computation resources (e.g., processors and hardware accelerators) and communication resources (e.g., network-on-chip supporting communication among tasks mapped to different cores). The T-CREST platform [21] is an example of an FPGA-based multi-core platform targeting real-time systems. All components are designed with a focus on time-predictability and WCET analysis aiming to reduce the complexity and pessimism of the analysis. The platform consists of a number of processing nodes [22] and two NoCs: a NoC, called Argo [23], that provides message-passing to support inter-processor communication and a NoC that provides shared memory access [24, 25]. T-CREST is used in this research as a prototyping platform for the proposed solutions supporting reconfiguration in multi-core real-time systems.

In general, reconfigurable computing is used to increase the performance of a system by dynamically starting and stopping tasks executed on the dedicated hardware resources implemented on a reconfigurable fabric [8]. However, when taking into account multi-core platforms for real-time applications, the reconfiguration can be extended to the infrastructure supporting inter-processor communication. Reconfiguration may be used to dynamically set up, tear down, and modify the communication channels between the tasks that are reconfigured. As our research targets multi-core real-time systems, we explore reconfigurable computing from both the computation and communication perspectives.

We hypothesise that the use of a hardware platform supporting reconfiguration of both computation and communication resources can provide substantial benefits by allowing run-time changes in the platform. More specifically, a system that uses a reconfigurable approach can be more efficient, flexible, and smaller in terms of hardware size compared to the equivalent static version, while maintaining comparable computational performance. We also hypothesise that the combination of hardware acceleration and reconfiguration can result in both lower and tighter bounds on the WCET since, in general, hardware has a simpler and more time-predictable behaviour than software executing on a processor. These hypotheses are further detailed in the thesis into a set of expected outcomes and evaluation metrics for our reconfiguration approach.

1.2 Thesis Overview

This thesis investigates the use of run-time reconfiguration in the context of multi-core real-time embedded systems. Such a run-time reconfiguration relates to both communication resources and computation resources of a multi-core platform. The main idea is to dynamically adapt the hardware platform to the actual needs of the software applications running on it. The reconfiguration of computation resources is achieved using the DPR feature of FPGAs for swapping between a set of hardware accelerators. The reconfiguration of communication resources is achieved by dynamically altering the bandwidth and latency guarantees of the communication channels offered by the inter-processor message-passing NoC. Since this research targets real-time systems, the time-predictability specification must also apply to the reconfiguration techniques.

Our approach is to associate reconfiguration with operational mode changes where the system, during normal operation, changes a subset of the executing software tasks to adapt its behaviour to new environmental conditions. Since different modes of a real-time application can have different requirements for the guaranteed services offered by the platform, run-time reconfiguration is used to modify the platform to meet the requirements of the current mode. This may lead to a reduction of hardware cost compared to a static solution where the hardware platform must meet the overall requirements of all the modes of the application. Moreover, the possibility to move functionality into hardware may increase the performance of the entire system in terms of speed and WCET compared to a pure software solution.

In addition to exploring the reconfiguration from a real-time perspective, we develop a hardware/software infrastructure to support the reconfiguration of the computation and communication resources targeting the existing time-predictable multi-processor platform T-CREST. For the reconfiguration of computation resources in T-CREST, we develop a hardware controller and software tools that enable time-predictable reconfiguration of the hardware accelerators using the DPR feature offered by Xilinx FPGAs. For the reconfiguration of the communication resources, we develop a new version of the T-CREST message passing NoC allowing instantaneous reconfiguration between different sets of virtual circuits without affecting those that persist across the reconfiguration.

The reconfiguration of the communication and computation resources are evaluated and discussed both independently and in conjunction. The independent evaluation of the reconfiguration of the computation resources is carried out using synthetic benchmarks and hardware accelerators implemented using high-level synthesis tools. Synthetic traffic benchmarks are also used for the independent evaluation of the reconfiguration of the communication resources. Finally, a

proof-of-concept case study that includes reconfiguration of both the computation and communication resource features is carried out using an audio digital signal processing (DSP) application.

1.3 List of Contributions

The main contributions of this thesis concern the proposed approach to reconfiguration and the hardware/software infrastructure we develop to support it. The following list summarises the features of these contributions.

- We present an approach for using reconfigurable computing in real-time multi-core embedded systems where the reconfiguration of computation and communication resources is associated with operational mode changes. This approach is analysed, especially focusing on the effects on the time predictability of the system. Particular attention is given to the extraction of the requirements for the guaranteed-services provided by the platform and to the role of reconfiguration into ensuring that these requirements are met for a multi-mode application.
- We explore and evaluate how to use the DPR feature of modern FPGAs to support the computation aspects of a mode change by dynamically reconfiguring the computation resources in the platform. More specifically, we present a hardware architecture of a lightweight time-predictable reconfiguration controller, named RT-ICAP, and the associated software tool that supports the controller. A reconfiguration time analysis for the reconfiguration controller is also presented.
- We supplement the message-passing NoC of the T-CREST platform with run-time reconfiguration, thereby supporting the reconfiguration of the communication resources. More specifically, we develop a new NoC architecture, named Argo 2, that supports instantaneous reconfiguration of end-to-end communication channels. An analysis of the reconfiguration effects on the time-predictability of the multi-core platform is also presented.
- We provide an extensive evaluation of the use of reconfiguration in real-time systems using synthetic benchmarks and a multi-core DSP audio application, aiming to confirm the thesis hypotheses.
- We significantly extend the T-CREST project by supplementing the existing time-predictable multi-core platform with reconfiguration capabilities for both hardware resources and the message-passing NoC.

1.4 Source Access

The entire software and hardware infrastructure developed in relation to the work presented in this thesis and the T-CREST multi-core platform are released under the terms of the simplified BSD open-source license. The full T-CREST platform and the source code related to this thesis are available at *https://github.com/t-crest/*, which is a collection of *git* repositories. The hardware source code related to the configuration of computation and communication resources can be found in the *reconfig* and in the *argo* repositories, respectively. The software source code can be found in the C folder of the *patmos* repository. The hardware and software source code related to the audio DSP application case study is available in the branch *reconfig-audio* of the *patmos* and *aegean* repositories. *README* files describing the file structure and the build instructions are also included in the repositories.

1.5 Thesis Structure and Outline

The thesis begins by presenting general background and a review of related work. Next, we present an overview of our reconfiguration approach followed by the implementation details of the hardware and software infrastructure we developed in relation to the proposed approach. Finally, we provide results and evaluate the proposed approach and architecture. The discussion is integrated with the evaluation. The following list of chapters provides a more detailed outline.

- Chapter 2 introduces the background for the topics related to the thesis, including reconfigurable computing, the DPR feature of FPGAs, real-time systems, the T-CREST multi-core platform, and the Argo message-passing NoC.
- Chapter 3 reviews related work covering two research areas: reconfiguration of computation resources and reconfiguration of communication resources.
- Chapter 4 presents our approach to reconfiguration by introducing a clear definition of computation and communication resources as well as the relationship between operational mode changes, guaranteed service requirements, and reconfiguration.
- Chapter 5 presents the hardware and software infrastructure we developed to support the reconfiguration of computation resources according to the proposed approach. More specifically, it presents the RT-ICAP controller

and the associated software tool we developed to enable time-predictable reconfiguration of the hardware accelerators using DPR.

- Chapter 6 describes the new version of the T-CREST message-passing NoC, called Argo 2, that we developed to support the reconfiguration of communication resources focusing on its reconfigurable features which allows instantaneous reconfiguration between different sets of virtual circuits.
- Chapter 7 evaluates and discusses our proposed reconfiguration approach through reviews of the developed infrastructure supporting DPR and the Argo 2 NoC as well as a proof-of-concept audio DSP application that combines the reconfiguration of both computation and communication resources.
- Finally, Chapter 8 concludes the thesis by summarising the contributions and the results, and by presenting future work.

Chapter 2

Background

This chapter provides the background for the main topics related to the thesis, including reconfigurable computing, the DPR feature of FPGAs, real-time systems, the T-CREST multi-core platform, and the Argo message-passing NoC. The background on each of the topics is presented independently to one another and aims to supply the reader with the knowledge needed from the perspective of this work. For additional background information, references to academic literature and technical reports are provided at the beginning of each section.

2.1 Reconfigurable Computing

The main focus of this work is reconfiguration. In this section, we provide background related to reconfigurable computing and overview its evolution and challenges. An extensive review of the state-of-the-art of reconfigurable computing can be found in [8].

2.1.1 Overview

Computing systems are typically implemented using two solutions. The first consists of employing a general-purpose processor to run software that realises the required functionality. This solution is very flexible since the same processor can be reprogrammed to implement multiple functionalities. However, the sequential nature of program execution, combined with the fact that the program must be compiled into a limited set of general-purpose instructions, may limit the efficiency and the performance. The second solution consists of using an ASIC chip that realises the required functionality directly in hardware. This is more efficient and has higher performance than the software-based solution since the hardware can be designed to match the required functionality and to exploit parallelism. However, ASICs lack flexibility, since the implemented functionality cannot be changed. Moreover, the design process of an ASIC is complex and time-consuming, and, thus, very expensive.

Reconfigurable computing is an approach that aims to combine the flexibility of software with the high performance of hardware. This is achieved by using reconfigurable fabrics, such as FPGAs, to implement computationally-demanding tasks in hardware. The hardware can be reconfigured when it is no longer needed allowing the re-use of resources of the reconfigurable fabric to realise new functionalities. The key advantage over a software-based solution is the ability to modify the hardware architecture to offer more complex, high-level instructions, as far as implementing entire tasks in hardware. The main advantages over an ASIC-based solution are the increased flexibility and the easier and less time-demanding design process.

2.1.2 Evolution and Technology

Reconfigurable computing was first presented in 1960 in [26], where the author proposed the concept of an architecture consisting of a fixed unit and a variable unit. The fixed unit would offer a simple interface to a user and manage the hardware implemented on the variable unit. The variable unit could be reconfigured to implement the hardware that better fit the current user application. This concept was not implemented and adopted in the 1960s due to limited technology. The invention of FPGAs in the early 1980s [27] brought new interest in reconfigurable computing from both academia and industry. Several pioneering FPGA-based designs demonstrated the potential of FPGA-based reconfigurable computing. For example, the SPLASH architecture [28] outperformed contemporary supercomputers on a DNA sequence matching problem by using a design based on 16 FPGAs.

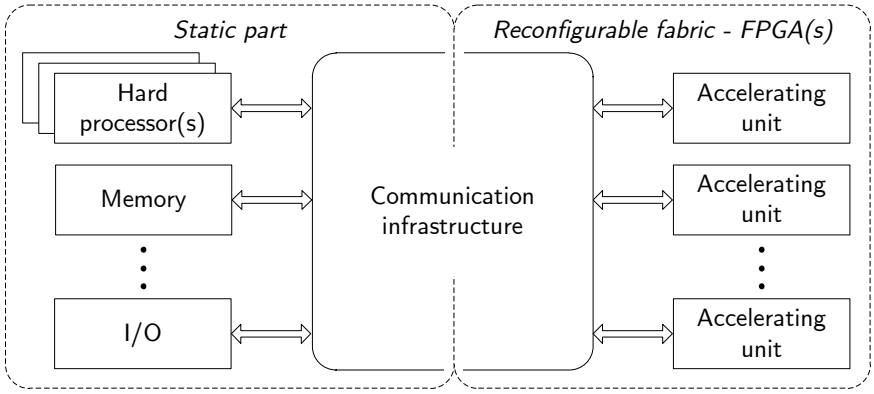

Figure 2.1: A block diagram of a reconfigurable computing system where a static (non-reconfigurable) processors-based architecture is combined with accelerating units implemented in a reconfigurable fabric. An infrastructure that extends in both the static part and the reconfigurable fabric supports the communication between all the system units.

Today, reconfigurable computing is mainly used to increase the speed of software algorithms by using hardware accelerators implemented on FPGAs. As previously mentioned, the common approach is to combine a processor-based computing architecture with reconfigurable accelerators. Figure 2.1 shows a block diagram of a possible computer architecture based on this approach. The system consists of a static (non-reconfigurable) part and a reconfigurable fabric (i.e. one or more FPGAs). The static part includes the processor-based computing system, which may consist of one or more hard processors, memory, inputs/outputs (I/Os), and possibly other peripherals. The reconfigurable fabric is used to implement acceleration units that match the current needs of an application executing on the system. A communication infrastructure supplies the communication channels between the units of the system, and it extends through both the static part and the reconfigurable fabric.

Hybrid SoCs available on the market are based on this architecture. Examples include the Xilinx Zynq SoCs [9], the Microsemi SmartFusion2 SoC [10], and the Intel Stratix 10, Arria V and 10, and Cyclone V SoCs [11]. These devices contain a single- or multi-core ARM processor together with an FPGA. In contrast, architectures incorporating one or more discrete FPGAs with one or more discrete hard processors also exist. For example, Microsoft Azure servers combine Intel FPGAs and processors to create a cloud that can be reconfigured to optimise a diverse set of applications and functions [29, 30]. The reconfiguration capabilities offered by the commercially-available FPGA and hybrid-SoC devices

are further extended by the DPR feature, which enables the reconfiguration of a portion of the FPGA, while the rest of the device continues to operate without interruption [14, 15, 16]. This feature is extensively used in this work to reconfigure computation resources. Thus, a detailed background related to DPR is presented in Section 2.2.

In a reconfigurable system such as the one presented in Figure 2.1, a set of tasks are executed by the hard processors, and another set of tasks are executed on the dedicated hardware resources implemented on the reconfigurable fabric. The communication between these two sets is supported by the communication infrastructure. The idea of reconfiguration, which generally refers to the reconfiguration of the computation resources, can also be applied to the infrastructure that implements the communication among the tasks. In this case, reconfiguration can be used to dynamically modify the characterising parameters (e.g., bandwidth, latency, and transmission policies) of the communication channels, leading to more efficient utilisation of the communication resources. In Section 3.2, we review the related work regarding reconfiguration of NoC-based communication infrastructures.

2.1.3 High-Level Synthesis

To accelerate algorithms on the reconfigurable fabric, application developers must have access to hardware implementations of the desired functionalities. The design of these accelerators can be performed manually or with the help of automated tools, such as high-level synthesis (HLS) tools. HLS is an automated design process that translates a software program into a functionally equivalent hardware architecture expressed in a hardware description language (HDL). Xilinx provides the Vivado HLS [31] tool, which can translate C and SystemC code into a register-transfer level implementation in VHDL or Verilog. An equivalent open-source tool developed in academia is LegUp [32]. A complete survey of available HLS tools is provided in [33].

HLS tools take the source code as input along with a set of constraints and directives provided by the designer defining design specifications for clock frequency, resources utilisation, and performance. First, the source code is analysed and the basic operations to be mapped into the available hardware components are identified. Next, a control flow graph is constructed, and the operations are scheduled, which means assigning the operations to control steps (i.e. to clock cycles). During this process, the operations are assigned to the hardware components aiming to exploit parallelism to the extent allowed by the user constraints on the resource utilisation as well as by data dependencies. A finite-state machine is then generated to orchestrate the execution of the operations according to the

identified schedule. In general, HLS-generated accelerators perform worse than manually implemented ones [34]. However, the speed-up over a pure software implementation is still acceptable taking into account the reduction in time and effort for the development process. In our research, we use the Vivado HLS tools to generate hardware accelerators from C benchmarks for evaluation purposes.

2.2 Dynamic Partial Reconfiguration

The reconfiguration approach presented in the thesis uses the DPR feature of FPGAs to reconfigure the hardware resources dedicated to accelerating computational tasks. In this section, we provide the background information regarding DPR for Xilinx FPGAs. Further background can be found in [14, 15].

2.2.1 Overview

DPR is a feature of modern FPGAs that allows for dynamic change of hardware modules of an operating FPGA. After the initial configuration of the FPGA with a full-bit-stream, partial bit-streams can be loaded to reconfigure the hardware design implemented in selected regions without compromising the integrity and the functionality of those parts of the device that are not affected by the reconfiguration.

A system using DPR can be conceptually considered as divided into static and dynamic parts. The static part is configured only once at boot-time with a full bit-stream. The dynamic part, which may consist of several independent reconfigurable regions, can be reconfigured multiple times during run-time with different partial bit-files. Figure 2.2 shows an example of an FPGA divided into static and dynamic parts. The dynamic part consists of two reconfigurable regions (A and B). For each region, a partial bit-stream can be loaded from a set to change the hardware implemented in the selected region without interfering with the functionality of the hardware implemented in the static part or in the other reconfigurable regions. For example, in Figure 2.2, the hardware architecture implemented in the reconfigurable region A is modified by loading one of the partial bit-streams A_0, A_1, ..., A_n.

In reconfigurable computing, DPR can be used to increase the flexibility in the choices of algorithms available to an application. Moreover, it can lead to a reduction of the size of the FPGA with a consequent reduction in cost and power consumption, since it allows the re-use of FPGA resources to implement

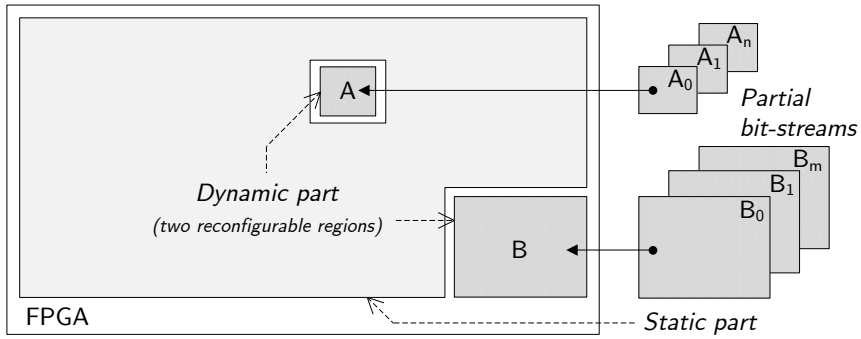

Figure 2.2: An example of an FPGA divided into a static part and a dynamic part. The dynamic part consists the two reconfigurable regions A and B. Partial bit-streams can be loaded to reconfigure these regions.

different functionalities. Various approaches may be used when employing DPR in a reconfigurable system. If the entire computing system is implemented on the FPGA, then the static part of the FPGA is used to implement the processing resources that need to run uninterrupted for the entire execution time of an application, such as processors, on-chip communication fabric, and on-chip memory. The dynamic part of the FPGA is shared between hardware accelerators, specialised co-processors, I/O peripherals, etc. that are only needed for a limited period. Alternatively, if the hardware architecture of a computing system is a hybrid SoC or an ASIC equipped with an eFPGA, then the static part of the FPGA is used to implement only the communication fabric between the hard processor (or ASIC logic) and the dynamic part of the system. In this work, we address the class where the entire computing system is implemented on the FPGA.

2.2.2 ICAP Interface

From a functional point-of-view, an FPGA can be modelled as a two-layered device, as shown in Figure 2.3. The top layer consists of the configurable logic components and interconnections available to implement the user design. The configuration bit-streams are stored in an SRAM configuration memory in the bottom layer. Depending on the content of the configuration memory, the logic functions implemented by the components and their interconnection are modified to construct the desired digital circuit. In other words, the content of the configuration memory defines the hardware design implemented in the

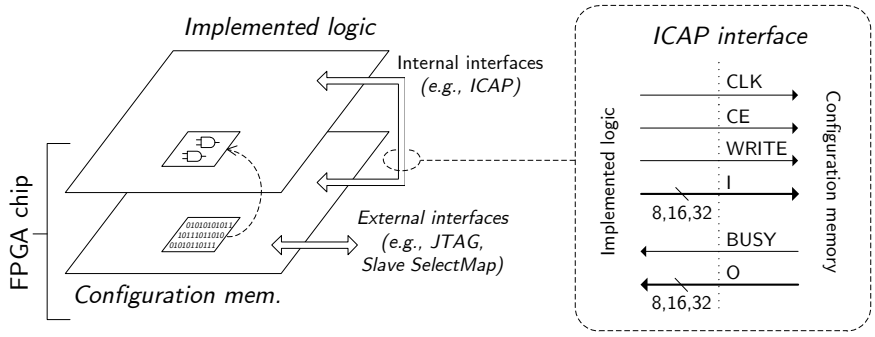

Figure 2.3: An FPGA chip modelled as two layers: one is the configuration memory, and the other is the configurable logic components and interconnections. The external and internal interfaces allow the access to the configuration memory. The ICAP interface is expanded to show the internal signals.

FPGA. In Figure 2.3, the dashed arrow between the two FPGA layers shows this dependency.

Technically, performing a partial reconfiguration consists of changing the content of selected segments of the configuration memory, which corresponds to the modification of the hardware design implemented in the respective regions of logic. The smallest reconfigurable region is called base region, and it corresponds to the smallest addressable segment, called frame, of the FPGA configuration memory space. The size of a frame depends on the FPGA model. For example, in the Xilinx Virtex-6 FPGA, a frame is equivalent to 320 6-input look-up tables (LUTs) and 640 flip-flops (FFs) (80 slices), or 16 DSP elements, or 8 blocks of RAM (BRAMs).

For Xilinx FPGAs, DPR can be performed by loading a partial bit-stream at run-time through one of the FPGA configuration interfaces [35, 36], which can be either off-chip or on-chip as shown in Figure 2.3. The off-chip interfaces are accessible externally from the FPGA chip through dedicated pins. Commonly available off-chip interfaces are JTAG and Slave SelectMAP. The on-chip interfaces are accessible by the user logic implemented on the FPGA itself. In this work, we use the on-chip internal configuration access port (ICAP) on-chip interface. The ICAP is a hardware primitive that provides read/write access to the FPGA configuration memory.

In Figure 2.3, the ICAP interface is expanded to show the internal signals. The naming of some signals of the ICAP interface changes between FPGA families; however, the functionality remains unchanged (Figure 2.3 uses Virtex-6 naming).

Table 2.1: Bit-stream size and calculated reconfiguration time for three different reconfigurable region sizes for a Xilinx Virtex-6 FPGA.

Reconf. region	Hardware resources				Bit-stream size (Byte)	Reconf. time
	FF	LUT	DSP	BRAM		
Base region	640	320	0	0	5 832	$\sim 16\,\mu$s
Patmos proc.	20 480	10 240	64	32	611 712	~ 1.5 ms
Large region	102 720	51 360	256	140	3 074 112	~ 7.6 ms

The ICAP is synchronous and has separate read (O) and write (I) buses, which can be configured to support data widths of 8, 16, or 32 bits. The ICAP is a streaming interface and, for DPR purposes, it receives a partial bit-stream as a continuous input stream through the write bus (I). As shown in Figure 2.3, the address of the FPGA configuration memory and the control signals are not directly available on the interface. All the control information needed to manage the reconfiguration, such as commands and frame address, are encoded into the bit-stream together with the data to be written to the configuration memory. In addition, the ICAP provides information about the current state of the ongoing reconfiguration and communicates when a region is successfully reconfigured through the read bus (O). The ICAP interface and the timing diagram for the signals *enable* (CE), *read/write select* (WRITE), and *busy bit* (BUSY) are explained in [36, 35]. A reconfiguration controller is needed to manage the bit-stream transfer through the ICAP. One of the contributions of this work is the design of a time-predictable reconfiguration controller for the ICAP as presented in Section 5.2. A review of some ICAP controllers from industry and academic environments is presented in Section 3.1.

The time to perform a DPR depends on the amount of data to be transferred over the ICAP as well as its speed. The ICAP for Virtex-5, -6, and 7-series FPGAs has a maximum operation frequency of 100 MHz. Assuming the widest possible interface (32 bits) and the fastest possible clock, a bit-stream can be written at a maximum speed of 400 MB/s. To provide an idea of the reconfiguration time for real applications, Table 2.1 reports calculated results for three different sizes of the reconfigurable region assuming the maximum ICAP transfer speed of 400 MB/s. The values shown in Table 2.1 refer to the Xilinx Virtex-6 FPGA. We expect to have similar reconfiguration times for other FPGA families. The first row reports the results for a base region size, and the second row reports the results for a Patmos processor [22], which is a medium-sized processor described in Subsection 2.4.2. The last row reports the results for a large region equivalent to one-third of the entire FPGA.

2.2.3 Design Flow and Requirements

The full bit-stream associated to the static part and the partial bit-streams associated to the dynamic part of the architecture must be generated using the Xilinx tools. For the 7-series FPGAs, the tool used is Xilinx Vivado, while for older FPGAs the tools are Xilinx ISE and Xilinx PlanAhead. This introduces some differences in the design flow between the 7-series FPGAs [14] and older ones [15]. In the following, we provide a summary of the steps of the design flow, which are common for all Xilinx FPGA families.

At first, the HDL description for the static part of the design and each HDL description targeting the reconfigurable regions must be synthesised independently. The HDL descriptions targeting reconfigurable regions are called reconfigurable modules. The synthesis generates a netlist for the static part and a set of netlists for the reconfigurable modules. The synthesis also produces an estimation of the resources needed to implement each reconfigurable module. The next step is to define the number of reconfigurable regions and to assign the netlists of the reconfigurable modules to each region. This is followed by the definition of the size and the location of each region on the FPGA floor-plan. The resources included in each region should be enough to implement the largest module assigned to it. At this point, one reconfigurable module for every region must be promoted to be part of the full bit-stream for the initial configuration (if nothing should be implemented, then a blank reconfigurable module can be used). The static part is then implemented (place and route) together with the promoted reconfigurable region. Then, the rest of the reconfigurable module are implemented respecting the routing of the signals on the border between the previously implemented static part and the reconfigurable regions. Finally, the static bit-stream and the partial bit-streams are generated. If needed, design constraints for the static part and each reconfigurable module can be provided in every step of the design flow. If the differences between two configurations for the same reconfigurable region are minimal, then the Xilinx tools allow the generation of a differential bit-stream that stores only the differences between a previous configuration and the new one. In this work, we do not use differential bit-streams. Possible extensions of this work using differential bit-streams are presented as future work in Section 8.2.

The current FPGA technology and tools introduce some requirements and limitations related to the use of DPR. During the design flow, specific checking procedures are used to verify that the DPR requirements are respected. In the following, we list the most relevant requirements and limitations for the work presented in the thesis.

- During reconfiguration, the interface between the static part and a reconfigurable region may take unknown values. This may affect the functionality of the design implemented in the static part of the FPGA. Therefore, a specially designed border interface is needed between the static and dynamic parts of a design to decouple the affected reconfigurable region from the rest of the design during reconfiguration. The border interface belongs to the static part of the design and may consist of FFs with enable or 2-to-1 multiplexers on all the signals toward the static part.

- A reconfigurable region must contain a super-set of all interface signals used by the possible hardware modules assigned to the region. The unused output signals need to be forced to a constant value by the logic implemented in the reconfigurable region.

- In a design that includes multiple reconfigurable regions in the dynamic part of the system, only one region can be reconfigured at a time. Even if most FPGAs that support DPR offer two ICAP interfaces, it is still not possible to reconfigure more than one region at a time.

- The reconfigurable region must be rectangular-shaped. Moreover, it is recommended that it is a multiple of the base region. This topological constraint, combined with the fact that the base regions containing DSPs and BRAMs are uniformly distributed in the FPGA chip layout, may lead to an over-inclusion of resources into a reconfigurable area.

- From a system point-of-view, particular attention should be given to avoid deadlocks. For example, if some transactions across a reconfigurable region boundary take multiple cycles to complete, then performing DPR after a transaction has started but before it has completed could cause the system to experience deadlocks. The same could apply for some software polling a register that no longer exists.

2.3 Real-Time Systems

This study explores reconfiguration in real-time systems. In the following, we provide an overview of this class of systems, as well as background on timing-analysis. Further background on real-time systems can be found in [17, 18]. An extensive overview of the methods and a survey of timing-analysis tools can be found in [19].

2.3.1 Overview and Classification

Real-time systems are a class of computing systems characterised by strict constraints on the execution time of tasks in addition to the correctness of the produced result. This definition introduces a set of design specifications on both the software and hardware aspects of a computing system. From a software perspective, this translates to the application of a programming policy that avoids unpredictable or non-analysable behaviours of software tasks (e.g., unbounded loops) and to the use of techniques that guarantee that deadlines are met when multiple tasks share a common resource (e.g., scheduling protocols). From a hardware perspective, the main specification is time predictability of the hardware architecture of a platform targeting real-time applications, which must be developed in a way that allows and simplifies the analysis of the time behaviour of the system.

Real-time systems are commonly used for those applications where a failure to respond in time may lead to undesirable consequences. Depending on the severity of these consequences with respect to the ability to respond in time, real-time systems can be classified into the following three categories:

- **Hard real-time:** Failure to respond in time is not permitted since it can lead to catastrophic consequences, such as loss of life, severe injury, or significant economic loss. Common hard real-time application examples include flight, train, and automotive control systems, medical devices, and industrial control systems.
- **Firm real-time:** Failure to respond in time is allowed within certain limits. If a deadline is missed, then the produced result cannot be used but does not cause excessive problems. A firm real-time application example can be found in assembly and production lines where a sporadic production error can be tolerated.
- **Soft real-time:** Failure to respond in time is allowed within certain limits. If a deadline is missed, then the produced result can still be used. However, this may lead to a degradation of the computing performance of the system. Common soft real-time application examples are audio/video signal processing, quality-of-service management in packet-switched networks, and gaming engines.

In this study, we provide solutions that can satisfy the specification of predictability of hard real-time systems. For this reason, the proposed solutions can also find application in soft and firm real-time systems.

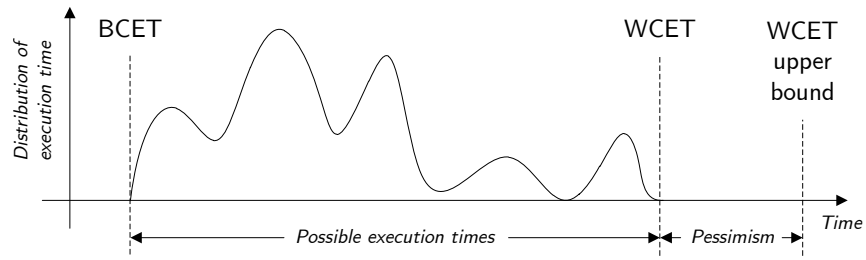

Figure 2.4: Example of a distribution of execution times of a software task. The minimum possible execution time is the BCET, and the maximum is the WCET. The timing-analysis produces an upper bound for the WCET.

2.3.2 Timing-Analysis

In the general case, a real-time application consists of a set of software tasks delivering the required functionality. Typically, the execution time of a task shows a variation that depends on the input data or different operating environments. Figure 2.4 shows an example of a distribution of execution times for a task and defines related terminology. The WCET of a software task is the maximum time interval that a task may take to execute on a specific hardware platform. The minimum interval is called best-case execution time (BCET).

In real-time systems, it is the WCET of the tasks that determines the system performance and its ability to respond in time. For realistic applications, it is impossible to determine the exact WCET of a task, since the space of execution times is too large to be exhaustively explored. Timing-analysis is used to calculate or estimate a safe upper bound of the WCET. The difference between the calculated safe upper bound and the real WCET is called pessimism, as shown in Figure 2.4. The pessimism should be as low as possible and small enough to be acceptable to the system designer.

The WCET of software tasks is dependent on the code structure, data inputs, and targeted hardware architecture. There are two classes of methods used for timing-analysis: measurement-based methods and static methods. In the measurement-based methods, the entire task code or a segment of it is executed on the target hardware or on simulators for a predetermined set of inputs from which the execution time is measured. If the set of inputs coincides with all possible inputs or if the set of inputs that trigger the WCET is known, then this method provides precise WCET results. In real cases, the level of complexity of the tasks to be analysed does not allow for testing all possible inputs, or the set of inputs that trigger the WCET is not known. In these cases, the measurement

can be performed using a subset of all possible input. However, this can only produce an estimate of the WCET and not a safe upper bound. This estimate, usually compensated with a safety margin, can still be used for soft and firm real-time applications.

For hard real-time applications, where only a safe upper bound for the WCET is acceptable, static methods or a combination of measurement-based methods and static methods should be used. Static methods do not rely on the execution of code on the target hardware or simulators. Instead, static methods use an abstract model of the hardware architecture and perform an analysis of the entire task code or segments of it to estimate a safe upper bound for the WCET. Static analysis tools work on source code or disassembled binary executables to extract a control flow graph that models the structure of a program. The control flow graph is then combined with annotations provided by the user (e.g., loop iterations bound and input values interval) and with low-level information regarding the hardware architecture on which the task will execute. The resulting control flow graph is therefore analysed aiming to find the longest path, which corresponds to an upper bound on the WCET of a task running on a given hardware platform. A commercial example of a timing-analysis tool is the aiT WCET Analyzer by AbsInt [37, 38]. The tool statically computes tight bounds for the WCET by directly analysing binary executables and taking cache and pipeline behaviour into account.

The extraction of low-level hardware information needed for the static analysis and the determination of tight WCET bounds can be very difficult or even infeasible for general-purpose architectures designed with the goal of improving the average-case performance. These architectures typically include advanced features such as, complex instruction pipelines, out-of-order execution, branch prediction, and caches, which can make timing-analysis very complex and, thus, increase the pessimism. For this reason, special architectures targeting real-time applications have been developed with special focus on time-predictability and reduction of the WCET and timing-analysis complexity. Examples include the CarCore processor in the multi-core platform developed by the MERASA project [20], the family of processors based on the PRET approach [39, 40], and the T-CREST multi-core platform [21]. The latter is used in this work as a prototyping platform and is presented in Section 2.4.

A possible approach to implement dependable real-time systems is to use a time-triggered architecture (TTA) [41]. This approach is based on the assumption that all the component of a system (e.g., the processing cores of a multi-core platform) share a common knowledge of time. Thus, tasks can be executed according to a pre-defined schedule leading to a deterministic and analysable timing-behaviour, especially for multi-core platforms. Using common knowledge of time to synchronise task execution and a static time-division multiplexing (TDM)

schedule to share in time resources between multiple users (e.g., a communication channel or a hardware accelerator) can be considered as an application of the time-triggered approach. The work presented in this thesis with regards to the reconfiguration of computation resources relies on the TDM approach to provide guarantees on the communication channels provided by the NoC.

2.4 The T-CREST Platform

The T-CREST multi-core platform is used as a target platform for the infrastructure we developed to support reconfiguration. In this section, we provide background on the platform and the associated timing-analysis tools. Further background can be found in [21], and technical background is available in [42].

2.4.1 Overview

T-CREST is a multi-core platform specially developed to be used in real-time applications [21]. All components of the platform are designed with a focus on time-predictability and reducing the complexity and pessimism of the WCET analysis. Figure 2.5 shows a block diagram of T-CREST consisting of a number of processing nodes and two NoCs: one is used for traffic between cores and the shared external memory, and the other is used for message-passing traffic between the nodes. The message-passing NoC is supplemented with reconfigurable features in this work, and it is described in detail in Section 2.5. T-CREST is supported by a C compiler [43] and WCET analysis tools [37, 44], which are described in the following subsections. In this work, we use T-CREST as a bare-metal platform. Applications are run in the platform without the support of an operating system and, instead, dedicated C libraries are used to interact with low-level primitives offered by the hardware.

2.4.2 Patmos Processor

A processing node of T-CREST includes the time-predictable processor Patmos [22]. Patmos is a dual-issue, in-order, reduced instruction set computer (RISC) processor especially optimised for reducing the WCET and simplifying its analysis. For example, the architecture of the pipeline is organised in a way that avoids timing dependencies between instructions. Figure 2.6 shows a

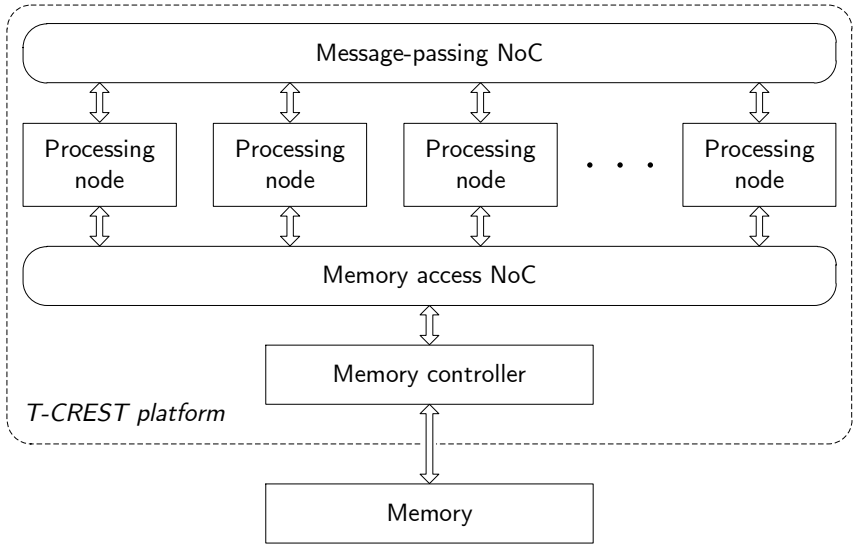

Figure 2.5: Block diagram of the T-CREST multi-core platform.

block diagram of a processing node consisting of the Patmos processor, three different caches, and a local scratch-pad memory (SPM). The communication protocol used by Patmos for memory and I/O devices is a subset of the Open Core Protocol (OCP) [45, 42].

Patmos is equipped with specialised instruction and data caches. The method cache (M$ in Figure 2.6) acts as an instructions cache, and it is characterised by the property of always storing entire functions (in C) [46]. In this way, it ensures that a cache miss can only happen on a function call or return. The compiler splits large functions into smaller ones to fit within the method cache. The stack cache (S$ in Figure 2.6) stores the data stack [47]. At function entry and return, the compiler inserts additional instruction to guarantee that the stack is valid. In this way, it ensures there are not stack cache misses during a function execution. Finally, the data cache (D$ in Figure 2.6) stores the data heap and the statically allocated data.

Patmos is also equipped with a local data SPMs. An SPM is a relatively small private memory coupled to a processor characterised by a single clock cycle access-time. The SPM can be used to store access-time sensitive data structures, and it also acts as data buffer for the message-passing NoC, as is further described in Subsection 2.5.3.

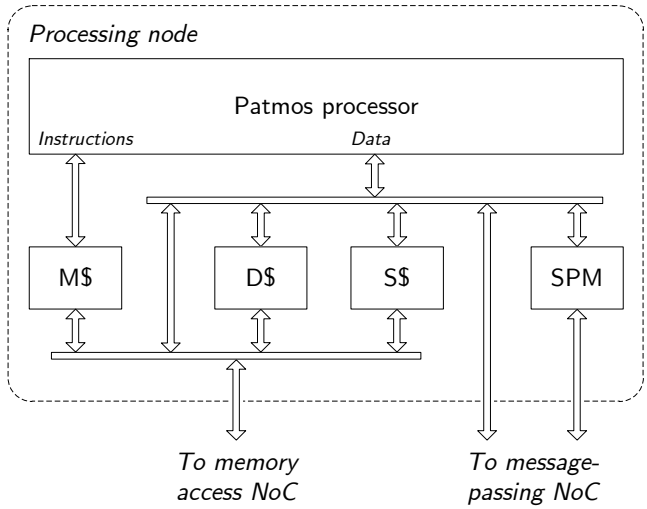

Figure 2.6: Block diagram of a T-CREST processing node and its interfaces to the two NoCs. The node contains the Patmos processor, the method (M$), data (D$), and stack (S$) caches, and the local data SPM.

2.4.3 Support Tools

T-CREST includes an LLVM-based C compiler [43], which supports the instruction-set of Patmos and the special features associated with cache management. Moreover, it preserves the information available during the compilation process that can be valuable for an automated and precise timing analysis. This includes the control-flow structure, as well as user-flow fact annotations provided by the user (e.g., loop iterations bound). This information is used as input to the WCET analysis tools. In addition, the compiler can use the results produced by the WCET analysis tools as feedback, to further optimise compilation aiming to reduce the worst-case performance.

T-CREST is supported by the WCET analysis tool aiT [37, 38] from AbsInt and by the Portable LLVM-based Annotation and Timing-Analysis Integration tool or, in short, platin [44]. In this work, we use the platin tool to compute the WCET of a software task running on the Patmos processor. Therefore, in the following, we provide a brief description of the WCET analysis performed by platin. A detailed description can be found in [44].

The platin tool is a comprehensive framework for WCET-aware compilation and WCET analysis. Analogously to the compiler, platin offers dedicated support

for the specific architecture of Patmos (e.g., the method caching) and it allows the derivation of tight WCET bounds using static methods. This means that to estimate the WCET, platin only examines the software structure without code execution on real hardware. The tool works at both the bit-code level, which is the intermediate representation in LLVM, and at the machine code level. It uses the information generated and preserved during the compilation process to determine a control-flow graph annotated with flow facts. The control-flow graph, combined with low-level timing information of the processor architecture, is therefore analysed for the longest paths, which correspond to a safe upper bound of the WCET of the analysed code segment.

2.4.4 Memory Access NoC

For code and larger data structures, external memory is used. All processors are connected by a NoC to a memory controller and then to the shared external main memory. For this NoC and memory controller, two time-predictable solutions exist: the Bluetree [24] memory NoC with the memory controller presented in [48] and the memory NoC presented in [25] with the memory controller presented in [49] supporting synchronous dynamic random-access memory (SDRAM). In this work, we use the latter solution, which is briefly described in the following.

The memory access NoC is a many-to-one NoC with channels toward the memory controller and a return path for read data. Each processing core is connected to a network interface (NI) offering the same interface provided by the memory controller. The arbitration policy of the NoC is TDM-based. The NI of each core executes a common TDM schedule which reserves a time slot for each processing node. When the time slot for the core arrives, and a memory transaction is pending, the NI immediately acknowledges the transaction to the processing node. The transaction freely flows down the network tree reaching the memory controller. For read transactions, the read data is sent back to all the NIs, which filter the data targeting other cores using transaction timing information.

The TDM-based approach, combined with a memory controller that guarantees static latency for a memory transaction, allows for easy computation of the worst-case memory timing for the processing nodes. The worst-case memory timing includes the maximum waiting time for the time slot assigned to a specific node, the static time needed for a memory transaction, the latency needed by the data to travel back to the NI for read transactions, and, if needed, the time needed by the SDRAM periodic refresh. The latter can be modelled as a periodic additional time slot in the TDM-schedule.

2.5 Argo Message-Passing NoC

The Argo NoC supports message-passing for inter-processor communication in the T-CREST platform. Message-passing consists of moving data from the local memory of a sender processor into the local memory of a recipient processor. This is an efficient way to implement communication between the platform nodes without using shared data structures stored in main memory. In this work, we supplement the Argo NoC with reconfiguration capabilities. Therefore, in the following, we provide an extensive background regarding its functionality and architecture. Further background can be found in [23, 50].

2.5.1 Overview

Argo is a packet-switched and source-routed NoC that uses TDM to statically allocate the network resources to provide VCs characterised by guaranteed communication bandwidth and latency. The Argo NoC offers primitives for moving a block of data from the local memory of a processing node into the local memory of a remote node. This is implemented using direct memory access (DMA) controllers. A dedicated DMA is assigned to the source-end of every virtual circuit (VC) and manages the data transfer. In Argo, the DMA controllers are integrated with the TDM mechanism in the NI [51], which is a very efficient architecture, since it avoids resources for arbitration, buffering, and flow control commonly found in most NoCs. Message-passing between tasks is performed at a higher level in software using the primitives provided by the Argo NoC [52].

The Argo NoC is composed of the packet-switched structure and the NIs. The packet switched structure of the NoC consists of the routers and the links between them. The NIs interface the packed-switched structure with the processing nodes and manage the transmission and reception of packets to and from the packed-switched structure.

Argo can support custom network topologies depending on how the routers are connected in the packet-switched structure. Moreover, the packet-switched structure is available in both globally-synchronous and asynchronous versions. The latter supports a globally-asynchronous/locally-synchronous implementation and uses asynchronous routers and mesochronously clocked NIs. This enables different forms of relaxed synchrony across the entire platform without the use of synchronisation FIFO queues, such as the ability to tolerate skew in both the clock and the reset signal between NIs. More details can be found in [53, 54].

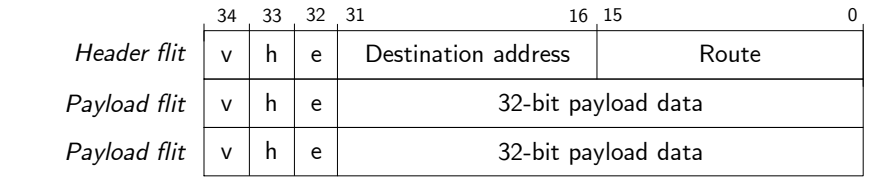

Figure 2.7: Packet structure used in the Argo NoC consisting of a header flit and two 32-bit payload flits.

In this work, we utilise the globally-synchronous version of the packet-switched structure in a bi-torus topology.

2.5.2 TDM-Schedule

The NoC uses TDM-based static routing to share the resources of the packet-switched structure between multiple VCs. The time is divided into periods of constant length, which are subdivided into time slots and statically assigned to VCs.

The scheduler is an offline software tool based on time-expanded graphs and meta-heuristic methods [55]. It uses the bandwidth requirements for the VCs and a description of the NoC topology to generate a schedule that avoids deadlocks, collisions and ensures in-order arrival of packets. The generated schedule and the related predefined routes are stored in the NIs. These inject packets into the packet-switched structure in the time slots as indicated by the schedule. Static scheduling is time predictable since the waiting time to access a time slot can be bounded offline.

In Argo, a NoC packet carries a 64-bit payload. A packet is divided into three flits of 35 bits, as shown in Figure 2.7. The flits are injected in the packet-switched structure one after the other; thus, a time slot corresponds to three clock cycles. The first flit is the header and contains the 16-bit destination address and the 16-bit route. The other two flits contain the 32-bit payloads. The destination address specifies which location of the local memory of the recipient node the data should be written to. Each flit also contains three control bits that specify a valid flit (v), the header flit (h), and the last flit (e). These bits are used by the packet-switched structure and the recipient NIs.

Figure 2.8 shows a 2-by-2 platform along with an example of a TDM schedule for the NoC traffic from the processing node P_0 towards the other nodes. The VCs *c1*, *c2*, and *c3* implement the communication channels between the processing

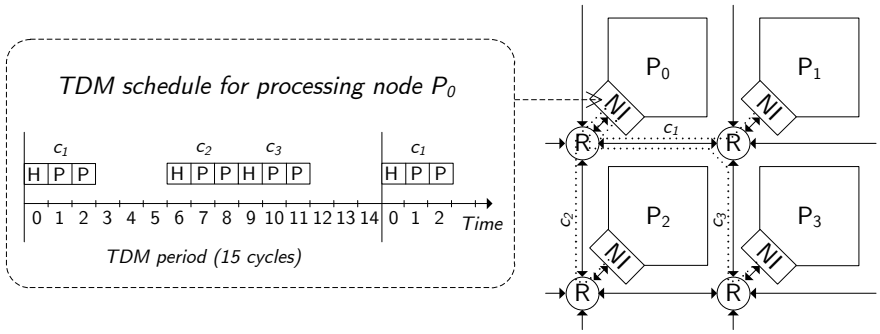

Figure 2.8: An example of a TDM schedule for processing node P_0 towards the other nodes the platform. The route of the VCs *c1*, *c2*, and *c3* are shown in the 2-by-2 section of the platform with a dotted line.

node P_0 and nodes P_1, P_2, and P_3, respectively. The route of these VCs through the NoC is shown in Figure 2.8 with dotted lines. The schedule has a period of 15 clock cycles, and each of the VCs is assigned three clock cycles (a time slot), one for the header and two for the payload flits. The NI of P_0 injects packets to the switched-structure only in the assigned time slots.

The maximum latency needed for sending a message consisting of multiple packets has two contributions: the time interval needed to send the message itself, which depends on the available VC bandwidth, and the maximum waiting time for the first time slot assigned to a specific VC. The typical length of a TDM schedule of Argo is 10-100 clock cycles [55], which is considerably shorter than the execution time of the intercommunicating tasks running on the processing nodes. Moreover, the data moved in a single NoC packet is in general smaller than a typical message size. In other words, the scheduler does not schedule for entire messages between tasks, but for small and frequent packets. Due to the fine granularity of the schedule, the waiting time is negligible with respect to the sending time of the entire message. This leads to efficient utilisation of the allocated bandwidth for both periodic and aperiodic tasks.

2.5.3 NoC Architecture

Router

The router used in Argo is a pipelined crossbar that routes incoming packets according to the routing information contained in the packet itself. As previously

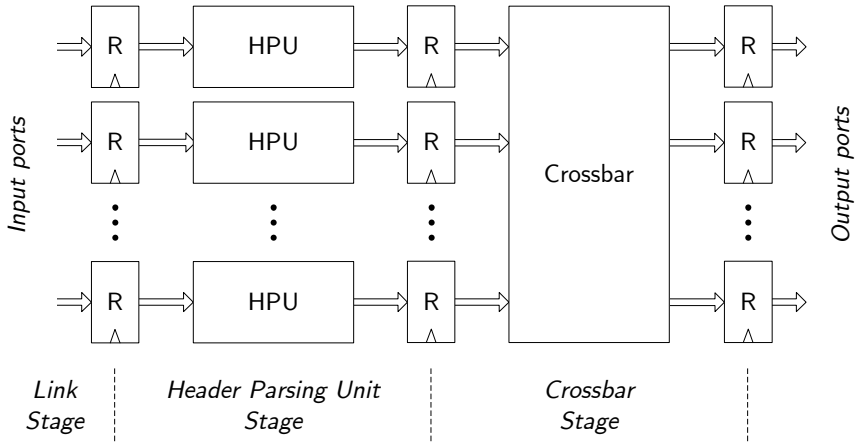

Figure 2.9: A block diagram of the synchronous version of the router used in Argo. The same router is also used in the new version of the NoC presented in Chapter 6.

mentioned, Argo supports both synchronous and asynchronous implementations of the router [56]. In this work, we use the synchronous version.

The implementation of the synchronous router is shown in Figure 2.9 and it consists of three pipeline stages: link, header parsing unit ('HPU' in the figure), and crossbar. The header parsing unit reads the header flit of an incoming packet and, depending on the route field (see Figure 2.7), the packet is properly routed to the output port in the crossbar stage. The control bits in each flit are used by the router to identify the header flit and to keep the crossbar stable until the last flit of a packet has traversed the router.

Network Interface

Figure 2.10 shows a block diagram of the NI and its connection with a processor and with a router [51]. The NI accesses a dual-port SPM. The other port of the SPM is connected to the processor, as shown in Figure 2.6. Moreover, the processor is also directly connected to the NI through the configuration interface, which is used for the initial NI configuration (i.e. loading the static schedule at boot-time) and to set up the DMAs in order to manage the transfer of blocks of data stored in the SPM. The SPM is mapped in the local address space of the processor, which sees it as a regular data SPM.

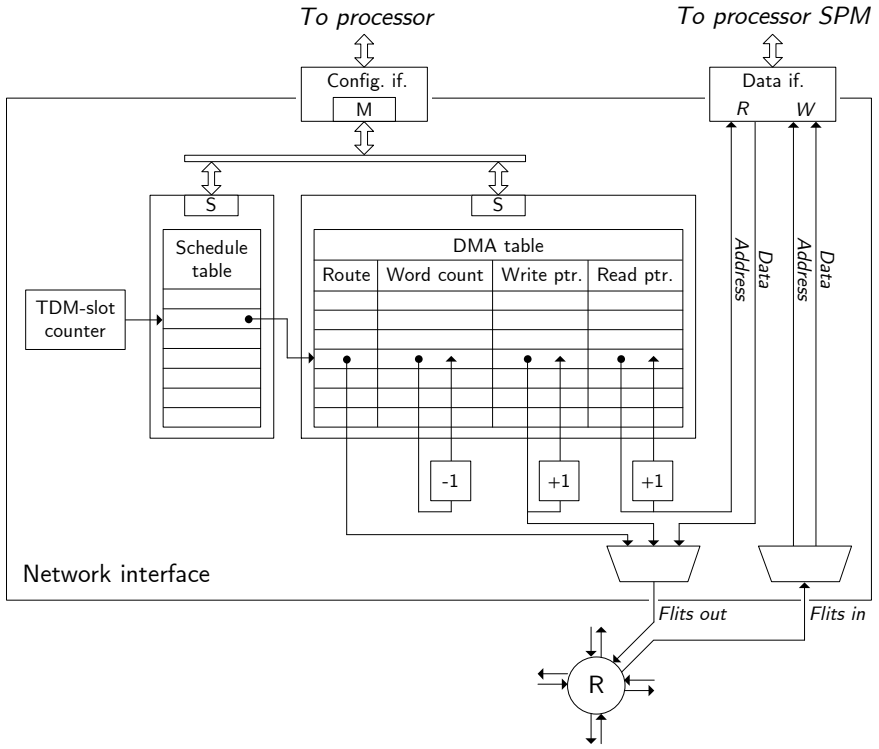

Figure 2.10: A block diagram of the NI used in the Argo NoC and its connection with a processing node (at the top) and with the switched-structure (at the bottom).

As previously mentioned, the NI of Argo stores the schedule and integrates the DMAs that manage the transfer of a block of data from the data SPM of a processing node into the data SPM of a remote node. In the following, we explain the functionality of the NI with reference to Figure 2.10. Each NI contains a TDM slot counter that operates synchronously between all NIs and wraps around at the end of the TDM period. The TDM slot counter indexes the schedule table, which contains the schedule information with one entry per time slot. Every entry specifies which DMA is enabled to send in each time slot according to the schedule. Only one DMA controller can be active at any given time. Entries in the schedule table can be marked as not valid if the NI is not allowed to send in a certain time slot. A DMA is represented by an entry in the DMA table, which holds the route that a packet should follow through the network, the word count, the write pointer into the destination SPM, and the read pointer into the local SPM. The enabled DMA controller reads the payload data of a packet from the local SPM and assembles the outgoing packet containing the route, the write

pointer, and the payload data. The three flits of a packet are then injected in the packet-switched structure one after the other. The word counter is decremented, and the write and read pointers are incremented every time a packet is sent. When a packet is received by the NI, the payload is directly written into the SPM at the destination address specified in the packet header.

A contribution of this work is the extension of the Argo NoC with run-time reconfiguration capabilities. Specifically, we develop a new NI that enables to instantaneously switch between different TDM schedules. The NoC architecture using this new NI is named Argo 2, and it is described in Chapter 6.

Chapter 3

Related Work

This chapter reviews related work in two main research areas: reconfiguration of computation resources and reconfiguration of communication resources. For the first research area, we present system and application level methods and tools supporting the use of DPR, as well as a selection of hardware controllers supporting DPR through the Xilinx ICAP interface. For the second research area, we present a selection of NoCs, which are based on flow control or TDM approaches, offer guaranteed service (GS) communication and support run-time GS reconfiguration. In addition to the coverage of the two main research areas, we also include works with regards to other topics that relate to this thesis, such as task mapping and computation of NoC configurations and a real-time platform supporting multiple use-cases and reconfiguration.

3.1 Reconfiguration of Computation Resources

This section is dedicated to related work regarding the reconfiguration of computation resources. To our knowledge, the use of DPR in real-time systems represents a relatively novel field of research and published works mainly address reconfiguration from a general-purpose perspective. For this reason, in the following, we discuss both works that address and introduce DPR in general-purpose systems and works with a specific focus on real-time systems.

3.1.1 Methods and Tools

Comprehensive surveys of the most relevant hardware aspects and an overview of the state-of-the-art with regards to reconfigurable computing can be found in [57] and [8]. These surveys explore the challenges involved in the design of reconfigurable hardware, addressing both single-chip and multi-chip architectures. In addition, particular focus is given to the problems regarding coupling between static and reconfigurable hardware and to the characteristics of the software that targets reconfigurable machines. This also includes compilation tools that map high-level algorithms directly to the reconfigurable fabric.

In the academic literature, it is possible to identify several frameworks that support run-time reconfiguration. These frameworks are complete top-to-bottom solutions offering compile-time tools for scheduling, analysis, and simulation, given a certain hardware and software characterisation of the system. In the following, we present a selection of these frameworks.

In the PhD thesis presented in [58], the author studies the dynamic behaviour, the run-time management strategies and the design methodologies of reconfigurable architectures, also focusing on the use of DPR. The work proposes a simulation framework for reconfigurable architectures that comprises an application model and an architecture model. The generic application model captures the behaviour of a software application using reconfiguration, and the architecture model captures the behaviour of the hardware supporting reconfiguration. The combination of these two models is used to describe the full dynamic behaviour of the reconfigurable architectures and to enable simulation of this kind of system. The main focus of this work is not on low-level time-predictability, but on performance and modelling.

Another relevant framework is the one presented in [59], called ReCoBus-Builder. The main idea is to use DPR to generate dynamically reconfigurable systems providing one or more run-time reconfigurable areas and using a fixed bus infrastructure or dedicated point-to-point links for the communication between the reconfigurable resources and other parts of the system. The framework addresses component-based reconfigurable architectures for general-purpose systems. Therefore, strict timing constraints are not taken into account, and performance is measured in terms of the average case speed-up.

The work presented in [60] proposes a software framework, called FRED, that exploits hardware accelerators in combination with DPR in the development of safety-critical real-time application on a hardware platform that includes a processor and an FPGA. The authors present a model of the platform and of the computational tasks where sections of the workload of tasks are acceler-

ated using reconfigurable hardware. Using this model, the authors propose a scheduling infrastructure supporting response-time analysis and verification of the schedulability of a real-time task set under given constraints and assumptions. In contrast to this modelling and scheduling work, our work is more focused on providing efficient hardware/software solutions, low-level timing-analysis and practical design experiments.

The work presented in [61, 62] proposes the RAMPSoC multi-processor SoC and approach. RAMPSoC is a run-time adaptive multi-core system that uses DPR to reconfigure at run-time the processors, the communication infrastructure, the instruction set of the processors, and the hardware accelerators in order to better meet performance, area, and power constraints. The RAMPSoC approach also includes a software toolchain of custom and commercial tools that provide support for the entire design flow including application partitioning, profiling, and hardware and bit-stream generation. RAMPSoC is supported by a dedicated real-time operating system, called CAP-OS [63], which manages the task scheduling, the resource allocation and the reconfiguration process.

From a system-level perspective, the software/software and hardware/software partitioning as well as the scheduling of tasks are the most relevant topic related to reconfigurable architecture. The work presented in [64] provides an overview of the hardware/software partitioning, scheduling, and placement issues. It proposes an exact approach and a heuristic approach for hardware/software partitioning in a system that uses DPR. The paper shows that taking into account key factors, such as placement implications and configuration pre-fetching, leads to a reduction of the tasks schedule length.

Another work presenting methodologies and tools for both software/software and hardware/software partitioning is presented in [65]. In this work, the authors use a custom library to generate the call graph for the application tasks (functions) and perform the software/software partitioning taking into account the inter-tasks communication requirements. In addition, a specific tool is used to profile the code in order to identify computationally-intensive sections that can be accelerated using reconfigurable hardware.

In [66], the authors present the PaRA-Sched automated design methodology for task scheduling in reconfigurable systems. This methodology takes into account DPR in the scheduling infrastructure to improve overall performance, attempting to automatically mask reconfiguration time when possible. This allows rapid exploration of the impact of the DPR use during early stages of the design process.

As previously mentioned, the software applications using the reconfigurable features presented in this thesis are executed bare-metal on the T-CREST

platform. Specialised C libraries are therefore used to interact with these features. Nevertheless, operating systems supporting reconfiguration are an active topic of research. An example is the CAP-OS operating system [63] supporting the RAMPSoC platform presented above.

Another example is the RECONOS operating system presented in [67], which extends the multi-threading programming model to support hardware threads also defining a standardised interface between them and the operating system. RECONOS introduces the concept of delegate thread, which is a lightweight software thread characterising and interfacing a reconfigurable hardware thread implemented on an FPGA.

An earlier work on operating systems that also includes methods addressing the topology of the reconfigurable area is presented in [68]. The work proposes a heuristic algorithm for run-time scheduling of real-time tasks implemented in hardware and reconfigured using DPR. The scheduling algorithm takes into account topological constraints with regards to the placement of reconfigurable modules. In other words, the task scheduling is executed both in time and in space (e.g., applying policies to reduce the fragmentation of the reconfigurable area). Solutions are provided considering both one degree of freedom in placing hardware tasks in the configurable region (1-dimension model) and two degrees of freedom (2-dimensions model).

3.1.2 Reconfiguration Controllers

The design of a hardware/software infrastructure to support time-predictable reconfiguration of the computation resources is one of the contributions of this thesis. This infrastructure is based on our time-predictable DPR controller RT-ICAP, presented in Section 5.2. In the following, we present a selection of DPR controllers from industry or developed in the academic environment. These controllers are used for comparisons with our controller in Subsection 7.1.1.

The XPS_HWICAP reconfiguration controller [69], provided by Xilinx, is an IP core that provides the support for DPR using a set of software functions offered in processor-specific libraries and it is designed to be connected to the processor local bus [70]. The AXI_HWICAP controller [71] by Xilinx delivers the same functionality as the XPS_HWICAP controller, but it can be interfaced to the AXI4-Lite bus [72]. The associated software library allows an application programmer to write and read configuration bit-streams, and it enables the modification of single look-up tables and flip-flop properties.

Other reconfiguration controllers managed by the software running on a processor are presented in [73]. The work presents and investigates the performance of three controller architectures, named MST_HWICAP, DMA_HWICAP, and BRAM_HWICAP, strongly inspired by the Xilinx XPS_HWICAP controller, and therefore very similar in terms of functionality.

The ZyCAP controller [74] is a custom controller for hybrid SoCs, such as the Xilinx Zynq. The ZyCAP controller is connected to the hardcore processor through the AXI4-Lite bus and to the system memory where the bit-streams are stored through the AXI4 bus [72]. A DMA controller loads the bit-stream during reconfiguration using the ICAP interface. Software drivers are associated with the controller and allow the hard processor to manage the reconfiguration process.

The work presented in [75] proposes an approach where the ICAP interface is integrated into the datapath of a processor, enabling it to directly access the FPGA configuration memory. A minimal finite-state machine (FSM) is used to manage the low-level communication with the processor and to handle the interfacing with the ICAP.

For all the controllers and solutions mentioned above, most of the functionality is provided by software executing on a processor that reads from or writes to the controller interface. These frequent accesses to the controller through the system bus affect the reconfiguration speed and may increase the WCET pessimism, since I/O functions may be difficult to analyse. Moreover, for the Xilinx controllers, only the netlists are available, making the WCET analysis very complicated or even impossible. In our solution, we aim to minimise the interaction between the processor and the DPR controller to increase time-predictability.

Another class of controllers perform reconfiguration without the assistance of a processor, somewhat similar to a DMA controller. The PRC controller [76], provided by Xilinx, is an IP core designed to independently manage DPR in reconfigurable designs targeting the Xilinx 7 series FPGAs. The controller is interfaced to a processor through the AXI4-Lite bus [72]. When it receives a software or hardware trigger, it can independently manage the reconfiguration of multiple regions by reading bit-streams from a memory connected to the AXI4-Lite bus and writing these into the ICAP interface. Also for this Xilinx controller, only the netlist is available.

The DPRM controller presented in [77] and the ICAP-I controller presented in [78] offer similar functionality for Xilinx Virtex-5 and Virtex-4 FPGAs, respectively. The DPRM controller supports only bit-stream transfers from off-chip flash memories into the FPGA configuration memory, while the ICAP-I controller also supports the transfer of bit-streams stored in on-chip BRAMs. The authors use

an FSM to transfer the partial bit-streams stored in on-chip BRAMs or off-chip flash memories into the FPGA configuration memory. The architecture of these two controllers and the BRAM_HWICAP are the ones that most resemble the architecture of our RT-ICAP controller.

The D^2PR controller presented in [79] is an example of a minimal custom DPR controller connected to the ICAP interface. The controller can be configured to include circuitry for error detection and correction on the bit-streams, aiming to improve safety and reliability of DPR. Our controller relies on the default checksum-based bit-stream checking already supported by Xilinx FPGAs.

Some controllers offer fine-grain reconfiguration of individual LUTs, which can be exploited to reduce the number of needed pre-computed bit-streams. The controller still needs to write a bit-stream into the FPGA, but it only needs to modify specific fields of the bit-stream to reconfigure the Boolean function performed by a particular LUT. This approach is vendor-specific and device-specific since the structure of the bit-stream varies depending on the FPGA model. Moreover, the FPGA vendors do not release official information regarding the structure of the bit-streams, and a certain level of uncertainty is always included when this kind of approach is used. An example is a high-speed ICAP controller, named AC_ICAP, presented in [80]. It is entirely implemented in hardware and supports the reading and writing of full bit-streams and the modification of individual LUTs for Virtex-5 and Kintex-7 FPGAs. An earlier work that implements the reconfiguration of LUTs is the ICAP controller for Virtex-II FPGAs presented in [81].

3.2 Reconfiguration of Communication Resources

This section reviews related work regarding the reconfiguration of the communication resources. In this thesis, we present a NoC that offers GS in terms of bandwidth and latency for end-to-end communication flows and that supports run-time reconfiguration of the provided GS. Therefore, in the following, we present a selection of NoCs based on flow control or TDM that support GS traffic and run-time reconfiguration.

3.2.1 NoCs Based on Flow Control

One possible approach to provide GS connections is to use non-blocking routers in combination with mechanisms that constrain packet injection rates (flow control). This class of NoCs is reconfigured simply by changing at run-time the parameters that regulate the packet injection rates to the new requirements.

The NoC used in the Kalray MPPA-256 processor uses flow regulation [82], output-buffered routers with round-robin arbitration, and no flow control. Network calculus [83] is used to calculate the injection rate parameters, such that buffer overflows are avoided and GS requirements are fulfilled. Reconfiguration is performed by modifying the routing tables and injection rate parameters in the NIs.

The IDAMC NoC presented in [84] is a source-routed NoC that uses a combination of credit-based flow control and virtual channel input buffers to provide GS by implementing the back-suction scheme [85]. The idea is to properly manage the priority of non-critical traffic to allow the critical traffic to meet the deadline in term of latency.

Another example is the Mango NoC presented in [86], which also uses non-blocking routers and flow regulation. Each end-to-end connection is allocated to a unique buffer in the output port of every router in the path of the connection, while links are shared between different virtual channels. The buffers are managed using credit-based flow control between them. The bandwidth and latency of the different connections are configured by setting priorities in the output port arbiters of each router and by setting the injection rate at the source NI. Due to its structure, the reconfiguration of the Mango NoC directly interacts with the rate-control mechanism in the NIs, and the crossbars and the arbiters in the routers. Moreover, since the crossbars and the arbiters in the routers are configured using best-effort traffic, the time-predictability of performing a reconfiguration may be affected.

To our knowledge, detailed descriptions on how reconfiguration is handled in Kalray, Mango and IDAMC have not been published. However, we can assume that during reconfiguration, the set up of a new connection must involve the modification of flow regulation parameters and the tearing down an existing connection must involve draining in-flight packets from the buffers in the end-to-end connection path.

The work presented in [87] proposes the concept of using a centralised NoC manager with the scope of searching and allocating the routes to support GS traffic between two nodes. The NoC manager is implemented in hardware,

and it bases its search functionality on the HAGAR approach [88] for graph exploration. The core manager, which manages the execution of real-time tasks, co-operates with the NoC manager and requires the allocation of the needed communication channels. The GS channel search process is executed in a time interval proportional to the length of the found path. In the worst case, the search may not succeed leading to aborting the related real-time tasks and compromising the predictability of the systems.

3.2.2 NoCs Based on TDM

An alternative to the flow-regulation approach is the use of VC switching implemented using static scheduling and TDM. Major examples are the Æthereal family of NoCs [89, 90] and the original Argo NoC [23, 50]. In this thesis, we extend the latter with run-time reconfiguration capabilities, as presented in Chapter 6. These TDM-based NoCs are configured by initialising schedule tables and routing tables in the NIs and/or in the routers.

The original Æthereal NoC [91] supports both GS and best-effort traffic. The scheduling and routing tables are stored in the NIs and in the routers. Reconfiguration is supported by changing the content of these tables using best-effort packets. This approach is analogous to the Mango NoC one, where the time-predictability of a reconfiguration is compromised since best-effort traffic is used. The dAElite NoC [90], which focuses on multicast, overcomes this problem by introducing a separate dedicated NoC with a tree topology for the distribution of the schedule and routing information during run-time reconfiguration.

The aelite NoC [92, 89] only supports GS traffic and it is based on source routing. Therefore, the routers are simple pipelined switches, and both schedule tables and routing tables are in the NIs. For aelite, reconfiguration involves sending messages across the NoC itself using GS connections from a reconfiguration master to the schedule and routing tables that are required to change. These GS connections are reserved for this purpose only.

It must be noted that for all the presented cases, reconfiguration is done incrementally. This implies that VCs that persist across the reconfiguration cannot be re-mapped, which can lead to fragmentation and sub-optimal use of resources. Moreover, if re-mapping of VCs is needed, the entire application must be suspended during the reconfiguration. Our solution supports instantaneous reconfiguration, including re-mapping of the VCs that persist between configurations.

The NoC reconfiguration features of the Æthereal NoC are supported and abstracted by the Æthereal Run-Time library [93]. This library offers a set of function for controlling the reconfiguration process, also addressing the problem of leaving the system in a state from which operation can continue after the reconfiguration. The same principles presented in this work for the Æthereal NoC can be applied to other NoCs supporting run-time reconfiguration.

The Nostrum [94] NoC supports GS and best-effort traffic. The GS traffic is supported by loading information in a container that continuously loops between the source and the destination node. Thus, a VC corresponds to a set of predefined containers looping between two nodes. The TDM period is implicitly related to the number of containers in a VC and to the length of the loop. The Nostrum NoC allows some form of reconfiguration of the GS VCs. The route is determined at design-time, and it cannot be changed during operation, but the bandwidth can be varied at run-time by modifying the numbers of containers used by a VC. To reduce bandwidth, containers can be removed from the loop. To increase bandwidth, containers can be inserted to increase bandwidth. However, the time interval needed to insert a container cannot be guaranteed since the new container does not have guaranteed access to the network.

The TTNoC presented in [95] supports the transmission of periodic time-triggered messages between nodes. The period and the phase of the messages are defined and set at boot-time. The NoC supports run-time reconfiguration since the schedule can be updated by the trusted network authority component of the NoC to adapt the offered bandwidths. The update of the schedule must be performed in several steps and following the correct order of disabling and enabling the receiving and the transmitting nodes in order not to disturb the time slots not affected by the reconfiguration.

A more recent TTNoC that supports time-triggered traffic is the one presented in [96]. For this NoC, a detailed description of how the reconfiguration is performed is not provided. However, we can assume that the reconfiguration should be carried out by a trusted network authority component in a similar way to the TTNoC presented above.

The original version of the Argo NoC [23, 50], already presented in Section 2.5, has some functional similarity with the aelite NoC: it only supports GS traffic and it also uses a TDM router with source routing. The original Argo NoC uses an efficient NI [51] in which the DMA controllers are integrated with the TDM scheduling, avoiding VC buffers and the credit-based flow control, which account for most of the area of the NIs of the Æthereal, aelite, and dAElite NoCs. The original Argo NoC does not support reconfiguration. A preliminary solution for supplementing the Argo NoC with reconfiguration capabilities was studied and implemented in the work presented in [97], where a dedicated asynchronous

tree network is used to broadcast the new TDM schedule and the commands to trigger the reconfiguration. Essentially, the extension of the Argo NoC presented in this thesis implements the same functionality, but using the NoC itself for schedule and commands distribution.

The scientific literature also includes more abstract/theoretical approaches towards modelling reconfigurable NoCs. An example of this is the mathematical framework developed in [98], where the authors develop a model for the dynamic behaviour of reconfigurable NoCs and use it to formulate NoC reconfiguration as a dynamic optimisation problem. In our work, we mainly focus on the NoC hardware architecture and implementation and on the low-level functionality and timing analysis.

3.3 Other Related Topics

This section collects the works with regards to other topics that relate to this thesis, such as task mapping and computation of NoC configurations and a real-time platform supporting multiple use-cases and reconfiguration.

Subsection 3.1.1 reviewed some related work addressing methodologies and tools for software/software and hardware/software partitioning. Once an application is partitioned in software and hardware tasks depending on the computation workload and the other parameters driving the partitioning algorithms, the mapping of this tasks to the processing nodes of the multi-core platform is executed. The T-CREST platform is not supported by a tool for automatically mapping tasks to the cores. However, solutions have been developed for specific applications. For example, in the audio DSP application presented in paper [C3] in our list of publications and used as case study in Section 7.3, a set of multiple audio effects are mapped to the cores of the T-CREST platform by a specialised software tool at compile-time. This tool receives in input the required chain of effect and the WCET of each effect, and performs the mapping of the effects to the cores aiming to maximise the utilisation of each processor without exceeding the audio sampling period (multiple effects can be mapped to the same processor). Then, the tool produces the GS communication requirements for the inter-processors message-passing NoC.

The work presented in [99] proposes a mapping/resource management method for NoC-based platforms containing reconfigurable processing elements implemented on an FPGA. The algorithm receives as input the application specification (task graph), the user requirements, and the current resource utilisation of the platform. Then, it uses heuristics to map tasks to the best fitting processing element of

the platform. The mapping is executed at run-time, and task migration is also supported.

Another work that describes a run-time mapping technique is presented in [100]. This work uses a minimisation algorithm to map at a set of tasks to a set of processor aiming to reduce the total amount of energy consumption to the minimum, without compromising the quality-of-service provided by the mapped application.

The work presented in [101] extends the heuristic mapping algorithm presented in [102] to be used for multiple use-cases. A use-case is defined as a set of applications that run on the platform at the same time. Assuming that all the use-cases use the same NoC configuration, the work proposes the construction of a synthetic worst-case use-case from the provided use-cases on which the mapping algorithms is applied. Thus, satisfying the requirements of the single use-cases. The paper also explores the possibility of reconfiguring the NoC and using voltage and frequency scaling during the switch between use-cases to adapt the services provided by the NoC to the current use-case.

The work presented in [103] addresses the problem of mapping multiple use-cases and generating NoC configurations without the use of a synthetic worst-case use-case. The work proposes a heuristic method that takes into account the possible use-cases and the transitions between them to generate NoC configurations. The transition between these configurations is achieved by removing and adding virtual communication channels by modifying the TDM schedule of the NoC during a transition between use cases without disrupting the services provided by the NoC. The possibility of partially modifying the guaranteed services provided by the NoC by modifying the TDM schedule must be supported by the NoC architecture.

A complete NoC-based multi-core platform supporting multiple use-cases real-time applications and reconfiguration is CompSOC [104, 105]. The CompSOC platform consists of a set of computing tiles interconnected with the one of the reconfigurable NoCs Æthereal, dAElite, and aelite presented in the previous subsection. A computing tile includes a processor, local memories, and DMA controllers. CompSOC is supported by a tool-chain for the generation of the hardware platform, including the NoC and the controller/arbiter for main memory. In addition, tools for automatic mapping of cyclo-static data-flow applications to the platform, DRAM power estimation, and WCET and ACET analysis for real-time memories are also provided. Application based on other models of computations, such as time-triggered, are also supported.

The CompSOC platform and design flow offer composability (complete absence of any interference) and time-predictability. Composability is achieved by offering

a virtual execution platform per application (or use-case), which is equivalent to offer independent design, verification, and execution isolation. Due to the isolation offered by the visualisation, different applications can be started and stopped at run-time without affecting other ones, and different applications can use different models of computation. Time-predictability is achieved by ensuring that each virtual platform is characterised by well-defined timing properties. TDM-based policies are widely used in the CompSOC platform to obtain both composability and time-predictability (e.g., there is not cross-application interference when the access to a certain resource is scheduled and the worst-case access time is bounded).

In CompSOC, run-time reconfiguration is achieved by using software-programmable hardware components (i.e. components that can be configured via software such as the processors and the NoC) and by allocating enough resources to support the addition of an application without affecting the ones already running. In our approach for the reconfiguration of the communication resources, we also use software-programmable hardware components since the extension to the Argo NoC presented in this work allows the NoC to be reconfigured via software. However, for the reconfiguration of computation resource, DPR is used to reconfigure the resources that belong to a different use-case. This could lead to a reduction of the hardware cost with respect to the CompSOC since not all the resources (or a subset of them) need to be implemented simultaneously. For the CompSOC platform, the problem of time-predictable reconfiguration of tasks at run-time has been addresses also from a software perspective. For example, the work presented in [106] proposes a software architecture that allows to manage multi-core partitions and offers composable and time-predictable dynamic loading of applications. In contrast, our work mainly addresses the hardware architecture supporting the reconfiguration, with only small contributions from the software perspective.

Chapter 4

Approach to Reconfiguration

This chapter introduces our approach to reconfiguration of both computation and communication resources. At first, we present our approach for supporting reconfiguration in multi-core real-time systems. This is followed by an explanation of how GS requirements can be extracted from applications with multiple modes of operation and by a description of how the reconfiguration process can be modelled at task-level. Finally, we list a set of expected outcomes and evaluation metrics deriving from the use of reconfiguration according to the proposed approach. The approach and the techniques presented in this chapter were in part published in the papers [J1], [J3], and [C6] in our list of publications.

4.1 Definition of Communication and Computation Resources

Multi-core platforms can be modelled as a set of computing nodes (the cores) and one or more communication fabrics connecting the nodes to each other, as well as to main memory. In the scope of this thesis, we refer to the hardware infrastructure that implements the computing nodes as computation resources and to the hardware infrastructure that implements the communication fabric as communication resources. More specifically, a computation resource can

be defined as a hardware infrastructure that provides computation primitives used for execution of tasks. Similarly, a communication resource can be defined as a hardware infrastructure that provides communication primitives used to exchange data between tasks or between a task and main memory. Examples of computation resources are processors, co-processors, and hardware accelerators. Examples of communication resources are NoCs and buses.

As previously mentioned, in real-time systems, the WCET of the software tasks of an application determines its ability to respond in time. Therefore, a platform that supports real-time applications must provide GS for both computation and communication resources. This translate into design specifications for the functionalities offered by the hardware infrastructure. For example, computation resources must provide guarantees on the execution time of the operations performed by the hardware (e.g., an upper bound to the WCET of a task running on a hardware accelerator). Communication resources must provide guarantees on the latency and the bandwidth of end-to-end communication channels. In the case of the Argo NoC, these are the VCs.

In this thesis, we extend a multi-core platform targeting real-time systems with run-time reconfiguration capabilities. Therefore, also the reconfiguration process itself, which can be considered as a hardware primitive offered by the platform, must be performed in a time-predictable manner. In other words, a platform that supports reconfiguration to be used as part of an application must provide GS reconfiguration capabilities. This is equivalent to guaranteeing that the time interval needed to perform a reconfiguration has a static upper bound.

4.2 Reconfiguration at Mode Changes

Real-time applications often have multiple modes of operations (sometimes referred to as use-cases) [107]. In a multi-core platform, a mode of operation is defined as a set of tasks running on a set of processors and communicating across a set of channels provided by the communication fabric. A mode change is defined as a change in the subset of the executing tasks during normal operation of the system or in response to an external event [108, p.340]. A mode change happening during normal operation of the system is triggered at a well-defined moment in the application execution, while a mode change occurring in response to an external event is triggered to adapt the system behaviour to new environment conditions.

A simplified, but very clear, example of mode changes can be found in the control system of an aeroplane going through three phases: take-off, cruise, and landing.

Each phase has a well-defined set of tasks to manage the aeroplane functionality. These sets of tasks constitute the modes of operation. Normal operation mode changes happen in the transitions between take-off and cruise and between cruise and landing. A mode change in response to an external event can happen, for instance, if an alarm requires the execution of a set of tasks to manage the specific situation that triggered the alarm.

Since each mode of operation consists of as a set of communicating tasks running on a set of processors, different modes may have different GS requirements for computation and communication resources. A single configuration of the hardware platform may be unable to support the requirements of a given real-time application or, in the general case, the hardware platform would include resources used only for a limited period of time during the execution of a specific mode. Therefore, the proposed approach is to associate reconfiguration to a mode change. More specifically, we suggest exploiting run-time reconfiguration during a mode change to adapt the GS provided by the hardware platform to the requirements of the current mode running on it.

We address the reconfiguration of the computation and the communication resources with two different architectural approaches presented in two dedicated thesis chapters, as described in the following:

- **Reconfiguration of computation resources:** This consists of the modification of the hardware architecture of selected computation resources, such as hardware accelerators, and it is supported using the DPR feature of FPGAs. The main idea is to share the dynamic region of the FPGA between the accelerators that belong to different modes of operation and thus not simultaneously used. The hardware/software infrastructure developed to support this is presented in Chapter 5.

- **Reconfiguration of communication resources:** This consists of setting up, tearing down, and modifying the bandwidth and latency of the end-to-end channels provided by the communication fabric. This is supported by extending the message-passing NoC Argo with run-time reconfiguration capabilities. The architecture developed to support this is presented in Chapter 6.

For both types of reconfiguration, the presented approaches and architectures provide GS reconfiguration capabilities, and they are prototyped and evaluated using the T-CREST multi-core platform.

4.3 Extraction of Guaranteed Service Requirements

According to the proposed approach, each mode of operation is characterised by a set of GS requirements for the computation and the communication resources. In the following, we present how the GS requirements can be extracted from a multi-mode application when mapped to a multi-core platform. Figure 4.1 presents the logical steps needed by this process.

Figure 4.1(a) models a real-time application characterised by multiple modes as an oriented graph where the vertices represent the modes of operation M_i, $i \in \mathbb{N}$ and the edges represent the possible transitions between the modes. Each mode consists of a set of communicating tasks, and it is also modelled as an oriented graph in Figure 4.1(b). Here, the vertices represent the computational tasks τ_i, $i \in \mathbb{N}$ and the edges represent the communication channels c_i, $i \in \mathbb{N}$ between them.

It should be noted that there are three types of tasks τ_i:

- Tasks that are purely implemented in software and need to be mapped to a processor (indicated with the subscript '$\_sw$').
- Tasks that can completely be implemented in hardware and need to be mapped to a dedicated hardware accelerator or to a reconfigurable region able to implement the accelerator (indicated with the subscript '$\_hw$').
- Tasks that are implemented in software, but require the use of an accelerator to execute sections of the code. These tasks need to be mapped to a processor equipped with the required accelerator or with a reconfigurable region where the required accelerator can be implemented (indicated with the subscript '$\_hy$', standing for 'hybrid').

The tasks τ_i shown in Figure 4.1(b) are mapped to the nodes of the multi-core platform as shown in Figure 4.1(c). For this graph, the vertices represent the platform nodes, and the edges represent the communication channels between each pair of nodes. Some nodes are processors P_i, $i \in \mathbb{N}$, some are hardware accelerators (or reconfigurable regions) HW_i, $i \in \mathbb{N}$, and some are processors equipped with dedicated hardware accelerators (or reconfigurable regions) P_{i_hy}, $i \in \mathbb{N}$. The tasks are mapped to the node type matching the resources requirements of each task. Further details and discussion regarding the implementation of these three types of nodes are provided in Section 5.1.

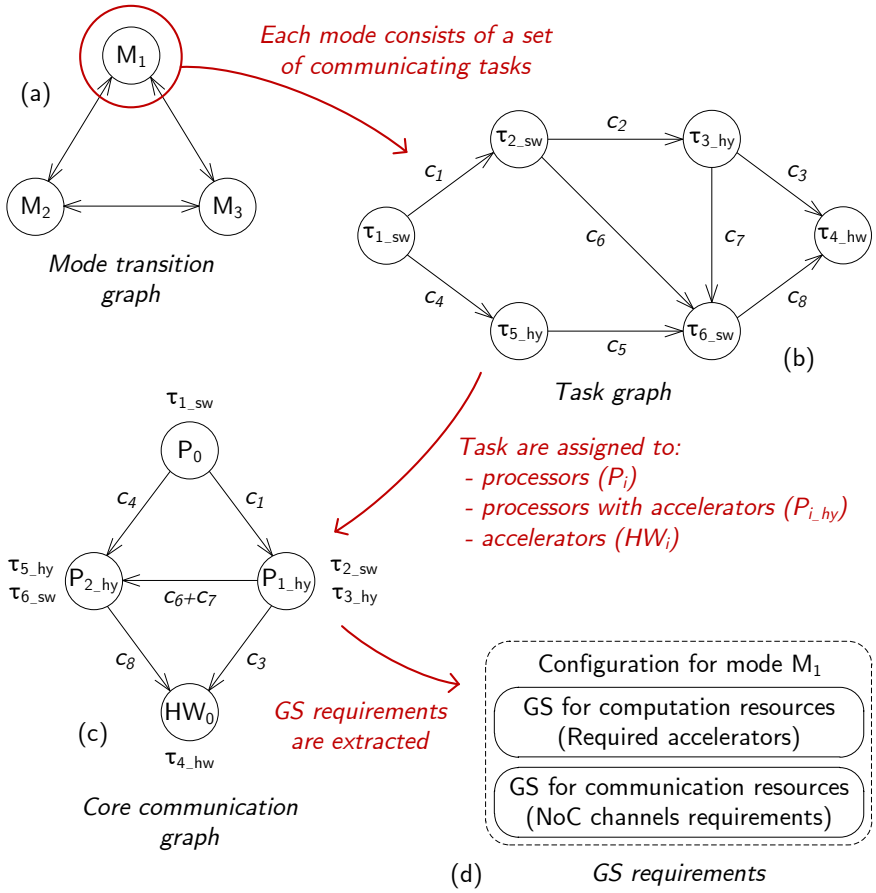

Figure 4.1: Steps needed for the extraction of the GS requirements for mode M_1 of a multi-mode example application. The steps need to be repeated for each mode of operation.

The graph shown in Figure 4.1(c) defines the computation resource requirements for the platform nodes and the communication resource requirements for the communication fabric connecting the nodes, as shown in Figure 4.1(d). The steps shown in the figure for the mode M_1 need to be repeated for all the other modes of the platform. In this way, it is possible to obtain the GS requirements for all the modes of operation for both computation and communication resources. These sets of requirements are called configurations. Reconfiguration is used to switch platform configuration during a mode change. The mapping process and the generation of the configurations are currently performed manually in the T-CREST platform. In the following chapters, we assume that the

communication resources requirements for each mode are provided as input to the tools we have developed to support reconfiguration.

4.4 Model of the Reconfiguration Process

From a task-level point of view, the reconfiguration process can be modelled as a task representing the operations that the processor in charge of the reconfiguration must execute during a mode change. These include interacting with the hardware to trigger the reconfiguration, stopping the tasks belonging to the old mode, and starting the tasks belonging to the new mode. The mode transition graph presented in Figure 4.1(a) can be extended to include the reconfiguration process. The modified graph is shown in Figure 4.2 where reconfiguration modes RM_{ij}, i, $j \in \mathbb{N}$ have been added in the transitions between the modes. A reconfiguration mode can be considered as an independent mode of operation where the task that models the reconfiguration process is executed together with the tasks that persist through a mode change.

To clarify this concept, we provide a small example related to the mode transition graph shown in Figure 4.2. Figure 4.3 shows a time diagram of the execution of the tasks during a mode change between the modes M_1 and M_2. M_1 consists of the tasks τ_1 and τ_2, while M_2 consists of the tasks τ_1 and τ_3. The reconfiguration process is modelled by the task τ_{rec_12}. Tasks τ_1, τ_2, and τ_3 run on the processor P_1, while task τ_{rec_12} runs on processor P_0. We assume that τ_2 and τ_3 need different computation resources to be implemented and different communication GS provided by the inter-processor NoC. We also assume that the periodic execution of τ_1 cannot be suspended during the mode change.

In the time diagram of Figure 4.3, we can observe that during the reconfiguration mode RM_{12}, task τ_1 continues to run, and task τ_{rec_12}, which handles the reconfiguration process, is executed by the processor P_0. The value C_{rec_12} is the WCET of the task τ_{rec_12}. In general, the WCET C_{rec_nm} of the task τ_{rec_nm} for a reconfiguration mode RM_{nm}, $n, m \in \mathbb{N}$ is the time needed to perform a reconfiguration from a system point of view. An upper bound of the WCET C_{rec_nm} can be computed at compile time for every possible mode change using WCET analysis tools and by performing the low-level timing analysis on the hardware/software infrastructure we have developed to support time-predictable reconfiguration

Task scheduling policies and schedulability analysis are beyond the scope of this work. However, modelling the reconfiguration process as a task belonging to a reconfiguration mode enables an application programmer to apply scheduling

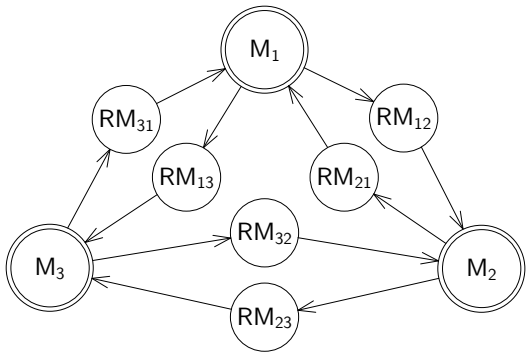

Figure 4.2: Mode transition graph of the multi-mode application presented in Figure 4.1(a) extended with reconfiguration modes RM_{ij}, $i, j \in \mathbb{N}$ to model the reconfiguration process.

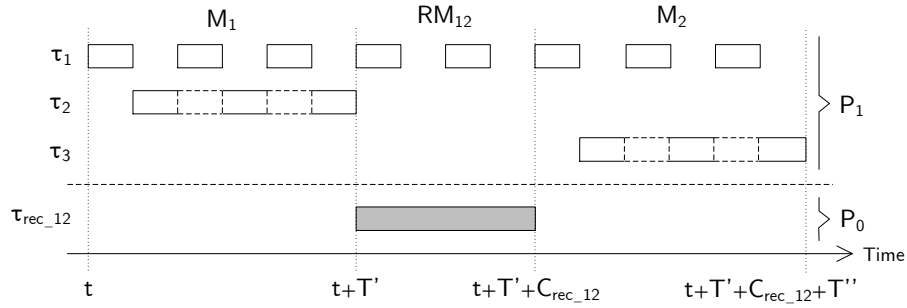

Figure 4.3: Time diagram example of the reconfiguration process using reconfiguration modes. The tasks τ_1, τ_2, and τ_3 execute on the processor P_1. The reconfiguration is performed by the task τ_{rec_12} executed on processor P_0. The figure shows the transition between modes M_1 and M_2 through the reconfiguration mode RM_{12}.

policies and to perform schedulability analysis at a system-level in an independent manner for each mode of operation.

4.5 Expected Outcomes and Evaluation Metrics

In the introduction, we stated the general hypotheses underlying this work. In the following, we detail these hypotheses into a set of expected outcomes,

deriving from the use of reconfiguration according to the proposed approach. This also defines a set of metrics used for the evaluation presented in Chapter 7, where the trade-offs between outcomes are assessed and discussed.

With regards to the reconfiguration of computation resources, and taking into account our approach of using DPR to implement hardware accelerators only when they are needed, we expect the following outcomes:

- **Reduction of the hardware size:** Implementing different functionalities that are needed only for a limited period of time in the dynamic region of the FPGA reduces the overall hardware size when compared to a fully static solution. This enables a more extensive use of accelerators and possible reduction of costs.
- **Simplification of the WCET analysis:** Moving the functionality performed by a software task into hardware may lead to a simplification of the WCET analysis. In general, the timing-analysis of hardware used to implement software-equivalent tasks is often easier to perform than analysis of a pure software solution. For example, timing-analysis of the instruction cache may not be needed when using accelerators.
- **Reduction of the WCET pessimism:** An interesting consequence of the simplification of the WCET analysis when using accelerators to perform selected tasks, is the possibility to reduce WCET analysis pessimism. A properly designed accelerator may have a very limited and predictable variance on the task execution time.
- **Speed-up from HW accelerator use:** In general, executing a computationally intensive task in hardware delivers a speed-up in the task execution time, leading to an increase of the overall system performance. This can apply for both the average-case execution time for general-purpose systems and the WCET for real-time systems.
- **Use of specialised accelerators:** DPR allows for the use of more efficient accelerators that are specialised in the execution of small specific tasks instead of generic accelerators that cover a more broad set of tasks. For example, a generic accelerator is a matrix multiplier that can take in input matrices of any size up to 32×32. Specialized accelerators are designed to perform the same operation, but only with a predefined matrix size (e.g., 4×4, 16×16, 32×32).
- **Increase of the design complexity:** The hardware architecture of a system that includes the DPR feature is more complex. For example, the use of a dedicated configuration controller is required, and some interface logic may be needed to isolate the reconfigurable region from the rest of

the system during reconfiguration. We mitigate this negative outcome by using our lightweight configuration controller RT-ICAP, characterised by a minimal hardware cost overhead and easy usability.

- **Increase of the memory requirements:** The partial bit-streams associated to each configuration must be stored in memory. Therefore, we expect an increase in the memory resources utilisation when using DPR. This memory increase goes against the hardware size reduction obtained by sharing the reconfigurable region between multiple accelerators. In our approach, we mitigate this negative outcome by applying compression techniques to the stored partial bit-streams.

With regards to the reconfiguration of communication resources, and taking into account the use of a statically scheduled TDM-based NoC as a communication fabric, such as the Argo NoC, we expect the following outcomes:

- **Increase of the flexibility:** The possibility of setting up and tearing down virtual channels at run-time increases the flexibility of the NoC. This is particularly relevant when taking into account that statically scheduled TDM-based NoCs, such as Argo, are usually only configurable at boot-time.
- **Increase of the bandwidth and reduction of the latency of VCs:** In a TDM-based NoC, the length of the TDM schedule grows with the number of VCs. Having the possibility of setting up and tearing down VCs or modifying their bandwidth depending on the current mode of operation allows the use of the minimum possible schedule period that satisfies the current mode requirements. Reducing the TDM period translates into higher bandwidth and lower latency with respect to a schedule that satisfies the requirements of all the modes.
- **Increase of the hardware cost:** The hardware architecture of the NI supporting reconfiguration is indeed more complicated and more costly in terms of hardware resources with respect to the NI without the reconfiguration feature. Moreover, since multiple schedules need to be stored, the memory resource requirements are also expected to increase.

Chapter 5

Reconfiguration of Computation Resources

This chapter presents the hardware/software infrastructure we developed to support reconfiguration of computation resources in real-time systems using the DPR feature of FPGAs. At first, we give an overview of the multi-core platform supporting DPR, and we present the architecture and functionality of our time predictable DPR controller, called RT-ICAP. This is followed by the description of the technique used to compress the bit-streams and of the software tool associated with the controller. Finally, we perform the reconfiguration time analysis, and we present a single-core application example, which is also used for part of the evaluation. The architectures and the techniques presented in this chapter were in part published in the papers [J1], [C1], [C4], and [C6] in our list of publications.

5.1 A Multi-Core Platform Supporting DPR

In this work, we envision to provide a multi-core platform with DPR support. Since we target platforms that may support a multitude of applications, one of the main challenges is defining how the reconfigurable regions are distributed in the platform and how they are accessed by the cores. In our approach, we

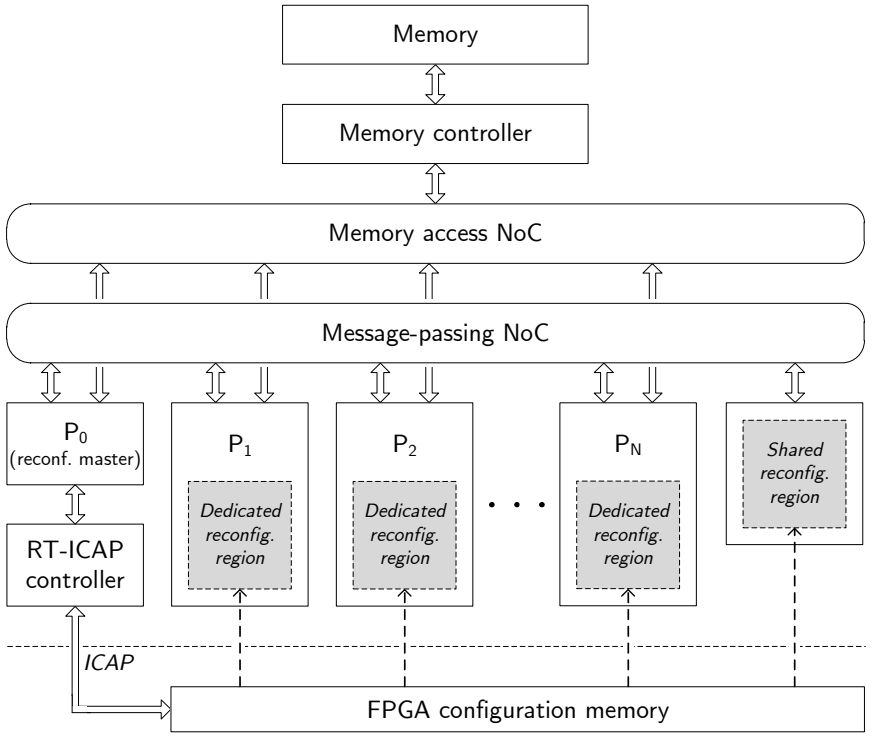

Figure 5.1: Block diagram of an example of a multi-core platform supporting DPR according to our approach. The reconfiguration master P_0 can reconfigure the hardware implemented in the dedicated reconfigurable regions of the processors P_1, ..., P_N and in the shared region by modifying the content of the FPGA reconfiguration memory.

consider two types of reconfigurable regions: dedicated to a single processor and shared by multiple processors. Figure 5.1 shows the block diagram of an example of a multi-core platform supporting DPR based on this approach. The platform consists of N processors (P_0, P_1, ..., P_N) connected by a message-passing network-on-chip for fast communication between cores and by a memory access NoC. This architecture matches the one of the T-CREST platform.

In Figure 5.1, the processors P_1, ..., P_N are equipped with dedicated reconfigurable regions. This type of region can be used to accelerate the execution of sections of software tasks running on that specific processor. In other words, the visibility of the functionalities provided by the hardware implemented in

these regions is limited to a single processor. In the general case, only a subset of processors may be equipped with a dedicated reconfigurable region.

The figure also shows one shared reconfigurable region connected to the message-passing NoC. In the general case, the shared regions can be interfaced to the multi-core platform with other communication fabrics. The shared regions are meant to implement hardware accelerators that offer functionalities that are not related to a specific processor. For example, it can be used to implement entire computational tasks in hardware, specific I/O communication protocols, or accelerators that can be shared between multiple processors. With regards to this, we developed a specific interface and integration technique for the T-CREST platform, as presented in paper [C7] in our list of publications. This interface enables the integration of stateless hardware accelerators using the Argo NoC and exploits its TDM properties to allow the accelerator to be shared between a set of processors in an interleaved fashion, without any form of reservation.

The reconfiguration process must be handled by a controller interfacing the FPGA configuration memory with the logic implemented on the FPGA itself. Even if multiple independent reconfigurable regions may coexist, one of the limitations of the current FPGA technology is that it allows the reconfiguration of only one region at a time. Therefore, our approach is to have a single processor of the multi-core platform, called reconfiguration master, interfacing the reconfiguration controller and managing the reconfiguration during a mode change. When the master processor starts a reconfiguration, the controller loads a partial bit-stream, pre-stored in a dedicated memory, into the FPGA configuration memory in a predictable time interval. In Figure 5.1, processor P_0 has the role of reconfiguration master, and it is connected to the ICAP interface through an ICAP controller. Therefore, P_0 is the only processor that can modify the content of the FPGA configuration memory. By writing a partial bit-stream into this memory, the hardware implemented in the reconfigurable regions of the platform is dynamically modified. The dashed arrows in the figure indicate the dependency.

5.2 RT-ICAP Controller Architecture

Most of the available reconfiguration controllers described in Section 3.1 offer a range of functionalities that are not strictly required by our approach to support reconfiguration of computation resources, where DPR is used to switch hardware accelerators during an operational mode change. Examples of these functions are the read-back, which is typically used for FPGA scrubbing for error mitigation [109], or the LUT-based reconfiguration for fine-grain DPR.

These additional functionalities increase the complexity of the reconfiguration controller and of the low-level reconfiguration timing-analysis. Moreover, for some of the controllers, the detailed hardware and software architecture is not public, making it impossible to perform a precise and reliable timing-analysis.

Our RT-ICAP controller is a lightweight open-source hardware component specifically designed to support DPR for mode changes in a time-predictable manner. The controller is developed and prototyped targeting the T-CREST platform; however, it can be easily used by other platforms where a small hardware footprint and time-predictable reconfiguration are strong requirements. The controller supports DPR in the Virtex-5, -6, and the 7-series FPGAs from Xilinx using the ICAP configuration interface described in Section 2.2.2. The RT-ICAP controller is designed to assist processor-initiated DPR of computation resources, such as hardware accelerators, in the same way as a DMA controller assists in moving data. The controller uses run-length encoding (RLE) compression to minimise the size of bit-streams, and it does not support read-back. This lightweight and simple design makes DPR and its timing analysis straightforward and easy. Moreover, it results in a small hardware implementation. In the following, we provide a description of the controller architecture and functionality.

Figure 5.2 shows a block diagram of the RT-ICAP controller. The controller is connected to the reconfiguration master processor through an OCP interface [45, 42], to a local SPM, and to the ICAP of the FPGA. In our architecture, the SPM is used to store the partial bit-streams and also acts as a local general purpose memory for the processor. In comparison to a data cache, the access time for an SPM is guaranteed to be a single cycle. The fact that the SPM is not strictly dedicated only to store bit-streams is one more aspect that distinguishes our RT-ICAP controller from the ones discussed in Section 3.1.

The processor controls and manages the functionality of the controller through the OCP interface, where it can access a set of 32-bit registers (mapped in the address space of the processor). A control FSM manages the ICAP interface, the registers, and all the functionalities of the controller. Table 5.1 describes the purpose of all the registers. The *status* register can be read by the processor to monitor the status of the controller and the reconfiguration. The *control* register can be written by the processor to manage the controller operations. Table 5.1 presents the list of the register and their description. The possible *status* register values *control* register commands reported in the table in a textual form correspond to values associated to specific fields of the register as defined in the C library associated with the controller. The role of the registers in the controller operations is explained in the following.

The RT-ICAP controller can operate in two different modes depending on where the partial bit-stream used in the reconfiguration is stored. If the bit-stream is

Table 5.1: List of registers of the RT-ICAP controller and their description.

Register	Description
status	This register holds the status of the controller. Read-only. Possible status values are: - READY: The controller is ready, waiting for the next command (post-reset status). - READY_AND_DONE: The configuration was successfully completed. The controller is ready, waiting for the next command. - READY_AND_FAIL: The configuration has failed or was aborted. The controller is ready, waiting for the next command. - WAIT_BUSY_ICAP: The configuration process is waiting for the ICAP interface to be free in order to start. - WRITE_IN_PROGRESS: The reconfiguration process is in progress. The controller is writing the bit-file into the ICAP interface. - WAIT_END: The reconfiguration process is in progress and the entire bit-stream has been written. The controller is waiting for the ICAP interface to confirm successful or failed reconfiguration. - ABORT_IN_PROGRESS: An abort command has been received. The controller is waiting for the ICAP interface to confirm the abort.
control	This register receives the command to execute. Write-only. Possible commands: - START_SPM_STREAM: Start the reconfiguration process in *SPM-stream* mode. - START_CPU_STREAM: Start the reconfiguration process in *CPU-stream* mode. - ABORT: Abort the current reconfiguration process. - SW_RESET: Software reset of the controller.
bs_length	This register contains the length, in bytes, of the bit-stream to be used in the reconfiguration. Write-only.
bs_addr	This register contains the address, in bytes, that points at the beginning of the bit-streams stored in the SPM to be used for the reconfiguration. Write-only.
stream_in	In *CPU-mode*, this register received the bit-stream data to be written through the ICAP interface in *CPU-stream* mode. Write-only.

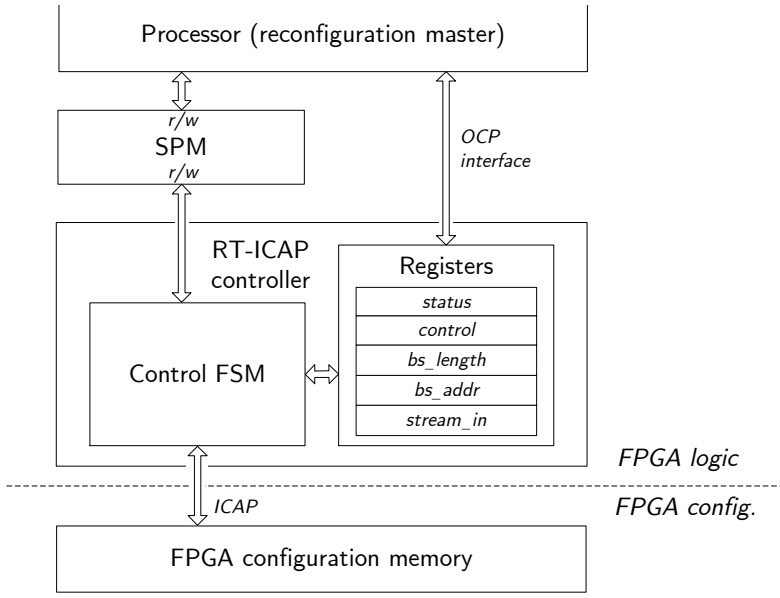

Figure 5.2: A block diagram of the RT-ICAP reconfiguration controller and its interfaces.

stored in the local SPM, the controller autonomously fetches the bit-streams from the SPM and writes it to the reconfiguration memory through the ICAP. We refer to this mode as *SPM-stream*. If the bit-stream is not stored in the local SPM, it can be copied from an external memory directly into the RT-ICAP controller by the reconfiguration master processor. We refer to this mode as *CPU-stream*.

When the controller operates in *SPM-stream* mode, the bit-stream to be used must be already stored in the SPM when the reconfiguration is triggered. To start a reconfiguration, the reconfiguration master configures the *bs_addr* register with the SPM address that points at the beginning of the bit-stream and the *bs_length* register with the length (in bytes) of the bit-stream. By writing the *START_SPM_STREAM* command into the *control* register, the processor starts the transfer from the SPM to the ICAP. The *status* register reports the controller status, including the end of the reconfiguration process. In *SPM-stream* mode, the controller can achieve the maximum transfer speed of the ICAP (400 MB/s); however, it is not always possible to fit the partial bit-stream associated to all the modes of operation in the SPM. If a reconfiguration is scheduled to happen at a particular point in time, the processor can pre-fetch the bit-stream from an external memory into the SPM. This will minimise the reconfiguration

time during a mode change. In summary, the *SPM-stream* mode is particularly suitable for small bit-streams associated with reconfigurable regions that require a fast reconfiguration.

When the controller operates in *CPU-stream* mode, the bit-stream is received from the reconfiguration master processor as a sequence of writes. The processor is tasked with the copying of the bit-stream from an external memory to the RT-ICAP controller, which forwards it to the ICAP and manages its interface. In order to start a reconfiguration, the processor must first configure the *bs_length* register and write the *START_CPU_STREAM* command into the *control* register. Then, the bit-stream is written into the *stream_in* register of the controller by the reconfiguration master one data element at a time. This operating mode is slower *SPM-stream* mode, and the bottleneck is typically determined by the bandwidth to the memory from where the processor reads the bit-stream. The *CPU-stream* mode should be used when it is not possible to store the partial bit-stream associated to all the modes of operation in the SPM and is not possible to perform bit-stream pre-fetching, such as in case of an unexpected mode change triggered by an aperiodic event.

5.3 Bit-Stream Compression

The size of the SPM is a limiting factor when the controller is operating in *SPM-stream* mode since it limits the number of bit-streams that can be locally stored and, thus, being able to be loaded at the maximum speed offered by the ICAP. To reduce this limitation, we use lossless compression techniques to reduce the size of the partial bit-streams and, therefore, the memory needed for bit-stream storage. The application of the most common compression techniques (e.g. RLE, Huffman, Arithmetic, Lempel-Ziv, etc.) on Xilinx bit-streams is a well-known topic, and it is explored in the work presented in [110], [111], and [112]. Advanced techniques, such as Lempel-Ziv, are very efficient in terms of compression ratio. However, this comes at the expenses of complex de-compression algorithms that in most cases are performed in software, potentially compromising the time-predictability of the reconfiguration or increasing the complexity of the timing-analysis.

For our implementation, we have selected an RLE compression technique for the bit-streams. The size of the data element used by the compression technique (referred as 'character' in the following) is the same as the input bus used by ICAP interface (8, 16, or 32 bits). The main idea used in the RLE compression is to store sequences of repeated characters (called 'data-run') as a single character combined with a character occurrence count. The compressed data-run is

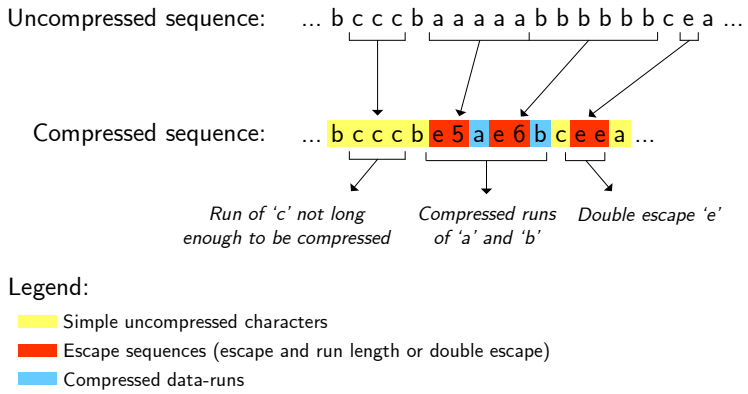

Figure 5.3: An example of the RLE compression technique applied to an arbitrary sequence of characters. In this example, the escape character is 'e'.

identified by an escape character. When a data-run gets compressed, it appears as an escape value to signal the beginning of a compressed sequence, followed by the count and the data itself. Therefore, only data-runs longer than three elements are compressed. Single escape values in the uncompressed original bit-stream are represented in the compressed ones as replicated escape character in order to distinguish it from a compressed data-run. Figure 5.3 shows an example of RLE compression on a sequence of characters. In this example, the escape character is 'e'. In the general case, the escape character should be selected as the character with less single, double, or triple occurrences in the bit-stream.

The bit-streams are compressed by the software tool associated with the RT-ICAP controller (presented in the following section), and it is decompressed in hardware by the controller. Implementing the RLE decompression in hardware instead of a software task executed by the reconfiguration master contributes to the reduction of the complexity of the WCET analysis. In addition, the hardware overhead needed for decompression is minimal due to the simplicity of the RLE compression.

The tools provided by Xilinx offer a native bit-stream compression functionality based on writing identical configuration frames once, instead of writing each frame individually. A frame is the smallest addressable segment of the FPGA configuration memory space, and its size is in the order of KB, depending on the FPGA model. If more than one frame has identical data, the frame is loaded into the configuration logic and written to multiple address locations of the FPGA

Listing 5.1: XML file structure used for configuring the *convbitstream* tool.

```
<?xml version="1.0" encoding="utf-8"?>

<bitfiles>
    <description>Bit-streams to be processed.</description>

    <parameters datasize=["8","16","32"] compression=["true","false
    "] bitswapped=["true","false"]/>

    <bitfile id="bit-stream_id" file=["path/bit-stream_name.bin"/>
    ...
    <bitfile id="bit-stream_id" file=["path/bit-stream_name.bin"/>
</bitfiles>
```

configuration memory in parallel. In this case, the decompression is executed by the FPGA logic that manages the reconfiguration (not accessible by the user).

The use of the Xilinx native compression not only reduces the size of the bit-stream but also increases the reconfiguration speed since multiple frames can be written in parallel. Our RLE technique can be used stand-alone or in addition to the frame-based compression offered by the Xilinx tools. Since the granularity of the RLE compression (character) is considerably smaller than the one of the Xilinx native compression (frame), the combination of both techniques leads to high compression ratios and introduces a trade-off between compression ratio and reconfiguration time. This trade-off is discussed and evaluated in Subsection 7.1.2.

5.4 Tool Support

The RT-ICAP controller is supported by a software tool, named *convbitstream*. The *convbitstream* tool compresses the partial bit-streams produced by the Xilinx tools and converts them into the format used by our controller. Most importantly, it performs the low-level timing-analysis to compute, for each compressed bit-stream, the time needed by the controller to perform the reconfiguration. The reconfiguration time is computed for both the *SPM-stream* and *CPU-stream* mode, and it can be used during the WCET analysis of a software application that uses the DPR feature.

The *convbitstream* tool is implemented in C and receives as input an extensible markup language (XML) file specifying the parameters that control the

compression and format conversion, as well as a path to a directory containing the bit-streams files to be processed. Listing 5.1 shows the structure of the XML file used for configuring the *convbitstream* tool. The parameters *datasize*, *compression*, and *bitswapped* in the XML file indicate the size of the ICAP interface and whether the compression and bit-swapping (i.e. reordering of data bits according to Xilinx specifications [36, 35]) should be enabled. In Listing 5.1, these parameters are followed by the list of the bit-streams to be processed. For each bit-stream provided as input, the tool produces the converted bit-stream in two formats: as a binary file to be used if the bit-streams are stored in an external memory, and as an array declaration in a C file to be used to embed the bit-streams into a C program. The name of the C array is the *bit-stream_id* parameter passed in the XML file, as shown in Listing 5.1. The length of each C array is also produced in the output file. This information is needed by the C library supporting the RT-ICAP controller to properly initialise the reconfiguration process.

5.5 Reconfiguration Time Analysis

The most relevant feature of the RT-ICAP controller is the ability to perform time-predictable DPR. For each bit-stream, the reconfiguration time T_{rec_dpr} is computed by the *convbitstream* tool. The reconfiguration time T_{rec_dpr} is from the moment in time when the master processor starts the reconfiguration until the partial bit-stream is entirely written into the configuration memory of the FPGA.

The reconfiguration time, in clock cycles, is computed using Equation 5.1. It consists of the sum of three contributions: the number of cycles needed to invoke (initialise and finalise) a reconfiguration, the number of cycles needed to transfer the compressed bit-streams into the RT-ICAP controller, and the number of additional cycles required to expand compressed data-runs and write these into the reconfigurable area of the FPGA.

$$T_{rec\_dpr} = T_{clk}\{n_{oh} + n_{if}(n_s + 2n_e + n_r) + \sum_{i=1}^{n_r}(R_{i\_len} - 1)\} \tag{5.1}$$

The following list explains in details the contributions to Equation 5.1:

- n_{oh} is the overhead required by the RT-ICAP controller for starting and finishing a reconfiguration.

- n_{if} is the number of cycles to write a character into the RT-ICAP controller; for *SPM-stream* mode this is 1 cycle, and for *CPU-stream* mode it is 2 clock cycles (the latency of an OCP-transaction [42]). In the processor and the RT-ICAP controller are in different clock domains, n_{if} should include the latency of the clock domain crossing.

- The term $(n_s + 2n_e + n_r)$ is the length of the compressed bit-stream; n_s is the number of simple (uncompressed) symbols, n_e is the number of escape sequences (each comprising a pair of symbols), and n_r is the number of the repeated symbols in compressed data-runs. In the example shown in Figure 5.3, these symbols are respectively marked in yellow, red, and blue.

- The third contribution is the number of additional clock cycles required for writing repeating characters into the reconfigurable area of the FPGA. Here, R_{i_len} is the number of times a compressed character repeats in the i-th data-run.

The parameters n_{oh} and n_{if} characterize the RT-ICAP controller and n_s, n_e, n_r, and R_{i_len} characterize the bit-streams.

In *SPM-stream* mode, a symbol of uncompressed data can be written into the RT-ICAP controller and further into the reconfigurable area of the FPGA in every clock cycle. When the RT-ICAP controller expands a compressed run of symbols, the interface towards the SPM stalls while the controller writes the expanded sequence of symbols into the ICAP. When the controller operates in *CPU-stream* mode, the bit-stream is loaded from an external memory. The time needed for this is independent of the RT-ICAP controller and needs to be analysed separately taking into account the behaviour of the communication fabric and the memory controller providing access to the memory.

The equation presented above is derived by applying low-level analysis on the architecture of the RT-ICAP controller. An application programmer needs to add the software overhead of setting up the controller to trigger a reconfiguration. When pre-fetching is used or when the controller operates in *CPU-stream* mode, the transfer time of the bit-stream from an external memory should also be included. In T-CREST, the WCET analysis tools platin [44] and aiT [37] can be used to perform this analysis. The software overhead along with the reconfiguration time, which is calculated by the *convbitstream* tool, correspond to the WCET of the task τ_{rec_nm} that performs the reconfiguration during a reconfiguration mode, as explained in the task-level model presented in Section 4.4.

5.6 Single-Core Application Example

In the following, we present an application example of the presented RT-ICAP controller in combination with a single Patmos processor (introduced in Subsection 2.4.2). The example aims to show a possible hardware architecture where a reconfigurable region is used to implement hardware accelerators dedicated to the processor. Figure 5.4 shows a block diagram of the hardware architecture, which consists of the Patmos processor and a set of I/O devices connected through an OCP bus [45]. The I/O devices include the RT-ICAP controller, a dedicated accelerator controller ('HwA-controller' in the figure), a hardware accelerator ('HwA' in the figure), and several SPMs.

The reconfiguration is managed by our RT-ICAP controller, which can modify the content of the FPGA configuration memory through the ICAP interface. By writing a partial bit-stream stored in the bit-stream SPM into the FPGA configuration memory, the content of the reconfigurable region is dynamically modified. The dashed arrow in Figure 5.4 shows this dependency.

An SPM ('HwA-SPM' in Figure 5.4) is used for data exchange between the hardware accelerator and the Patmos processor. The HwA-SPM is divided into a certain number of banks mapped to the address space of the processor as a single and continuous address space. However, the banks can be accessed in parallel by the accelerator in order to increase the memory bandwidth towards it. Note that the HwA-SPM is not the local SPM of Patmos, which is used by the processor to store easily accessible data and instructions. Both the accelerator and the HwA-SPM reside in the reconfigurable region since the number of used memory banks may vary depending on the currently implemented accelerator. Therefore, the HwA-SPM needs to be reconfigured together with the hardware accelerator. The dedicated HwA-controller manages the accelerator and reports its current status to the processor.

Assuming that the bit-streams are available in the bit-stream SPM, the RT-ICAP controller can operate in *SPM-stream* mode. If the bit-streams are not available in the bit-stream SPM, the controller can operate in *CPU-stream* mode and the processor directly loads the bit-streams into the ICAP without using the bit-stream SPM. Alternatively, the processor can perform pre-fetching to transfer them into the bit-stream SPM before performing reconfiguration.

The operational flow to use the accelerator after a reconfiguration has been triggered by a mode change is as follows:

1. Patmos moves the data to be processed into the HwA-SPM.

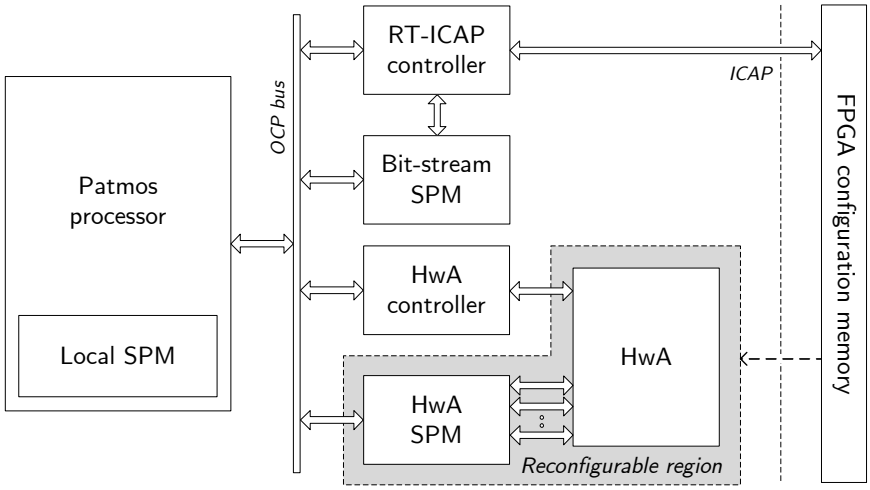

Figure 5.4: A block diagram of an example single-core architecture that includes a reconfigurable region used to implement hardware accelerators. The same architecture is used in the evaluation of the reconfiguration capabilities of the RT-ICAP controller presented in Subsection 7.1.3

2. Patmos activates the accelerator by interacting with the HwA controller. When the accelerator is running, the processor is free to execute other operations.

3. When the accelerator has finished, Patmos can read the processed data from the HwA-SPM. Note that the HwA-SPM is not only dedicated to the accelerator, but it can also store temporary data until the next reconfiguration.

This architecture is used for the evaluation of the reconfiguration capabilities of the RT-ICAP controller presented in Subsection 7.1.3. For the evaluation, we use benchmarks from the TACLe suite [113] and hardware accelerators generated using the Xilinx Vivado HLS [31] tool.

Chapter 6

Reconfiguration of Communication Resources

This chapter describes the new version of the message-passing NoC Argo 2 that we developed to support the reconfiguration of communication resources. At first, we give an overview on how we approach the NoC reconfiguration and on the differences between the new Argo 2 and the original Argo NoCs. This is followed by a description of the Argo 2 NI and a detailed presentation on how reconfiguration is supported. Finally, we provide the reconfiguration time analysis. The architectures and the techniques presented in this chapter were in part published in the papers [J3] and [C5] in our list of publications.

6.1 Overview

The original Argo NoC presented in Section 2.5 and in [23, 50] supports message-passing between the processing nodes of a multi-core platform. Argo offers a set of VCs characterised by guaranteed bandwidth and latency. TDM-based static routing is used to share the resources of the packet-switched structure between the VCs. The NIs inject packets into the packet-switched structure according to the TDM schedule. The packets are routed to the destination node according to

the route included in the packet header. Therefore, it is the TDM schedule that guarantees the bandwidth and latency of the VCs provided by the NoC.

To support reconfiguration, we have implemented a new version of the Argo NI which allows switching between a set of static TDM schedules generated at compile-time. Switching TDM schedule at run-time corresponds to changing or modifying the characteristics of the set of VCs offered by the NoC. A unique feature of the new NI is the possibility to switch instantaneously between sets of VCs without affecting the VCs that persist across the reconfiguration. Moreover, to comply with the time-predictability specification, the NI ensures that the switch between two schedules is executed in a bounded time interval. The NoC that uses the new version of the NI supporting reconfiguration is called Argo 2. The Argo 2 NI is a new design based on the original Argo NI presented in Subsection 2.5.3.

In addition to supporting reconfiguration, the Argo 2 NI also implements additional functionalities with respect to the original Argo NI. This includes the use of variable-length packets to reduce the header overhead, a more compact representation of the TDM schedule, the support of interrupt packets, as well as the possibility to configure and initialise the NI of a remote node using the NoC itself.

6.2 Argo 2 NI Architecture

Before discussing how reconfiguration is supported, we present the architecture and the basic operation of the Argo 2 NI. At first, we describe the packet format and the schedule representation. Then, we present the NI architecture, we explain how packets are sent and received, and we describe the remote initialisation feature.

6.2.1 Packet Format and Schedule Representation

The Argo 2 NI supports three types of packets: data packets, interrupt packets, and configuration packets. Figure 6.1 shows the general format of the packet. The packet consists of a 35-bit header flit followed by n 35-bit payload flits. For interrupts and configuration packets $n = 1$ and for data packets $n \in [1, 15]$. The variable-length for data packets allows to reduce the overhead of the header for those VCs that require high bandwidth. The header contains three fields: the packet type, the destination address specifying the location where the data must

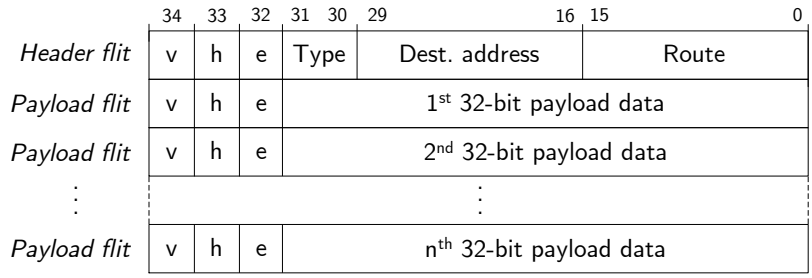

Figure 6.1: Packet structure used in the Argo 2 NoC consisting of an header flit and n 32-bit payload flits. For data packets $n \in [1, 15]$ and for interrupts and configuration packets $n = 1$.

be written in the SPM of the receiving node, and the route that the packet should follow through the NoC. The position of the route field and the three control bits specifying a valid flit (v), the header flit (h), and the last flit (e) remain unchanged with respect to the original Argo since the packet-switched structure of the NoC is the same.

Data packets are used to transfer data from the SPM of a processing node to the SPM of a remote node, as in the original Argo. In Argo 2, the sender node can notify the receiver when a data transfer is complete. This is achieved by marking the last packet of a data transfer to generate an interrupt at the destination core. We refer to this as local interrupt, since it is generated and processed in the in the receiving node.

Interrupt packets are used to generate an interrupt in the receiving processing node and they carry one 32-bit word payload to be used as interrupt identifier. This feature may be needed to support multi-core operating systems and to manage the tasks executed in remote nodes (e.g., terminating a task during a mode change). We refer to this as remote interrupt since it is directly triggered by a remote node.

The configuration packets are used to write configuration data into the tables of the NI of a remote node. This packet type plays a fundamental role in how the reconfiguration is implemented and further description is provided later. The configuration packets are used to trigger a reconfiguration, to distribute a new schedule, or to initialise a remote NI.

In the original Argo NoC, the size of a TMD time slot is fixed (3 clock cycles) and correspond to the packet length. The slot counter is incremented for each time slot and it indexes directly into the schedule table. The entries in the schedule table that correspond to a time slot in which the NI is not allowed to

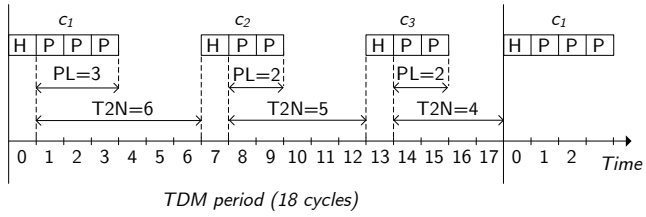

Figure 6.2: An example of the TDM schedule representation used in Argo 2 for the three VCs c_1, c_2, and c_3. 'PL' stands for packet length. 'T2N' stands for time-to-next.

send are marked as not valid. This straightforward approach leads to a relatively simple hardware implementation, but it cannot support variable packet length and it may waste entries of the schedule table represent the unused slot entries. To overcome this, in the Argo 2 NI, we represent the schedule in a different format, where the entries of the schedule table correspond to packets instead of time slots. Each entry uses two fields specifying the number of payload words of the packet (packet length or 'PL') and the clock cycles until the header of the next packet (time-to-next or 'T2N'). Figure 6.2 shows an example of the TDM schedule representation used in Argo 2 for the three VCs c_1, c_2, and c_3. The packet associated to c_1 has a 3 words payload, the ones associated to c_2 and c_3 have 2 words payload. For each packet, the fields PL and T2N indicate the packet length and the distance between packet headers. If the schedule contains entries marked as not valid, the representation used in Argo 2 leads to a reduction of the number of entries in the schedule table. For the example in the figure, the TDM period is 18 clock cycles and the schedule requires 3 entries in the schedule table, one for each packet. For comparison, the original Argo would require 6 entries in the schedule table to represent a schedule of 18 clock cycles (i.e. 6 TDM slots of 3 clock cycles each).

6.2.2 Transmit Module

The transmit module of the Argo 2 NI is shown in the left side of Figure 6.3 and it consists of the TDM controller, the schedule table, the DMA table, the packet manager, and the reconfiguration controller. The reconfiguration controller manages the reconfiguration and it is described in detail in Section 6.3. In the following, we explain the functionality of the transmitting module, with reference to Figure 6.3.

The Argo 2 NI contains several tables where pointers in one table index into the next. Each NI is equipped with a TDM controller, which contains a TDM

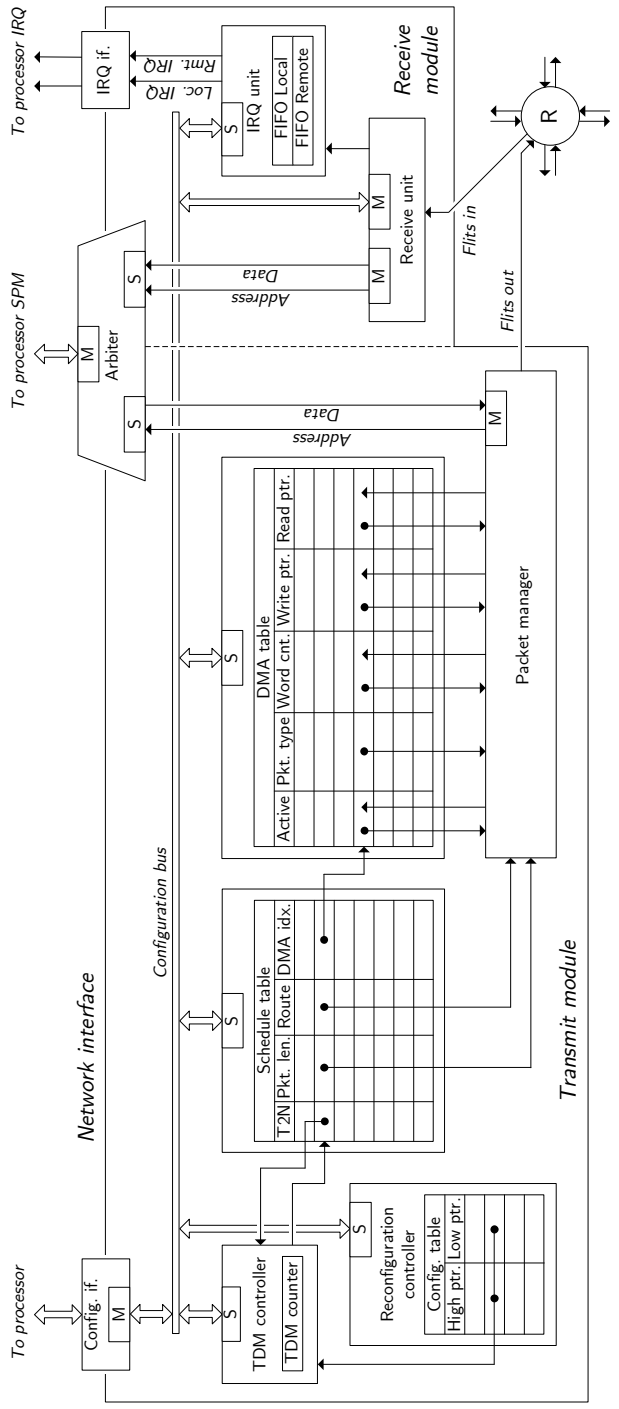

Figure 6.3: A block diagram of the Argo 2 NI and its connection with a processing node at the top and with the switched-structure at the bottom. The connection to a processing node matches the one shown in Figure 2.6. The NI is divided into two modules: the transmit module is on the left and manages the transmission of packets and the receive module is on the right and handles the reception of incoming packets. Source: paper [J3] in our list of publications.

counter. The TDM counter is incremented every clock cycle to keep track of time progression and it wraps around at the end of a TDM period. The TDM controller uses the TDM counter in combination with the time-to-next field in the schedule table to generate a pointer for the schedule table. The schedule table contains the schedule, one entry per packet. Therefore, the pointer is incremented only when a new packet should be sent. The schedule table can hold entries belonging to multiple schedules used for reconfiguration. The TDM controller uses the information provided by the reconfiguration controller (high and low pointers) to generate a pointer for the schedule table. This pointer is in the range of entries belonging to the currently active schedule.

An entry in the schedule table contains the route of the packet that the NI has to send, the packet length, and the index into the DMA table specifying which DMA entry is enabled to send. Each entry in the DMA table represents a VC. The packet manager assembles and sends out packets using information from the schedule table and the DMA table. The header of an outgoing packet is assembled using the packet type and the write address fields from the enabled entry in the DMA table, and the route field from the active entry in the schedule table. The payload words following the header are read from the SPM. The packet length field from the schedule table determines the maximum number of payload flits that can follow the header flit. Analogously to the original Argo NI, the data to be sent is read word-by-word from the SPM using the read address field from the enabled entry in the DMA table. During transmission, the packet manager updates the read address, the write address, and the word count fields in the DMA table. The active field in the DMA table is set by the processor that initialises the transfer. The field indicates if the DMA has data pending for sending and it is cleared by packet manager when the last packet of a DMA transfer is sent. The processor can poll this field to know when a transfer is completed.

In contrast with the original Argo, the Argo 2 NI places the route field into the schedule table instead of the DMA table. This allows routing multiple packets belonging to the same VC along different paths. The scheduler [55] ensures in-order arrival of the packets belonging to the same multi-path VC. This is achieved by generating schedules where all the possible paths that belong to a VC have the same number of hops through the NoC.

6.2.3 Receive Module

The receive module of the Argo 2 NI is shown on the right side of Figure 6.3 and it consists of the receive unit and the interrupt (IRQ) unit. Depending on the packet type, the incoming packets are processed differently by the receive unit.

Data packets carry the target address as part of the header and the data payload is written directly into the SPM as soon as it is received. The receive unit increments the target address for each write into the SPM. If the data packet is the last of a DMA transfer and it is marked to generate an interrupt, the local interrupt signal is asserted. The target address of the last word, which serves as interrupt identifier, is written into the IRQ FIFO for local interrupts in the IRQ unit. The interrupt signal is de-asserted when the lRQ FIFO is emptied by the processor.

For configuration packets, the data payload is written into one of the NI tables in the transmit module through the internal configuration bus. The data stored in these tables are mapped into a private address space of the NI and the address of the configuration packet header points into this address space.

For interrupt packets, the data payload is written into the SPM and the target address is written into the IRQ FIFO for remote interrupts in the IRQ unit as identifier. The remote interrupt signal remains asserted until the IRQ FIFO is emptied by the processor.

The transmit and receive modules of the NI share one port to the SPM. To support concurrent 32-bit reads and writes, the SPM is equipped with a double width read/write port (64 bits). An arbiter manages the interface and offers the needed buffering. Similarly to the original Argo, there is no need for buffers or flow control in the NI, or for extra DMA controllers in the processor to copy the received data out of the NI, since the payload of incoming packets is written directly to its target address in the processor SPM. This leads to an efficient and small hardware implementation of the receive module.

6.2.4 Remote Initialization

In the original Argo, each NI is initialised at boot-time by the processor to which it is directly connected. This works well if all the platform nodes are processors since they are able to read the initialisation data (i.e. TDM schedule information) from shared memory and load them into the NI tables. However, if a processing node is a hardware accelerator, it might not be able to perform the initialisation. In our approach for reconfiguration of computation resources, we envision to implement accelerators in the shared reconfigurable region connected to the message-passing NoC, as explained in Section 5.1. Therefore, we have supplemented the Argo 2 NI with the capability of being remotely initialised using configuration packets.

The remote initialisation is performed using configuration packets to write into the TDM controller, the schedule table, and the reconfiguration controller of a remote NI since these tables are mapped into a private address space for each NI. After having initialised its own NI, the master processor can send configuration packets to the remote NI to load the TDM schedule information needed to operate. When all the tables in the remote NIs are set, the master processor can enable the rest of the NIs to operate and start to inject packets in the NoC.

6.3 Support for Reconfiguration

This section describes how we support reconfiguration in the Argo 2 NoC. At first, we present the ideas and observations underlying our reconfiguration approach. Then, we introduce the reconfiguration process, also discussing possible ways in which reconfiguration can be used.

6.3.1 Key Ideas and Observations

In the Argo 2 NoC, reconfiguration consists of switching TDM schedule at run-time. This corresponds to modifying the set or the characteristics of the VCs offered by the NoC to match the GS requirements of the current mode. In our approach to reconfiguration, we use the NoC itself for the transmission of reconfiguration commands and new schedules. This implies that the VCs dedicated for reconfiguration purposes must be set up alongside the VCs that are used for transmission of regular data.

Similarly to the reconfiguration of the computation resources, the reconfiguration of the NoC is managed by one processor of the multi-core platform, which has the role of reconfiguration master. This processor triggers the reconfiguration process by sending commands to all the NIs of the platform and, if needed, distributes the new schedules to all the NIs. For this reason, a VC dedicated to reconfiguration must be allocated from the master to each of the other cores in the platform. The impact of this addition on the TDM schedule period is around 10 % on average, as evaluated in paper [J3] in our list of publications. Another approach for allocating the VCs dedicated to reconfiguration could involve the use of channel trees [114]. This approach multiplexes multiple VCs in a single TDM slot by using an additional level of scheduling. The VCs dedicated to reconfiguration could be multiplexed, leading to a reduction of the impact on the TDM schedule. This would come at the cost of increasing the hardware complexity to support the additional scheduling.

Reconfiguring a NoC at run-time requires accessing and modifying the state of the NoC, and typically flushing the VCs that are torn down and initialising the ones that are set up between configurations. The Argo 2 NoC NI is characterised by some properties that simplify significantly the reconfiguration process. At first, the routers are characterised by the absence of buffers, flow control, or arbitration. This translates into the property of not preserving any state when switching between VCs. Therefore, the reconfiguration process does not involve the routers, but only the NIs. In addition, Argo 2 does not use VC buffers in the NIs and credit-based flow control among these buffers. Therefore, these do not need to be flushed and credits counters do not need to be initialised when new connections are set up. Finally, the scheduler maps VCs to time slots in the TDM schedule in a way where the network can be considered as conceptually empty at the end of each TDM period. This property is explained in detail later and it allows instantaneous switching from one TDM schedule to another in a way that is completely transparent to VCs that persist across the reconfiguration. These VCs can even be re-mapped to different time slots and different routes.

6.3.2 Reconfiguration Process

The reconfiguration controller handles the reconfiguration process in hardware and it is connected to the receive unit through the internal configuration bus, as shown in Figure 6.3. Therefore, it can receive write transaction triggering a reconfiguration from a remote node through incoming configuration packets. As previously mentioned, the schedule table may hold several different schedules, each spanning a range of entries. Each range is represented by a couple of pointers (i.e. high pointer and low pointer) stored in the configuration table of the reconfiguration controller. Only one range can be active at a time. In principle, a reconfiguration simply requires that the TDM counter is set to the start entry of the new schedule when the TDM counter itself reaches the end of the previous schedule.

All the NIs are equipped with a small counter (2 bits) incremented every TDM period and used to identify the TDM period in which all the NIs should synchronously switch to the new schedule. A master processor invokes a reconfiguration by sending a configuration packet to the reconfiguration controller of all the slave NIs, announcing that they must switch to the new schedule. This packet contains two parameters: the index of the configuration table entry in the reconfiguration controller that holds the high and low pointers for the new schedule and a number that identify the TDM period in which the switch to the new schedule must happen.

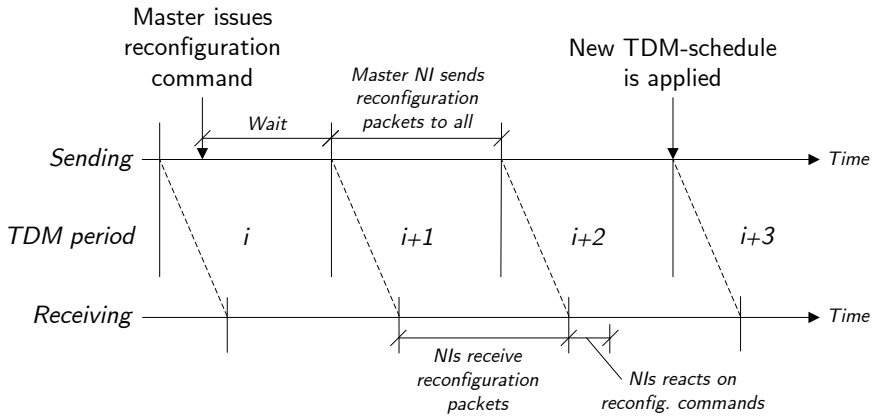

Figure 6.4: Time diagram showing the reconfiguration process of the Argo 2 NoC.

Figure 6.4 shows a time diagram of the reconfiguration process. In the figure, the reconfiguration master issues the reconfiguration command in TDM period i. Thus, the NI of the master transmits configuration packets announcing a reconfiguration to all the other NIs in period $i + 1$. The reconfiguration packets sent during a TDM period $i + 1$ arrive in a time window that is phase-shifted by 6 clock cycles. This shift corresponds to the pipeline depth in a shortest possible path between two NIs. To minimise the period of a TDM schedule and avoid wasting bandwidth, the scheduler allows an equivalent phase-shift at the end of the TDM period. This means that the flushing of the NoC at the end of a TDM period and the filling at the beginning of the next TDM period is overlapped and that any injected packet at the beginning of a new period cannot cause collisions in the NoC. Due to this shift, a reconfiguration issued by the master in TDM period i can only take effect in TDM period $i + 3$. In addition to the phase-shift of 6 clock cycles, the receiving NIs need 2 additional clock cycles to react to the reconfiguration command. This implies that the period of a TDM schedule must be longer than the $6 + 2$ clock cycles. If the schedule is shorter, it can be unrolled two or more times to reach the required minimum length.

The reconfiguration process implemented in the Argo 2 NoC can be used to support reconfiguration in three ways, as described in the following.

- If the schedule table has sufficient capacity to store all possible configurations, these can be loaded into the NIs at boot-time. In this case, a master only needs to send reconfiguration requests to the NIs. This is the preferred method since it has the lowest reconfiguration latency.

- If the schedule table can only store two configurations at a time, it possible to first transmit the new schedule and then send a reconfiguration request. This is possible since all the NI tables that contain configuration data can be remotely written using configuration packets. The transmission of a new schedule does not affect the functionality of the NoC and it is completely transparent to the receiving nodes.
- If the schedule table can only store a subset of all possible configurations, a hybrid of the above two methods is also possible. In this case, the mode graph can be divided into partitions and all the schedules required by the modes of a partition are loaded into the schedule table. In this way, the reconfiguration master can rapidly switch between the schedules of one partition, but switching between partitions requires the transmission of the new schedules.

6.4 Reconfiguration Time Analysis

Argo 2 supports time-predictable reconfiguration by ensuring that the reconfiguration process is executed in a bounded interval of time. The switch between two schedules happens instantaneously and it is transparent from an application point-of-view. However, there is a latency between the moment when the reconfiguration master issues a reconfiguration and the moment when the new schedule is applied. In the following, we present the low-level timing-analysis for both the case where the new schedule is already stored in the schedule table of the NIs and for the case where the new schedule needs to be transmitted to all NIs before reconfiguration.

For the first case, the worst-case reconfiguration time T_{rec_noc} can be calculated with Equation 6.1, where, P_{curr} is the TDM period length of the current schedule. From Figure 6.4, we can observe that the T_{rec_noc} depends only on the currently executing schedule and it is, in the worst-case, three times its period.

$$T_{rec\_noc} = 3 \cdot P_{curr} \tag{6.1}$$

If the new schedule is not stored in the schedule table, the worst-case transmission time T_{send_sched} needed by the reconfiguration master to transfer the schedule has to be added to the worst-case reconfiguration time T_{rec_noc}. The size of a TDM schedule is different in each NI due to the Argo 2 schedule representation. Therefore, T_{send_sched} is the maximum of the individual worst-case transmission times for each NI, due to the interleaved way for transmitting the schedule through

the NoC. The transmission time T_{send_sched} depends on the characteristics of the VCs reserved for the configuration packets in the current schedule and on the size of the new schedule, and it can be calculated with Equation 6.2.

$$T_{send\_sched} = \max_{i \in N} \left(L^i_{\text{curr}} + \left\lceil \frac{S^i_{\text{new}} - 1}{B^i_{\text{curr}}} \right\rceil \cdot L_{\text{curr}} + L^i_{\text{chan}} \right) \tag{6.2}$$

In Equation 6.2, i represents the i-th NI from the set of N nodes in the platform, L^i_{curr} is the worst-case waiting time for a time slot to NI i, S^i_{new} is the number of words of the new schedule to be sent to NI i, B^i_{curr} is the bandwidth of the current schedule towards NI i, L_{curr} is the TDM period of the current schedule, and L^i_{chan} is the NoC latency to NI i. Overall, Equation 6.2 is the sum of three contributions: the first is the waiting time for a time slot, the second contribution is the time needed to send the entire schedule, and the last is the time for the last packet injected in the NoC to reach the destination node.

The equations presented above are obtained by performing low-level analysis on the architecture of the Argo-2 NoC. Therefore, an application programmer needs to add the software overhead of setting up DMA transfers for all of the NIs and triggering a reconfiguration request to the low-level worst-case reconfiguration time produced with Equation 6.1 and 6.2. With reference to the task-level model presented in Section 4.4, the overall worst-case time interval including the software overhead and the low-level worst-case reconfiguration time correspond to the WCET of the task τ_{rec} that performs the reconfiguration during a reconfiguration mode. In T-CREST, the WCET analysis tools platin [44] and aiT [37] can be used to perform this analysis.

Chapter 7

Evaluation and Discussion

This chapter evaluates and discusses the developed architectures and the proposed reconfiguration approach. The chapter is divided into three sections. The first and the second sections are respectively dedicated to the evaluation of the reconfiguration of computation and communication resources. The last section presents an evaluation of the reconfiguration of both computation and communication resources carried out using a multi-core audio DSP application as a case study.

7.1 Reconfiguration of Computation Resources

This section presents the evaluation of the developed hardware/software infrastructure and the reconfiguration proposed approach with regards to the computation resources. At first, we evaluate the developed RT-ICAP controller in terms of hardware cost, operating frequency, and reconfiguration speed. This is followed by the evaluation of the bit-stream compression technique and the reconfiguration time. Finally, we evaluate our reconfiguration approach using synthetic benchmarks and HLS-generated hardware accelerators. The results presented in this section were in part published and obtained in collaboration with the co-authors of the papers [J1], [C1], and [C4] in our list of publications.

7.1.1 RT-ICAP Controller Characterization

This subsection evaluates the hardware cost and performance of our RT-ICAP controller against some of the controllers presented as related work in Section 3.1. Table 7.1 presents the hardware size, the maximum operating frequency, and the reconfiguration speed results for our controller and the other controllers listed in the first column of the table. For these controllers, the results presented in the table are retrieved from the respective publications.

For each controller, Table 7.1 reports the target FPGA used to produce the results. All the results regarding our architecture are produced using Xilinx Vivado (v16.4) when targeting the Xilinx Kintex-7 FPGA (model XC7K325T-2FFG900C) and using Xilinx ISE and PlanAhead (v14.7) when targeting the Xilinx Virtex-6 FPGA (model XC6VLX240T-1FFG1156). All the synthesis properties are set to their defaults. The data size used for the ICAP interface is 32 bits. Table 7.1 also shows the hardware size of the controllers in terms of FFs, LUTs, and BRAMs. The hardware results for 7-series, Virtex-6, and -5 FPGAs can be quantitatively compared since all these FPGAs use 6-input LUTs. Virtex-4 FPGAs use 4-input LUTs; therefore, the LUT results are reported only for qualitative comparison. The BRAM used to store the bit-streams needed to produce the hardware results of Table 7.1 is not taken into account for any controller, except for the BRAM_HWICAP (marked with the superscript '3').

From the hardware results, we can observe that our controller is comparable in size to the controllers DPRM [77], D^{2}PR [79] (synchronous version without error check), and ICAP-I [78]. However, it must be taken into account that these controllers offer only simple transfer of bit-stream from a memory to the ICAP. Our controller offers more functionalities, such as support of two operating modes, bit-stream decompression, and a *status/control* register-based interface. Our controller is considerably smaller than the other controllers listed in Table 7.1 since it does not implement any functionality not strictly needed by our reconfiguration approach, such as bit-stream read-back.

The sixth column of Table 7.1 presents the maximum operating frequency of our controller and the other designs. In practical applications, the controller typically runs at the same frequency as the ICAP interface (maximum 100 MHz). From the results, we can observe that all the controllers are able to meet this constraint, except for the ICAP-I controller.

The last column of Table 7.1 shows the bit-stream transfer speed computed as a ratio between the bit-stream size and the reconfiguration time. The operating frequency is assumed to be 100 MHz for all the controllers, except for the ICAP-I (90 MHz). The reconfiguration speed is computed assuming that the bit-stream

Table 7.1: Hardware resources, maximum clock frequency, and reconfiguration speed results for our RT-ICAP controller and comparison with related published designs presented as related work in Chapter 3.

Controller	Target FPGA	Hardware resources			f_{max} (MHz)	Recon. speed (MB/s)
		FF	LUT	BRAM		
RT-ICAP	Kintex-7	101	245	0	>300	382.2
PRC [76]	Kintex-7	1270	1075	0	>100	n/a
ZyCAP [74]	Zynq-7000	806	620	0	>100	382
RT-ICAP	Virtex-6	88	190	0	323	382.2
DPRM [77]	Virtex-6	77	109	0	379	6.6[1]
D^{2}PR [79] [2]	Virtex-6	112	249	0	>100	400
XPS_HWICAP [69, 80]	Virtex-5	745	741	3	>100	1.3[2]
AC_ICAP [80]	Virtex-5	1667	1161	7	>100	381.0
ICAP-I [78]	Virtex-4	303	177	0	90	180.0
DMA_HWICAP [73]	Virtex-4	4277	977	0	121	82.1[1]
MST_HWICAP [73]	Virtex-4	1083	918	2	200	234.5
BRAM_HWICAP [73]	Virtex-4	963	469	32[3]	121	332.1

[1] Using off-chip memory. [2] Synchronous version - no error check. [3] Includes bit-streams storage.

is stored in an on-chip memory. For some controllers, the reconfiguration speed with the bit-streams stored in an on-chip memory is not available. In these cases, we report the reconfiguration speed for a bit-stream stored in an off-chip flash memory (marked with the superscript '1') for qualitative comparison.

For our controller, the table shows the speed for the *SPM-stream* mode calculated as the average of the reconfiguration speed of 6 different RLE-compressed sample bit-streams of size between 13.1 KB and 129.7 KB. From the results, we can observe that the reconfiguration speed of our controller is comparable or faster than the other controllers. The only exception is the D^2PR [79] controller (synchronous version, no error check), which is faster and matches the maximum transfer speed of the ICAP interface. Our controller is slower than this due to the presence of escape sequences in the bit-stream to decompress. If compression is not used, our controller can match the maximum speed of the ICAP interface.

7.1.2 Bit-Stream Compression and Reconfiguration Time

This subsection evaluates the compression capability of the *convbitstream* tool associated with the RT-ICAP controller and presents the trade-off between compression and reconfiguration time.

Table 7.2 reports the compression ratios obtained by applying different compression techniques on bit-streams characterised by different values of utilisation of the reconfigurable region. The compression ratio is defined as the size of the uncompressed bit-stream divided the by the size of the compressed one. The utilisation is defined as the percentage of the reconfigurable region, in term of slices, used by the hardware accelerator implemented in it. To produce these results, we use a reconfigurable region of 880 slices (equivalent to 7040 FF, 3520 LUTs and 16 DSPs) for all the experiments. Accelerators of different size are implemented in the reconfigurable region to produce the bit-stream with the desired utilisation value. The uncompressed bit-stream size, which depends only on the size of the reconfigurable region, is 184.8 KB.

To obtain the utilisation value of 91 %, a double-precision floating-point unit containing a multiplier and an adder is implemented in the reconfigurable region. This floating-point unit is generated with FloPoCo [115] and has a size of 805 slices (equivalent to 2406 FFs, 2524 LUTs, 12 DSPs). A double-precision floating-point adder of size 485 slices (equivalent to 1665 FFs, 1499 LUTs) is used to produce utilisation of 55 %. Leaving the reconfigurable region without any hardware implementation produces a blank bit-stream with utilisation of 0 %. The first two columns of Table 7.2 specify the accelerator used to generate the bit-stream and the corresponding reconfigurable region utilisation value.

Table 7.2: Compression ratios using our RLE compression, the Xilinx compression only, and our RLE compression on top of the Xilinx one for three bit-streams with different utilization ratios. The reconfigurable region size is 880 slices and the uncompressed bit-stream size is 184.8 KB.

Bit-stream	Recon. region utilization	Compression ratio				RLE+Xilinx size (KB)
		Ideal	RLE	Xilinx.	RLE+Xilinx	
Mult. & Add.	91 %	4.1	1.4	1.3	1.4	132.5
Adder	55 %	6.4	2.0	1.4	1.9	95.1
Blank	0 %	38.9	14.1	2.2	8.7	21.2

Table 7.3: Computed and measured reconfiguration time, expressed in clock cycles (CC), for the uncompressed and compressed sample bit-streams (*SPM-stream* operating mode).

Bit-stream	Computed (CC)				Measured (CC)
	Uncompr.	RLE	Xilinx	RLE+Xilinx	RLE+Xilinx
Mult. & Add.	47 311	48 745	37 384	38 994	38 999
Adder	47 311	49 480	33 782	36 105	36 109
Blank	47 311	48 655	21 175	22 527	22 529

For each bit-stream, Table 7.2 shows the ideal compression ratio and the ratios obtained by our RLE compression applied on a bit-stream, by the Xilinx tools compression only, and by our RLE compression applied on a bit-stream already compressed by the Xilinx tools. For the latter, the last column of the table also reports the compressed bit-stream size. The ideal compression ratio specifies a theoretical upper bound, and it is derived from the 32-bit-based information entropy of the entire bit-stream. All the compression ratios are calculated with reference to the size of the uncompressed bit-stream.

From the results, we can observe that the RLE compression introduces a significant reduction of the bit-stream size and that it performs better than the native compression offered by the Xilinx tools. Moreover, it can further decrease the size of the bit-stream already compressed by the Xilinx tools. This can be explained considering that the compression offered by the Xilinx tools is frame based (it does not repeat frames with the same content), while the granularity of our compression is considerably smaller than a frame (32-bit in this experiment). However, our RLE compression applied on a bit-stream already compressed by the Xilinx tools is not as good as the pure RLE compression, since the Xilinx compression reduces the amount of data-runs to be compressed by the RLE algorithm.

Table 7.3 shows the reconfiguration time for the uncompressed bit-streams and for the bit-streams compressed for the same techniques listed in Table 7.2. The reconfiguration time, expressed in clock cycles, is computed by the *convbitstream* tool for the RT-ICAP controller operating in *SPM-stream* mode.

For the bit-streams compressed with both the RLE and the Xilinx native compression, the last column of Table 7.3 reports the measured reconfiguration time. With this measurement, we aim to verify our computed WCET of the reconfiguration time. The measurement is executed using the RT-ICAP controller connected to a Patmos processor [22] and implemented on the Xilinx Virtex-6 FPGA. The time interval is measured by the software running on the processor. Since the end of the reconfiguration process is determined by polling the *status* register of the RT-ICAP controller in a loop, a small measurement overhead is observed. This overhead is in the order of few clock cycles and, when executing the WCET analysis of a task that uses reconfiguration, it is taken into account by the analysis tool which models the access to I/O devices. The results show that the computed reconfiguration time is correct. By removing the measurement overhead, it is possible to notice that there is no overestimation of the total reconfiguration time.

The most interesting observation can be made taking into account the results of both Table 7.2 and 7.3. Considering the reconfiguration time, we can notice that a bit-stream compressed with both the RLE and the Xilinx native compression

has a considerably shorter reconfiguration time (about 50 % on average) than the one compressed with RLE only. This is due to the fact that the Xilinx decompression logic writes identical configuration frames concurrently to multiple addresses of the configuration memory of the FPGA. This shows a trade-off between compression ratio and reconfiguration time, where the pure RLE compression leads to the highest compression ratio and slower reconfiguration and the Xilinx native compression leads to the lowest compression ratio and faster reconfiguration. We consider the combination of the RLE and the Xilinx native compressions as the best compromise between a high compression ratio and a low reconfiguration time. The above observations on the trade-off between compression and reconfiguration time are particularly valid for our RT-ICAP controller, where we aim to reduce the hardware size of the controller as much as possible. In general, it could be possible to run the controller with a faster clock than the one of the ICAP interface and to use clock domain crossing and FIFOs between the controller and the ICAP. In this case, it would be possible to utilise the ICAP interface at the maximum available speed, at the cost of larger and more complex hardware implementation.

7.1.3 Synthetic Benchmarks Experiments

This subsection evaluates our configuration approach by investigating the potential benefits of using DPR to implement hardware accelerators in real-time systems using synthetic benchmarks. Each benchmark can be considered as a mode of operation characterised by different computation resources GS requirements. In the following, we compare a static approach, in which non-reconfigurable accelerators are used to implement software tasks, against a reconfigurable approach, in which DPR is used to switch between accelerators belonging to different modes. Therefore, we analyse the trade-offs between hardware-resource utilisation and the computational performance loss due to the reconfiguration time overhead of DPR, aiming to determine whether the use of DPR can be advantageous over a static approach. For one of the test cases, we also investigate whether using DPR to switch between multiple specialised accelerators can provide a lower WCET bound with respect to the use of a more general one.

Experimental Setup and Benchmarks

The evaluation is carried out using the single-core architecture presented in Section 5.6 and shown in Figure 5.4. All the hardware results, are produced using Xilinx Vivado (v16.4) targeting the Xilinx Artix-7 FPGA (model XC7A100T-1CSG324C). The platin WCET analysis tool presented in Subsection 2.4.3 is used

to perform the WCET analysis of the accelerated tasks and the reconfiguration process. The size of the reconfigurable region is 1500 slices, which is equivalent to 6000 LUTs, 12000 FFs, 30 BRAMs, and 40 DSP elements.

The hardware accelerators used for the evaluation are based on code from four benchmarks of the TACLe suite, which is a collection of open-source C programs for timing analysis and real-time related research [113]. For the selected benchmarks, the computationally intensive part of the program (the algorithm itself) is manually identified and moved into hardware. The accelerators are generated using Xilinx Vivado HLS [31], which transforms C code from the TACLe benchmarks into VHDL. The benchmark is then modified to interact with the accelerator in order to perform the section of the program that was moved into hardware. With reference to Figure 5.4, the accelerator uses the *ap_ctrl_hs* interface protocol to communicate with the HwA-ctrl, as defined in [31, p. 89].

The benchmarks have been chosen to be representative of accelerators working on large data sets, small data sets, and data streams. The data type used in all the benchmarks is single-precision floating-point. The functionality of each benchmark and the associated hardware accelerator are described in the following list.

- **Matrix multiplication**: This benchmark executes the multiplication of two square matrices. For this benchmark, we have generated three different specialized accelerators for matrices of size 4×4, 16×16, and 32×32. Moreover, we have generated a generic accelerator for matrix multiplication which can take any given matrix of size up to 32×32. The latter is used to compare the use of specialised accelerators combined with DPR against the use of a generic static accelerator.
- **Matrix inversion**: This benchmark computes the inversion operation on a square matrix. Similarly to the matrix multiplication, we have also generated accelerators for matrices of size 4×4, 16×16, and 32×32.
- **2D FIR filter**: This benchmark performs a bi-dimensional finite impulse response (FIR) filtering on a matrix of size M×N using a 3×3 coefficient mask. This kind of filtering is commonly used for smoothing or sharpening bi-dimensional data sets. More specifically, the benchmark computes the cross-correlation between a 3×3 area surrounding each value of the input matrix and the coefficient mask. Zero-padding is performed at the matrix edges. For this benchmark, we have generated an accelerator for a matrix of size 3×3, corresponding to the minimum input matrix size.
- **Filterbank**: This benchmark implements a bank of with FIR filters for multi-rate signal processing. The input signal is passed through eight

independent FIR filters. The filtered signals are then down-sampled and up-sampled again. The up-sampled signals are passed through a second set of FIR filters and, finally, the outputs are summed together. Normally, some data processing is performed between the down-sampling and the up-sampling. In our benchmark, we do not perform any processing. For this benchmark, we have generated an accelerator where the input data and the two sets of filter coefficients are passed as arguments.

Reconfiguration Overhead Results

The first set of results aims to characterise the overhead of reconfiguration over the total WCET of the benchmarks. Three factors contribute to the WCET of a benchmark: C_{sw}, C_{hwa}, and T_{rec_dpr}. The first contribution C_{sw} is the WCET of the software section of the benchmark. This value is produced by the platin timing-analysis tool. C_{sw} also includes the WCET of the data transfer to and from the HwA-SPM and the WCET needed for the setup of the hardware accelerator. The second contribution C_{hwa} is the time needed by the accelerator to perform the computation. This result depends on the hardware architecture of the accelerator, and it is produced by the Vivado HLS tool. The third contribution T_{rec_dpr} is the reconfiguration time needed to perform DPR. This time interval is computed by our *convbitstream* tool using Equation 5.1, as presented in Section 5.5. Table 7.4 presents the values of these three contributions for all the benchmarks considered in this work, expressed in clock cycles.

The total WCET of a benchmark that uses DPR is denoted C_{tot_dpr}, and it can be calculated using Equation 7.1, where N is the number of computations executed by the accelerator after a reconfiguration.

$$C_{tot\_dpr} = T_{rec\_dpr} + N\,(C_{sw} + C_{hwa}) \tag{7.1}$$

The reconfiguration time T_{rec_dpr} is an overhead that cannot be avoided. This means that using reconfiguration always reduces the performances of the systems. However, its weight depends on the number of times N that the accelerator is used. To quantify the effect of this overhead, the last column of Table 7.4 ($N_{95\,\%}$) shows the number of times the accelerator needs to be used in order to reach a performance that is 95 % of that of a static solution. We can observe that, for the accelerators that perform computationally intensive tasks such as *Filterbank* and *Matrix multiplication* and *Matrix inversion* operating on 32×32 matrices, the value of $N_{95\,\%}$ is very low. Low values of $N_{95\,\%}$ show that the reconfigurable solution may be beneficial even if only a small number of computations are

Table 7.4: Contributions to the WCET in clock-cycles and the number of times ($N_{95\,\%}$) the accelerator has to be used to reach a performance that is 95 % of the one of a system that uses a static accelerator.

Benchmark	Software (C_{sw})	Hardware (C_{hwa})	Reconfig. (T_{rec_dpr})	$N_{95\,\%}$
Matrix mult.				
- 4×4	3 203	42	133 436	781
- 16×16	30 345	1 107	130 786	79
- 32×32	114 816	8 351	133 823	21
Matrix inv.				
- 4×4	2 307	793	130 772	802
- 16×16	21 363	12 168	131 885	75
- 32×32	78 859	55 223	130 071	18
2D FIR filter	3 174	137	132 098	758
Filterbank	46 450	216 264	132 152	10

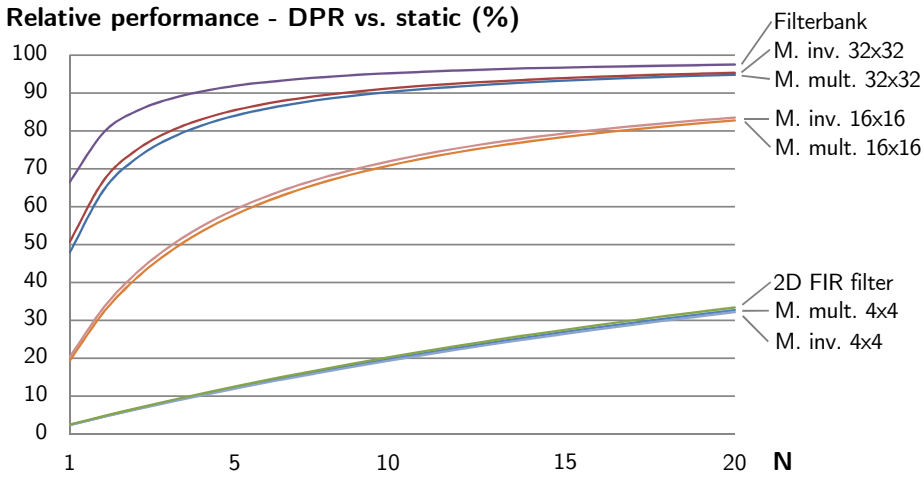

Figure 7.1: Plot showing the performance when using DPR relative to the performance when using static accelerators for values of $N \in [1, 20]$, calculated as the ratio $(C_{tot\_dpr} - T_{rec\_dpr})/C_{tot\_dpr}$.

required in a certain mode since the performance reduction is compensated for by lower hardware cost. This will be discussed in detail in the next subsection.

Figure 7.1 provides additional insight into the relation between reconfiguration overhead and the number of times, N, that the hardware accelerator is used. The figure shows the performance when using DPR, relative to the one of a static solution, for values of $N \in [1, 20]$. The relative performance is calculated as the ratio $(C_{tot\_dpr} - T_{rec\_dpr})/C_{tot\_dpr}$. In the figure, we can observe how results for benchmarks characterised by similar computation complexity tend to cluster together. In the top of the plot, we can find the curves related to *Filterbank* and the benchmarks operating on 32×32 matrices. Right below, in the second group, we can find the curves related to the benchmarks operating on 16×16 matrices. For these two groups, the solution using DPR becomes comparable to the static approach, in terms of computational performance, even for low values of N. Finally, the curves related to the *2D FIR filter* and to the benchmarks operating on 4×4 matrices are located in the lower part of the plot, showing that it is unlikely that a real application can benefit from using DPR unless a very large amount of computations are performed between reconfigurations.

Hardware Cost

One of the advantages of using DPR for the reconfiguration of computation resources is the possibility to reduce the overall hardware size of the system. This can be achieved by sharing the reconfigurable region between the hardware resources that are only utilised for a limited amount of time (e.g., only during a specific mode of operation). Table 7.5 shows the hardware cost results in terms of LUTs, FFs, BRAMs, and DSP elements for the main components of the hardware platform (shown in Figure 5.4) and for all the hardware accelerators used for this set of experiments.

Considering a hypothetical application where the functionality of each benchmark is a mode of operation, we can observe that the minimum size of the reconfigurable region should be enough to contain the largest hardware requirements across all the hardware accelerators. In our case, this corresponds to 5 126 LUTs, 7 411 FFs and 3 BRAMs to fit the *Filterbank* accelerator, and 20 DSP elements to fit the *Matrix multiplication* and *Matrix inversion* accelerators. The last two rows of the table report the total hardware cost of the static and the reconfigurable solutions. The total cost for a static solution includes the processor, the HwA-controller, the accelerators operating on 32×32 matrices (these are able to operate also on smaller matrices), the *Filterbank* accelerator, and the *2D FIR filter accelerator*. The total cost for a solution that uses DPR includes the processor, the HwA-controller, the RT-ICAP controller, the bit-stream SPM, and the largest resource

Table 7.5: Hardware cost results for the hardware platform and all the accelerators used in the experiments.

Entity	LUT	FF	BRAM	DSP
Patmos	4 931	3 602	8.5	4
HwA controller	7	4	0	0
RT-ICAP controller[1]	289	105	0	0
Bit-stream SPM[1]	72	7	64	0
Matrix mult.				
- 4×4[1]	1 270	1 138	0	20
- 16×16[1]	1 979	2 523	0	20
- 32×32	2 763	4 048	0	20
Matrix inv.				
- 4×4[1]	2 051	2 017	0.5	5
- 16×16[1]	3 425	3 725	0.5	20
- 32×32	4 080	4 636	0.5	20
2D FIR filter	1 816	1 987	0	10
Filterbank	5 126	7 411	3	10
Generic matrix mult.[1, 2]	2 912	4 037	0	5
Total cost static	18 723	21 688	12	64
Total cost DPR	10 425	11 129	75.5	24

[1] Not included for the total hardware cost for the static solution.

[2] Results used for the specialised vs. generic accelerator evaluation.

requirements across the accelerators. Using the total cost for a static solution as a baseline, the reconfigurable solution leads to the saving of around 44 % of LUTs, 49 % of FFs and 62 % of DSP elements. However, since the RT-ICAP controller works in *SPM-stream* mode, the utilisation of BRAMs is considerably higher in the reconfigurable solution. If these memory resources are needed by the system, the *CPU-stream* mode or pre-fetching can be used to eliminate or reduce the BRAM consumption. Taking into account the hardware cost results and the performance results previously presented, we can conclude that in our case, for a value of N sufficiently high, DPR leads to a more efficient use of FPGA resources and delivers comparable computational performance with respect to a static solution.

Table 7.6: Size, reconfigurable region utilisation, and compression ratio of the bit-streams for all the benchmark. The compression ratio accounts for the RLE compression performed by the *convbitstream* tool.

Benchmark	Bit-stream size (KB)	Rec. region utilization	Compression ratio
Matrix mult.			
- 4×4	209.3	35 %	2.3
- 16×16	274.3	62 %	1.7
- 32×32	328.1	83 %	1.5
Matrix inv.			
- 4×4	217.9	49 %	2.2
- 16×16	336.1	86 %	1.4
- 32×32	354.5	94 %	1.3
2D FIR filter	233.8	49 %	2.1
Filterbank	369.8	100 %	1.3

Bit-Stream Size and Compression Ratio

As previously explained, we apply compression in the bit-streams in order to maximise the number of bit-streams that can be stored locally in the SPM. Table 7.6 reports the size, the reconfigurable region utilisation, and the compression ratio of the partial bit-streams for all the benchmarks. The bit-stream size shown in the table refers to the bit-stream compressed with our RLE compression. The utilisation is the percentage of the reconfigurable region, in term of slices, used by the hardware accelerator implemented in it. The compression ratio is defined as the size of the uncompressed bit-stream divided the by the size of the compressed one.

From the results, we can observe that the RLE compression produces bit-streams ranging from 1.3 to 2.3 times smaller than the original size, with an average compression ratio of 1.7. This leads to a memory saving of 1.5 MB over a total size of 3.7 MB needed for storing all uncompressed bit-streams. As expected, the compression delivers a higher ratio for those accelerators that have lower utilisation of the reconfigurable region, since these bit-streams contain long runs of zeros for the unused logic.

Comparison with Software

In this set of experiments, we evaluate the speed-up in terms of WCET between the solution that uses hardware accelerators in combination with DPR and a pure software solution. As previously explained, using a reconfigurable solution instead of a static one introduces an overhead that decreases with the number of computations N executed by the accelerator after a reconfiguration. For this set of experiments, we evaluate the speed-up assuming $N = N_{95\%}$ (reported in the last column of Table 7.4). Table 7.7 presents the WCET and the speed-up results for all the benchmarks. All WCETs listed in the table are normalised with respect to N, in order to show the WCET of a single execution.

The Patmos processor is not equipped with a floating-point unit. Therefore, to execute floating-point operations, it utilises equivalent software functions from the LLVM project [43, 116]. Comparing a solution that uses floating-point accelerators against software running on a processor that is not equipped with a floating-point unit can produce biased results, since the speed-up does not characterise the ability of the accelerator to exploit parallelism, but mainly the speed-up related to faster floating-point operations. To have a fairer comparison, in addition to the results where the floating-point operations are executed in software, we also present results for the software benchmarks that use 32-bit integer data instead of single-precision floating-point, since these operations are supported in hardware by the processor. For the pure software solution, the data to be processed is placed in the local SPM as it is done in the accelerator-based solution. This avoids data cache misses to be accounted into the WCET.

The first column of the table reports the WCET of the reconfigurable solution C_{tot_dpr} computed using Equation 7.1 and the values presented in Table 7.4. The second and third columns present the WCET of the pure software solution obtained with the WCET analysis tool platin for floating-point C_{sw_fp} and integer data types C_{sw_int}, respectively. Finally, the last two columns show the speed-up calculated as the ratio between the WCET of the solution that uses accelerators in combination with DPR and a pure software solution.

From the results, we can observe that the speed-up results for the software using the floating-point data type are very high. As previously mentioned, this can be explained by the fact that Patmos is not equipped with a floating-point unit. The speed-up results for the software using the integer data type are more realistic and give a concrete grasp of the value that can be obtained by using reconfigurable accelerators, which deliver speed-ups for the WCET ranging from 1.2 to 4.1. In one case, for the 4×4 *Matrix multiplication*, the speed-up is less than 1. This means that the reconfigurable accelerator solution performs worse than the software one. This can be explained by the fact that the execution time

Table 7.7: WCET and relative speed-up for the solution that uses accelerators in combination with DPR and pure software solutions using floating-point and integer data types.

Benchmark	HwA + DPR (C_{tot_dpr})	Software (float.) (C_{sw_fp})	Software (int.) (C_{sw_int})	Speed-up (float.)	Speed-up (int.)
Matrix mult.					
- 4×4	3 416	208 451	1 725	61.0	0.5
- 16×16	33 108	13 072 787	71 493	394.9	2.2
- 32×32	129 540	104 301 204	530 534	805.2	4.1
Matrix inv.					
- 4×4	3 263	280 084	48 894	85.8	15.0
- 16×16	35 289	14 607 172	497 190	413.9	15.1
- 32×32	141 308	113 617 924	2 169 670	804.0	15.4
2D FIR filter	3 485	471 460	4 155	135.3	1.2
Filterbank	275 929	864 106 383	15 279 271	3 131.6	55.4

Table 7.8: WCET results for general and specialized accelerators, reconfiguration time for the specialized accelerators, and the minimum number of operations $N_{100\,\%}$ for which using DPR becomes advantageous in terms of performance. The results are in clock cycles.

	4×4	16×16	32×32
Generic HwA	8 028	49 438	152 933
Specialized HwA	3 245	31 452	123 167
Reconfig. time	133 436	130 786	133 823
$N_{100\,\%}$	28	8	5

of the benchmark is dominated by data transfer and the acceleration provided by using hardware is not enough to compensate for the reconfiguration overhead. In contrast, the execution time of the *Filterbank* benchmark is dominated by computation with little data transfer, leading to a speed-up of one order of magnitude higher than the other benchmarks.

Specialised vs. Generic Accelerator Trade-off

In this last set of results, we aim to determine what benefits can be derived from the use of specialised hardware accelerators combined with DPR instead of a generic one. As previously mentioned, for *Matrix multiplication* we have generated specialized accelerators for matrix sizes of 4×4, 16×16, 32×32, and a generic accelerator that can take in input matrices of any size up to 32×32. The specialised accelerators execute the multiplication of a specific matrix size faster than the generic one. The idea is to investigate the trade-offs between using DPR to switch between the use of multiple specialised accelerators and the use of a static generic accelerator. The trade-offs are measured in terms of WCET and hardware utilisation. Table 7.8 shows the WCETs, in clock cycles, for both the specialised accelerators and the generic one for the *Matrix multiplication* benchmark for the three different matrix sizes. The table also shows the reconfiguration time for the specialised accelerators.

In contrast to the experiment evaluating the reconfiguration overhead, where the solution using reconfiguration is always slower than the static one, in this experiment the solution using reconfiguration can be faster since the specialised accelerators are faster than the generic ones. After a certain amount of computations, the overhead introduced by the reconfiguration time will be compensated for by the speed difference between the specialised and the generic accelerators. The value $N_{100\,\%}$, shown in the last row of Table 7.8, is the threshold value

of N (number of times the accelerator is used) for which the generic and the specialised accelerators, including reconfiguration, are equivalent in performance. For values of $N > N_{100\,\%}$, the use of the specialised accelerator combined with reconfiguration outperforms the general accelerator.

In terms of hardware cost, it is possible to observe from Table 7.5 that the minimum size of the reconfigurable region should be enough to contain the specialised 32×32 *Matrix multiplication* accelerator. This size is smaller than the resources needed to implement the generic accelerator (third to last row of Table 7.5).

7.2 Reconfiguration of Communication Resources

This section presents the evaluation of the architecture and of the proposed reconfiguration approach with regards to the reconfiguration of communication resources. At first, the Argo 2 NoC architecture is evaluated in terms of hardware cost and maximum operating frequency. Then, the reconfiguration technique of Argo 2 is evaluated in terms of schedule size and worst-case performance, using synthetic traffic. The results presented in this section were in part published and obtained in collaboration with the co-authors of the papers [J3] and [C5] in our list of publications.

7.2.1 Argo 2 Characterization

This subsection evaluates the hardware cost and the maximum operating frequency of the Argo 2 FPGA implementation compared with the original Argo NoC and with some of the NoCs presented as related work in Section 3.2. The results are produced using Xilinx ISE Design Suite (version 14.7) targeting the Xilinx Virtex-6 FPGA (model XC6VLX240T-1FFG1156). All the synthesis properties are set to their defaults, with the only exception of the synthesis optimisation goal, which is set to area or speed.

Table 7.9 presents the hardware cost in terms of LUTs, FFs, and BRAMs for one 5-ported router and one NI. The NI used to produce the results is equipped with 256-entries schedule table and 64-entries DMA table. These tables are implemented using BRAMs. We consider these sizes reasonable for a large platform ($\geq$ 16 nodes). The table reports two set results, optimised for area or speed, for both the Argo 2 and the original Argo NoCs.

Table 7.9: Hardware cost and maximum operating frequency of the Argo 2 and the original Argo NoC architectures, for one 5-port router and one NI. The NI is equipped with 256-entries schedule table and 64-entries DMA table.

	Optimized for area		Optimized for speed	
	Argo	Argo 2	Argo	Argo 2
LUT	926	1 071	1 155	1 358
FF	897	908	923	931
BRAM	4	4	4	4
f_{max} (MHz)	155	166	167	179

From the results in Table 7.9, we can observe that our extension of the original Argo with the hardware needed to support reconfiguration, variable length packets, and interrupt and configuration packets only adds a small amount of hardware resources overhead. The Argo 2 NoC is around 16 % larger in terms of LUTs and 1 % larger in terms of FFs. With regards to the maximum operating frequency, Argo 2 is around 7 % faster than the original Argo.

Table 7.10 shows the comparison of the Argo 2 NoC to the aelite, dAElite [90], and IDAMC [84] NoCs. The aelite and dAElite [90] NoCs are TDM-based. The IDAMC [84] NoC uses a classic router designed with VC buffers and flow control. The table shows the results of the four architectures for one NI and one 3-ported router. The table also reports the number of schedule table and DMA table entries per node used in the experiment (2 entries are the minimum for the DMA table in Argo 2). The results related to the aelite and dAElite NoCs we compare against are available for a 2-by-2 platform with mesh topology, from which we derived the hardware consumption of one 3-ported router and one NI. For a fair comparison, we forced the synthesis tool to use FFs to implement memory in the Argo 2 NI.

From the results in Table 7.10, we can observe that the Argo 2 NoC implementation is overall smaller than the other NoCs. Moreover, the results show that the numbers for the IDAMC NoC are much higher than that of the aelite, dAElite, and Argo 2 NoCs. This can be explained considering that using VC buffers and flow control mechanisms is more complex than a TDM-based approach. With regards to the maximum operating frequency, the Argo 2 implementation is comparable to aelite and dAElite for a 3-port router. The operating frequency results for the IDAMC NoC are not available for comparison. Finally, we can observe that the Argo 2 5-port router implementation optimised for area (results

Table 7.10: Hardware resources utilization and maximum operating frequency of the Argo 2 architecture and three similar designs for one 3-port router and one NI.

	Optimized for area			Optimized for speed			
	aelite	dAElite	Argo 2	aelite	dAElite	Argo 2	IDAMC
Sch. tab. entries	8	8	8	8	8	8	–
DMA entries	1	1	2	1	1	2	8
LUT	1 916	2 506	1 185	2 351	3 117	1 342	9 160
FF	3 861	3 081	1 021	3 960	3 243	1 047	5 462
BRAM	0	0	0	0	0	0	7
f_{max} (MHz)	119	122	125	200	201	204	n/a

in Table 7.9) is around 33 % faster than the 3-port one. This is due to the use of FFs to implement memory, instead of BRAMs.

7.2.2 Synthetic Traffic Experiments

This section evaluates the Argo 2 NoC in terms of schedule size, worst-case reconfiguration time, and worst-case schedule transmission time. For the experiments, we use synthetic communication traffic patterns derived from the MCSL benchmark suite [117] and an *All-to-all* schedule. The target platform size is 4-by-4 with a bi-torus topology.

Schedule Size

In the following, we evaluate the schedule size for all the benchmarks. Table 7.11 shows the minimum and the maximum number of bytes needed to store the schedule in the schedule table of one node in the platform. Moreover, it reports the minimum and maximum number of VCs of one node in the platform. The sum of the maximum schedule table sizes of all the benchmarks is 696 bytes for one node. Therefore, this value represents the minimum schedule table size that is required to store all the schedules at the same time. We believe that in most cases, a schedule table size of 1 KB is sufficient to store the schedules associated with all the modes of an application, avoiding the need to transmit a new schedule from the reconfiguration master to all the other cores through the NoC.

Table 7.11: Minimum and maximum number of bytes needed to store the schedule in the schedule table, and minimum and maximum number of VCs for one node in the platform.

Benchmark	Sched. table size (Byte)		Number of VCs	
	min	max	min	max
FFT-1024	52	108	13	27
Fpppp	56	108	13	26
RS-dec	24	76	4	18
RS-enc	20	68	1	16
H264-720p	20	72	1	16
Robot	32	84	1	15
Sparse	8	60	1	15
All-to-all	60	120	15	30

Table 7.12 shows the number of schedule table entries for the original Argo NoC and the Argo 2 NoC, and the reduction in the number of entries delivered by the compact schedule representation used in Argo 2. In the original Argo NoC, the number of entries is the same in all the nodes, while for the Argo 2 NoC the size varies between nodes. Therefore, for the Argo 2 NoC, the Table 7.12 reports the average number of entries. From the results, we can observe that the average reduction in the number of schedule table entries of each node due to the schedule format adopted in Argo 2 is 58 %. This does not imply that the memory requirements for the schedule table are smaller for the Agro 2 NoC, but only that there are fewer entries in the table. If the implementation targets FPGAs, where the data width of the BRAMs used to implement the schedule table is fixed (typically 18 bits), having fewer entries may also reduce the overall BRAM requirements. To a certain extent, this may mitigate the impact on the memory requirements of having to store multiple schedules at the same time in the NIs to perform an instantaneous reconfiguration.

Worst-Case Reconfiguration Time

The worst-case reconfiguration time T_{rec_noc} in the case where the new schedule is already stored in the schedule table of the NIs depends only on the currently executing schedule, and it is three times its period, as explained in Section 6.4. Equation 6.1 (presented in Section 6.4) can be used to calculate the worst-case reconfiguration time. Table 7.13 reports the TDM schedule period P_{curr} and

Table 7.12: The number of schedule table entries in each core of the original Argo NoC and the average number of entries in the Argo 2 NoC.

Benchmark	Original Argo (entries)	Argo 2 (entries)	Reduction (%)
FFT-1024	21	15	28.6
Fpppp	40	16	60.0
RS-dec	30	8	73.3
RS-enc	28	6	78.6
H264-720p	30	7	76.7
Robot	57	10	82.5
Sparse	9	4	55.6
All-to-all	18	16	11.1

Table 7.13: TDM schedule period P_{curr} and worst-case reconfiguration time T_{rec_noc}, starting from each benchmark as the current schedule.

Current schedule	P_{curr}	T_{rec_noc}
FFT-1024	74	222
Fpppp	95	285
RS-dec	77	231
RS-enc	73	219
H264-720p	78	234
Robot	127	381
Sparse	30	90
All-to-all	75	225

the worst-case reconfiguration time T_{rec_noc}, starting from each benchmark as the current schedule.

From the results, we can observe that in Argo 2 it is possible to switch the complete schedule in a time interval between 90 and 381 clock cycles. In Æthereal and dAElite, the reconfiguration mechanism involves tearing down and setting up VCs individually, which requires respectively 246 clock cycles for Æthereal and 60 clock cycles for dAElite [90]. For comparison, we consider the case where the NoC performs a reconfiguration from benchmark *RS-dec* to benchmark *RS-enc*. *RS-dec* has a maximum of 18 outgoing VCs and *RS-enc* a maximum of 16 outgoing VCs per node. We also assume that two-thirds of the VCs stay the same between the two configurations, resulting 6 VCs to be torn down and 4 VCs to be set up. On dAElite this would require $(6 + 4) \cdot 60 = 600$ clock cycles. In Argo 2 this reconfiguration from *RS-dec* to *RS-enc* needs 231 clock cycles.

Table 7.14: Worst-case schedule transmission time T_{send_sched} between the schedules of all the benchmarks.

	New schedule							
Current schedule	FFT-1024	Fpppp	RS-dec	RS-enc	H264-720p	Robot	Sparse	All-to-all
FFT-1024	–	2 010	1 418	1 270	1 341	1 560	1 122	2 229
Fpppp	2 577	–	1 814	1 624	1 716	2 004	579	2 862
RS-dec	2 091	2 088	–	1 318	1 398	1 626	1 164	2 316
RS-enc	1 983	1 980	1 396	–	1 326	1 548	1 104	2 196
H264-720p	2 115	2 112	1 494	1 338	–	1 644	1 176	2 355
Robot	3 435	3 438	2 422	2 174	2 292	–	1 914	3 822
Sparse	822	549	579	519	546	639	–	912
All-to-all	2 034	2 037	1 431	1 281	1 365	1 587	1 137	–

We can conclude that, in the case where two modes differ by more than few VCs, Argo 2 is characterised by a shorted reconfiguration time than Æthereal and by a comparable or shorter reconfiguration time than dAElite.

Worst-Case Schedule Transmission Time

In the case where the new schedule is not stored in the schedule table, Equation 6.2 (presented in Section 6.4) can be used to calculate the worst-case transmission time T_{send_sched} needed by the reconfiguration master to transfer a schedule to all the NIs. The worst-case transmission time depends on the characteristics of the current schedule and on the size of the new schedule. Table 7.14 reports the worst-case transmission time T_{send_sched} between the schedules of all the benchmarks. The first column shows the currently active schedule. The rest of the columns report the T_{send_sched} needed to transmit the new schedule indicated as column heading.

From the results in Table 7.14, we can observe that T_{send_sched} ranges between 519 and 3822 clock cycles. Overall, the *Sparse* benchmark has the lowest values of T_{send_sched}, due to the fact that it is characterised with the shorted TDM period and, therefore, the highest bandwidth between the reconfiguration master and all the other nodes. When the schedule needs to be transmitted, the reconfiguration time of the Argo 2 NoC is still comparable with the one of the Æthereal and dAElite NoCs. For example, the maximum T_{send_sched} of 3822 clock cycles is for

the switch from *Robot* to *All-to-all*, which corresponds to the reconfiguration of 255 VCs. In the same time interval, the Æthereal and dAElite NoCs can set up 16 and 64 VCs, respectively.

7.3 Audio DSP Application

This section presents an evaluation for both the reconfiguration of computation and communication resources carried out using a real-time audio DSP application. At first, we give an overview of the goals of this case study, and we present the overall functionality of the audio DSP application. Then, we present the hardware platform on which the application executes. This is followed by an explanation of the audio effects implemented in software and the ones implemented in hardware. Finally, we present and discuss the result in terms of reconfiguration time and hardware cost.

7.3.1 Overview

This case study aims to shows that the presented approaches for reconfiguration of both computation and communication resources can be effectively used in real-world embedded applications. Moreover, it further evaluates the presented architectures and techniques. To do this, we have supplemented a real-time DSP audio application with the reconfiguration feature. The application was first developed for the T-CREST multi-core platform and presented in the paper [C3] in our list of publications.

The DSP audio application functionality is based on a homogeneous synchronous data-flow model of computation [118]. In this model of computation, a digital signal is processed by a statically ordered sequence of actors. When an actor receives enough input tokens, it starts the computation to produce output tokens. In our case, the tokens are stereo samples of an audio signal, and the actors are audio effects running on processors or implemented in hardware. In other words, the DSP audio application consists of a sequence of audio effects applied to a continuous stream of data.

The individual effects, indicated with FX_i, can be modelled as a processing task equipped with an input buffer and an output buffer. Therefore, the audio application can be modelled as a chain of communicating task where the effect FX_{i+1} depends on the data produced by the effect FX_i. The first effect of the processing chain receives in input the audio samples from an analog-to-digital

converter (ADC) buffer and the last effect produces samples in output for the digital-to-analog converter (DAC) buffer. For simplicity, in this case-study, we use audio effects operating on a single sample. The execution steps performed by an effect are:

1. The effect receives the sample in the input buffer from the previous effect in the system (or from the ADC buffer if it is the first effect).
2. The sample is processed according to the DSP algorithm that the current effect implements. When processing is complete, the produced sample is stored in the output buffer.
3. The effect sends the data from its output buffer to the input buffer of the next effect in the system (or to the DAC buffer if it is the last effect).

The effects can be mapped to different cores of the platform, and multiple subsequent effects can be mapped to the same core. When effects are mapped to the same core, the output buffer of effect i coincides with the input buffer of effect $i + 1$, and it is stored in the local SPM of the core itself. The communication between effects mapped to different cores is achieved using the VCs offered by the message-passing NoC Argo 2. In this case, the output buffer of effect i is stored in the local SPM of the sending core, the input buffer of effect $i + 1$ is stored in the local SPM of the receiving core, and the transfer between the two buffers is managed by the DMAs in the NIs of the Argo 2 NoC.

Figure 7.2 shows an example of a set of four effects mapped to three cores and the respective communication channels. Effect FX_0 is mapped to P_0, effects FX_1 and FX_2 are mapped to P_1, and effect FX_3 is mapped to P_2. The communication channel C_{12} is a same-core channel, while the rest are NoC channels. The ADC and DAC buffers are large buffers located respectively in the input and the output of the processing chain. In our hardware platform, these are connected to processor P_0, as explained later. Once pre-filled with a predetermined amount of samples, these buffers can compensate for temporary interruptions of the data stream in the processing chains, such as in case of a cache miss.

The system uses a flow control communication paradigm to exchange data. The execution of each effect is triggered by a new sample in the input buffer and data is sent from the output buffer to the next effect as soon as it is available. The flow control communication mechanism provides the system with elasticity since the samples may be located in different elements of the platform at different times, depending on the execution time of the single effects. However, the number of samples in the effect chain is constant at any moment in time, and it corresponds to the latency of the system.

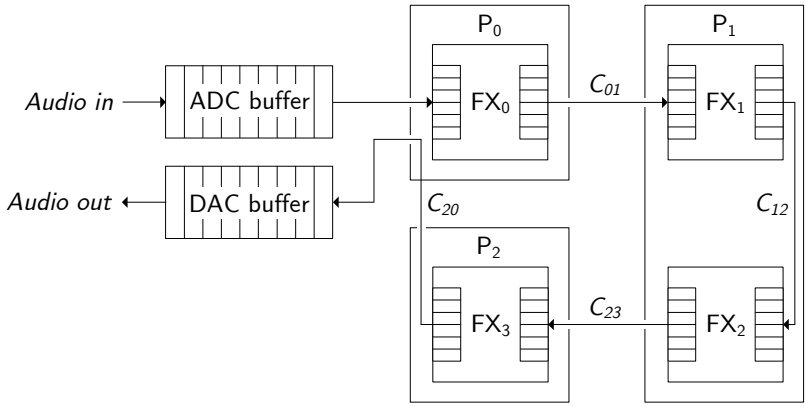

Figure 7.2: An example of a set of four effects FX_0, FX_1, FX_2, and FX_3 mapped to three processors P_1, P_2, and P_3. The communication channels between the effects are also shown.

The mapping of effects to the platform cores is executed at compile-time by a tool that, depending on the desired chain of effects, the sampling period, and the WCET of each effect, allocates the effects aiming to maximise the utilisation of the processing resources. After allocation, the tool provides the GS communication requirement between the nodes, which can be used as input to the TDM scheduler of the Argo 2 NoC.

The original DSP application does not support NoC reconfiguration and audio effects implemented in hardware. For this case study, we have modified the application and the hardware platform on which it is running to include multiple modes of operation, NoC reconfiguration, and to support reconfigurable hardware implementations of audio effects.

7.3.2 Hardware Platform

The original DSP application was developed targeting the Terasic DE2-115 development board, which is equipped with an Intel FPGA. Since we target Xilinx FPGAs, we have ported the application and the hardware platform to the Digilent Genesys 2 development board, which is equipped with a Xilinx Kintex-7 FPGA (XC7K325T-2FFG900C). All the results related to this case study are related to this FPGA model and are produced using Xilinx Vivado (v16.4). In addition to porting the hardware platform to the Genesys 2 board, we have also included a reconfigurable region where hardware effects are implemented and reconfigured during a mode change. We also added the RT-ICAP controller, the

bit-stream SPM, and additional infrastructure needed to preserve the continuity of the audio data stream during reconfiguration.

Figure 7.3 shows a block diagram of the hardware platform used for this case study. The T-CREST multi-core platform is shown at the right end of the figure, and it consists of four Patmos processors, the message-passing Argo 2 NoC, and the memory access NoC. The latter provides access to the off-chip memory through a memory controller. Processor P_0 has the role of reconfiguration master for both computation and communication resources. Therefore, the hardware infrastructure needed to support time predictable DPR, consisting of the RT-ICAP controller and the bit-stream SPM, is connected to this processor, as shown in the top-left end of Figure 7.3. Processor P_0 is also the master core for the DSP audio application. Therefore, it is connected to the ADC and DAC buffers for receiving the input samples and sending the output processed sample.

The Genesys 2 board includes an Analog Devices ADAU1761 audio codec chip [119]. This chip executes the analog-to-digital and digital-to-analog conversion of the audio signal and also offers additional functionalities such as volume control, mute, etc. The ADAU1761 audio codec provides and receives audio samples through a dedicated serial interface that uses the inter-integrated circuit sound (I^2S) protocol, as shown in Figure 7.3. A serialiser/deserialiser block is used to interface the ADAU1761 with our system, which uses a parallel representation of the samples (i.e. data and a valid bit). The ADAU1761 audio codec needs to be initialised and controlled through an interface based on the inter-integrated circuit (I^2C) protocol. This is managed by processor P_0 through a dedicated Audio controller block, as shown in the figure.

The hardware platform includes a reconfigurable region for hosting the effect implemented in hardware. For this case study, the reconfigurable region is connected to the input audio stream, as shown in Figure 7.3. This means that the effect implemented in hardware can only be the first one of the processing chain. We consider this sufficient for a proof-of-concept application. In the general case, reconfigurable regions hosting effects implemented in hardware could be placed anywhere in the processing chain. We placed the bypass buffer *Bypass_0* parallel to the reconfigurable region. This buffer has the same latency as the hardware effect implemented in the reconfigurable region, and it provides an alternative path for the samples during DPR. The multiplexer *MUX_0* selects from the *Bypass_0* output and from the reconfigurable region output. Moreover, *MUX_0* also decouples the reconfigurable region during DPR. A similar approach is implemented by the bypass buffer *Bypass_1* and the multiplexer *MUX_1* for the processing executed in software on the multi-core platform during NoC reconfiguration. Processor P_0, which manages the reconfiguration process, controls these multiplexers through the Audio controller block. Even if the reconfiguration process is extremely fast compared to the sampling frequency,

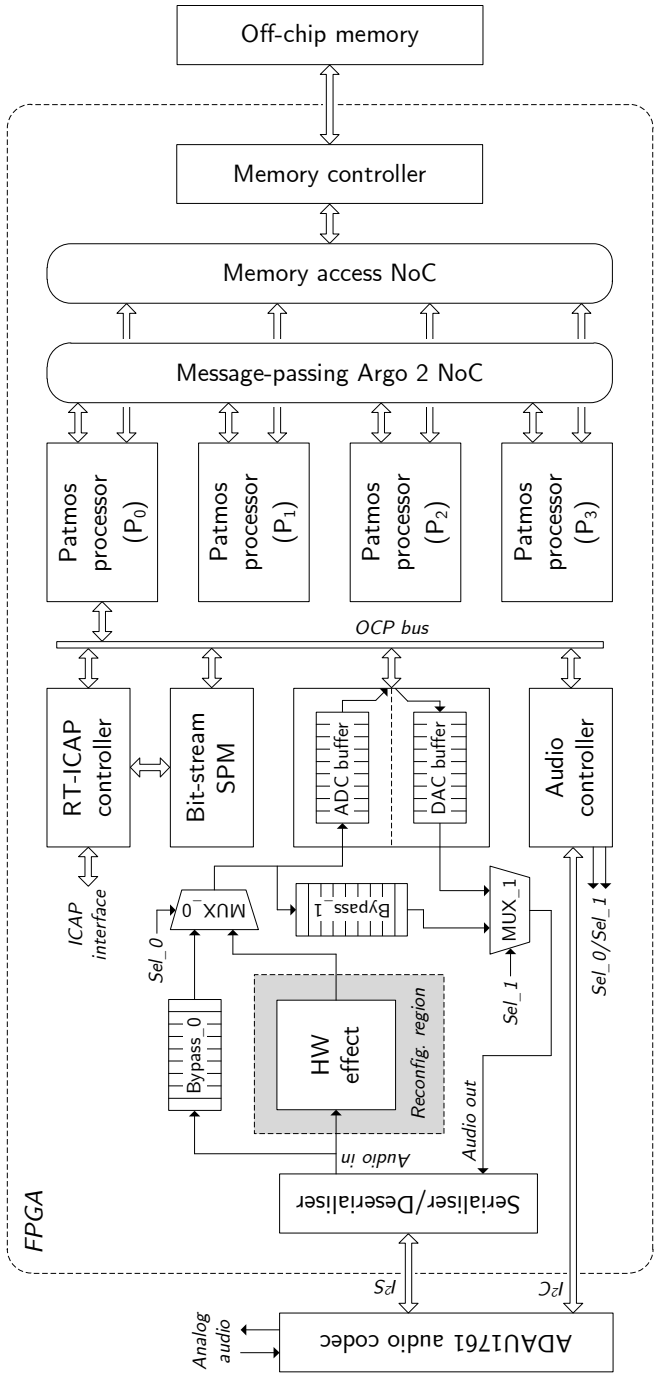

Figure 7.3: A block diagram of the hardware platform used for the audio DSP application case study.

the temporary interruption in the sample stream would result in an audible 'click' in the processed audio. Using bypass buffers prevents these interruptions, making the reconfiguration process imperceptible in the processed audio.

The operating frequency of the hardware platform is 100 MHz. The sampling frequency used by the ADAU1761 audio codec is 48 KHz and each sample consists of two 16-bit words (signed integer): one for the right channel and one for the left channel. Once having received a sample in input, each effect must be able to produce a sample in output and send it to the next actor in an interval of time that is shorter than the sampling period (i.e. 2083 clock cycles in our case). In Subsection 7.3.4, we show that the WCET of each effect implemented in software is less than the sampling period.

7.3.3 Effects and Modes of Operation

For this case study, we use six audio effects implemented in software and four effects implemented in hardware. The following list provides a brief description of the effects implemented in software, which were already available in the original DSP application. Further details on the DSP algorithms used in the effects can be found in [120].

- **DRY:** It just copies the input signal to the output with no processing performed. Useful for test and debugging purposes.
- **WAHWAH:** It is a band-pass filter applied on the original signal, where the central frequency of the filter is modulated.
- **OVERDRIVE:** It is a modulation effect that applies wave-shaping to enhance the character of the signal by adding some harmonic content to it.
- **DELAY:** It sums the original signal with a set of delayed attenuated replicas; thus, generating an echo.
- **CHORUS:** It is a set of second order comb filters, which generates a 'choir' effect.
- **TREMOLO:** It modulates the amplitude of the original signal, generating a 'vibration' effect.

The effects implemented in hardware are FIR filters. The coefficients and the hardware implementation of the filters are generated with MATLAB [121]. The order of all the filters is 50 and the latency is 9 samples, which corresponds to

the pipeline depth of the hardware implementation. The following list provides a brief description of the effects implemented in hardware:

- **PASS:** It copies the input signal to the output introducing a latency of 9 samples to match the other filters.
- **LOW-PASS:** It is a low-pass filter characterised by a cutting frequency of 700 Hz.
- **HIGH-PASS:** It is a high-pass filter characterised by a cutting frequency of 3500 Hz.
- **BAND-PASS:** It is a band-pass filter characterised by a low cutting frequency of 500 Hz and a high cutting frequency of 3400 Hz. The human voice belongs to this range of frequencies.

The DSP application we developed for this case study has 16 modes of operation. Each mode of operation is characterised by a specific chain of software effects running on the multi-core platform and by one of the hardware effects implemented in the reconfigurable region. We indicate the modes with $M_{ij}, i, j \in [0, 3]$. The index i specifies which hardware effect is implemented in the reconfigurable region in the current mode according to Table 7.15. The index j specifies which chain of software effects is running on the multi-core platform in the current mode according to Table 7.16. By combining the information of Table 7.15 and 7.16, it is possible to identify the complete chain of effects for each of the 16 modes. For example, mode M_{13} consists of LOW-PASS, CHORUS, and TREMOLO. Mode M_{00} does not execute any DSP processing on the audio signal (PASS, DRY, DRY).

Table 7.16 specifies, in parenthesis, to which processor of the multi-core platform each effect is mapped. The last column of the table shows the VCs that the NoC must offer in each mode. The message-passing library requires two VCs between each couple of communication processors: a forward VC for sending the data and a return VC for acknowledgement and synchronisation [52]. The required GS bandwidth offered by the NoC must be sufficient to move samples between cores at a rate of at least 1 sample every 2083 clock cycles (48 KHz sampling frequency with 100 MHz operating frequency). It should be noted that no effects are mapped to processor P_0 since this core has the role of reconfiguration master for both computation and communication resources. However, NoC channels for the audio stream are needed to and from this processor, since P_0 is also the audio master processor connected to the ADC and DAC buffers (see Figure 7.3).

Table 7.15: Hardware effect implemented in the reconfigurable region for each mode of operation.

Mode	Hardware effect in reconf. region
M_{0x}	PASS
M_{1x}	LOW-PASS
M_{2x}	HIGH-PASS
M_{3x}	BAND-PASS

Table 7.16: Chain of software effects and sets of VCs that the NoC must offer in each mode of operation. The processor of the multi-core platform in which each effect is mapped is reported in parenthesis.

Mode	Chain of software effects	Required VCs
M_{x0}	DRY (P_1) $\rightarrow$ DRY (P_1)	$P_0 \leftrightarrow P_1$, $P_1 \leftrightarrow P_0$
M_{x1}	WAHWAH (P_1)	$P_0 \leftrightarrow P_1$, $P_1 \leftrightarrow P_0$
M_{x2}	OVERDRIVE (P_1) $\rightarrow$ DELAY (P_2)	$P_0 \leftrightarrow P_1$, $P_1 \leftrightarrow P_2$, $P_2 \leftrightarrow P_0$
M_{x3}	CHORUS (P_1) $\rightarrow$ TREMOLO (P_2)	$P_0 \leftrightarrow P_1$, $P_1 \leftrightarrow P_2$, $P_2 \leftrightarrow P_0$

7.3.4 Observations and Results

As previously mentioned, the main goal of this case study is to show that the presented approach to reconfiguration can be successfully used in real-world applications. In the following, we present some observations on the implications of using reconfiguration in the DSP application and we provide and discuss results in terms of reconfiguration time for both computation and communication resources and hardware cost.

The first observation regards the techniques we have used to maintain the number of samples that are in the entire effects chain constant during reconfiguration. During a mode change that requires changing the effect implemented in hardware, the master processor performs the following operations:

1. It diverts the audio stream through the *Bypass_0* buffer.
2. It triggers the RT-ICAP controller to perform DPR of the hardware effect.
3. It waits for the hardware effect to fill with samples.
4. Finally, it diverts back the audio stream from the reconfigurable region

A mode change that requires changing the effects implemented in software is more challenging. In this case, the master processor performs the following operations:

1. It diverts the audio stream through the *Bypass_1* buffer.
2. It flushes the samples in the software effects chain.
3. It stops the tasks related to the old mode.
4. If needed, it performs a NoC reconfiguration.
5. It starts the tasks related to the new mode.
6. It pre-fills the software effects chain with a predetermined amount of samples.
7. Finally, it diverts back the audio stream from the DAC buffer.

In our case, the *Bypass_0* buffer holds 9 samples, which is the latency of the hardware effect implemented in the reconfigurable region. The *Bypass_1* buffer holds 64 samples, which is the amount of samples pre-filled in the chain of effects implemented in software (including the ADC and DAC buffers). The flushing is performed by stopping samples to enter the ADC buffer, while the pre-filling is performed by stopping samples from exiting the DAC buffer.

Table 7.17 shows the reconfiguration time for the hardware effects. In order to give a real grasp of this time interval, the table reports the results in clock cycles, milliseconds, and audio sampling periods. The reconfiguration time shown in the table only refers to the DPR. An additional 9 sampling periods (equivalent to 18 747 CC or 18.7 μs) needed by the hardware effect to fill with samples before *MUX_0* can be switched, should be added to the reconfiguration time. From this set of results, we can observe that the DPR takes at most 4.3 ms to switch between different hardware effects, making our approach to reconfiguration of computation resources a concrete and viable solution for the DSP audio application, especially taking into account the saving in the hardware cost as presented in the following.

From Table 7.16, we can observe that only two NoC schedules are needed by the DSP application. The first schedule (Schedule_01) serves the modes M_{x0} and M_{x1}, while the second schedule (Schedule_23) serves the modes M_{x2} and M_{x3}. Table 7.18 reports the total number of VCs, the schedule period length, and the NoC worst-case reconfiguration time needed to switch the schedule (starting from the current schedule). The latter is equivalent to three times the schedule period

Table 7.17: Reconfiguration time needed to perform DPR for the hardware effects (*SPM-stream* mode).

Hardware effect	Reconfiguration time		
	(CC)	(ms)	(samples)
PASS	205 414	2.1	99
LOW-PASS	381 647	3.8	183
HIGH-PASS	433 148	4.3	208
BAND-PASS	359 786	3.6	173

Table 7.18: Total amount of VCs, schedule period length, and worst-case reconfiguration time for the two schedules used by the DSP application.

TDM schedule	Number of VCs	TDM period (CC)	Rec. time (CC)
Schedule_01	4	15	45
Schedule_23	6	18	54

when all the schedules are already stored in the NIs. In addition to the VCs required by the DSP application, three additional VCs are added from processor P_0 to the other processors for sending configuration packets. From the results, we can observe that the NoC reconfiguration time is negligible compared with the sampling period. However, during a mode change that requires changing the effects implemented in software, the effect chain needs to be flushed and then re-filled. Since our chain of effects is pre-filled with 64 samples, a time interval of 128 sampling periods (equivalent to 266 624 CC or 2.7 ms) must be added to the NoC reconfiguration time presented in Table 7.18. For our DSP application running on a 2-by-2 platform and characterised by VCs bandwidth requirements in the order of few words per sampling period, an All-to-all schedule would be able to provide similar performances without the need to reconfigure the NoC. Nevertheless, the case study hints that the Argo 2 NoC reconfiguration is usable and viable since it can be performed in a very short amount of time.

Table 7.19 presents the WCET bounds produced by the platin timing-analysis tool and the experimental execution time measurements, in clock cycles, for all the software effects used in the case study, as presented in the paper [C3] in our list of publications. The WCET bounds and the measured execution times refer to the time interval required to process a single sample from the moment when the processing starts until the processed sample is placed in the output buffer.

Table 7.19: WCET bounds produced by the platin timing-analysis tool and experimental execution time measurements in clock cycles for all the software effects used in the case study for both cold and warm cache situations.

Software effect	Cold cache		Warm cache	
	WCET	Measured	WCET	Measured
WAHWAH	6 292	4 192	846	336
OVERDRIVE	1 874	1 518	113	84
DELAY	3 646	1 749	808	504
CHORUS	5 127	1 369	1 029	336
TREMOLO	2 167	207	331	84

The results in Table 7.19 are provided for both cold and warm cache situations. We can observe that the WCET and the execution time for cold cache are always larger than the ones for warm cache due to the stalls caused by main memory access. Moreover, as expected, the measured execution times are always smaller than the WCET values. Also, we can notice that all the warm cache WCETs are lower than the sampling period of 2083 clock cycles, while cold cache WCETs exceed the sampling period. Cold cache situations only happen after a reconfiguration. In this cases, the audio stream is diverted through the *Bypass_1* buffer and diverted back when the software effect chain (including the ADC and DAC buffers) is pre-filled with samples. In general, these buffers can compensate for temporary interruptions of the audio stream, such as in an occasional case of cache miss.

A similar audio application is used as case study in [122]. In this work, the application consists of a producer (ADC) a consumer (DAC) and an intermediate stage (the audio effect) communicating through the Æthereal NoC [91]. The authors use the formal model presented in the same work to verify the end-to-end temporal behaviour of their audio application. Our application relies on the assumption that the WCET of each software effect, together with the time interval of the receiving and sending operations, is faster than the sampling period. This assumption, in combination with the fact that all the audio effects have an input and an output buffer of a single sample, ensures the correct functionality of the system. As previously mentioned, the total latency of our audio DSP application corresponds to the constant amount of samples that are stored in the effect chain at any moment in time.

Finally, Table 7.20 presents the hardware cost for the platform used in this case study. The rows of the table are divided into groups. The first group reports results for the entire T-CREST platform and the memory controller. The first

Table 7.20: Hardware cost for the hardware platform used in the audio DSP application case study.

Entity	LUT	FF	BRAM	DSP
Multi-core platform	27 900	23 857	53	16
Memory controller	6 894	5 797	0	0
Audio controller	111	102	0	0
Audio buffer	118	67	1	0
Bypasses and MUXs	108	2 477	0	0
RT-ICAP controller[1]	294	171	0	0
Bit-stream SPM[1]	110	55	256	0
PASS effect	288	0	0	0
LOW-PASS effect	3 910	3 664	0	102
HIGH-PASS effect	2 002	3 730	0	126
BAND-PASS effect	3 902	3 656	0	102
Total cost static	45 223	43 350	54	346
Total cost DPR	39 445	36 256	310	142

[1] Not included for the total hardware cost for the static solution.

row includes the Patmos processors, the two NoCs, and OCP bus infrastructure. The second group and the third group report the hardware cost for all the components related to the audio stream and the infrastructure supporting DPR, respectively. The fourth group presents the results for the hardware effects. Only one effect at a time is implemented in the reconfigurable region. Therefore, this should be large enough to accommodate 3 910 LUTs required by the LOW-PASS effect and 3 730 FFs and 126 DSP blocks required by the HIGH-PASS effect. Finally, the last two rows report the total hardware cost of the platform that uses DPR and of a static solution where all the hardware effects are implemented at the same time. The total for the static solution does not include those resources required due to reconfiguration (marked with the superscript '1' in the table).

Using the total cost for the static solution as a baseline, we can observe that the use of DPR leads to the savings of around 13 % of LUTs, 16 % of FFs and 59 % of DSP blocks. However, the reconfigurable solution uses considerably more BRAMs due to the bit-stream SPM. This amount of memory is only needed if the RT-ICAP controller operates in *SPM-stream* mode. In our case, the entire design not including the bit-stream SPM consumes a very limited amount of BRAMs. Therefore, we opted for using the available BRAMs to implement the bit-stream SPM and benefit from the maximum reconfiguration speed offered by the RT-ICAP controller in *SPM-stream* mode. If the controller operates in

CPU-stream mode, the bit-stream SPM would not be needed. However, this would lead to a slower reconfiguration process and require the master processor to perform the bit-stream transfer from an external memory.

Chapter 8

Conclusion

This chapter concludes the thesis by summarising the contributions of this work and discusses future work.

8.1 Summary and Final Remarks

In this thesis, we explored the use of run-time reconfiguration in the context of multi-core real-time embedded systems addressing both computation and communication resources.

At first, we presented an approach for using reconfigurable computing in real-time multi-core systems where the reconfiguration of computation and communication resources is associated with operational mode changes. Reconfiguration is used to adapt the hardware platform to meet the functional and temporal GS requirements of the current mode of operation. The main goal is to minimise the hardware cost by only implementing what is needed by the current mode of operation rather than what is required by all possible modes. With regards to this, we presented how the GS requirements can be extracted from a multi-mode application, and we suggested modelling the reconfiguration process as a task characterised by a certain WCET and belonging to a reconfiguration mode. This allows the application of system-level scheduling policies and schedulability

analysis in an independent manner for each mode of operation and for each reconfiguration mode.

For the reconfiguration of the computation resources, we used the DPR feature of FPGAs to switch between different hardware accelerators. To support this, we developed the time-predictable RT-ICAP controller and the associated *convbitstream* tool for bit-stream compression and reconfiguration time analysis. The evaluation has shown that the hardware size of our RT-ICAP controller is comparable to or smaller than other available controllers developed in industry and academia. The evaluation has also shown that the RLE-based compression introduces a significant compression ratio of the partial bit-streams, ranging from 1.3 to 2.2 for the set of sample bit-streams used in the experiments. Moreover, our compression can further reduce the size of bit-streams that are already compressed with the Xilinx native compression.

The idea of using DPR for reconfiguring hardware accelerators during a mode-change was evaluated using synthetic benchmarks from the TACLe suite and accelerators generated with the Xilinx HLS tool. The results have shown that using DPR leads to a reduction of the hardware cost in comparison with a static solution that implements all the needed accelerators at the same time. From the selected benchmarks, we also concluded that for computationally-intensive tasks, the reconfiguration time overhead does not significantly affect the worst-case performance. In comparison with a pure software solution, the results have shown that using accelerators in combination with DPR leads to speed-ups, in terms of WCET, ranging from 1.2 to 4.1. For two cases, the speed-ups were 0.5 and 55.4; we considered these as special cases. Overall, we can conclude that our approach for reconfiguration of computation resources: (1) leads to a reduction of the hardware cost maintaining computation performance that is comparable to a static approach and (2) can offer WCET speed-up over a pure software solution.

For the communication resources, we developed the new NI used in the Argo 2 NoC, which allows instantaneous switching between a set of TDM schedules. The evaluation has shown that the Argo 2 NoC architecture is less than half of the size of other NoCs offering similar reconfiguration capabilities and only slightly larger than the original Argo NoC. The performance evaluation of the Argo 2 NoC was carried out using communication traffic patterns derived from the MCSL benchmark suite and an *All-to-all* schedule. In terms of reconfiguration time, the results have shown that the worst-case reconfiguration time of Argo 2 is comparable to the functionally-similar NoCs and considerably shorter if the new schedule is already loaded into the schedule table. Finally, we mention that the reconfiguration of the Argo 2 NoC is completely transparent for the VCs that persist across a reconfiguration and VCs can even be re-mapped to different

slots or to follow different paths through the NoC without disruption in the end-to-end communication flow.

The thesis also presents low-level reconfiguration time analysis for the developed architectures supporting reconfiguration. These architectures were prototyped targeting the existing time-predictable multi-processor platform T-CREST, which was supplemented with reconfiguration capabilities by this work.

The reconfiguration of both the computation and communication resources was used in a final case study based on a multi-core DSP application implementing a selection of audio effects. We modified the DSP application and the hardware platform to support run-time reconfiguration of effects implemented both in hardware and software. The case study, characterised by 16 modes of operation, has shown that the presented approaches for reconfiguration can be effectively used in a real-world multi-mode application. Compared to a static approach, the use of our techniques delivered a reduction of the overall hardware size and a better utilisation of the platform resources while maintaining comparable computation performance.

8.2 Future Work

In the following, we present possible extensions to the work presented in this thesis with regards to task mapping and scheduling, energy consumption, and use of differential bit-streams for DPR.

Task Mapping and Scheduling

Task-level methodologies, such as task mapping and scheduling, represents an important aspect of research on real-time systems. The focus of this thesis was mainly on hardware architectures and low-level timing-analysis. Further investigation could be conducted in relation to the proposed approach where reconfiguration is associated to mode changes.

A possible extension would include exploring possibilities and developing specialised tools for mapping of tasks to the cores of a multi-core platform. Currently, this process is carried out manually or with the use of simple tools specialised in the mapping of a specific application, such as in the case of the audio DSP application. A dedicated tool would execute the mapping of tasks taking into account the reconfiguration capabilities of both computation and communication

resources. A possible approach would aim to minimise the number of reconfigurations of hardware resources per mode change, as well as the total number of bits-stream needed by the application. For the NoC reconfiguration, the mapping should aim to reduce the TDM schedule length, possibly leading to a reduction of the schedule size and of the reconfiguration time.

With regards to task scheduling, we proposed a model where the reconfiguration process is modelled as a task in a reconfiguration mode, as explained in Section 4.4. An alternative approach to be investigated would include the reconfiguration process into the normal execution of task aiming to mask the reconfiguration time or the pre-fetching of bit-streams from main memory into the reconfiguration SPM.

Energy Consumption

Using hardware accelerators to implement computing tasks instead of software may lead to a reduction of the total energy required to perform a certain task due to the use of specialised and efficient hardware. Further investigation of this aspect can be conducted with regards to the reconfiguration of computation resources.

Employing DPR to switch between different accelerators may enable saving energy needed to execute a task, but it also introduces an energy consumption overhead for the reconfiguration process. It would be interesting to quantify the trade-off between the energy needed to execute a task purely in software and the energy required to execute the task on reconfigurable hardware taking into account the overhead of the reconfiguration process. We would expect results similar to the ones related to the reconfiguration time overhead, where the use of DPR becomes beneficial only when the accelerator is used for a number of times greater than a certain threshold.

In addition, the use of DPR to perform blanking of an unused reconfigurable region in order to save energy could also be analysed. Blanking consists of writing a bit-stream that implements no active hardware in the reconfigurable region. This could be used as an alternative to clock gating to reduce energy consumption by cancelling the hardware architecture of certain areas of the FPGA when not used for an extended period of time.

Use of Differential Bit-Streams

In Subsection 2.2.3, we briefly presented the possibility offered by the Xilinx tools to generate differential partial bit-streams for a reconfigurable region. Differential bit-streams only store the differences between a previous configuration and the new one. If the differences between two configurations are minimal, a differential bit-stream has a considerably smaller size than one that contains the full configuration information for a reconfigurable region (complete bit-stream).

A possible extension of the work presented in this thesis could be to use differential bit-streams aiming to reduce the memory needed to store the configurations. If the mode transition graph of a multi-mode application is not fully connected and it is possible to identify modes that can be reached by one or by a sufficiently small subset of other modes, differential bit-stream can be associated to the transitions to these modes. In this case, a differential bit-stream is associated to each transition to one of these modes, in contrast to having only one complete bit-stream associated to the mode itself. A reduction in memory utilisation is achieved if the total size of all the differential bit-streams for a mode is smaller than the size of the complete bit-stream. The *convbitstream* tool could be extended to receive in input the mode transition graph and the information regarding the size of all the partial bit-streams (differential and complete) to make a decision, for each mode, whether the use of differential bit-streams is convenient.

Bibliography

[1] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (MPSoC) technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, 2008.

[2] W. Dally, "Route packets, not wires: On-chip interconnection networks," in *Proc. Design Automation Conference*. ACM, 2001, pp. 684–689.

[3] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[4] J. Andersson, J. Gaisler, and R. Weigand, "Next generation multi-purpose microprocessor," *European Space Agency, Esa Sp. (Special Publication)*, vol. 682 SP, pp. 83–86, 2010.

[5] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, "A distributed run-time environment for the Kalray MPPA-256 integrated manycore processor," *Elsevier Procedia Computer Science*, vol. 18, pp. 1654–1663, 2013.

[6] A. Varghese, B. Edwards, G. Mitra, and A. P. Rendell, "Programming the Adapteva Epiphany 64-core network-on-chip coprocessor," *International Journal of High Performance Computing Applications*, vol. 31, no. 4, pp. 285–302, 2015.

[7] A. Olofsson, T. Nordström, and Z. ul Abdin, "Kickstarting high-performance energy-efficient manycore architectures with Epiphany," in *Proc. Asilomar Conference on Signals, Systems and Computers*. IEEE, 2014, pp. 1719–1726.

[8] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, 2015.

[9] XILINX, "DS190: Zynq-7000 All-programmable SoC data sheet: Overview (v1.11)," Tech. Rep., 2017, online (last accessed: March 2018).

[10] Microsemi, "PB0115: Product brief - SmartFusion2 SoC FPGA," Tech. Rep., 2017, online (last accessed: March 2018).

[11] Intel SoC FPGA Website, "Webpage: *https://www.altera.com/products/soc/overview.html*," (last accessed: March 2018).

[12] Menta-eFPGA Website, "Webpage: *http://www.menta-efpga.com/*," (last accessed: March 2018).

[13] Achronix Website, "Webpage: *https://www.achronix.com/*," (last accessed: March 2018).

[14] XILINX, "UG909: Vivado design suite user guide - Partial reconfiguration (v2017.1)," Tech. Rep., 2017, online (last accessed: March 2018).

[15] ——, "UG702: Partial reconfiguration user guide (v14.5)," Tech. Rep., 2013, online (last accessed: March 2018).

[16] ALTERA Corporation, "QII51026: Design planning for partial reconfiguration," Tech. Rep., 2013, online (last accessed: March 2018).

[17] K. Shin and P. Ramanathan, "Real-time computing - A new discipline of computer-science and engineering," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, 1994.

[18] G. C. Buttazzo, *Hard real-time computing systems.* Kluwer Academic Publishers, 1997.

[19] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - Overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, 2008.

[20] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quiñones, M. Gerdes, M. Paolieri, and J. Wolf, "MERASA: Multi-core execution of hard real-time applications supporting analysability," *IEEE Micro*, vol. 30, no. 5, pp. 66–75, 2010.

[21] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch,

P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Elsevier Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[22] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, and T. Thorn, "Towards a time-predictable dual-issue microprocessor: The Patmos approach," in *Proc. Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, 2011, pp. 11–20.

[23] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. Müller, K. Goossens, and J. Sparsø, "Argo: A real-time network-on-chip architecture with an efficient gals implementation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 2, pp. 479–492, 2016.

[24] J. Garside and N. C. Audsley, "Investigating shared memory tree prefetching within multimedia NoC architectures," in *Proc. Memory Architecture and Organisation Workshop*, 2013, pp. 1–7.

[25] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparsø, "A time-predictable memory network-on-chip," in *Proc. International Workshop on Worst-Case Execution Time Analysis*, 2014, pp. 53–62.

[26] G. Estrin, "Organization of computer systems," in *Proc. Western Joint IRE/AIEE/ACM Computer Conference*. ACM, 1960, pp. 33–40.

[27] W. S. Carter, K. Duong, R. H. Freeman, H. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo, and S. L. Sze, "A user programmable reconfigurable logic array," in *Proc. Custom Integrated Circuits Conference*. IEEE, 1986, pp. 233–235.

[28] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and using a highly parallel programmable logic array," *IEEE Computer*, vol. 24, no. 1, pp. 81–89, 1991.

[29] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. International Symposium on Computer Architecture*. ACM/IEEE, 2014, pp. 13–24.

[30] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proc. International Symposium on Microarchitecture*. ACM/IEEE, 2016, pp. 1–13.

[31] XILINX, "UG902: Vivado design suite user guide - High-level synthesis (v2017.1)," Tech. Rep., 2017, online (last accessed: March 2018).

[32] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. International Symposium on Field Programmable Gate Arrays*. ACM, 2011, pp. 33–36.

[33] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[34] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan, "An empirical evaluation of high-level synthesis languages and tools for database acceleration," in *Proc. International Conference on Field Programmable Logic and Applications*. IEEE, 2014, pp. 1–8.

[35] XILINX, "UG470: 7-Series FPGA configuration - User guide (v1.12)," Tech. Rep., 2017, online (last accessed: March 2018).

[36] ——, "UG360: Virtex-6 FPGA configuration - User guide (v3.9)," Tech. Rep., 2015, online (last accessed: March 2018).

[37] R. Heckmann and C. Ferdinand, "Worst-case execution time prediction by static program analysis," AbsInt Angewandte Informatik GmbH, Tech. Rep., online (last accessed: March 2018).

[38] AbsInt aiT WCET Analyzers Website, "Webpage: *https://www.absint.com/ait/*," (last accessed: March 2018).

[39] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Proc. Conference on Design Automation*. ACM, 2007, pp. 264–265.

[40] S. A. Edwards, S. Kim, E. A. Lee, I. Liu, H. D. Patel, and M. Schoeberl, "A disruptive computer design idea: Architectures with repeatable timing," in *Proc. International Conference on Computer Design*. IEEE, 2009.

[41] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[42] M. Schoeberl, F. Brandner, S. Hepp, W. Puffitsch, and D. Prokesch, "Patmos reference handbook," Technical University of Denmark, Tech. Rep., 2014.

[43] P. Puschner, R. Kirner, B. Huber, and D. Prokesch, "Compiling for time predictability," *Springer Computer Safety, Reliability, and Security: Lecture Notes in Computer Science*, vol. 7613, pp. 382–391, 2012.

[44] S. Hepp, B. Huber, J. Knoop, D. Prokesch, and P. P. Puschner, "The platin tool kit - The T-CREST approach for compiler and WCET integration," in *Proc. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, 2015, pp. 1–16.

[45] OCP official website, "Webpage: *http://www.accellera.org/downloads/standards/ocp/*," (last accessed: March 2018).

[46] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl, "A method cache for Patmos," in *Proc. Symposium on Object/Component/Service-oriented Real-time Distributed Computing*. IEEE, 2014, pp. 100–108.

[47] S. Abbaspour, F. Brandner, and M. Schoeberl, "A time-predictable stack cache," in *Proc. Workshop on Software Technologies for Embedded and Ubiquitous Systems*. IEEE, 2013, pp. 1–8.

[48] M. D. Gomony, B. Akesson, and K. Goossens, "Architecture and optimal configuration of a real-time multi-channel memory controller," in *Proc. Design, Automation Test in Europe Conference and Exhibition*. IEEE, 2013, pp. 1307–1312.

[49] E. Lakis and M. Schoeberl, "An SDRAM controller for real-time systems," in *Proc. Workshop on Software Technologies for Embedded and Ubiquitous Systems*. IEEE, 2013, pp. 1–8.

[50] E. Kasapaki, "An asynchronous time-division-multiplexed network-on-chip for real-time systems," PhD Thesis, PhD-2015-361, Department of Applied Mathematics and Computer Science, Technical University of Denmark, 2015.

[51] J. Sparsø, E. Kasapaki, and M. Schoeberl, "An area-efficient network interface for a TDM-based network-on-chip," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2013, pp. 1044–1047.

[52] R. B. Sørensen, W. Puffitsch, M. Schoeberl, and J. Sparsø, "Message passing on a time-predictable multicore processor," in *Proc. Symposium on Real-time Distributed Computing*. IEEE, 2015, pp. 51–59.

[53] E. Kasapaki and J. Sparsø, "Argo: A time-elastic time-division-multiplexed NoC using asynchronous routers," in *Proc. International Symposium on Asynchronous Circuits and Systems*. IEEE Computer Society Press, 2014, pp. 45–52.

[54] ——, "The Argo NoC: Combining TDM and GALS," in *Proc. European Conference on Circuit Theory and Design*. IEEE, 2015, pp. 1–4.

[55] R. B. Sørensen, J. Sparsø, M. R. Pedersen, and J. Højgaard, "A metaheuristic scheduler for time division multiplexed networks-on-chip," in *Proc. Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*. IEEE/IFIP, 2014, pp. 309–316.

[56] E. Kasapaki, J. Sparsø, R. B. Sørensen, and K. Goossens, "Router designs for an asynchronous time-division-multiplexed network-on-chip," in *Proc. Euromicro Conference on Digital System Design*. IEEE, 2013, pp. 319–326.

[57] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[58] K. Wu, "Reconfigurable architectures: From physical implementation to dynamic behavoir modelling," PhD Thesis, IMM-PhD-2007-180, Department of Informatics and Mathematic Modelling, Technical University of Denmark, 2007.

[59] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder - A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs," in *Proc. International Conference on Field Programmable Logic and Applications*. IEEE, 2008, pp. 119–124.

[60] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable FPGAs," in *Proc. Real-Time Systems Symposium*. IEEE, 2016, pp. 1–12.

[61] D. Göhringer and J. Becker, "High performance reconfigurable multiprocessor-based computing on FPGAs," in *Proc. International Symposium on Parallel Distributed Processing, Workshops and PhD Forum*. IEEE, 2010, pp. 1–4.

[62] D. Göhringer, M. Hübner, V. Schatz, and J. Becker, "Runtime adaptive multi-processor system-on-chip: RAMPSoC," in *Proc. International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–7.

[63] D. Göhringer, M. Hübner, E. N. Zeutebouo, and J. Becker, "CAP-OS: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures," in *Proc. International Symposium on Parallel Distributed Processing, Workshops and PhD Forum*. IEEE, 2010, pp. 1–8.

[64] S. Banerjee, E. Bozorgzadeh, and N. D. Dutt, "Integrating physical constraints in HW-SW partitioning for architectures with partial dynamic reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 11, pp. 1189–1202, 2006.

[65] D. Göhringer, M. Hübner, M. Benz, and J. Becker, "A design methodology for application partitioning and architecture development of reconfigurable multiprocessor systems-on-chip," in *Proc. International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2010, pp. 259–262.

[66] R. Cattaneo, R. Bellini, G. Durelli, C. Pilato, M. D. Santambrogio, and D. Sciuto, "PaRA-Sched: A reconfiguration-aware scheduler for reconfigurable architectures," in *Proc. Parallel Distributed Processing Symposium Workshops*. IEEE, 2014, pp. 243–250.

[67] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An operating system approach for reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, 2014.

[68] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, 2004.

[69] XILINX, "DS586: LogiCORE IP XPS HWICAP product specifications (v5.01a)," Tech. Rep., 2011, online (last accessed: March 2018).

[70] ——, "DS586: Processor Local Bus (v4.6)," Tech. Rep., 2019, online (last accessed: March 2018).

[71] ——, "PG134: AXI HWICAP LogiCORE IP product guide (v3.0)," Tech. Rep., 2015, online (last accessed: March 2018).

[72] ——, "UG761: AXI reference guide (v13.1)," Tech. Rep., 2011, online (last accessed: March 2018).

[73] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time partial reconfiguration speed investigation and architectural design space exploration," in *Proc. International Conference on Field Programmable Logic and Applications*. IEEE Computer Society, 2009, pp. 498–502.

[74] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41–44, 2014.

[75] M. Hübner, D. Göhringer, J. Noguera, and J. Becker, "Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs," in *Proc. International Symposium on Parallel Distributed Processing, Workshops and PhD Forum*. IEEE, 2010, pp. 1–8.

[76] XILINX, "PG139: LogiCORE IP PRC product guide (v1.0)," Tech. Rep., 2015, online (last accessed: March 2018).

[77] J. Tarrillo, F. A. Escobar, F. L. Kastensmidt, and C. Valderrama, "Dynamic partial reconfiguration manager," in *Proc. Latin American Symposium on Circuits and Systems.* IEEE, 2014, pp. 1–4.

[78] V. Lai and O. Diessel, "ICAP-I: A reusable interface for the internal reconfiguration of Xilinx FPGAs," in *Proc. International Conference on Field-programmable Technology.* IEEE Computer Society, 2009, pp. 357–360.

[79] S. D. Carlo, P. Prinetto, P. Trotta, and J. Andersson, "A portable open-source controller for safe dynamic partial reconfiguration on Xilinx FPGAs," in *Proc. International Conference on Field Programmable Logic and Applications.* IEEE, 2015, pp. 1–4.

[80] L. A. Cardona and C. Ferrer, "AC-ICAP: A flexible high speed ICAP controller," *International Journal of Reconfigurable Computing*, vol. 2015, pp. 1–15, 2015.

[81] C. Schuck, B. Haetzer, and J. Becker, "An interface for a decentralized 2D reconfiguration on Xilinx Virtex-FPGAs for organic computing," *International Journal of Reconfigurable Computing*, vol. 2009, pp. 1–11, 2009.

[82] B. Dupont de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti, "Guaranteed services of the NoC of a manycore processor," in *Proc. International Workshop on Network on Chip Architectures.* ACM, 2014, pp. 11–16.

[83] R. L. Cruz, "A calculus for network delay. Part I: Network elements in isolation," *IEEE Transactions on Information Theory*, vol. 37, no. 1, pp. 114–131, 1991.

[84] B. Motruk, J. Diemer, R. Buchty, R. Ernst, and M. Berekovic, "IDAMC: A many-core platform with run-time monitoring for mixed-criticality," in *Proc. International Symposium on High-Assurance Systems Engineering.* IEEE, 2012, pp. 24–31.

[85] J. Diemer and R. Ernst, "Back suction: Service guarantees for latency-sensitive on-chip networks," in *Proc. International Symposium on Networks-on-Chip.* ACM/IEEE, 2010, pp. 155–162.

[86] T. Bjerregaard and J. Sparsø, "A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip," in *Proc. Design, Automation and Test in Europe Conference and Exhibition.* IEEE, 2005, pp. 1226–1231.

[87] M. Winter and G. P. Fettweis, "Guaranteed service virtual channel allocation in NoCs for run-time task scheduling," in *Proc. Design, Automation Test in Europe Conference and Exhibition.* IEEE, 2011, pp. 1–6.

[88] ——, "A network-on-chip channel allocator for run-time task scheduling in multi-processor system-on-chips," in *Proc. Euromicro Conference on Digital System Design Architectures, Methods and Tools*. IEEE, 2008, pp. 133–140.

[89] K. Goossens and A. Hansson, "The AEthereal network on chip after ten years: Goals, evolution, lessons, and future," in *Proc. Design Automation Conference*. ACM/IEEE, 2010, pp. 306 –311.

[90] R. A. Stefan, A. Molnos, and K. Goossens, "dAElite: A TDM NoC Supporting QoS, multicast, and fast connection set-up," *IEEE Transactions on Computers*, vol. 63, no. 3, pp. 583–594, 2014.

[91] K. Goossens, J. Dielissen, and A. Radulescu, "AEthereal network on chip: concepts, architectures, and implementations," *IEEE Design Test of Computers*, vol. 22, no. 5, pp. 414–421, 2005.

[92] A. Hansson, M. Subburaman, and K. Goossens, "aelite: A flit-synchronous network on chip with composable and predictable services," in *Proc. Design, Automation Test in Europe Conference and Exhibition*. IEEE, 2009, pp. 250–255.

[93] A. Hansson and K. Goossens, "Trade-offs in the configuration of a network on chip for multiple use-cases," in *Proc. International Symposium on Networks-on-Chip*. IEEE, 2007, pp. 233–242.

[94] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip," in *Proc. Design, Automation Test in Europe Conference and Exhibition*, vol. 2. IEEE, 2004, pp. 890–895.

[95] M. Schoeberl, "A time-triggered network-on-chip," in *Proc. International Conference on Field Programmable Logic and Applications*. IEEE, 2007, pp. 377–382.

[96] C. Paukovits and H. Kopetz, "Concepts of switching in the time-triggered network-on-chip," in *Proc. International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2008, pp. 120–129.

[97] I. Kotleas, "Mode changes in network-on-chip based multiprocessor platforms," Master's thesis, Department of Applied Mathematics and Computer Science, Technical University of Denmark, 2014.

[98] Y. Xue and P. Bogdan, "Improving NoC performance under spatio-temporal variability by runtime reconfiguration: a general mathematical framework," in *Proc. International Symposium on Networks-on-Chip*. IEEE, 2016, pp. 1–8.

[99] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J. Y. Mignolet, "Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles," in *Design, Automation and Test in Europe*. IEEE, 2005, pp. 234–239.

[100] L. T. Smit, G. J. M. Smit, J. L. Hurink, H. Broersma, D. Paulusma, and P. T. Wolkotte, "Run-time mapping of applications to a heterogeneous reconfigurable tiled system on chip architecture," in *Proc. International Conference on Field- Programmable Technology*. IEEE, 2004, pp. 421–424.

[101] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. D. Micheli, "Mapping and configuration methods for multi-use-case networks on chips," in *Proc. Asia and South Pacific Conference on Design Automation*. IEEE, 2006, pp. 1–6.

[102] A. Hansson, A. Radulescu, and K. Goossens, "A unified approach to constrained mapping and routing on network-on-chip architectures," in *International Conference on Hardware/Software Codesign and System Synthesis*. IEEE/ACM/IFIP, 2005, pp. 75–80.

[103] A. Hansson, M. Coenen, and K. Goossens, "Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip," in *Proc. Design, Automation Test in Europe Conference and Exhibition*. IEEE, 2007, pp. 1–6.

[104] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, pp. 1–24, 2009.

[105] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha, "Virtual execution platforms for mixed-time-criticality systems: The CompSOC architecture and design flow," *ACM SIGBED*, vol. 10, no. 3, pp. 23–34, 2013.

[106] S. Sinha, M. Koedam, G. Breaban, A. Nelson, A. B. Nejad, M. Geilen, and K. Goossens, "Composable and predictable dynamic loading for time-critical partitioned systems on multiprocessor architectures," *Elsevier Microprocessors and Microsystems*, vol. 39, no. 8, pp. 1087 – 1107, 2015.

[107] J. Real and A. Crespo, "Mode change protocols for real-time systems: A survey and a new proposal," *Springer Real-Time Systems*, vol. 26, no. 2, pp. 161–197, 2004.

[108] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX*. Addison-Wesley, 2001.

[109] C. Carmichael, M. Caffrey, and A. Salazar, "XAPP216: Correcting single-event upsets through Virtex partial configuration (Xilinx Corporation - v1.0)," Tech. Rep., 2000, online (last accessed: March 2018).

[110] S. Hauck and W. D. Wilson, "Runlength compression techniques for FPGA configurations," in *Proc. Symposium on Field-programmable Custom Computing Machines*. IEEE Computer Society, 1999, pp. 286–287.

[111] Z. Li and S. Hauck, "Configuration compression for Virtex FPGAs," in *Proc. Symposium on Field-programmable Custom Computing Machines*. IEEE, 2001, pp. 147–159.

[112] D. Koch, C. Beckhoff, and J. Teich, "Hardware decompression techniques for FPGA-based embedded systems," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 2, no. 2, pp. 1–23, 2009.

[113] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Waegemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *Proc. International Workshop on Worst-Case Execution Time Analysis*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 1–10.

[114] A. Hansson, M. Coenen, and K. Goossens, "Channel trees: Reducing latency by sharing time slots in time-multiplexed networks on chip," in *Proc. International Conference on Hardware/Software Codesign and System Synthesis*. IEEE/ACM/IFIP, 2007, pp. 149–154.

[115] F. De Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design and Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.

[116] C. Lattner and V. Adve, "The LLVM instruction set and compilation strategy," Computer Science Department, University of Illinois at Urbana-Champaign, Tech. Rep. UIUCDCS-R-2002-2292, 2002.

[117] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, and Z. Wang, "A NoC traffic suite based on real applications," in *Proc. Computer Society Annual Symposium on VLSI*. IEEE, 2011, pp. 66–71.

[118] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[119] Analog Devices, "ADAU1761 datasheet (Rev. C)," Tech. Rep., 2019, online (last accessed: March 2018).

[120] U. Zöler, *DAFX: Digital Audio Effects, 2nd edition.* John Wiley & Sons, 2011.

[121] MATLAB: Basic FIR fiter/HDL coder Website, "Webpage: *https://se.mathworks.com/help/hdlfilter/basic-fir-filter.html*," (last accessed: March 2018).

[122] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij, "Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis," *IET Computers Digital Techniques*, vol. 3, no. 5, pp. 398–412, 2009.