João Gabriel Reis

A FRAMEWORK FOR PREDICTABLE HARDWARE/SOFTWARE COMPONENT RECONFIGURATION

Dissertation presented in partial fulfillment of the requirements for the degree of Master in Electrical Engineering.
Advisor: Prof. Dr. Eduardo Augusto Bezerra
Co-advisor: Prof. Dr. Antônio Augusto Fröhlich

Florianópolis

2016

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Reis, João Gabriel
A Framework for Predictable Hardware/Software Component Reconfiguration / João Gabriel Reis ; orientador, Eduardo Augusto Bezerra ; coorientador, Antônio Augusto Fröhlich. - Florianópolis, SC, 2016.
119 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia Elétrica.

Inclui referências

1. Engenharia Elétrica. 2. Field-Programmable Gate Array. 3. Computação Reconfigurável. 4. Sistemas Embarcados. 5. Reconfiguração Dinâmica. I. Bezerra, Eduardo Augusto. II. Fröhlich, Antônio Augusto. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia Elétrica. IV. Título.

João Gabriel Reis

A FRAMEWORK FOR PREDICTABLE HARDWARE/SOFTWARE COMPONENT RECONFIGURATION

This dissertation was accepted in its present form by the Programa de Pós-Graduação em Engenharia de Elétrica as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

Florianópolis, October 6, 2016.

Prof. Dr. Marcelo Lobo Heldwein
Graduate Program Coordinator

Prof. Dr. Eduardo Augusto Bezerra (video conference)
Advisor

Prof. Dr. Antônio Augusto Fröhlich (video conference)
Co-advisor

Master's committee:

Prof. Dr. Giovani Gracioli

Prof. Dr. Héctor Pettenghi Roldán

Prof. Dr. Leandro Buss Becker

Prof. Dr. Luigi Dilillo (video conference)

To my grandmother, Iracema.

ACKNOWLEDGEMENTS

This work would not be possible without the unconditional support of my family during those years, specially from my mother, Marlene, and my sister, Gabriela. I could only achieve my goals due to the love and education they provided me. I am also grateful for the constant encouragement and optimism of my partner, Tânia. Her emotional support and understanding were fundamental for this work's conclusion.

I would like to thank my advisor, Prof. Eduardo Augusto Bezerra, for accepting me as his advisee and his constant support during the last years. I would also like to thank my co-advisor, Prof. Antônio Augusto Fröhlich, for the opportunity of working at the Software/Hardware Integration Laboratory. The scientific environment maintained by him at the laboratory was fundamental for achieving this dissertation goals. I owe my gratitude to many of my colleagues specially Mateus Krepsky Ludwich, Arliones Hoeller Jr., Lucas Wanner, and Davi Resner for the valuable technical and scientific discussions.

I am also grateful to Prof. Giovani Gracioli, Prof. Héctor Pettenghi Roldán, Prof. Leandro Buss Becker, and Prof. Luigi Dilillo for serving on my dissertation committee and for their insightful comments and suggestions.

Finally, I would like to show my gratitude to CAPES for the financial support I received for realizing this work and to the UFSC for fostering a dynamic academic community.

ABSTRACT

Rigid partitions of components or modules in a hardware/software co-design flow can lead to suboptimal choices in embedded systems with dynamic or unpredictable runtime requirements. Field-Programmable Gate Array (FPGA) reconfiguration can help systems cope with dynamic non-functional requirements such as performance and power, hardware defects due to Negative-Bias Temperature Instability (NBTI) and Process, Voltage and Temperature (PVT) variations, or application requirements unforeseen at design time. This work proposes a framework for reconfigurable components whereby the reconfiguration of a component implementation is performed transparently without user intervention. The reconfiguration process is confined in system's idle time without interfering with or being interfered by other activities occurring in the system or even peripherals performing I/O. For components with multiple implementations, our approach opportunistically and speculatively monitors system load and performance parameters to check when the reconfiguration can start. The framework differs from previous approaches in its syntax and semantics for reconfigurable components which are preserved across the multiple implementations in different substrates and the reconfiguration process that can be split into multiple steps. To quantify the impact of I/O interference on FPGA reconfiguration, we measured the execution time when loading bitstreams containing hardware components implementations from memory to the FPGA reconfiguration interface with multiple peripherals performing I/O in parallel. Moreover, a Private Automatic Branch Exchange (PABX) case study investigated the deployment of reconfigurable components in a scenario with timing constraints. A reconfiguration policy for the PABX components was proposed to deal with the unpredictable number of calls it receives by using reconfigurable hardware resources without degrading voice quality due to reconfiguration. Furthermore, we explored trade-offs between power consumption, execution time, and accuracy in a set of reconfigurable mathematical components.

Keywords: Field-Programmable Gate Array. Reconfigurable Computing. Embedded Systems. Dynamic Reconfiguration.

RESUMO

O particionamento estático de componentes ou módulos ao realizar o co-design hardware/software pode levar a escolhas insatisfatórias em sistemas embarcados com requisitos dinâmicos e imprevisíveis durante tempo de execução. A reconfiguração dinâmica de Field-Programmable Gate Arrays (FPGAs) pode ajudar sistemas a se adaptar em requisitos dinâmicos e não funcionais como desempenho e consumo de energia, defeitos de hardware devido ao fenômeno Negative-Bias Temperature Instability (NBTI) e variações de *Processo, Tensão e Temperatura* ou ainda requisitos da aplicação que não foram levados em consideração em tempo de projeto. Esse trabalho propõe um framework para componentes reconfiguráveis onde a reconfiguração da implementação de um componente é realizada de maneira transparente e sem a intervenção do usuário. O processo de reconfiguração é confinado no tempo ocioso do sistema sem interferir ou sofrer interferência de outras atividades ou mesmo periféricos realizando operações de entrada/saída. Para componentes com múltiplas implementações, nossa abordagem monitora de maneira especulativa a carga do sistema e contadores de desempenho para escolher o momento em que a reconfiguração deve se iniciar. O framework se difere de trabalhos anteriores devido à sintaxe e semântica para componentes reconfiguráveis que são preservadas nas múltiplas implementações e em diferentes substratos e no processo de reconfiguração que pode ser dividido em diversos passos. Para quantificar o impacto da interferência de entrada/saída na reconfiguração de FPGAs, foi medido o tempo de execução para carregar *bitstreams* contendo implementações de componentes em hardware da memória para a interface de reconfiguração de FPGA com diversos periféricos realizando operações de entrada/saída em paralelo. Além disso, o estudo de caso de um Private Automatic Branch Exchange (PABX) investigou o uso de componentes reconfiguráveis num cenário com requisitos temporais. Uma política de reconfiguração para os componentes do PABX foi proposta para lidar o número imprevisível de chamadas recebidas através de recursos reconfiguráveis sem degradar a qualidade da reprodução da voz devido à reconfiguração. Foram também explorados os *trade-offs* entre consumo de energia, tempo de execução e exatidão dos resultados num conjunto de componentes implementando operações matemáticas.

Palavras-chave: Field-Programmable Gate Array. Computação Reconfigurável. Sistemas Embarcados. Reconfiguração Dinâmica.

LIST OF FIGURES

Figure 1 Generic FPGA architecture. 32
Figure 2 SPREAD programming model (AL., 2013b). 35
Figure 3 FUSE system architecture (ISMAIL; SHANNON, 2011). 36
Figure 4 The ViRUS framework (WANNER; SRIVASTAVA, 2014). . . 39
Figure 5 ADESD domain decomposition (FRöHLICH, 2001). 46
Figure 6 Example of the parametrized traits class for a generic component. 49
Figure 7 Reconfigurable component framework. 50
Figure 8 `Handle` implementing method forwarding and reconfiguration. 51
Figure 9 Possible `Component::Base` implementation. 52
Figure 10 A minimal implementation of `Handle's` constructor. 53
Figure 11 Example of helper functions for saving and restoring a component's state. 55
Figure 12 Cross-domain communication using proxies and agents as proposed by Mück (MüCK; FRöHLICH, 2013). 56
Figure 13 Generic interconnect architecture modeled using queuing theory. 60
Figure 14 Mapping of the architecture into the model. 63
Figure 15 Bitstream path from a memory device to a PIO-based and a DMA-based partial reconfiguration interface in a SoC with a central interconnect. 64
Figure 16 Contention time for transfering an L words bitrstream from the main memory to a PIO-based and a DMA-based reconfiguration interface. The arrival rate of the devices sharing the interconnect with the reconfiguration interface and the memory is presented as a fraction of the memory service rate μ_m. 65
Figure 17 Reconfiguration triggering and resource allocation sequence diagram. 74
Figure 18 Component bitstream loading sequence diagram. 75
Figure 19 `Component_Manager` powering down two threads and the peripherals being used by them. 77
Figure 20 State handling and rebinding sequence diagram. 79
Figure 21 EPOSSoC block diagram. CPU, IO, and rec nodes are interconnected by RTSNoC routers. 85

Figure 22 RTSNoC internals. 85
Figure 23 CPU node block diagram. Software components are deployed in it by an RTOS. 86
Figure 24 EPOS component framework metaprogram. 87
Figure 25 Resulting architecture for a component deploying a software and hardware implementation. 89
Figure 26 Experiment software architecture. The application used and the operating system run on a CPU node of the EPOSSoC. 91
Figure 27 Hardware architecture of the first experiment. Threads start DMA transactions in peripherals connected to the AXI interconnect as PCAP. 92
Figure 28 Results for the interconnect contention experiment. Reconfiguration time for different normalized bitstream sizes with interference inflicted by peripherals performing DMA. Different configurations of peripheral's AXI transaction burst length, threads number, and peripheral operating frequency. 94
Figure 29 Hardware architecture of the second experiment. Threads start DMA transactions in peripherals connected to the AXI interconnect isolated from PCAP but still share the DDR memory. 95
Figure 30 Results for the memory contention experiment. Reconfiguration time for different normalized bitstream sizes with interference inflicted by peripherals performing DMA. Different configurations of peripheral's AXI transaction burst length, threads number, and peripheral operating frequency. 96
Figure 31 PABX telephony system. 98

LIST OF TABLES

Table 1 Comparison of related works that provide support for using reconfigurable resources present in FPGAs. 43
Table 2 PABX reconfiguration policy. 99
Table 3 Hardware resources utilisation for the communication infrastructure in the XC7Z020 SoC supporting the invocation of a method with a 4 B argument and a 4 B return value. 100
Table 4 Execution time of each function call in the reconfiguration process of different reconfigurable components. 101
Table 5 Hardware resources to implement the FFT in the XC7Z020 SoC. 103
Table 6 FFT implementations characteristics. 104
Table 7 Hardware resources needed to implement the exponental function in the XC7Z020 SoC. 105
Table 8 Natural exponential implementations characteristics. 105

LIST OF ABBREVIATIONS AND ACRONYMS

ACP Accelerator Coherency Port.

ADC Analog-to-Digital Converter.

ADESD Application-Driven Embedded System Design.

ADPCM Adaptive Differential Pulse-Code Modulation.

AES Advanced Encryption Standard.

ALU Arithmetic Logic Unit.

AMBA ARM Advanced Microcontroller Bus Architecture.

AOP Aspect Oriented Programming.

API Application Program Interface.

ASIC Application-Specific Integrated Circuit.

AXI Advanced eXtensible Interface.

DAC Digital-to-Analog Converter.

DDR Double Data Rate.

DFT Discrete Fourier Transform.

DMA Direct Memory Access.

DSP Digital Signal Processor.

DTMF Dual-Tone Multi-Frequency.

EDA Electronic Design Automation.

EDF Earliest Deadline First.

FBD Family-Based Design.

FFT Fast Fourier Transform Transform.

FPGA Field-Programmable Gate Array.

FPP Fast Passive Parallel.

GPU Graphics Processing Unit.

HDL Hardware description language.

HLS High-Level Synthesis.

HPC Hardware Performance Counter.

ICAP Internal Configuration Access Port.

iLBC Internet Low Bitrate Codec.

ISR Interrupt Service Routine.

LAN Local Area Network.

LLF Least Laxity First.

LRG Least Recently Granted.

MAPE Mean Absolute Percent Error.

MPSoC Multiprocessor System-on-Chip.

NBTI Negative-Bias Temperature Instability.

NoC Network on Chip.

OOP Object Oriented Programming.

PABX Private Automatic Branch Exchange.

PBD Platform Based Design.

PCAP Processor Configuration Access Port.

PCM Pulse-Code Modulation.

PIO Programmed Input-Output.

PMU Performance Monitoring Unit.

PVT Process, Voltage and Temperature.

QoS Quality of Service.

RAM Random-Access Memory.

RM Rate-Monotonic.

RMI Remote Method Invocation.

RMSD Root-Mean-Square Deviation.

RTL Register-Transfer Level.

RTOS Real-Time Operating System.

RTSNoC Real-Time Star Network-on-Chip.

SIMD Single Instruction, Multiple Data.

SoC System-on-a-Chip.

SRAM Static Random-Access Memory.

TDMA Time Division Multiple Access.

WCET Worst Case-Execution Time.

CONTENTS

1	**INTRODUCTION**	**25**
1.1	GOALS	27
1.2	ACKNOWLEDGEMENT OF PREVIOUS GROUP WORKS	29
1.3	DOCUMENT ORGANIZATION	30
2	**RECONFIGURABLE COMPUTING**	**31**
2.1	FPGAS	31
2.2	RUNTIME SUPPORT FOR RECONFIGURABLE COMPUTING	34
2.3	DYNAMIC REQUIREMENTS ADAPTATION	38
2.4	DISCUSSION	40
3	**DESIGNING RECONFIGURABLE COMPONENTS**	**45**
3.1	ADESD	45
3.2	DECOUPLING INTERFACE AND IMPLEMENTATION	47
3.3	STATE HANDLING	54
3.4	CROSS-DOMAIN INTERACTION	54
3.5	SUMMARY	57
4	**I/O INTERFERENCE MODELING**	**59**
4.1	LATENCY ANALYSIS	59
4.2	BITSTREAM LOAD MODELLING	62
4.3	DISCUSSION	66
5	**SPECULATIVE RECONFIGURATION**	**67**
5.1	ASSUMPTIONS	67
5.2	RECONFIGURATION ISOLATION	68
5.3	I/O TRAFFIC MONITORING	69
5.4	I/O INTERFERENCE MITIGATION	71
5.5	RECONFIGURATION TRIGGERING	73
5.6	COMPONENT LOADING	74
5.7	POWER MANAGEMENT	76
5.8	STATE HANDLING AND REBINDING	78
5.9	DISCUSSION	80
6	**EVALUATION**	**83**
6.1	IMPLEMENTATION	84
6.1.1	**EPOSSoC**	84
6.1.1.1	Real-Time Star Network-on-Chip (RTSNoC)	84
6.1.1.2	CPU nodes	86
6.1.1.3	Rec nodes	86
6.1.2	**EPOS**	87

6.1.3 Unified design of implementations 88
6.1.4 Software/hardware communication 89
6.2 I/O INTERFERENCE DURING FPGA RECONFIGURATION 90
6.2.1 Interconnect contention 92
6.2.2 Memory contention 93
6.3 PABX SOC 95
6.3.1 Reconfiguration policy 97
6.3.2 Resources usage 99
6.3.3 Reconfiguration time 100
6.4 NON-FUNCTIONAL TRADE-OFFS ANALYSIS 102
6.4.1 Fast Fourier Transform 102
6.4.2 Natural Exponential 104
6.4.3 Discussion 106
7 CONCLUSION 107
7.1 LIMITATIONS 108
7.2 FUTURE WORK 109
Bibliography 111

1 INTRODUCTION

Embedded systems are computer systems designed for specific tasks capable of interacting with the environment through sensors and actuators (MARWEDEL, 2006). They are commonly part of larger physical systems ranging from mobile handheld devices to avionics (LEE; SESHIA, 2015). In 2005 the number of embedded systems in use was already bigger than the number of humans on Earth and more than 99 % of the microprocessors produced were used to build embedded systems (POP, 2005).

As the semiconductors industry advances, embedded systems become more complex and integrate a wider range of features implemented using more sophisticated techniques and computing resources. The restrictions in terms of energy consumption, memory usage and processing power become stricter leading to longer design cycles. To shorten the gap between restrictions and requirements, most functionalities in embedded systems are partitioned and synthesized into different cuts of software and hardware through co-design techniques. Common substrates for implementing functionalities include microprocessors, Digital Signal Processors (DSPs), programmable logic devices, Graphics Processing Units (GPUs), or even Application-Specific Integrated Circuits (ASICs) which in many cases are integrated in a single chip as a System-on-a-Chip (SoC). Nevertheless, developing a dedicated SoC architecture tailored for a single application is a laborious and expensive engineering process with a lengthy time to market (BERGAMASCHI; COHN, 2002).

As complexity escalates, strategies focused on simplifying and standardizing the design of embedded systems arise. Designing SoCs by integrating previously validated hardware and software components abstracts complexity and increases reusability in embedded systems. Platform Based Design (PBD) embraces this approach by proposing reusable hardware and software platforms able to comply with the restrictions and requirements of groups of applications (SANGIOVANNI-VINCENTELLI; MARTIN, 2001). As the development cost of a platform is dissolved by the number of applications it targets, platforms must be carefully designed to fit a large number of applications. Moreover, a proper methodology for component selection, configuration, and adaptation to cope with application requirements must be specified for the platform (POLPETA; FRöHLICH, 2005).

Embedded systems requirements evolve during development phase and in many cases continue to evolve in the next phases of its life cycle. While patching the embedded software can cope with the evolution of most application requirements during deployment, there are scenarios where only by upgrading the hardware it is possible to extend embedded system life cycle

reducing further development costs. Telecommunication systems are a clear example in which the constant advancements in standards and protocols can only be followed by the constant evolution of hardware circuitry to keep up with the desired performance.

Moreover, rigid partitions of components or modules in a hardware/software co-design flow can lead to suboptimal choices in systems with dynamic or unpredictable runtime requirements. A partition can be seem as a combination of the micro architectural choices made to fulfill the application requirements and the substrate where it is deployed (e.g. multi-core CPU, hardware accelerator, a remote machine) captured by the component implementation. While there are choices in implementation and variation in quality and costs for each substrate (RAHIMI et al., 2015), choices are typically static (decided at design time) and variations are typically not quantified and exploited, but simply suffered. Reconfiguring the underlying implementation of a component during runtime can help systems cope with dynamic non-functional requirements such as performance and power (LI et al., 2013), hardware defects (MARTINS et al., 2015) due to Negative-Bias Temperature Instability (NBTI) and Process, Voltage and Temperature (PVT) variations, or application requirements unforeseen at design time. A reconfigurable system could for example migrate the implementation of a multimedia encoder from a high quality software version running on a general-purpose core to a dedicated hardware accelerator or to an alternative lower quality version running on a low-power embedded microcontroller according to system load or power consumption.

Despite being idealized in the early 1960s (ESTRIN, 1960), reconfigurable hardware devices, such as Field-Programmable Gate Arrays (FPGAs), only started being commercialized in the mid 1980s. Such devices allow designers to leverage hardware's power efficiency and parallelism while keeping software's flexibility. They offer drastic reduction of power consumption and speedup factors by up to several orders of magnitude compared to using the Von Neumann paradigm (HARTENSTEIN, 2011). Hardware resources usage is raised by the device's ability to multiplex functionalities in time, i.e. different functionalities can be loaded into the device only when they are effectively necessary. FPGA's high performance and low energy consumption to execute groups of algorithms when compared to general purpose microprocessors allow designers to explore trade-offs between components implementations depending on different substrates. Moreover, the flexibility associated to its usage allow embedded systems exploit and adapt to quality and cost variations suffered by each substrate during runtime through reconfiguration.

FPGA reconfiguration allows systems to change portions of a hardware dynamically while other parts of the circuit are still active. There are three main requirements for enabling effective and efficient reconfiguration

in FPGAs: 1) hardware reconfiguration support, 2) application programming interfaces for reconfiguration, and 3) non-intrusive reconfiguration policies. Device manufacturers handle the first requirement: the Xilinx Vivado suite, for example, allows users to change the configuration of part of an FPGA while keeping the rest of the device operational. The runtime support for reconfigurable computing handles the second requirement by providing support to dynamic reconfiguration. A typical runtime support may provide, minimally and in addition to the standard hardware drivers and task scheduling functions, mechanisms for migrating a component between hardware and software implementations, and for managing hybrid software/hardware inter-process and inter-component communication. The final requirement is often left to application programmers, who must monitor any important parameters and decide when and under what circumstances to reconfigure the system.

An operating system designed disregarding hardware reconfiguration fails to provide the needed functionality. Indeed, it will usually disturb the procedure by inadvertently concurring with it. A new task being scheduled causing a context switch at the wrong instant can disrupt FPGA reconfiguration and cause several drawbacks to applications that depend on it. The reconfiguration can also be compromised by background I/O operations. The bitstream used to store the FPGA configuration must be loaded from memory while, for example, a video stream is being moved from the memory to a video decoder. The sharing of hardware resources (interconnects and memory interfaces) by both streams of data results in an increased execution time for both operations due to interconnect scheduling policies and limited bandwidth.

1.1 GOALS

This work proposes a framework whereby reconfiguring the current component implementation is a deterministic process performed without modifying its interface. The reconfiguration is a deterministic process aiming not to disrupt critical system activities. We consider that a reconfigurable component can have multiple implementations with different trade-offs in terms of quality, timeliness, and cost. The framework delivers a tailored wrapper for each component according to the number of implementations it has. Each implementation can use different resources in different substrate (e.g. processor core, FPGA) and better suit the application at given moment according to the embedded system environmental conditions. Reconfigurable components have a single interface such that from the application point of view, components implementation using software or reconfigurable hardware resources can be invoked seamlessly.

The reconfigurable components implementations deployed in the framework are designed following Application-Driven Embedded System Design (ADESD) techniques (FRöHLICH, 2001), a domain engineering methodology. Reconfigurable components are described through unified high-level models (MüCK; FRöHLICH, 2013) that allow the synthesis of both hardware and software implementations. Interaction between components is transparent regardless of implementation, that is, functions provided by a component can be called identically regardless of how that component is instantiated at any point in time. Software adapters implemented as aspect programs abstract component-to-component communication, and helper functions for each component perform state migration when transitioning from software to hardware and vice-versa.

The process of reconfiguring the component implementation is transparent from the application point of view, confined in the system's idle time, and designed not to interfere with critical threads executed by the system. Each thread in our system dynamically creates (and destroys) any components it may need (e.g. a multimedia codec, or a cryptographic accelerator). For components with multiple software/hardware implementations, our system opportunistically monitors system load and performance parameters to check when the operating system can perform a reconfiguration without disrupting timing constraints. The reconfiguration process of each component is divided into small tasklets such that its largest atomic step can typically be performed within available system slack as long as processor utilization is under 100 %.

Despite being temporally isolated from other threads, the reconfiguration can have its timing determinism compromised when interfered by background I/O operations, specially when a new implementation must be reconfigured in an FPGA. A large FPGA bitstream used to store its configuration must be moved from memory to the FPGA reconfiguration interface while, for example, a video stream is being moved from the memory to a video interface. The sharing of hardware resources (interconnects and memory interfaces) by both streams of data results in an increased execution time for both operations due to interconnect scheduling policies and limited memory bandwidth. Our reconfiguration process speculatively monitors the I/O traffic sources to predict when to deploy FPGA reconfiguration without the hazard of being interfered by other peripherals. It is also capable of powering down devices being used by non-critical threads to reduce I/O interference and prioritize the reconfiguration.

This work focus on providing the infrastructure necessary for transparently deploying runtime reconfiguration for computing systems willing to exploit the variations in their environment and adapt to unpredictable application requirements. The reconfiguration process goal is being transparent to

the application and isolated from other tasks performed not to disrupt their timing constraints. Nevertheless, it is out of its scope to propose general guidelines to define when to reconfigure a component or even which is the best implementation for a particular set of system conditions. We believe that those tasks should be delegated to alternate software/hardware layers that posses a holistic understanding of the current system state (SARMA; DUTT, 2014).

1.2 ACKNOWLEDGEMENT OF PREVIOUS GROUP WORKS

This work is built upon the research being conducted by the Software/Hardware Integration Laboratory members in Federal University of Santa Catarina over the last decades supervised by professor Antônio Augusto Fröhlich. Below there is a list of the members followed by their contributions in chronological order.

- Fauze Valério Polpeta: ADESD concepts in the field of embedded systems development (POLPETA, 2006).
- Arliones Stevert Hoeller Junior: Power management interface for embedded systems (JUNIOR, 2007).
- Hugo Marcondes: Hybrid hardware and software components as a development artifact that can be deployed by combinations of hardware and software elements (MARCONDES, 2009).
- Giovani Gracioli: Operating system infrastructure for dynamic software reconfiguration (GRACIOLI, 2009).
- Tiago de Albuquerque Reis: Difference based partial FPGA reconfiguration only deployed after system hibernation started by the power management interface (REIS, 2010).
- Danillo Moura Santos: Hardware independent interfaces for digital signal processing algorithms allowing efficient migration of application between platforms (SANTOS, 2010).
- Tiago Rogério Mück: Unified C++ descriptions of embedded system components (MüCK, 2013).
- Marcelo Daniel Berejuck: RTSNoC (BEREJUCK, 2015).

This works relies on artifacts consolidated on the works above, specially artifacts from the EPOS component framework implemented in collaboration with Tiago Rogério Mück and also on the RTSNoC implemented by Marcelo Daniel Berejuck.

1.3 DOCUMENT ORGANIZATION

Chapter 2 has an overview on hardware technologies used by reconfigurable computing systems as well as a bibliographical review on runtime support systems used to cope with its complexity. Next, in Chapter 3, we present the guidelines used to design component possessing multiple implementations in different computing substrates. Besides presenting techniques to decouple the multiple implementations from the component itself, it exhibits details for cross-domain communication between components and how to represent their internal state.

In Chapter 4 we propose a model to quantify the I/O interference that can be suffered when moving data in a computing system from one point to another data through a shared chip resources such as interconnects and memory controllers. A reconfiguration process capable of isolating itself from the rest of the system as well as mitigating possible I/O interference is described in Chapter 5.

Chapter 6 explores technical aspects of implementing the ideas proposed in Chapter 3 and Chapter 5. It also presents the platform previously designed in the Software/Hardware Integration Laboratory for deploying SoCs on FPGAs. Finally, it quantifies I/O interference during FPGA reconfiguration, explore trade-offs between different component implementations, and to evaluate the proposed reconfiguration process using a Private Automatic Branch Exchange (PABX) as a case study.

2 RECONFIGURABLE COMPUTING

As applications become more complex, embedded systems must integrate even more more functionalities to follow the semiconductors industry pace. Most applications are statically partitioned in software and hardware components, constantly consuming system resources despite not being utilized during the whole application execution time. Reconfigurable computing allies software flexibility with hardware performance and power consumption allowing to increase the hardware resources usage during the application life span. Using reconfigurable devices, algorithms could be rapidly prototyped in hardware offering better performance compared to its software counterparts. With dynamic reconfiguration capabilities, designers are able to switch the algorithm implemented in the reconfigurable device during runtime, allowing different applications to be accelerated during the system execution.

This chapter initially presents an overview of reconfigurable hardware devices, software techniques, and runtime systems used for deploying reconfigurable computing systems in Section 2.1 and Section 2.2. Next, it presents works that address system adaptation due to unpredictable runtime variations in embedded systems in Section 2.3. Finally, it compares the related works with this work's proposal in Section 2.4.

2.1 FPGAS

FPGAs are integrated circuits composed of reconfigurable logic blocks and hierarchical reconfigurable interconnects. Figure 1 shows a generic FPGA architecture comprising its main building blocks. Reconfigurable logic blocks can implement combinational logic such as boolean functions (e.g. AND, OR) and, if it contains memory elements, sequential logic using flip-flops. The device functionality will be defined by the configuration of the logic blocks and its wiring, performed by the FPGA interconnect. Modern FPGAs employ not only reconfigurable elements but also hard blocks that implement dedicated functions wired to the rest of the system by the interconnect without consuming logic resources. Compared to having the same functionality implemented by logic blocks, hard blocks occupy a smaller FPGA area and have performance and power consumption comparable to an ASIC. Multipliers, digital signal processing blocks, embedded processors, transceivers, external memory controllers and Static Random-Access Memory (SRAM) are commonly found as hard blocks in FPGAs. A detailed survey on hardware aspects of reconfigurable devices is presented by Compton and Hauck (COMPTON;

HAUCK, 2002).

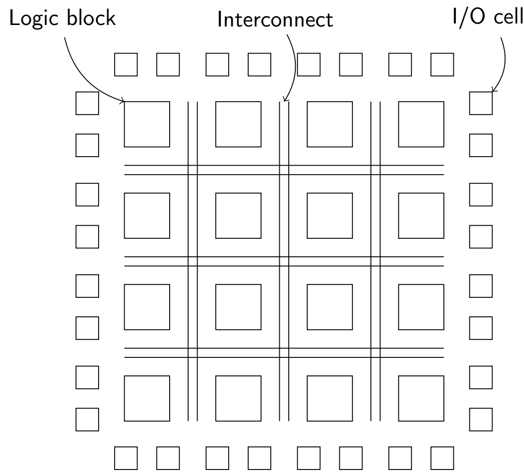


Figure 1 – Generic FPGA architecture.

FPGA development flow typically starts by the implementation of the desired functionality in Register-Transfer Level (RTL) using a Hardware description language (HDL). The following development phases are performed by Electronic Design Automation (EDA) tools provided by the FPGA vendor. Initially, the RTL description is synthesized into a netlist that contains the connectivity between the circuit elements that implement the functionality. The next phase, place and route takes as input the generated netlist and the target FPGA characteristics. It first places the circuit elements into specific hard and logic blocks available in the FPGA and then routes the wires that connect all logic blocks. Finally, an FPGA configuration binary file known as bitstream is generated and can be loaded into the FPGA through one if its programming interfaces.

Describing and verifying a complex algorithm in RTL using an HDL can be time consuming and error prone. Moreover, refining the algorithm micro-architecture to meet timing and power constraints requires several design iterations and consume project resources. To minimize the time spent in the initial design phase, major EDA and FPGA vendors are investing heavily in High-Level Synthesis (HLS) tools. HLS tools are able to generate an RTL description of a functionality from its algorithmic description. Not only design but also verification, which usually occupies a large slice of the development process, can benefit from HLS. Specifying the problem using a higher-level language allows designers isolate algorithm behavior from lower level RTL

concerns such as timing. Micro-architectural exploration of the algorithm is normally supported as a set of directives which can affect area, timing, parallelism of the synthesized RTL description. C and C++ are the high-level languages of choice by different HLS vendors such as Calypto and Xilinx.

To cope with higher integration, lower power consumption, and higher communication bandwidth between processors and FPGAs, vendors are proposing new devices called SoC FPGAs. SoC FPGAs integrate both hard CPU subsystems, a rich set of hard IP cores, high speed transceivers and FPGA fabrics into a single chip. For instance, Xilinx has its own line of SoC FPGAs called Zynq-7000 (XILINX, 2015d), a device that couples a dual-core ARM Cortex-A9 processor with a full-fledged FPGA. The processor comes with a set of hard peripherals such as Double Data Rate (DDR) memory controller, two gigabit Ethernet controllers, two 12 bit Analog-to-Digital Converters (ADCs), and several other peripherals. CPU-FPGA communication is performed though AMBA AXI3 ports including a cache coherent port that provides coherent low-latency access to the CPU subsystem memory space. Not only Xilinx but also Altera and Microsemi have a commercially available line of SoC FPGAs named Altera SoC, and SmartFusion SoC FPGA respectively.

Partial reconfiguration is the process of modifying reconfigurable hardware circuitry by downloading partial bitstreams while the remaining system continues to operate normally. For example, a video codec can be instantiated in the FPGA on demand when a given video standard is requested by the application without disrupting the operation of the other modules in the video processing chain. It allows designers to implement complex applications in smaller FPGAs by reconfiguring its resources during execution time, reducing idle power consumption and improving system flexibility. With partial reconfiguration, designers can make efficient use of silicon by only loading into the FPGA the necessary functionality at any point in time. Modern FPGAs families such as the Xilinx 7 Series (XILINX, 2015a) and Zynq-7000 support partial reconfiguration. Altera's FPGA reconfiguration interface, Fast Passive Parallel (FPP) (ALTERA, 2008), is quite similar to to Xilinx's except that it is only accessible from outside the FPGA.

As FPGA reconfiguration moves a significant amount of data (the binary file that contains the FPGA configuration) from memory to the FPGA reconfiguration interface, it is susceptible to being disrupted by other data flows occurring simultaneously in the chip (PELLIZZONI; CACCAMO, 2010). Dynamically reconfiguring an FPGA on critical systems is subject to non-deterministic contention time due to sharing chip resources as the interconnect and memory. On a broader perspective, a non-critical task performing I/O using a chip peripheral can interfere with the reconfiguration process, a fact that must be taken into account to guarantee a deterministic timing operation

of the system. Not only the system tasks interfere with the reconfiguration of and FPGA but the opposite also occurs, the reconfiguration can impact of the execution time of critical tasks that depend on I/O data for completion.

2.2 RUNTIME SUPPORT FOR RECONFIGURABLE COMPUTING

Several efforts focus on developing reconfigurable computing environments that can help developers to leverage FPGA's resources and simplify its development flow. Such computing environments provide abstractions and interfaces to application programmers familiar with systems software development. FPGA's intrinsic parallel capabilities are deeply explored by different works that propose using reconfigurable fabrics as a substrate for a multitasking environment (WALDER; PLATZNER, 2003) in which tasks are dispatched to CPUs or to reconfigurable devices (AHMADINIA et al., 2004).

The Erlangen Slot Machine (AL., 2005) is a reconfigurable computing platform based on uniform resource allocation for each reconfigurable module. It provides a solid infrastructure for reconfigurable computing allowing the dynamic reconfiguration of its slots. One advantage of this platform is that each module can access the FPGA's input and outputs pins independent from its location through a programmable crossbar, allowing an unrestricted relocation of modules on the device. Furthermore, the Erlangen Slot Machine has four intermodule communication structures: adjacent communication, shared memory, Reconfigurable Multiple Bus and crossbar.

A higher-level approach to reconfigurable computing is proposed by Abel that advocates describing reconfigurable modules in a Java-like language called Parallel Object Language (ABEL, 2010). Though partial reconfiguration techniques, reconfigurable modules can be created and destroyed at runtime just like regular software objects. The modules can be tested and verified using a Java Emulator and then translated to VHDL for synthesis. Communication architecture is centered around FIFOs and multiplexers for parallel data exchange controlled by a software scheduler. The scheduler also orchestrates reconfigurable modules loading into partially reconfigurable regions.

The SPREAD programming model (AL., 2013b) is a proposal for reconfigurable computing that focuses on high throughput point-to-point streaming applications. It presents a common software/hardware thread interface and unlike the other solutions, in SPREAD a thread can be set as reconfigurable and thus switch domain during runtime. The common reconfigurable thread interface is guaranteed by a stub thread that creates a wrapper around the thread's software and hardware implementations. After receiving a command to switch domain, the stub thread saves the thread context consisting of the pro-

cessed samples and snapshot registers and transfers them to the other domain. SPREAD allows switching threads domain during runtime but the decision of switching is delegated to the application programmer, and it is not clear if thread migration can deal with timing constraints. Figure 2 presents the mapping of two streaming computations with software, hardware, and switchable threads in the corresponding computing resources (e.g. CPU, FPGA).

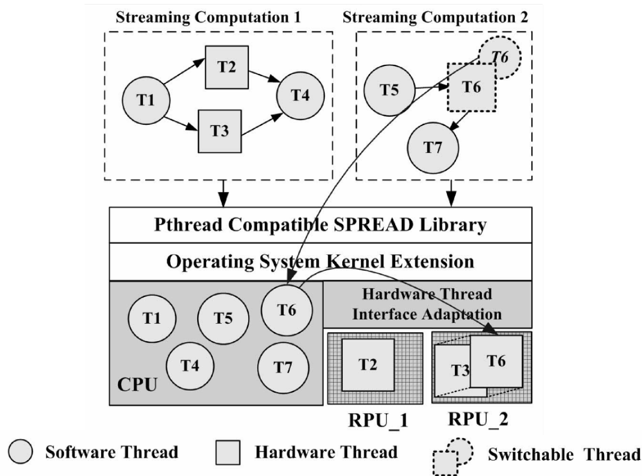


Figure 2 – SPREAD programming model (AL., 2013b).

Moreover, Cemin et al. propose a framework for multi-agent systems that supports hardware/software migration among different nodes of a distributed system (DYNAMICALLY. . . , 2014). Their framework allows the dynamic reconfiguration of distributed application to cope with variable system requirements. The communication between agents is handled transparently by proxies that handle all the low-level intricacies of hardware communication. Their approach is implemented on top of the Jade framework, a Java-based multi-agent systems framework.

A technique to reduce the number of reconfigurable resources used by critical applications described as directed acyclic graphs is proposed by Clemente et al (CLEMENTE; RESANO; MOZOS, 2014). Starting with a task set that meets its timing constraints, their objective is hiding the delays due to dynamic reconfiguration of hardware tasks while minimizing the number of reconfigurable units. They modify the original scheduling so that tasks introducing a delay due to their reconfiguration can be replaced by other

tasks as long as they are loaded back again before the end of the task graph execution.

Several works extend commodity operating systems using them as a back-end for reconfigurable computing. Such approach aims at reducing the development time of porting legacy applications to reconfigurable platforms. BORPH (AL., 2006) and FUSE (ISMAIL; SHANNON, 2011), depicted in Figure 3, extend the Linux kernel providing native support for FPGAs, treating them as computational resources instead of coprocessors. BORPH offers a homogeneous UNIX interface for software and hardware processes including ioreg and file I/O. In BORPH, FPGA designs are encapsulated in a new file format called BORPH Object File, which start the corresponding hardware process when executed. One of BORPH's limitations is that the number of executing hardware processes is limited by the number of FPGAs in the system as each hardware process occupies the whole FPGA. Unlike BORPH, FUSE partitions the FPGA in slots, thus one FPGA can contain multiple hardware threads. Both operating systems are not suited for critical applications due to the inherent non deterministic timing behavior of Linux. Figure 3 shows different layers in the FUSE with software tasks on the top and hardware accelerator logic on the bottom interacting with the FUSE Application Program Interface (API).

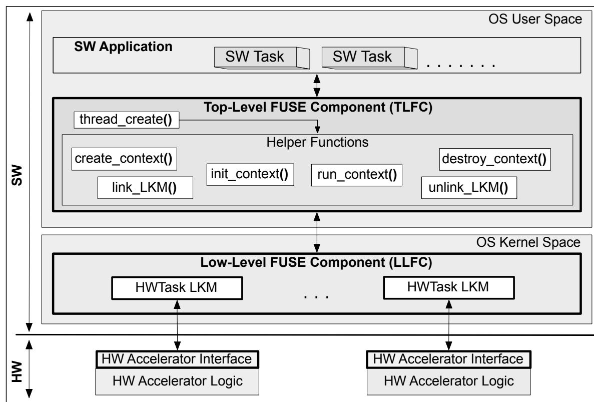


Figure 3 – FUSE system architecture (ISMAIL; SHANNON, 2011).

ReconOS (LUBBERS; PLATZNER, 2009) provides a common abstraction layer for software and hardware threads as well as a set of communication and synchronisation primitives for them. Low-level synchronisation and communication of hardware threads are managed by theirs Operating System Interface, an artifact embedded in hardware threads. It focuses on applications

with strict timing constraints and multithreading providing to the user an API based on the eCos operating system (LIMITED, 2016) as well as on Linux. Despite its rich set of features, thread reconfiguration is managed by the user in execution time. Moreover, when using eCos scheduler, hardware and software threads present cope with critical timing requirements but it is not clear if thread migration can do the same.

Andrews et al. propose HybridThreads (AL., 2008), an operating system written from scratch that allows the execution of software threads on a CPU and hardware threads on an FPGA simultaneously. Most synchronisation services, such as mutexes and the operating system scheduler, are implemented in hardware to boost system performance. The hardware threads are coupled with a Hardware Thread Interface, a component that provides hardware/-software interaction through POSIX threads and access to synchronisation mechanisms. This approach does not manage the dynamic reconfiguration of the FPGA; threads are allocated to the FPGA in design time and are not able to be switched by other hardware implementation or migrated to software.

Another operating system that supports reconfigurable computing is R3TOS (AL., 2013a). The system focuses on reliable computing and provides support for hardware threads in task sets with critical timing constraints. It differs from previous approaches in its capability of allocating hardware tasks in any FPGA region and not only reconfigurable slots defined in design time. R3TOS provides hardware tasks with virtual channels; the FPGA's Internal Configuration Access Port (ICAP) is used to transfer data between hardware threads. In R3TOS, the reconfiguration capabilities are also used to circumvent damaged FPGA resources, increasing system reliability against faults. R3TOS manages the reconfiguration of stateless hardware threads as single operation carried prior to thread execution.

CAP-OS was conceived to exploit reconfiguration in heterogeneous Multiprocessor System-on-Chips (MPSoCs) implemented in FPGAs (AL., 2011). Computations are performed either by software threads in a processor or by hardware threads in a processor coupled with a hardware accelerator handling compute-intensive tasks, and communication is based on Message Passing Interface. CAP-OS uses a preemptive priority-based scheduling algorithm divided into a static list scheduling and a dynamic scheduling step capable of scheduling the task reconfiguration during runtime. The scheduling algorithm considers resource constraints such as ICAP exclusiveness and intertask dependencies. Nevertheless, the prototype only supports reconfiguring processor's program memory. The capability to reconfigure hardware resources, as well as the number of processors in the system, is not supported.

2.3 DYNAMIC REQUIREMENTS ADAPTATION

Reconfigurable computing can be seem as a mechanism for system adaptation used to cope with dynamic application requirements. For instance, the dynamic choice of a given hardware implementation for certain system components (e.g., leading to different trade-offs between quality and resource usage) and the change in non-functional characteristics of such component (e.g., speed and energy consumption) are scenarios that can be explored using reconfigurable computing techniques.

One broad approach for coping with unpredictable runtime variations in performance, energy, and user behavior in embedded software is to have multiple code paths available to perform certain critical functions. Each of these paths may be better suited for a given system state or user input. Petabricks (ANSEL et al., 2009) and Eon (SORBER et al., 2007), for example, feature language extensions that allow programmers to provide alternate code paths. In Green (BAEK; CHILIMBI, 2010), a combination of a calibration phase and runtime accuracy sampling are used by the application to define which function to execute from a set of possible candidates. In Eon and Levels (LACHENMANN et al., 2007) the runtime system dynamically chooses paths based on energy availability. Compared to traditional algorithmic choice research, and in particular to the ViRUS framework (WANNER; SRIVASTAVA, 2014) that shares our motivations for dynamic handling of variability events, reconfigurable computing adds a dimension of hardware reconfiguration, whereby one of the code paths may be offloaded to specialized hardware. The ViRUS framework allows switching between function implementations that perform equivalent functionality with different Quality of Service (QoS) levels when the system is under stress with the ultimate goal of energy efficiency. According to a policy specified by the user and data collected from registered sensor, ViRUS choses the fittest implementation according to the environmental conditions. As presented in Figure 4, ViRUS stress daemon monitors system sensors and raises alarms for the monitor library to switch the underlying implementation of functions according to the enforced policy given by the configuration file.

The code used to implement a given function may also be changed at the binary or bytecode level. Dynamic recompilation techniques (VOSS; EIGEMANN, 2001) test different optimization techniques at runtime, so that code is matched to the capabilities of the hardware which is running it. Dynamic recompilation may be performed in a system-driven manner, with minimal support from applications, providing the same adaptation knobs as in compile time optimization, e.g., loop unrolling, memory optimization, and automatic parallelization. Our work resembles dynamic recompilation in that

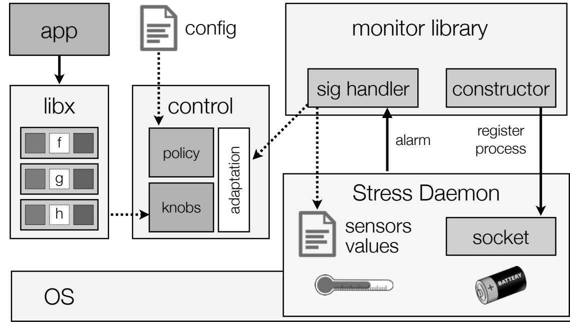


Figure 4 – The ViRUS framework (WANNER; SRIVASTAVA, 2014).

the multiple code paths are explored automatically by the runtime system without application intervention, but it is closer to algorithmic choice in that the alternative implementations are defined at design time. In a similar fashion to dynamic recompilation, reconfigurable computing can be used to explore multiple code paths automatically by the runtime system without application intervention if alternative implementations are defined at design time just like with algorithmic choice.

Application and hardware parameters can also be dynamically adapted to explore energy, quality, and performance trade-offs (HOFFMANN et al., 2011). Dynamic voltage and frequency scaling is the canonical example for hardware tuning. In software, Green (BAEK; CHILIMBI, 2010) provides an adaptation modality where the programmer provides “breakable” loops and a function to evaluate QoS for a given number of iterations. The system uses a calibration phase to make approximation decisions based on the quality of service requirements specified by the programmer. At runtime, the system periodically monitors quality of service and adapts the approximation decisions as needed. In the mobile context, Powerleash (FALAKI, 2012) is used to adapt the work of background tasks according to a desired rate of energy consumption. Adaptable software parameters can be used to maximally leverage the underlying hardware platform in presence of variations, for example in multimedia applications (PANT; GUPTA; SCHAAR, 2012). Finally, self-awareness can be extended to SoCs platforms coupled with cross-layer sensing and actuation (SARMA; DUTT, 2014). The general mechanism of observe-and-adapt from parametric control can be used in reconfigurable systems to add a dimension of choice in software and hardware components that can be dynamically chosen to implement a given functionality.

2.4 DISCUSSION

There is a large body of work in dynamic hardware reconfiguration, and particularly in leveraging reconfigurable resources in modern FPGA platforms. The reconfigurable computing environments found in the literature provide fundamental services for applications to explore FPGAs' functionalities such as hardware task loading and scheduling, FPGA resource allocation and bi-directional software/hardware communication without dealing with lower level hardware details. Runtime systems such as BORPH, and ReconOS provide drivers and operating system support (e.g. file interfaces) for reconfigurable resources, but typically leave the management of components migration and reconfiguration up to application software. Closer to our work is the SPREAD programming model whereby threads can migrate to specialized hardware resources and back to general purpose processors at runtime. Those works provide solid infrastructure for speeding up applications by exploring the inherent parallelism of reconfigurable hardware platforms. Nevertheless, they lack a unified interface from the application point of view when interacting with artifacts that can be deployed in hardware or in software. For instance, the application must be aware of the substrate where a given component (thread or hardware accelerator) is currently deployed to use the correct interface for it. Any sort of runtime adaptation of an component's underlying implementation must be performed by the application in a platform dependent fashion.

A subset of related works propose a unified interface for components mapped to hardware or software. For example, FUSE supports hardware and software threads accessible through a uniform interface despite not being able to switch their domains during runtime. In the framework proposed by Cemin et. al., components have a unified Java interface but are not statically partitioned as software or hardware and are able to migrate from one domain to the other. Such works decouple interface from implementation no matter in which substrate a functionality is deployed but do not address the isolation of the reconfiguration process from critical system tasks. Critical tasks can have severe timing requirements and without properly isolating them from non-critical ones, such as the reconfiguration, one might interfere with the timing characteristics of the other.

Several works target systems with critical timing requirements but do not explore the issues related to performing hardware reconfiguration in platforms that subjects the data intensive reconfiguration operation to I/O interference. They support reconfiguration during runtime but keeping its execution time deterministic by taking into account traffic from other subsystems flowing through the chip does not seem a concern. Some works rely on of-the-shelf operating systems as Linux which do not guarantee a deterministic timing

behaviour for reconfiguration due to background services which are hard to account for during application design.

In most related works, the reconfiguration process is a monolithic operation triggered by the application programmer and cannot be interrupted. Configuring a coarse-grained hardware threads or hardware accelerators is an operation that can take a considerable amount of time to finish depending on the size of the bitstream or the component's state. Unlike other works that treat hardware/software reconfiguration as a rigid operation, we split it into smaller steps executed while the system is idle. Even the bitstream loading process can be split in smaller chunks of data in order to fit partial reconfiguration in the system's idle time. Hence, even with small available idle time, reconfiguration is temporally confined and carried without disrupting timing requirements of critical systems tasks. This punctual approach allows on-demand switching between implementations of embedded system components during runtime in a transparent fashion.

In this context, we observe the need to provide a transparent interface for the underlying implementation (software or hardware) of a component with a well-behaved reconfiguration process. Such infrastructure provides the necessary services for reconfiguring a component implementation to address dynamic application requirements without modifying the application. Therefore, by reconfiguring the current component implementation the system can adapt to its environment and cope with dynamic non-functional requirements, hardware defects, or constraints unforeseen at design time. Moreover, the reconfiguration must be self-contained, temporally isolated and aware that it can move a large amount of data that might interfere and be interfered with background I/O traffic disrupting critical system activities.

Table 1 presents a comparison between this work's proposal with related ones that present reconfiguration support and a well-defined interface to handle reconfigurable resources. The table compares the abstractions used to handle FPGA resources stating if they are reconfigurable, i.e., the underlying implementation can be reconfigured or migrate to another computing substrate (e.g. CPU). Notice that some works have FPGA reconfiguration support to load abstractions to the FPGA prior to its execution but it does not mean that the abstractions are reconfigurable as they execute always in the same substrate. The usage API dictates how the application interacts with hardware and software abstractions being unified when from the application point of view there is no differentiation between an abstraction deployed in hardware or in software.

We provide similar mechanisms based on the state of the art pushed by previous efforts but differs from them in key aspects. Initially, the syntax and semantics of the component interface in our system are preserved across the

multiple implementations, such that an application sees no difference (other than changes in quality and cost associated with the component usage) when implementations are changed. Moreover, our reconfiguration process leverages the system's idle time to perform component implementation reconfiguration. By delegating reconfiguration to lower-level software layers that can reason on the system's current state, the reconfiguration process becomes transparent from the application point of view as the application programmer does not have to be aware that the reconfiguration is happening. Reconfiguration is divided into smaller steps and part of it occurs even when the available idle time is not enough to accommodate the full implementation reconfiguration. By performing reconfiguration only when the system is idle we can temporally isolate other critical operations from it and mitigate the inflicted interference. The reconfiguration process time takes extra care for I/O interference from external system components in a speculative fashion by monitoring system execution.

Table 1 – Comparison of related works that provide support for using reconfigurable resources present in FPGAs.

	Abstraction		FPGA reconfiguration process			Usage API		
	Granularity	Reconfigurable	Timing	Fragmentation	Interference aware	Unified	Hardware	Software
Erlangen Slot Machine	Task	No	Prior to task execution	Single step	No	No	Custom	N/A
BORPH	Process	No	N/A	N/A	N/A	No	Linux file I/O	N/A
HybridThreads	Thread	No	N/A	N/A	N/A	Yes	Custom	Custom
ReconOS	Thread	No	N/A	N/A	N/A	No	Custom	Pthreads/eCos
FUSE	Task	No	N/A	N/A	N/A	Yes	Custom	Custom
CAP-OS	Task	No	Prior to task execution	Single step	No	No	Xilkernel	Xilkernel
SPREAD	Thread	Yes	Prior to thread execution	Single step	No	No	Custom	Pthreads
R3TOS	Task	No	Prior to task execution	Single step	No	Yes	Custom	Custom
Cemin et al.	Agent	Yes	During agent runtime	Single step	No	Yes	JADE	JADE
This work	Component	Yes	During idle time	Multiple steps	Yes	Yes	EPOS	EPOS

3 DESIGNING RECONFIGURABLE COMPONENTS

This chapters recalls the main concepts behind ADESD and previous group works necessary to design reconfigurable components. It establishes a base and introduce guidelines necessary for the following chapters by initially presenting an overview on ADESD, an embedded systems project technique on which this work relies for different aspects. Next, it discusses how to decouple multiple implementations from the component interface, handle the component state in different implementations and finally on how components in different domains can communicate. Reconfigurable components allow for dynamic change between multiple software and hardware implementation choices, as well as adjustments in operation parameters for each of the choices. The resulting QoS and cost for each component depends on the specific choice of implementation as well as the physical state of the system at any point in time. Parts of this chapter have been previously published in the RSP'15 paper *X-Ware: Mutant Computing Substrates* (REIS; WANNER; FRöHLICH, 2015)

3.1 ADESD

For developing hardware and software implementations we rely on ADESD which is a multi-paradigm domain engineering methodology to decompose a problem domain into reusable and scenario independent abstractions while capturing scenario related variations into a set of aspect programs (FRöHLICH, 2001). Application domain entities are decomposed into abstractions by employing object-orientation and Family-Based Design (FBD) techniques. Abstractions variations that belong to its essence are separated from those that emanate from execution scenarios, the former shaping family members, and the latter yielding scenario aspects. Each family of abstractions exports an inflated interface that implements all family members functionalities, thus from the application point of view, each family is a single entity. The assignment of the most appropriate family member in each application can be postponed by the application developer or even performed by an automatic configuration tool (CANCIAN, 2011).

Factoring the different execution scenarios for each abstraction into aspects allows ADESD to improve reusability over traditional object-orientation decomposition and avoid combinatorial explosion of the family members number. For example, instead of modeling a new member for a family of communication mechanisms that is able to operate in the presence of multiple threads, concurrency can be modelled as a scenario aspect that blocks

the communication in critical sections. A scenario can incorporate a set of aspects and weave them to a target component is performed by an artifact called scenario adapter by mediating the interaction between the component and its clients. Moreover, the relation between components and aspects in each scenario enforced by scenario adapters is captured by a framework that establishes how component are interconnected.

The whole domain decomposition process is presented in Figure 5. Entities from the problem domain are grouped into families of abstractions exporting inflated interfaces for their clients. The scenario variability is captured in a set of aspects while slight modifications of component behavior or structure (e.g. thread stack size) are allowed using reconfigurable features.

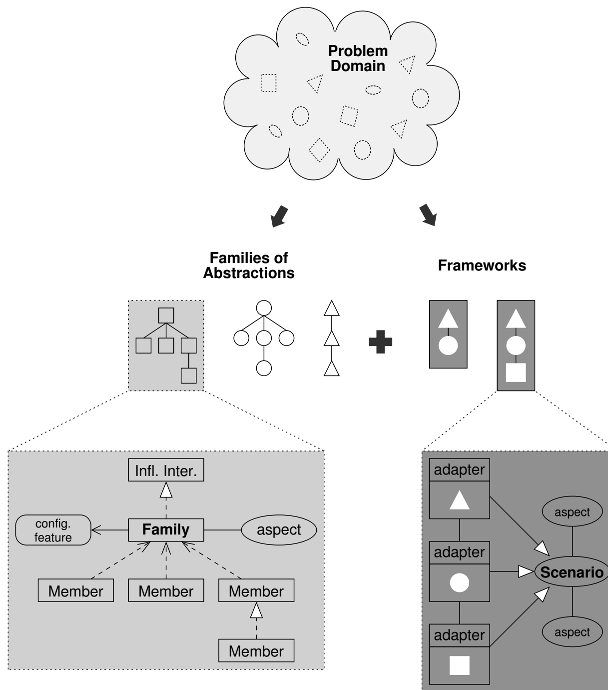


Figure 5 – ADESD domain decomposition (FRöHLICH, 2001).

Despite being initially proposed as a software engineering methodology used in the context of dedicated operating systems, ADESD concepts has been widely applied in the field of embedded systems development (POLPETA; FRöHLICH, 2005). ADESD was also used to explore common implementation and communication interface for components deployed in the software and

hardware domains by Marcondes (MARCONDES; FRöHLICH, 2009). This approach, called *Hybrid Components*, allows freely migrating the component from one domain to another in any phase of the system design process without needing to adjust the system itself. Nevertheless, the developer still needed to design separate hardware and software implementations. To overcome this limitation, Mück provides a technique for developing unified descriptions of hardware and software components (MüCK; FRöHLICH, 2013) based on Aspect Oriented Programming (AOP) and Object Oriented Programming (OOP) techniques. Components designed following such principles are susceptible to both software and hardware synthesis using standard compilers and HLS tools. This is possible through the isolation of specific hardware and software characteristics (resource allocation and communication interface) into aspect programs which are weaved with the unified descriptions only during the synthesis process. Therefore, the extraction of hardware/software implementation from the unified implementation happens directly through language-level transformations, thus facilitating compatibility with different C++-based HLS tools and design flows. For example, while Marcondes was able to keep hardware and software implementations of a scheduler and switch between them, Mück generated its hardware and software implementations from the scheduler description.

Reconfiguration was also previously studied leveraging ADESD concepts. ELUS provides the infrastructure for dynamic software reconfiguration in resource-constrained embedded systems (GRACIOLI; FRöHLICH, 2010). Despite not dealing with hardware reconfiguration, ELUS provides a configurable and transparent software reconfiguration mechanism without incurring overhead for non-reconfigurable components. Difference-based partial reconfiguration was also proposed as a viable alternative for SoC development if its bitstream generation unpredictability can be avoided (REIS; FRöHLICH, 2009). As hardware reconfiguration impacts the software running over it, the work also proposes a software reconfiguration interface that allows the application to inform the operating system about the hardware co-processors entering and leaving the system. This way, a partially reconfigurable SoC can be designed without inter-module communication structuring such as buses or interconnection networks on the FPGA, simplifying the design, saving design space and making it model-independent.

3.2 DECOUPLING INTERFACE AND IMPLEMENTATION

By using ADESD concepts we provide an approach for deploying reconfigurable components employing static metaprogramming techniques and

a transparent interface from the application developer point of view. Multiple implementations of a reconfigurable component are exposed to applications through a single, unified interface (which in our approach is called inflated interface (FRöHLICH, 2001)). Metaprogrammed software artifacts are used to handle differing function signatures, communication with hardware IP cores, and state migration between implementations. While all software implementations and IP cores bitstreams are included in the system image (typically kept in Random-Access Memory (RAM)), hardware accelerators are instantiated and freed on-demand, and therefore do not take up system resources (particularly FPGA area) when not in use. Per-application constraints are used to guide the reconfiguration process and dictate which version of a component should be used under different circumstances. From the application's standpoint, a call to a double precision exponential function is therefore identical to a call activating a hardware IP to calculate it or a remote call to a cloud resource.

Our approach assumes that applications dynamically instantiate (and destroy) the components they need, and the operating system automatically uses the component implementation that best suits its current needs. Multiple component implementations are available to applications through a single interface. For the application, a call to a software function is, hence, identical to employing a hardware IP core or a remote call to a cloud resource. The application programmer selects its preferred implementation of a given component for each application but, to cope with system load as well as environmental and manufacturing-related variations, the deployed implementation might change during runtime.

We divided implementations in three groups: software, hardware, and remote. Software implementations run in soft and hard core processors where the operating system is also being executed. Such implementations might use not only the CPU Arithmetic Logic Unit (ALU) but also tightly-coupled coprocessors such as floating-point units and Single Instruction, Multiple Data (SIMD) engines through special instruction sets. Hardware implementations can vary in microarchitecture details that result in different balances between power consumption, performance, and resource usage. They can be implemented as dedicated circuits in ASICs or as reconfigurable modules in FPGAs. In both cases, the CPU executing the operating system interacts with them through I/O mechanisms such as Network on Chip (NoC) communication, Direct Memory Access (DMA), or memory-mapped I/O. The last group, remote implementations, comprises software and hardware implementations that are not being deployed on the same machine as the application. The implementation is accessed through a network device, which will forward the call to a remote machine and fetch the results.

The C++ programming language is used to illustrate the usage of recon-

figurable components, the same concepts could be applied to any programming language supporting object-orientation and static metaprogramming techniques. C++ classes abstract components, each component has a parametrized class whose static constant members describe the properties (traits) of a certain type. The only additional information the user must provide to the operating system is which component implementations suit best each application in order of preference through the component's traits. In Figure 6, the parameterized class Traits<Component> denotes the traits of Component for a given application in which the implementation order of preference is Implementation_2, Implementation_0, Implementation_1. With this information, the system can adapt to process and environmental variations during runtime by using implementations that best suit system's requirements while trying to utilize the user's preferred implementations.

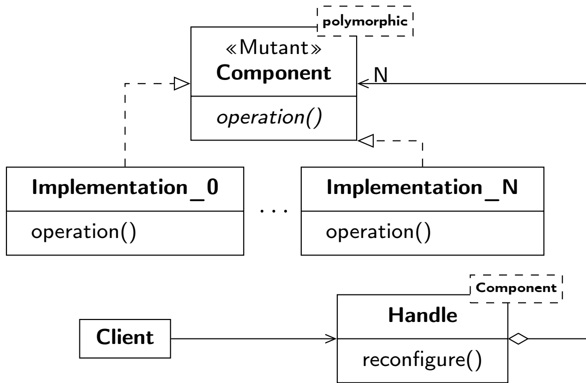
```
template<> struct Traits<Component>
{
    typedef LIST<Implementation_2, Implementation_0, Implementation_1> IMPS;
};
```

Figure 6 – Example of the parametrized traits class for a generic component.

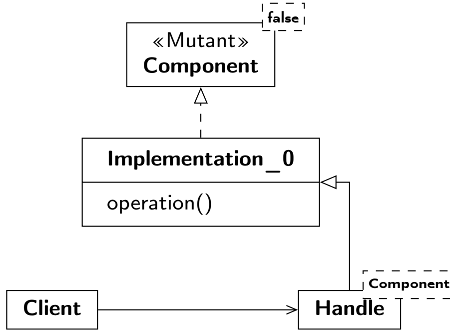
Template metaprogramming techniques enable the user to assign a priority to each component implementation in each application without inferring runtime overhead using the traits mechanism. LIST type is a metaprogrammed list of types populated by the user containing the preferred implementations of Component. Through C++ template metaprogramming, it is possible to obtain the number of types in the list and also a type stored under a given index during compilation without inferring runtime overhead.

Independent of the implementation deployed in a given moment in time, the component must provide a uniform interface to its clients. Also, changing the implementation behind a given component must be a transparent operation and a clean interface must be provided to the entity in charge of performing component reconfiguration. To comply with such requirements, this work proposes an indirection between the component interface and the implementation being currently deployed. Moreover, any infrastructure deployed for this purpose must not incur any overhead when the component has a single implementation. Polymorphism was employed to guarantee that the infrastructure remains scalable independent of the number of implementations and static metaprogramming techniques assure that polymorphism is deployed only when strictly necessary.

Figure 7a presents the building blocks of the infrastructure proposed for reconfigurable components. A parametrized class named Handle is introduced



(a) Component binding to a Handle with N implementations.



(b) Component binding to a Handle with a single implementation.

Figure 7 – Reconfigurable component framework.

between the reconfigurable component interface provided to its clients and the component implementations. It defines a first component wrapper whose main purpose is to ensure the usage of a proper allocator for components, independently of how they are statically or dynamically declared by the client. `Handle` realizes the interface of the component passed as a parameter and forwards invocations of its methods to the implementation deployed in a given moment. It can point to any of its N implementations while still maintaining references to the other implementations for later usage. It also provides the `reconfigure()` method as means of changing its component current implementation during runtime. Hence, reconfigurable components are manipulated through their `Handles`.

`Handle` keeps a reference to each deployed implementation by means of a `Base` pointers array, `_implementations`, as presented in Figure 8. `_current` is a pointer to the current implementation and is used by the `Handle` to dispatch invocations to the component's methods. The type `Base` is defined using the `IF` metaprogram according to the value of `Polymorphic` which is also a metaprogram that defines a boolean after searching the list of implementations for any implementation of a type different then that at the list's head. `IF` returns its second parameter, the `Component::Base<true>` class, if its first parameter evaluates as `true` else, it returns the third one, the `Component` single implementation.

```
template<typename Component>
class Handle
{
private:
    typedef typename Traits<Component>::IMPS IMPS;
    typedef typename IF<IMPS::Polymorphic, typename Component::template Base<true>,
            typename IMPS::template Get<0>::Result>::Result Base;

    static const unsigned int UNITS = IMPS::Length;

public:
    void operation() { _current->operation(); }

    void reconfigure(int i) {
        if (i < UNITS)
            _current = _implementations[i];
    }

private:
    Base * _current;
    Base * _implementations[UNITS];
};
```

Figure 8 – `Handle` implementing method forwarding and reconfiguration.

Notice that `Component::Base<true>` is the base class inherited by

each of the component's implementations. `Component::Base<true>` methods are pure virtual to comply with the fact that `_current` might point to multiple implementations. This allows `_current` to point to any implementation and invoke their methods. The `reconfigure()` method can be easily implemented by changing the element to which `_current` points to.

If `Polymorphic` is false, `Base` will be the implementation itself resulting in the architecture presented in Figure 7b. In this case, the methods will only be defined and implemented in the component's sole implementation with the `Base` inheriting from the implementation itself. The differentiation is necessary to remove the virtual function call overhead for a single implementation. The `Component::Base` behavior can be achieved with template specialization techniques depicted in Figure 9. By specializing the template `Component::Base` according to value of `polymorphic`, we can have a polymorphic base class with pure virtual methods or an empty class. Each `Component` implementation will choose from which to inherit based on the number of implementations.

```
class Component
{
public:
    template<bool polymorphic>
    class Base;
};

// Polymorphic Base
template<bool polymorphic = true>
class Component::Base: public Component
{
public:
    Base() {}
    virtual ~Base() {}

    virtual void m() = 0;
};

// Monomorphic Base
template<>
class Component::Base<false>: public Component
{
public:
    Base() {}
};
```

Figure 9 – Possible `Component::Base` implementation.

A minimal implementation of `Handle's` constructor is shown in Figure 10. The template parameter `polymorphic` is set to `true` if the number of implementations is bigger than 1 by checking the list's `Length`. `Handle's` constructor initializes each implementation and insert them in

_implementations by employing template recursion as a looping construct (CZARNECKI; EISENECKER, 2000). The first invocation of helper() initializes the first metaprogrammed list element defined in the component's traits as previously exemplified in Figure 6 and keeps a reference for it in the Base pointers array. helper() initializes each element until it reaches the last element in the list, with an index given by Traits<Component>::IMPS::Length.

```
// Type ENNUMERATOR
template<unsigned int N>
struct Index { enum { Result = N }; };

template<typename Component>
class Handle
{
private:
    typedef typename Traits<Component>::IMPS IMPS;
    typedef typename IF<IMPS::Polymorphic, typename Component::template Base<true>,
            typename IMPS::template Get<0>::Result>::Result Base;

    static const unsigned int UNITS = IMPS::Length;

public:
    Handle() {
        helper(Index<0>());
        _current = _implementations[0];
    }

private:
    template <unsigned int UNIT>
    void helper(const Index<UNIT> &) {
        _implementations[UNIT] = new typename IMPS::template Get<UNIT>::Result;
        helper(Index<UNIT + 1>());
    }

    void helper(Index<UNITS>) {};

private:
    Base * _current;
    Base * _implementations[UNITS];
};
```

Figure 10 – A minimal implementation of Handle's constructor.

C++ namespaces are used to separate component implementation and the decoupling artifacts from its interface exported to the application. The former is defined in the System namespace while the latter goes on the Application namespace. The client interacts directly with a Handle as in the application namespace the component is bind to its Handle already specialized with namespace Application { typedef System::Handle<System::Component> Component }.

3.3 STATE HANDLING

Not all related works that propose reconfiguring components from software to hardware and vice versa consider component's with an internal state (DYNAMICALLY..., 2014). Stateless components simplify reconfiguration as there is no need to deal with the state representation in different domains (i.e. software and hardware) with different syntaxes for representing a state. We overcome this issue with a unified description of both hardware and software components which allows the designer to express the component state uniformly among domains.

To keep the system sane when changing a component implementation, the old implementation must provide to the new one a snapshot of its state so that it can be restored in the new one. For simplicity, we consider that each component implementation (hardware or software) provides a data structure that captures its current state as a set of variables and each of its implementations provides methods to save and to restore its state using such data structure. The methods are invoked by the entity performing the reconfiguration which is responsible for transferring the state from one implementation to another as one atomic operation. Figure 11 shows both methods used to manage the implementations state, `save_state()` and `restore_state()`, for a whose current state comprises two variables, `_a` and `_b`.

By employing ADESD techniques for developing hardware and software implementations from a unified description, which in our case is written in C++, the state representation and the interface for handling it are uniform avoiding complex state migration issues. Techniques for migrating and changing the numerical representation of the current state from one implementation to another are a complex matter and are outside this dissertation's scope. For instance, there is no common convention for conversion between most numerical data types (e.g. how to convert a IEEE 754 binary64 number to a fixed point representation). Moreover, it is also not trivial to infer the type of each data value that constitutes the component state without embedding meta data in each value. Finally, the conversion must be performed only once (in the previous or in the next implementation) and a synchronization mechanism or convention must be employed to avoid multiple conversions.

3.4 CROSS-DOMAIN INTERACTION

Each implementation encapsulates the communication mechanisms necessary to exchange data and perform the computations on its substrate. In the software domain, components are objects which communicate using

```
class Implementation_0
{
public:
    struct State {
        int a;
        float b;
    };

public:
    Implementation_0() {}

    State save_state() {
        State state;

        state.a = _a;
        state.b = _b;

        return state;
    }

    void restore_state(State &state) {
        _a = state.a;
        _b = state.b;
    }

private:
    int _a;
    float _b;
};
```

Figure 11 – Example of helper functions for saving and restoring a component's state.

method invocation, while in the hardware domain, components communicate using I/O signals and specific handshaking protocols. For communication across different domains, the operating system must provide appropriate abstractions for hardware components and mechanisms for interrupt handling while the hardware must be aware that it is requesting a software operation. For instance, if given implementation uses a hardware IP to perform a calculation it is responsible for all mechanisms that must be deployed to exchange data between itself and the IP. The same applies for an implementation that depends on resources from different nodes of a network; it must operate the network stack to communicate with it.

To abstract communication patterns between components in different substrates, we employ an approach based on Remote Method Invocation (RMI) concepts from distributed object platforms (OSTROWSKI et al., 2008). Figure 12 illustrates an interaction between a component in the software domain with another in the hardware domain. Callee components are represented in the domain of callers by proxies. Channels deploy serializers and deserializers to marshal data structures exchanged by different domains. When an operation is invoked on a component's proxy, the arguments supplied are marshaled into a request message and sent through a communication channel to the corresponding component's agent. An agent receives requests, unpacks the arguments and performs local method invocations. The whole process is then repeated in the opposite direction, producing reply messages that carry eventual return arguments back to the caller

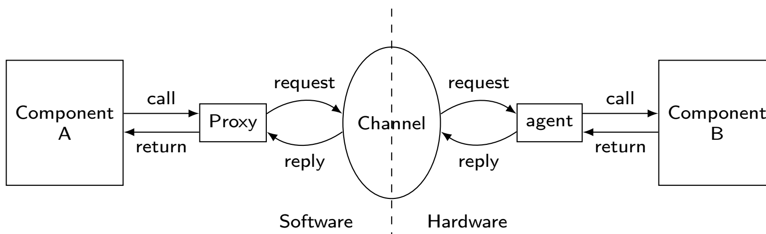


Figure 12 – Cross-domain communication using proxies and agents as proposed by Mück (MüCK; FRöHLICH, 2013).

The approach based on channels, proxies and agents that we have adopted for communication does not impose a single communication architecture. It can be used with different networking technologies, such as a shared bus, a NoC, or a DMA engine. For instance, when a bus-based communication channel is used, a proxy in hardware may be implemented as a memory-mapped slave device that notifies the CPU through interrupt requests when it

has a message ready to be read. Alternatively, on a NoC based implementation, a packet-oriented interface would be used to transmit request messages. Such variability is related to choices regarding the hardware/software architecture of the chosen implementation platform and should not affect the system components. Finally, the resulting communication bandwidth is limited mostly by the chosen underlying communication technology.

3.5 SUMMARY

This chapter presents the infrastructure for reconfiguring a component implementation during runtime while maintaining the component state by leveraging ADESD techniques. It allows components to have multiple implementations with the same interface so the application can be developed without being tied to a particular implementation. Implementations running in different substrates (as hardware or software) can be reconfigured as long as they follow guidelines for saving and restoring their current state. With reconfigurable components that can be deployed in different substrates the system to adapt and cope with conditions faced during runtime.

4 I/O INTERFERENCE MODELING

When developing on-chip interconnects, designers face a cycle of multiple iterations before reaching the most balanced architecture for their target platforms. Estimating the efficiency of an interconnect is not a straight-forward task and requires tools to evaluate statically or dynamically if the resulting architecture fits in the project requirements. Despite the chosen architecture, an interconnect is a shared resource and when used simultaneously by multiple hardware devices they can interfere with each other delaying transactions.

This chapter proposes a static performance-estimation technique for the time delay caused by I/O interference when moving data through shared chip resources such as interconnects and memory controllers. The model uses concepts from queuing theory and estimates the average time each peripheral data transaction has to wait on shared resources. The higher latency and throughput degradation is due to other peripherals' transactions that are passing through the same shared resources that queue the transactions before forwarding them. Such model is specially helpful to illustrate the increase in the FPGA reconfiguration time (specially bitstream loading) when multiple peripherals are contending for system I/O resources. Techniques to mitigate the issue will be discussed in Chapter 5. It is also meant to be used as a design tool for application designers to ensure a minimal interference between threads that depend on I/O resources.

4.1 LATENCY ANALYSIS

Threads using shared hardware resources can cause interference on each other due to several reasons. In our analyses, we address the impact of such interference from the perspective of the contention to access those shared resources, specially the when total bandwidth requested by the corresponding data flows is close to the system's interconnect capacity. We rely on queuing theory to model the interconnect, assuming the system is comprised of multiple software and hardware components that share a set of physical I/O resources. Figure 13 illustrates the modeling of an arbitrary interconnect using infinite queues. Components generating I/O traffic in the system, *sources*, are represented by circles with one or more arrows leaving it. They are connected to traffic destinations, *servers*, through an interconnect (e.g. hierarchical bus, NoC, crossbars) represented by *queues*. Each arrow arriving in a queue has an associated arrival rate λ_i dictated by the source connected to it. A server (e.g. memory, hardware accelerator, peripheral, CPU) consumes I/O data at a

rate given by its service rate μ_i. Interconnect characteristics such as topology, arbitration, routing, flow control, and any other influencing its performance are also captured by a service rate (μ_i) associated to each queue and define how it responds to different traffic patterns. In other words, the μ_i service rate of a server defines how fast it can consume data, while that of a queue specifies how fast data can flow through it.

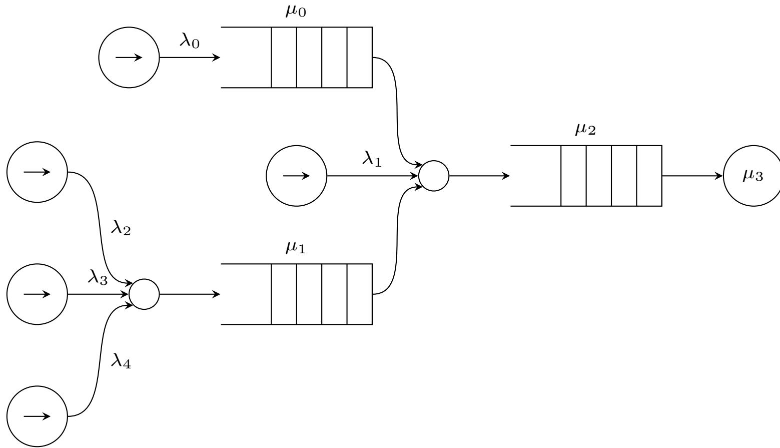


Figure 13 – Generic interconnect architecture modeled using queuing theory.

In our analysis, a *packet* is considered the minimum allocation unit of data flowing through an interconnect. Depending on the interconnect's architecture, it can be a real packet (on a network), or a flit (on a NoC), or a data transaction (on a bus). If multiple packets from different sources are directed to the same server, they will contend for the usage of the channels (modeled here as queues) along the path, eventually leading to additional latency (e.g. a hardware accelerator performing DMA and a CPU trying to access the main memory). Queues must handle multiple arriving packets and decide, according to their arbitration policy, which will be granted access to a requested output port. The time spent by arbitration dealing with contention is a major interference source for data flows depending on shared resources (servers). We therefore quantify interference in terms of such added latency. Latency is further characterized using the model proposed by Dally and Towles (DALLY; TOWLES, 2003). A queue's latency T is given by Equation 4.1 and represents the time a packet waits on that queue, from the moment it enters it to the moment it leaves it.

$$T = T^c + T^h + T^s \tag{4.1}$$

In order to properly model real interconnects, we further split latency in three composing elements: *contention*, *head*, and *payload serialization*. Even if a queue is representing a bus level or a NoC router, and a packet is representing a single flit, this decomposition is fundamental to model essentially distinct times. The contention time T^c models the time a packet waits to enter a queue. It increases with load and is highly dependent on the arbitration mechanisms in place. Several strategies have been proposed to estimate T^c, from static probabilistic analysis to dynamic simulation (PASRICHA; DUTT, 2008). In this work, we further rely on queuing theory to model T^c as an average estimate. In a queuing system, the contention time T^c can be given by the expected waiting time of the packet in the queue. According to Little's law (DALLY; TOWLES, 2003), the expected waiting time can be given by the ratio between the expected number of packets in the queue (occupancy) N^Q and the queue's service rate μ:

$$T^c = \frac{N^Q}{\mu} \tag{4.2}$$

The expected queue's occupancy depends on the characteristics of the interconnect and on the packet flows passing through it. For instance, a queue with deterministic service time and arrival rates that follow a Poisson distribution (M/D/1 queue) has an expected occupancy given by (DALLY; TOWLES, 2003)

$$N^Q = \frac{\lambda}{2(\mu - \lambda)} \tag{4.3}$$

The head latency T^h represents the time needed to allocate resources in interconnects for which that is a requirement. For instance, on a wormhole routing interconnect, T^h would be a function representing the times involved in establishing a path from source to destination. For time-division multiplexing, it would model the time needed to establish a virtual channel. The serialization latency T^s represents the time needed to inject L packets into a channel with a bandwidth of B packets per second as given by Equation 4.4.

$$T^s = \frac{L}{B} \tag{4.4}$$

Finally, we assume that each queue represents a stage or level of the whole interconnect bridging CPUs, memory, and I/O devices. Therefore, the total latency suffered by a packet on the interconnect from the source to the

destination server is modeled as the sum of the latency T_i suffered in each level i of the interconnect:

$$T = \sum_{i=0}^{N} (T_i^c + T_i^h + T_i^s) \tag{4.5}$$

or

$$T = \sum_{i=0}^{N} \left(\frac{\lambda_i}{2\mu_i(\mu_i - \lambda_i)} + T_i^h + \frac{1}{B_i} \right) \tag{4.6}$$

4.2 BITSTREAM LOAD MODELLING

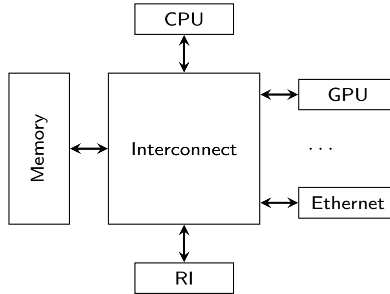
In FPGA partial reconfiguration schemes, the bitstream path from memory to the reconfiguration interface can cross the system interconnect shared by system peripherals and CPUs. The simultaneous usage of the interconnect by multiple peripherals implies in contention on its inputs ports. Such scenario lowers data transfer throughput and augments the reconfiguration time due to interconnect arbitration protocols and the resulting additional time each data flow must wait to cross the interconnect. A similar issue happens when multiple peripherals access different shared memory ports at the same time as, usually, the memory requests such as read and write from each port are served sequentially. If the reconfiguration is triggered while a critical threads performs I/O moving data through the system's shared interconnect, it might consume bandwidth previously allocated for the critical thread. This section presents the modeling using queuing theory of the I/O interference suffered when moving a bitstream from a storage device (e.g. RAM memory) to the FPGA reconfiguration interface due to shared resources contention (e.g. memory, interconnect).

Our analysis is based on an architecture containing a CPU and multiple devices (i.e. Ethernet interface, GPU) sharing a central interconnect with the reconfiguration interface presented in Figure 14a. Figure 14b presents the mapping of the architecture to the model: the interconnect is represented by a queue, the memory by a server while the GPU, Ethernet and the reconfiguration interface (RI in the Figure) by a source. Partial reconfiguration is carried by transferring the bitstream from the memory device to the FPGA reconfiguration interface. The granularity and reconfiguration method of each FPGA depends on the architecture dictated by its vendor. FPGAs usually employ two methods for bitstream loading (XILINX, 2015c; ALTERA, 2008):

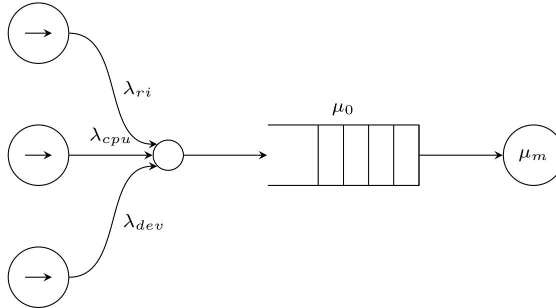
- The CPU performs Programmed Input-Output (PIO) by reading the

bitstream from the memory device and writing it, word by word, to the memory mapped register that controls the reconfiguration interface.

- The reconfiguration interface has a DMA engine that fetches the bitstream directly from the memory device; the CPU must only program the DMA to initiate the process.



(a) Architecture with a CPU, memory, reconfiguration interface and two peripherals.



(b) Resulting model.

Figure 14 – Mapping of the architecture into the model.

PIO-based reconfiguration interfaces, e.g. Xilinx's ICAP (XILINX, 2015b), rely on external reconfiguration controllers to load bitstreams for partial or full FPGA reconfiguration. The controller operates without interruption throughout reconfiguration and is implemented in the static region of the FPGA or even outside the reconfigurable fabric in an external processor. The bitstream data flow in a reconfiguration process through a PIO interface is depicted in Figure 15 as the dashed line. Bitstreams are stored in memory devices that can be accessed by the controller and later transfered to the reconfiguration interface. According to Equation 4.1 and considering

that the reconfiguration controller is implemented in the CPU, the latency of moving L data words from the memory to the CPU and from the CPU to the reconfiguration interface can be accounted as

$$T = 2L(T^c + T^h + T^s) = 2L\left(\frac{\lambda_0}{2\mu_m(\mu_m - \lambda_0)} + T_0^h + \frac{1}{B_0}\right) \tag{4.7}$$

in which $\lambda_0 = \lambda_{ri} + \lambda_{cpu} + \lambda_{dev}$. We have used the same μ_m both for read and write operations but notice that they might be different.

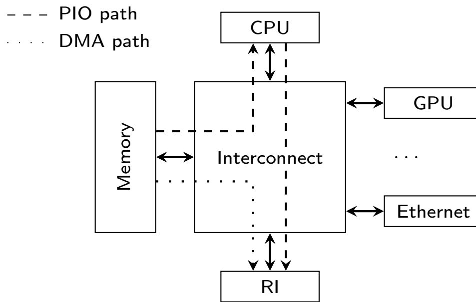


Figure 15 – Bitstream path from a memory device to a PIO-based and a DMA-based partial reconfiguration interface in a SoC with a central interconnect.

Given the overhead implied in having one load and one store operation for each word moved to the reconfiguration interface, newer devices propose alternative reconfiguration schemes to relieve the reconfiguration controller. For instance, instead of relying on an external reconfiguration controllers to move the bitstream from the memory into it, DMA-based reconfiguration interface haves their own DMA controller used to fetch bitstreams from the memory device as pictured in Figure 15. One example of this technology is Xilinx's Zynq-7000 family's reconfiguration interface, Processor Configuration Access Port (PCAP) (XILINX, 2015d).

Nevertheless, DMA-based reconfiguration interfaces are not completely autonomous; the reconfiguration controller still needs to set the bitstream length and its initial address in the memory device before issuing starting the DMA. Such interfaces can have control registers mapped into the reconfiguration controller memory and thus they can set all parameters necessary for starting a partial reconfiguration with load operations. Equation 4.8 presents the bitstream loading latency for a DMA-based reconfiguration interface which half the latency of the PIO-based one.

$$T = L(T^c + T^h + T^s) = L\left(\frac{\lambda_0}{2\mu_m(\mu_m - \lambda_0)} + T_0^h + \frac{1}{B_0}\right) \tag{4.8}$$

Figure 16 presents a comparison of time spent due to contention for both kinds of reconfiguration interfaces. The peripherals total arrival rate (λ_0) varies from zero to 90 % of the total memory service rate. The contention time grows as the arrival rate reaches the memory service rate becoming prohibitively high for values close to it. PIO-based reconfiguration contention time is two times longer that the DMA-based one as the bitstream crosses the interconnect two times before arriving to the reconfiguration interface.

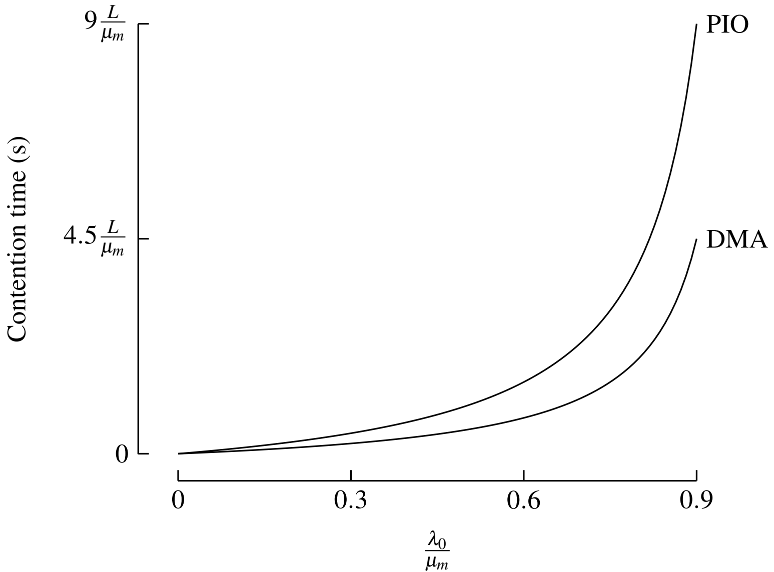


Figure 16 – Contention time for transfering an L words bitrstream from the main memory to a PIO-based and a DMA-based reconfiguration interface. The arrival rate of the devices sharing the interconnect with the reconfiguration interface and the memory is presented as a fraction of the memory service rate μ_m.

4.3 DISCUSSION

Different interconnect arbitration schemes can affect the arrival rate in a queue's input ports due to the back pressure exerted by the interconnect itself. Kim et. al. explore the modeling of fixed priority, round-robin and Time Division Multiple Access (TDMA) interconnect arbitration schemes to increase the accuracy of the queuing model for on-chip interconnects (KIM; IM; HA, 2005). In such sense, their model is complementary to ours and their techniques could be used to extend our model. Another technique used by Kim et. al. that could fit in our model, is the usage of memory traces to estimate the arrival rates of each source.

The analytical model proposed by Cho et. al. (CHO; CHOI; CHO, 2006) estimates the performance of multi-layer SoC interconnects based on the ARM Advanced Microcontroller Bus Architecture (AMBA) 2.0 AHB specification (ARM, 1999). Their model uses parameters such as the usage rate of the bus, the probability of a transaction to be single mode, and the probability of a data transfer to cross a bridge to estimate the average latency of communication. Despite targeting single interconnect architecture, such model is similar to ours in the sense that it is also based on probabilistic parameters.

5 SPECULATIVE RECONFIGURATION

This chapter describes a reconfiguration scheme where reconfiguring the implementation behind a given component's interface is a deterministic operation performed by the operating system in a speculative manner. The scheme is speculative in the sense that it is triggered based on the runtime awareness of shared resources utilization (e.g. I/O, CPU time, memory) in the system rather than static knowledge of its functioning. Reconfiguration is split in small steps executed while the operating system is idle and mitigates interference by powering down peripherals performing I/O operations. The decision of powering down a peripheral based on its I/O usage is taken by using data from the Performance Monitoring Unit (PMU) and also from peripheral's performance registers (e.g. the register that counts the number of bytes sent in a network card). Hence, even with little available idle time, reconfiguration can be carried transparently, comply with timing requirements, and be aware of possible interference sources in the system. The chapter shares content with the DSD'15 paper *A Framework for Dynamic Real-Time Reconfiguration* (REIS; FRöHLICH; WANNER, 2015), and the SBESC'15 paper *On the FPGA Dynamic Partial Reconfiguration Interference on Real-Time Systems* (REIS; FRöHLICH; HOELLER, 2015).

Section 5.1 contains the assumptions considered for deploying the speculative reconfiguration process. In Section 5.2 we discuss how to isolate the reconfiguration process to ensure it is only deployed when there is available idle time. Techniques for monitoring the I/O usage that can affect the reconfiguration and techniques to mitigate its impact are presented in Section 5.3 and Section 5.4 respectively. The reconfiguration process steps are described in Section 5.5 to Section 5.8. Section 5.9 presents a discussion on different aspects of the proposed techniques.

5.1 ASSUMPTIONS

This works relies on a set of assumptions to function effectively. Initially we consider that to perform the reconfiguration, there must be available idle time (slack) during application execution and that when no other thread is executing, an operating system thread usually called idle thread is scheduled. We assume that critical threads are periodic and the scheduling policy is selected from a group of scheduling algorithms for which the slack time can be computed. Next, we assume that the operating system has a power management mechanism capable of keeping track of which tasks are using each

device. Finally, we assume that components subject to power management, usually I/O devices, implement methods to dump and restore their states.

5.2 RECONFIGURATION ISOLATION

Operating system or application activities occurring at the wrong instant can disrupt FPGA reconfiguration and cause several drawbacks to applications depending on it. Moreover, contention time due to applications usage of I/O resources is usually not taken into account on reconfiguration execution time. For instance, operations that are most susceptible to I/O interference are those that move significant amounts of data from one subsystem to another such as loading a bitstream from memory into the FPGA reconfiguration interface. To overcome these concerns, we deploy the reconfiguration process only when there are no other threads to run. We consider that in this scenario a special thread with the lowest priority in the system called idle thread is scheduled. The reconfiguration process invoked by idle thread is the entity in charge of keeping the system configuration tuned with the reconfiguration policy specified by the user.

The idle thread must be able to retrieve the time available before the next scheduling event to ensure that the reconfiguration will not interfere with timing guarantees of the next scheduled thread. We consider that the operating system scheduler has a queue listing all threads that are ready to be executed but did not yet reached their activation periods and that such structure is accessible by the idle thread. Such list contains timing information on each thread and is used by the operating system to know when each thread should be scheduled and can be used to calculate the available slack time that can be used to perform the reconfiguration. The reconfiguration process is activated only if the available slack is large enough to hold its execution (or part of it) without delaying the next scheduling event. Nevertheless, even when the scheduler's ready queue is empty, we cannot assume that no I/O operation that might interfere with (and be interfered by) the reconfiguration is happening on background. For example, a thread might have started a DMA operation to transfer a network packet from the main memory to the network interface card and gone sleeping waiting for it to finish. When deployed by the idle thread, the reconfiguration process monitors I/O operations occurring in the system to infer when is the best time to trigger the reconfiguration so that its execution time does not exceed the available slack time. If the slack time is not enough to perform the reconfiguration, the reconfiguration process can power down one or more non-critical threads expecting a larger slack time in its next invocation.

The speculative reconfiguration triggered by the idle thread is a key element in our approach but, ensuring that critical threads will not have their deadlines compromised solely through it would require the reconfiguration procedure to be carried out without ever blocking the scheduler. Indeed, much of this procedure can be performed in parallel with the execution of user threads, including reconfiguration policy enforcement and bitstream loading. Nevertheless, the handling of a component's state and the redirecting of their clients to the new implementation are operations that have to be performed atomically. For instance, the state can be corrupted if we transfer half of it from the old implementation to the new one and part of the already transferred state changes before binding the interface to the new implementation. Since both atomic activities, state and client handling, have deterministic duration (state handling is directly proportional to the component's internal state size and client binding is a fixed cost operation), the reconfiguration mechanism can always determine whether a reconfiguration would compromise a deadline or not, postponing it in case it would. Combining these elements we provide a deterministic and self-contained reconfiguration process as will be shown in the next sections.

The decision of which device to power down to maximize the slack time and to reduce interference on the reconfiguration process depends on the proper monitoring of the interference sources. Before presenting a detailed description of the reconfiguration process, we present an overview of I/O monitoring and interference mitigation techniques.

5.3 I/O TRAFFIC MONITORING

As previously stated, activities carried during reconfiguration might move large amounts of data from one subsystem to another (e.g. loading a bitstream from memory to the reconfiguration interface). Not always the time interval until the next scheduling event calculated by the operating system is large enough to perform a full or a partial reconfiguration. To enlarge the slack time in subsequent invocations, the reconfiguration process can decide to suspend non-critical threads whenever their I/O usage trespasses a given threshold or based on usage statistics of peripheral usage. Such heuristic is fed with data from system monitors capable of gather metrics as the usage of peripherals I/O capabilities and the load on hardware communication bridges.

If each active peripheral has its own set of registers containing usage statistics, it is straightforward to infer their individuals I/O loads on the system and by consequence, how much I/O interference they generate. The statistics registers hold counts for various types of events associated with device specific

operations and are valuable to infer when it should be powered off in order to reduce interference. By standard, Ethernet controllers gather statistics like the number of frames and octets received and transmitted through the network. Moreover, modern I/O bridges are also equipped with transaction counters able to differentiate from read and write operations. Insights on I/O traffic can be derived by fusing data from peripheral counters with data from bridges to estimate the traffic of peripherals with no usage statistics registers at all. The overhead associated with using peripheral statistics registers for heuristics on peripheral I/O usage is comprised by the operation of reading them by polling.

In platforms where such fine grained monitoring capabilities are not available, it is still feasible to infer usage data from a PMU considering that the operating system is aware of the active peripherals. PMUs are specialized on-chip hardware units that monitor micro-architectural events on processors in real-time used mainly to tune and profile application and operating system (SPRUNT, 2002) and to perform online optimizations. The events count is stored in Hardware Performance Counters (HPCs) which are special registers available in most modern processors capable of monitoring events such as cache misses, page faults, elapsed CPU cycles, traffic on interconnects and many more. Newer PMUs are even capable of monitoring I/O traffic on individual interconnect slots. Despite modern PMUs being able to account hundreds of events, due to physical limitations only a restricted set of events can be routed to HPCs being unfeasible to monitor all events simultaneously. Specific libraries are used to mitigate the complexity of handling the low-level PMU interface and correctly mapping the desired events to the HPCs by the operating system. The Linux operating system has its own performance counter subsystem commonly accessed in user space by the *perf* tool to obtain and analyze application performance. Embedded libraries to abstract the access to HPCs by creating a simpler interface to configure and read them in embedded systems, without adding expressive overhead to the application performing dynamic optimizations are also proposed (GRACIOLI; FRöHLICH, 2011). Notice that peripheral statistics registers can provide a finer-grained information compared to PMUs data but the latency associated with HPCs registers is much smaller as PMUs are usually closer to the CPU.

Even with no hardware performance monitoring at all, the operating system might have counters implemented in software able to correlate software operations with I/O usage. By accounting the number of invocations from a thread to a given peripheral represented by its device driver, an operating system has an indicator of I/O traffic. In reflective systems it would even be possible to store as object meta-information the number of invocations of a method the performs heavy I/O to further use it as a traffic estimator. Off-the-shelf operating systems such as Linux can monitor I/O usage per task from

user land (e.g. the *iotop* command can monitor task I/O usage through the *taskstats* kernel interface). Software counters and hardware counters should not be seem as separate mechanism for I/O monitoring but as complementary, using data from both of them can provide deeper insights on the global I/O usage.

A different approach would be using static powering down order to define which threads should be powered down first to relief critical threads from I/O interference. Such strategy can work effectively on systems with statically predictable I/O loads in which a set of threads are clearly the ones to generate most of the I/O traffic in the application. For example, a thread that is constantly decoding a video stream can be a perfect candidate for powering down. Nevertheless, in complex applications that react to unpredictable external stimuli it is difficult to evaluate before runtime which threads and peripherals are the ones that will have a bigger impact on I/O usage.

Monitoring methods independent from hardware or software event counters can also be employed. An operation passive to suffer I/O interference from other system components can be profiled during system startup, when no other system peripheral is active. Subsequent invocations of the operation are also monitored and we can attribute significants increase of its execution time to the I/O interference caused by other peripherals.

5.4 I/O INTERFERENCE MITIGATION

Monitoring alone is not sufficient to allow a reconfiguration without interference. Not always the system can wait for an opportunity where there is enough CPU time and no I/O operations at all are being held to start the reconfiguration. It must also deploy techniques to mitigate interference sources when the reconfiguration is more important than the activities generating interference.

A hardware mitigation solution would be employing interconnects with QoS capabilities. QoS allow packets to be categorized in classes that may posses different performance requirements (e.g. latency, jitter, throughput) and priority. By means of strict service contracts between client (peripheral) and the interconnect, the desired level of performance is provided as long as the peripheral complies with a set of limitations (DALLY; TOWLES, 2003). For example, a QoS contract can guarantee a maximum latency of 1 ms to a peripheral moving data to the main memory if it injects less than 1 MB during a 1 ms interval. In a scenario where two packets with different priorities request the same resource, the resource will be yielded to the one with a higher QoS priority. To comply with contracts, some restrictions are propagated to

the software layer that must guarantee a peripheral I/O usage between the contract bounds. ARM's CoreLink NIC-301 Network Interconnect provides an optional QoS extension (ARM, 2011) that allows assigning priority levels for traffic coming from each master in the interconnect. Notice that solely using QoS for a peripheral attached to an interconnect does not solves the interference problem completely if the peripheral is being used by multiple threads. If a critical and a non-critical thread are accessing the same device, QoS will prioritize traffic generated by both of them without distinguishing between criticalities. The coarse grained control offered by QoS on I/O traffic priorities does not allow it to be used as the single mitigation mechanism, it must be coupled with software techniques for a holistic control on I/O traffic.

Similar to QoS, specialized hardware interfacing the peripheral with the interconnect can throttle the peripheral I/O operations and also buffer incoming data, waiting for the appropriate moment to transfer it to its destination without disrupting timing requirements from the rest of the system. An approach focused on the classical real-time systems theory would be to tame the I/O subsystem by putting it under a real-time scheduling discipline (BAK et al., 2009). Time allocated for communication can be viewed as a shared resource and be allocated by scheduling policies in a similar fashion to classical uniprocessor scheduling. Instead of worrying during runtime if a peripheral is active, specialized hardware IPs centralize all the decision making concerning scheduling the I/O transfers using real-time policies. Without employing monitoring techniques, this strategy must guarantee that only one peripheral is exchanging data at a time. Notice that allowing multiple peripherals to simultaneously access the main memory through the same interconnect can result in unpredictable latency and bandwidth allocation. A static strategy to mitigate possible interference sources is to deal with I/O interference by accounting for the contention time in the Worst Case-Execution Time (WCET) calculation of each task (PELLIZZONI; CACCAMO, 2010) based on traces of program execution. However, being aware of the global load bound of all peripherals in the system can be unfeasible for partitioned architectures that must be temporally and logically isolated from each other. To address this issue, specialized hardware is proposed to extend I/O isolation to different partitions with different criticalities.

Software power management techniques can also be adapted to function as an I/O mitigation mechanism assuming that the system can infer which peripherals are active. For instance, assuming that power management mechanism can throttle or disable a peripheral to save energy, the same procedure can be employed to shut down a peripheral when I/O monitors signal it is interfering with a critical thread. The power management mechanism must effectively map the abstractions to the peripherals used by them to transparently disable

not only the peripheral but also the higher level layers of the application. A detailed description on using a power management as a mechanism to mitigate interference will be presented in Section 5.7. When using the powering down mechanism to mitigate I/O interference we must acknowledge that powering down a component that is frequently used might increase the overall power consumption instead of decreasing it due to the inherent overhead of the powering down operation (WEISSEL; BEUTEL; BELLOSA, 2002). Heuristics to both decrease power consumption and reduce I/O interference must account not only for the magnitude of the I/O transactions a peripheral perform but also the frequency they occur.

5.5 RECONFIGURATION TRIGGERING

To explain the reconfiguration process in details, we will assume a reconfiguration policy that favors hardware implementations (i.e., it will try to push as many components to hardware as possible). Any other policy to guide when and which components should be reconfigured could be used. For instance, if energy awareness is a must, the system might start reconfiguring power hungry implementations into low power ones when running on batteries if the application can tolerate a decline in quality-of-service of its components (WANNER; SRIVASTAVA, 2014).

In the first step of the process, presented in Figure 17, the idle thread loads a given component implementation into the reconfigurable fabric. Initially, `get_slack()` asks the scheduler for the available slack time until the next scheduling event. The idle thread relies on an abstraction named `Component_Manager` for assistance with auxiliary tasks during reconfiguration. `Component_Manager` is charged of managing available hardware resources and manipulating component implementation's state during runtime It also interacts with other operating system component to power down components that might interfere with the reconfiguration process. If the slack is enough to allocate the necessary hardware resources in the reconfigurable fabric, the idle thread asks the `Component_Manager` if one of the system components needs to be reconfigured using the `reconfig_request()` method. The allocation of the hardware resources which will be used by the component such as reconfigurable partitions is performed in `alloc_resources()`. The time necessary to allocate the resource depend on the underlying reconfigurable fabric and how it is organized. For example, if well defined reconfigurable partitions are used as the reconfigurable blocks, `alloc_resources()` will look for a free one in a list of partitions. Otherwise, better FPGA area usage might be yielded with sophisticated placement algorithms (AHMADINIA et

al., 2004; STEIGER; WALDER; PLATZNER, 2004). Caching techniques can also be employed to reduce the resource allocation time by exploring temporal locality in hardware implementations usage (AHMADINIA et al., 2004).

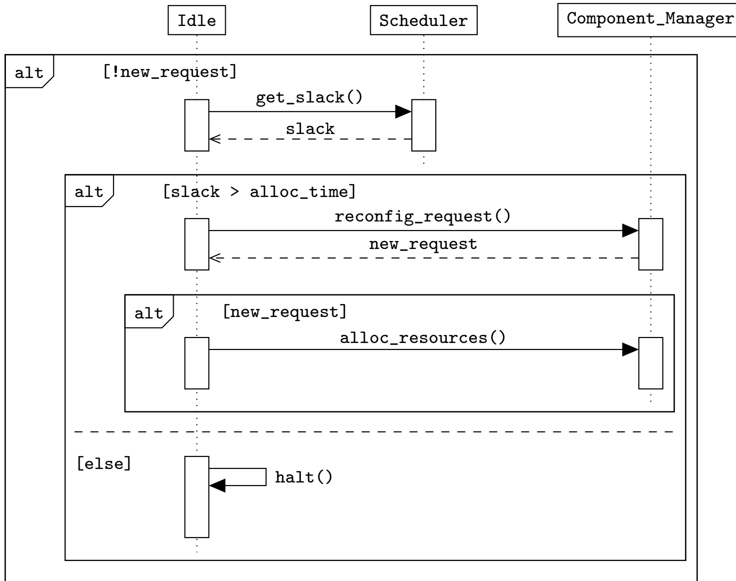


Figure 17 – Reconfiguration triggering and resource allocation sequence diagram.

5.6 COMPONENT LOADING

Next, a bitstream containing the implementation of the given component to the available partition is loaded into the FPGA as shown in Figure 18. Notice that the process described in Figure 18 happens right after the process depicted in Figure 17 is completed. `get_slack()` is invoked again to account for the impact of resource allocation in the available slack. The time necessary to load a bitstream to the reconfiguration interface is sensible to the usage of the interconnects through which the bitstream will cross. Thus, to keep the reconfiguration time deterministic, we rely on minimizing interference from other peripherals performing I/O operations during bitstream loading. Based on the heuristic knowledge of the I/O operations carried in the system, the idle thread can decide to perform or to postpone the bitstream loading. If

bitstream loading is postponed, the idle thread will use its heuristics to chose a non-critical thread to power down aiming to increase the slack time in its next invocation by calling `Component_Manager`'s `power_down()` method.

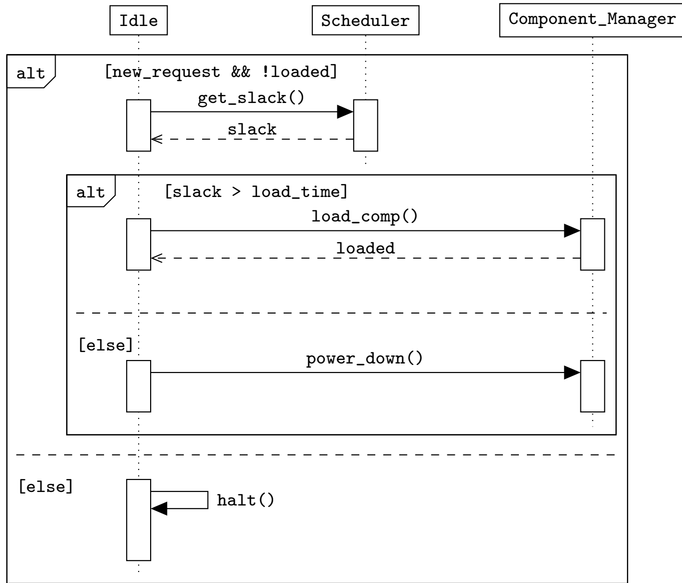


Figure 18 – Component bitstream loading sequence diagram.

With possible interference sources disabled and enough slack time available, bitstream chunks are fetched from memory and loaded into the FPGA each time `load_comp()` is invoked by the idle thread. The size of each chunk is adjusted to fit in the slack time available in the given scheduling period considering the FPGA's reconfiguration interface throughput. `load_comp()` signals that the bitstream is completely loaded by returning `true` which is assigned to the `loaded` variable. If the slack time is insufficient, the idle thread follows its original flow (usually halting the CPU). One could be tempted to assume that bitstream loading and user thread execution could be carried out in parallel, but since our focus is on critical systems, such an assumption would rise complicated questions about arbitration in the system buses that cannot be answered in a platform-independent way. We therefore check for amount of time remaining until the next thread activation. The first step ends when the bitstream in completely loaded, which might take multiple invocations of the idle thread, followed by the powering up of the components that were previously shut down.

The idle thread also monitors interference sources by profiling each `load_comp()` execution time, t_{lc}. The canonical `load_comp()` execution time without any source of interference, T_{lc}, is measured on system startup before launching the application. If $t_{lc} > T_{lc}$ for any `load_comp()` execution, we consider that critical threads issued I/O operations and are now sleeping waiting their completion and thus, interfering with reconfiguration. If peripherals being used by critical threads interfere with `load_comp()` operation, the idle thread can only abort the reconfiguration.

5.7 POWER MANAGEMENT

Both `power_up()` and `power_down()` methods are wrappers for a power management mechanism that allows changing operating modes of individual components, including the ability to turn them on and off (HOELLER; FRöHLICH, 2006). The power management mechanism consistently migrates component states among operating modes using the hierarchical organization of software and hardware components. It also keeps track of the relation between system components, ensuring consistency of operating mode transitions by navigating the hierarchical organization of components. System-wide power management takes place by issuing operating mode change commands to the `Component_Manager` component that contains references to all subsystems used by the application. `Component_Manager` handles the propagation of operating mode change commands to all components in the system. `power_up()` and `power_down()` use `Component_Manager`'s interface to navigate the components hierarchy to turn on and off the devices that are clients of non-critical threads.

The power management mechanism controls concurrent access to operating mode transitions of system devices. When threads instantiate components, the mechanism is notified so it can identify the client threads of each device. If more than one thread uses the same device, an operating mode transition only takes place if all client threads agree on the target operating mode. If two or more threads issue different operating mode transition commands, the targeted device stays at the requested operating mode that delivers the larger set of services or the higher performance.

Components deliver their power management capabilities through their power management interface. The interface is comprised by two methods: one to verify the component operating mode (`power()`) and other to change it (`power(status)`). The system also provides a set of predefined operating modes for portability purposes: `OFF`, `STANDBY`, `LIGHT`, and `FULL`. All components can have their operating modes managed by the interface, including

operating system abstractions such as threads or network communication interfaces, not only peripherals.

The Component_Manager uses Thread's power management interface to power down peripherals being used only by non-critical threads. Threads implement two operating modes: FULL and OFF. When an invocation of the thread's power(OFF) takes place, the Component_Manager forwards the same invocation to all components down the hierarchy to which the target thread is a client Each component can in turn use other components or devices, so the power management mechanism propagates the command until it reaches the bottom of the components hierarchy. Each device reached is shut down if the target thread is its sole client otherwise, the device stays in the higher level of operation requested by its client threads.

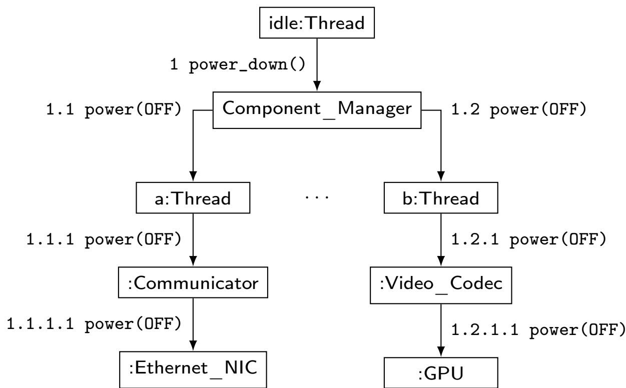


Figure 19 – Component_Manager powering down two threads and the peripherals being used by them.

Figure 19 shows an example of the power down process performed by the Component_Manager in two threads. After the idle thread calls the Component_Manager's power_down() method, Component_Manager traverses the scheduler queue to identify non-critical threads. Next, the Component_Manager proceeds to issue power(OFF) commands to all non-critical threads which, in this example, are threads a and b. a uses a Communicator abstraction[1] to which the power(OFF) command is forwarded. The Communicator accesses the power management interface of its clients to check which of them are active. If a is the only active client of Communicator, the command is forwarded to the Ethernet_NIC driver, which handles the Ethernet device shut down. Similarly to what happens to a,

[1] Communicator abstracts the protocol stack of a network communication subsystem.

in b the power(OFF) command is forwarded to the underlying components (Video_Codec), eventually reaching the GPU driver and shutting it down if it has no active clients.

When a critical and a non-critical thread share the usage of a peripheral, one might argue that it is not possible tell which of them is responsible for a high I/O usage rate using only information from low-level performance counters. Indeed, most PMUs cannot tell which logical entity (e.g. an operating system thread) started an I/O operation and thus, account for it in a separate counter. In such scenarios, software techniques can combine PMU data related to a given peripheral activity with the awareness of which and when each logical entity interacts with it to infer the peripheral usage share of each logical entity. For instance, each time the operating system intercepts a call from a thread to a peripheral, it can annotate the peripheral's performance statistics before and after to account for the difference and associate it with the thread's usage of the peripheral. Based on such usage hints associated to each thread, the operating system can employ statistical methods to guide its choice of which component to power down to minimize I/O interference. For example, if a non-critical and a critical threads share the same peripheral and historically the non-critical thread is responsible for most of the usage share of a peripheral, it is safe to assume that powering the non-critical thread down will minimize its interference on the critical thread.

5.8 STATE HANDLING AND REBINDING

To finalize the reconfiguration process, the Handle must save the component's state from the previous domain and restore it into the current one. Moreover, it must update its domain to dispatch future method invocations to the correct component implementation. This step, depicted in Figure 20, must be atomically executed and therefore requires enough slack time to be concluded in a single invocation of the idle thread. When the available slack is bigger than the time to perform the bind operation, the idle thread initially acquires the scheduler lock. One might argue that if the slack time is enough, the idle thread will not be preempted during the whole step thus the lock is pointless. Nevertheless, on multicore machines, the idle thread must acquire the lock to prevent cross-core interference. Next, the idle thread invokes the reconfigure() method of the component's Handle.

reconfigure() first fetches the software implementation's state by invoking save_state() and further transfers it to the hardware implementation using restore_state() At this point the hardware component is ready to operate and its Handle can change the component's domain using

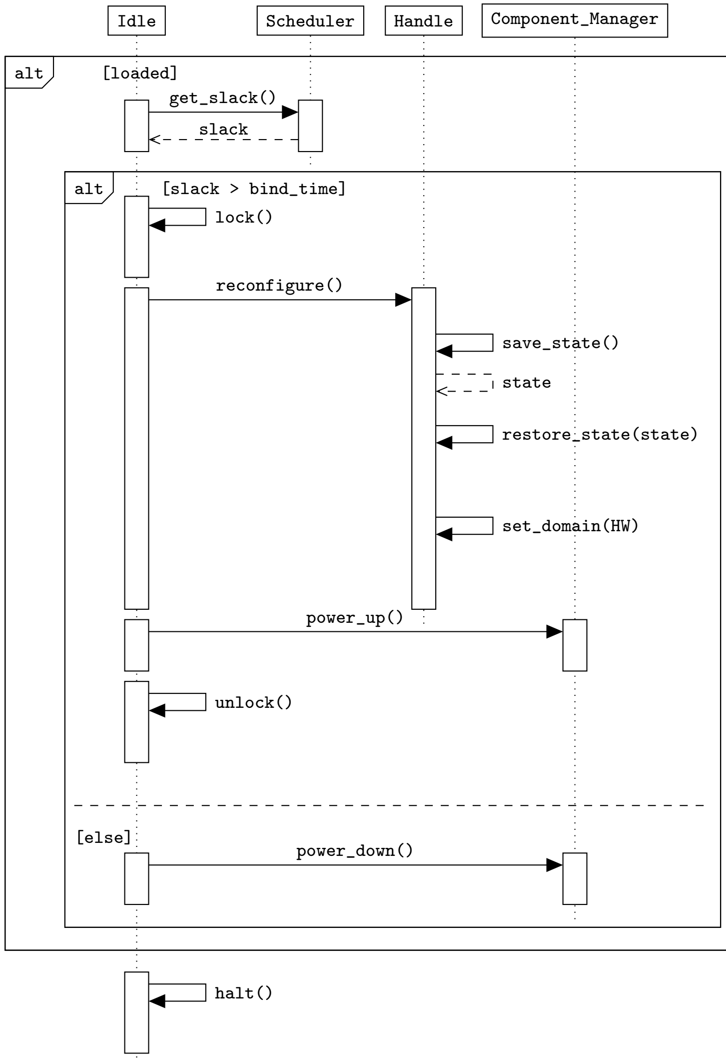


Figure 20 – State handling and rebinding sequence diagram.

`set_domain()`. From now on, the method invocations will all be dispatched to the component's hardware implementation. After the reconfiguration is finished (possibly after more than one idle thread invocation), the peripherals that were powered off are turned on by the `power_up()` method.

Extra attention to I/O interference is also taken when saving and restoring the state of the component being reconfigured. As it is an atomic operation, we cannot interrupt it as with the component loading step. We can, however, account for its worst interference case and use the same heuristics approach to power down components as in the previous step and defer `reconfigure()` execution until it becomes possible. When the whole reconfiguration process is finished, `power_up()` is called to awake all threads and peripherals that were powered down previously.

5.9 DISCUSSION

The strategy proposed in this chapter provides a speculative reconfiguration scheme that can be used without disrupting the embedded system's timing constraints postponing it otherwise. In opposite to most works that employ reconfiguration capabilities in time sensitive applications, we do not schedule statically when each reconfiguration will occur (LI et al., 2013; AL., 2011; CLEMENTE; RESANO; MOZOS, 2014). Instead, we provide the infrastructure to deploy it during application execution as a reaction to unpredictable environmental and process variations (e.g. availability of power sources, hardware defects). Such flexible reconfiguration process allows it to be seen as an adaptation mechanism in systems that are sensitive to its environment and use such self-awareness to better adapt itself to an application during runtime.

We do not specify exactly in which environmental conditions an implementation should be reconfigured and how to gather data to obtain such insights, those concerns are out of this work's scope. In this way, our work is complementary to efforts such as the CPSoC (SARMA; DUTT, 2014) that fill the gap of how to monitor an embedded system conditions through sensor data that crosses the multiple hardware and software layers. Moreover, algorithmic choice efforts such as the ViRUS framework (WANNER; SRIVASTAVA, 2014) provide rich insights on how to create a set of rules of when a certain component implementation can be switched by another without strict user intervention. It is possible to trigger adaptation according to a set of power consumption, temperature rules, and quality-of-service ranges defined by the user that guide a context-aware algorithmic choice. Our hardware reconfiguration scheme can be seem as an actuation mechanism in such works and utilize

their self-awareness capabilities to change a component implementation based on context.

The reconfiguration criticality depend directly on who is benefiting from it: generally, it is as critical as the thread that uses the component being reconfigured. Nevertheless, special environmental conditions might demand for a more flexible approach to evaluate the reconfiguration criticality. For example, if the system is running on batteries that are almost depleted, it is safe to state that reconfiguring components to their low-power implementation is a priority to keep the system alive.

Notice that the reconfiguration process executing in the idle thread results in an overall execution time increase that must be reserved for the operating system in the CPU capacity. On traditional computer architectures that employ deterministic hardware, the WCET of each method invoked by the speculative reconfiguration process can be estimated. Statistical methods based on static code analysis techniques and code profiling (WILHELM; AL., 2008) can be employed to determine the execution time of such operations. Nevertheless, modern architectures employ mechanisms (e.g. pipelines, hardware prefetchers) that sacrifice the timing determinism of its instructions set in favor of a lower average latency. In multicore processors estimating the WCET becomes even more complicated due to sharing of resources such as memory, cache, and I/O channels resulting in intercore interference. For such architectures, using static analysis to predict the WCET of a source code block is extremely difficult or overly pessimistic due to its interactions with non-deterministic hardware. An estimation of the WCET with an adjustable confidence level can be extracted from multiple executions of the application binary code on the target platform. In measurement-based approaches, the resulting value of the WCET is typically inflated further with an error margin (20 % to 30 % of the observed value) in order to account for unobserved conditions that can delay the execution (GRACIOLI et al., 2015). Consequently, measurement-based approaches can also lead to an overestimated WCET.

We do not enforce a specific thread scheduling policy for the reconfiguration scheme to work however, some scheduling policies might result in faster a reconfiguration than others. For instance, the Least Laxity First (LLF) scheduling policy for thread scheduling agglutinates the slack time of jobs running for multiple periods, yielding intervals that are more likely to be suitable for reconfiguration operations. Nevertheless, other scheduling policies such as Earliest Deadline First (EDF) and Rate-Monotonic (RM) could also be used.

6 EVALUATION

This chapter has four objectives: 1) Present the technical aspects behind the ideas in Chapter 3 and Chapter 5; 2) Show that I/O operations interfere with FPGAs reconfiguration; 3) Evaluate the proposed reconfiguration scheme in an industrial application with severe timing constraints; 4) Analyse possible trade-offs between different components implementations that might dictate reconfiguration policies.

For so, the chapter explores a proof of concept for the partial implementation of reconfigurable components and their reconfiguration process used to evaluate the ideas previously proposed. The focus is on the hardware platform used in the experiments and the Real-Time Operating System (RTOS) infrastructure proposed. To address the three final objectives, four experiments were performed and are described in the next sections. The first and the second evaluates the partial reconfiguration time in different scenarios of I/O interference inflicted by peripherals. The third analyses the feasibility of using reconfigurable components in a PABX telephony system SoC studying the heuristics that can be implemented to optimise performance based on the ability to reconfigure system components. The fourth explores possible trade-offs between mathematical components implementations that might be explored to adapt the system's non-functional characteristics.

In all experiments, the operating system and the applications were compiled with GCC 4.4.4 targeting the ARMv7 ISA using level 2 optimization enabled by GCC's `-O2` flag except in the last that uses GCC 4.7.2 with the `-O` flag activated. For the hardware flow, Calypto's Catapult UV 2011a was used to obtain RTL descriptions of the components described using the Unified Design methodology. The hardware platform was prototyped in ZedBoard, a development kit based on Xilinx's XC7Z020 SoC, using Xilinx's Vivado 13.4 for RTL hardware synthesis except in the first and the second experiment that used Vivado 14.2. Synthesis constraints were adjusted on Vivado and Catapult to minimize circuit area considering a maximum frequency of 100 MHz except in the last experiment where the maximum frequency is 150 MHz. Notice that the software versions were updated during this work to benefit from new features.

This chapter shares content with the DSD'15 paper *A Framework for Dynamic Real-Time Reconfiguration* (REIS; FRöHLICH; WANNER, 2015), the RSP'15 paper *X-Ware: Mutant Computing Substrates* (REIS; WANNER; FRöHLICH, 2015), and the SBESC'15 paper *On the FPGA Dynamic Partial Reconfiguration Interference on Real-Time Systems* (REIS; FRöHLICH; HOELLER, 2015).

6.1 IMPLEMENTATION

This section starts by presenting an overview of the EPOSSoC and its internals including details on RTSNoC, CPU nodes, and rec nodes. Next, the EPOS operating system is presented followed by the description of software and hardware components implementations and the details on hardware/software communication.

6.1.1 EPOSSoC

A platform called EPOSSoC was assembled to support the deployment of this work, Figure 21 shows its architecture. The CPU nodes contain software components and execute the RTOS that orchestrates hardware components deployment. Rec nodes are reconfigurable partitions that might contain hardware implementations of components deployed on an FPGA. An IO node accommodates peripherals used for input and output operations that are shared between all other nodes in EPOSSoC. All nodes are interconnected through a NoC based scheme implemented using RTSNoC routers. A bus-based interconnect was discarded as it is not the most suitable choice for more heterogeneous designs in which hardware components have active roles (MICHELI et al., 2010). Moreover, the rec nodes are connected to RTSNoC routers ports and one of the RTSNoC routers ports is connected to the CPU node through an AMBA bridge. The SoC was prototyped in ZedBoard (AVNET, 2014), a development kit based on Xilinx's Zynq-7000 FPGA SoC coupling a reconfigurable fabric with a hard core ARM Cortex-A9 dual core CPU.

6.1.1.1 RTSNoC

The RTSNoC targets real-time applications: routers arbiters implement a priority-based dynamic scheduling algorithm that establishes limits for the communication latency between two network nodes. RTSNoC routers, presented in Figure 22a, allow flit interleaving from different flows in the same communication channel while balancing the load without centralized network control (BEREJUCK; FRöHLICH, 2014). For this purpose, all flits flowing through the network carry embedded routing information used for flit arbitration and scheduling by the network routers. The network consists of star topology routers disposed to form a two dimensional mesh and each router has eight bi-directional channels, presented in Figure 22b, that can be connected to cores or channels of other routers.

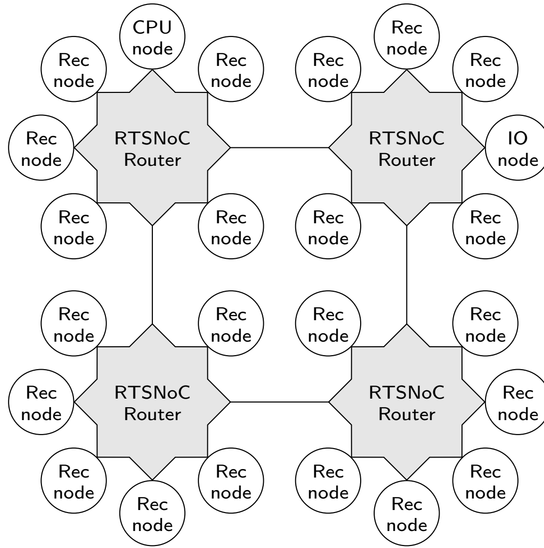
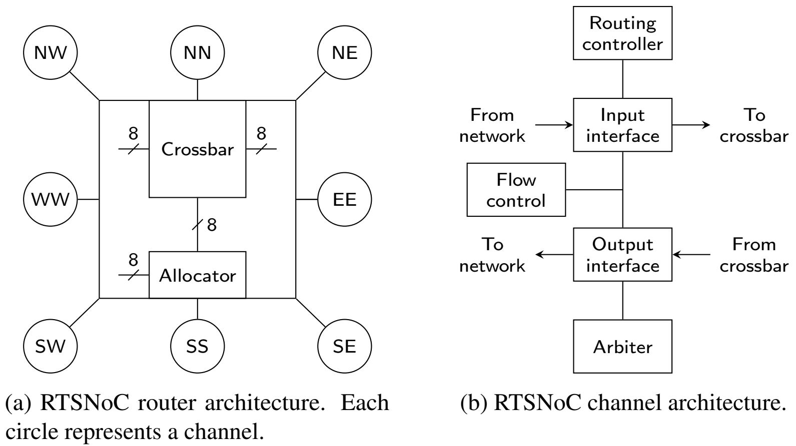


Figure 21 – EPOSSOC block diagram. CPU, IO, and rec nodes are interconnected by RTSNoC routers.



(a) RTSNoC router architecture. Each circle represents a channel.

(b) RTSNoC channel architecture.

Figure 22 – RTSNoC internals.

6.1.1.2 CPU nodes

Software implementations of components run on the CPU nodes that are hardcore or a softcore processors with its own peripherals. In our scenario, a CPU node, presented in Figure 23, is a dual-core Cortex-A9 processor coupled with several peripherals that executes an RTOS. Its internal interconnect structure has several levels that are based on the third generation of the AMBA protocols family called Advanced eXtensible Interface (AXI). The CPU node provides the necessary hardware support to implement the proxies and agents described previously when communicating between software and hardware. Hardware reconfiguration is held by the PCAP interface on the AMBA Interconnect, also managed by the RTOS.

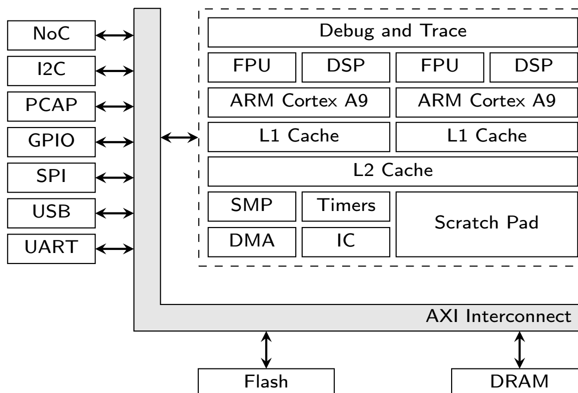


Figure 23 – CPU node block diagram. Software components are deployed in it by an RTOS.

6.1.1.3 Rec nodes

Rec nodes are FPGA partitions that contain a hardware implementation of a given component and can be individually reconfigured using PCAP. The implementation deployed in the node might change during application execution as well as the component using the node itself. For instance, component `A`'s `Imp_0` hardware implementation might be initially deployed in a rec node followed by component `A`'s `Implementation_1` followed by component `B`'s `Implementation_0`. During partial reconfiguration, rec node output pins behaviour is unpredictable which can lead to invalid packets sent through

its interface with the RTSNoC router (XILINX, 2015c). To mitigate this issue, the rec node interface with the RTSNoC router is decoupled during partial reconfiguration using registers that are disabled before starting partial reconfiguration and enabled when it finishes.

6.1.2 EPOS

The EPOS RTOS was used as the backbone for the infrastructure employed for seamlessly deploying reconfigurable components. EPOS is a multi-platform, component-based operating system that implements traditional operating system services through adaptable, platform-independent system components (FRöHLICH, 2001). It has a highly scalable architecture that is molded to accomplish the needs of applications. Moreover, EPOS supports aspects through a scenario adapter mechanism (FRöHLICH; SCHRöDER-PREIKSCHAT, 2000). Distinct combinations of system components and scenario aspects lead to different software architectures. In this context, EPOS implements a framework that defines how components can be arranged together into a functioning system. EPOS component framework, depicted in Figure 24, is realized by a C++ static metaprogram that is executed during component instantiation to adapt the component to coexist with other components as required by each application.

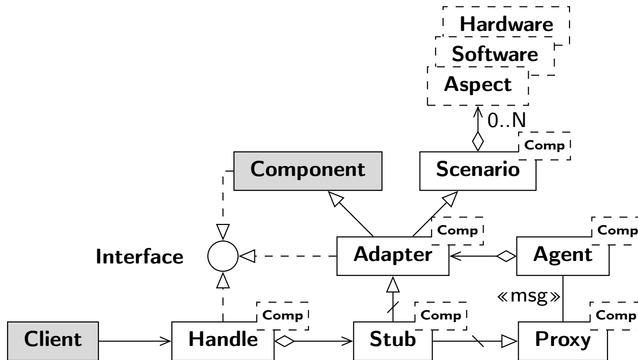


Figure 24 – EPOS component framework metaprogram.

`Stub` bridges `Handle` either with the abstraction's scenario `Adapter` or with its `Proxy`. The `Proxy` is used when the interaction between the component and the client crosses domains, such as SW/HW, user/kernel, or distinct machines on a network. Moreover, `Proxy` realizes the interface of

the abstraction it represents, forwarding method invocations to its Agent. Agent, likewise the Handle for a local scenario, forwards invocations to the abstraction's Adapter. Adapter performs scenario adaptations for the corresponding abstraction, adapting its instances to perform in the selected scenario. It applies primitives supplied by the aspect program collection Scenario (e.g. remote or hardware) to abstractions, without making assumptions about the scenario aspects represented by these primitives. Adaptations are carried out by wrapping the operations defined by the component within the enter() and leave() scenario primitives, and also by enforcing a scenario-specific semantics for creating, sharing and destroying its instances.

EPOS component framework provides most of the artifacts needed to deploy reconfigurable components such as Handles. The indirection between the client and the component in EPOS framework allows a transparent implementation of components with reconfigurable implementations. We modified EPOS so that Handle could be used as the switching point between implementations during runtime by aggregating multiple component implementations. Handle might point to any user-selected implementation while still maintaining internal references to other implementations for later usage so that only one component implementation is active as proposed in Chapter 3.

The idle thread in EPOS has the lowest priority among all threads in the system and it's used to carry the reconfiguration operation. The real-time scheduler in EPOS supports multiple scheduling policies (GRACIOLI et al., 2013) and ensures that the idle thread is only scheduled when there is no other thread ready to run. Another feature of EPOS scheduler useful for our reconfiguration strategy is that it keeps the list of threads that are ready to be executed but did not yet reached their activation periods[1] in an ordered, relative queue. Therefore, calculating the amount of time available for atomic reconfiguration activities becomes a deterministic operation that consists of subtracting the current time kept in the system timer from the time stamp of the queue's head.

6.1.3 Unified design of implementations

Hardware and software components implementations were developed from a unified description in C++ by leveraging AOP and OOP techniques. Such descriptions are obtained by isolating aspects specific to hardware and software scenarios. Aspects that differ significantly in each domain such as resource allocation and communication interface were isolated in aspects programs that are applied to the unified descriptions before compiled (MüCK; FRöHLICH,

[1] Each activation of a periodic thread is called a *job* that is released at each period.

2013). A single description can be compiled as a software implementation using regular C++ compilers and also to RTL through a HLS process and then synthesized to a target hardware platform.

The infrastructure previously presented in Figure 7a was used for handling one or more hardware and software implementations for each component. Figure 25 presents the resulting architecture for a component with a single software implementation and single hardware one, both generated from the component's unified description. The implementations are adapted to their scenarios (software and hardware) and inherit from `Component` itself so that the `Handle` can transparently dispatch method invocations to the implementation being currently deployed. The `Proxies` and `Agents` used for inter domain communication between software and hardware and the `Adapters` for scenario adaptation were not depicted in the figure for simplicity.

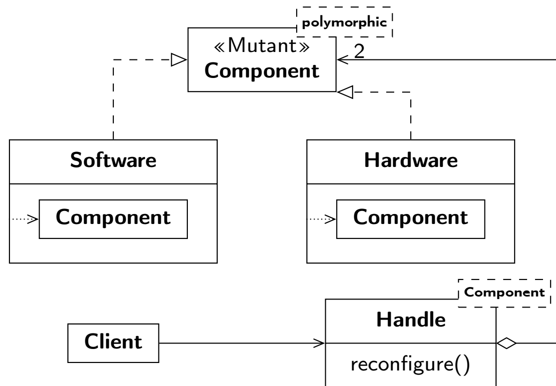


Figure 25 – Resulting architecture for a component deploying a software and hardware implementation.

As our components have a single high-level description for hardware and software implementations, their state in both domains can be captured by a group of internal variables. Methods `save_state()` and `restore_state()` are implemented by the component's developers themselves and not by the operating system. Therefore they must agree upon a common representation of each component's state so both methods can interoperate.

6.1.4 Software/hardware communication

Platform-specific specializations of `Proxies` and `Agents` support the RTSNoC based communication scheme within EPOS. In software, EPOS keeps

lists of all existing `Proxies` to hardware and `Agents` to software. Each component is associated with a unique identifier that is mapped by a resource table to a physical address in the RTSNoC. Upon a call request, this address is used to build a packet containing the target method identifier and its arguments. An EPOS Interrupt Service Routine (ISR) reads all pending packets and performs the necessary operations to transfer the packet to its destination `Agent` in a rec node. When EPOS receives a packet containing the return value of the invoked method, it forwards the packet to the corresponding blocked `Proxy`. When a packet contains data from a method call request, the information is forwarded to the dispatcher of the respective `Agent`

Method arguments are serialized on invocation by the `Proxy` and deserialized when received by the `Agent` to fit in the RTSNoC data channels. After finishing method invocation the `Agent` serializes the return value and send it through the RTSNoC for the block `Proxy` that will deserialize the data. Notice that the exchange of pointers to data structures or data structures containing pointers between hardware and software implementations is not supported due to HLS tool limitations. The tool must be able to determine statically all structures to which a pointer might point to function correctly. In our case, passing a pointer to a hardware component would mean sharing a data allocated in a CPU node's main memory that is not known at synthesis time by the HLS tool and is not accessible by the rec node.

Although RMI provides a transparent approach for method invocation across domains, it is clearly not optimized for transferring massive amounts of data as return values from the invoked methods. On components deployed in the realm of dataflow applications, RMI is meant solely to transport the commands and fixed address that are used in the specific dataflow mechanisms. For example, an application can invoke a video decoding component method whose arguments are the compressed video address in the main memory and an address to store the video output so the component can move data in and out autonomously. EPOS employs static metaprogramming techniques to provide a low overhead mechanism specialised for each type of method arguments and return values.

6.2 I/O INTERFERENCE DURING FPGA RECONFIGURATION

We designed two experiments to perform a quantitative evaluation of the interference during FPGA reconfiguration. Both experiments consist of evaluating the execution time increase of loading a bitstream from memory to the FPGA due to threads starting DMA transfers on the peripherals they use right before the start of the reconfiguration. The experiments reflect a

scenario that occur when the hardware implementation of a component is being configured in a Rec node by loading its bitstream from the DDR memory to the PCAP. At the same time, threads using I/O peripherals or components implemented in Rec nodes, from now on also referred to as peripherals, can exchange I/O data with the DDR memory.

In the experiment software architecture, exposed in Figure 26, each thread uses a different peripheral that performs DMA transfers to the DDR memory and they go to sleep one after another right after triggering data transfer operations. In parallel to the threads performing DMA, the reconfiguration of a Rec node is triggered moving a bitstream from the memory to the PCAP interface. Right after the last one goes to sleep, the FPGA reconfiguration starts i.e., the PCAP fetching the bitstream competes for shared resources with all other data streams. Also, the execution time of each peripheral DMA operation is bigger than the reconfiguration time even when there is no competition for shared hardware resources.

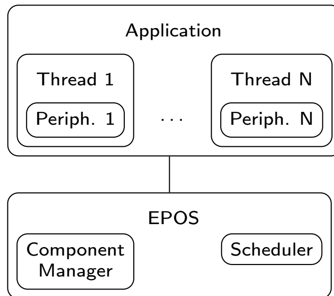


Figure 26 – Experiment software architecture. The application used and the operating system run on a CPU node of the EPOSSoC.

The heterogeneous nature of the CPU node interconnect allows us to propose two experiments to explore how the peripheral placement in the interconnect interfere with the FPGA reconfiguration. In the first experiment, the peripherals employed by each thread compete for the interconnect with the PCAP when trying to access the DDR memory. In the second experiment, PCAP and the peripherals are attached to independent interconnects and thus, share no hardware resource but the DDR memory itself. While in the first experiment the interference happens in the interconnect, in the second it happens in the DDR memory controller in both cases due to transaction scheduling policies and limited associated bandwidth. As for the bitstreams being loaded, they can be seen as hardware implementations of components requested by the application being loaded into Rec nodes due to the reconfiguration policy

previously described. Notice that the number of available AXI interfaces on Zynq's reconfigurable fabric that satisfy each experiment requirement limits the number of peripherals in both experiments i.e., two interfaces in the first experiment and four in the second.

In both experiments, we varied the peripheral's DMA controller operating frequency, its transaction burst length, and the size of the bitstream being loaded by the PCAP. 50 % of peripherals transactions were reads and the other 50 % were writes. Five burst length values were chosen, from the minimum to the maximum defined by the AXI specification: 1, 8, 32, and 256 words. For the operating frequencies, the chosen values were 100 MHz, which is a fair operating frequency for IPs implemented on mid-range FPGAs and the highest frequency the design could be synthesized for, and 10 MHz to represent slower devices. The partial bitstream used as the normalization base has 151 048 B, while a bitstream for fully reconfiguring the FPGA has 4 045 564 B. In a rough estimate, the partial bitstream reconfigures 4 % of all FPGA resources.

6.2.1 Interconnect contention

Figure 27 presents an overview of how peripherals, PCAP, and shared resources (memory and interconnect) are disposed in first experiment. Due to the placement of the peripherals in this experiment, they share an interconnect with PCAP that has access to one of the DDR memory controller ports. We quantified the impact of up to two threads using peripherals that interfere with the PCAP reconfiguration.

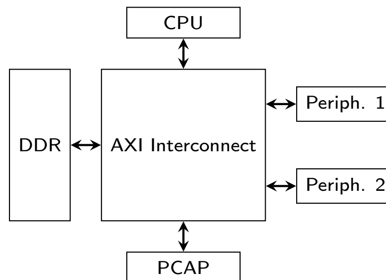


Figure 27 – Hardware architecture of the first experiment. Threads start DMA transactions in peripherals connected to the AXI interconnect as PCAP.

The AXI interconnect grants master access to peripherals following a Least Recently Granted (LRG) (XILINX, 2015d) arbitration scheme. After

performing the handshake for starting a transaction, the peripheral will gain permission to transfer a number of words according to the AXI interconnect read/write request capabilities configuration. These capabilities define the maximum number of words each peripheral can transfer per granted interconnect access. PCAP's read/write request capability is four while the capability of the peripherals used in this experiment is equal to eight. A preliminary analysis of the experiment might conclude that bigger burst lengths allow interconnect access to the peripheral for a larger share of time and interfere more in the reconfiguration process. The same argument is valid for increasing the operating frequency of the DMA controller: the peripheral will issue transactions more often generating more interference. Figure 28 shows the results concerning the time needed to load a bitstream for each of the configuration proposed by this experiment.

Examining the gathered results, for all combinations of burst length and operating frequency, the number of peripherals using the interconnect impacts the reconfiguration time. For a fixed frequency, the reconfiguration time, and thus interference, increases as the burst length grows as previously stated. The interference is more significant when the burst length is greater or equal to the peripheral read/write request capability, i.e., eight, due to the LRG arbitration scheme previously described. Also, the peripherals' operating frequency has a clear influence on reconfiguration time however, its impact inverts for burst lengths greater than or equal to eight. For smaller burst lengths, increasing the frequency increases the reconfiguration time while for bigger burst lengths, increasing the frequency decreases the reconfiguration time. Moreover, the reconfiguration time is directly proportional to the bitstream size in all cases, even with interference. When two real-time threads perform I/O operations in parallel, the reconfiguration time is 8800 % longer compared to the reconfiguration time without interference. This increase happens when both peripherals execute at 10 MHz with their DMAs issuing 256 words burst length AXI transactions.

6.2.2 Memory contention

Figure 29 presents the disposal of the peripherals deployed in the second experiment as well as the relative position of the PCAP. In this experiment was possible to quantify the interference of up to four threads using peripherals that interfere with the PCAP reconfiguration. All peripherals are connected to the DDR memory through an AXI interconnect that has access to two DDR memory controller ports while PCAP is the only active peripheral in its interconnect. Bitstream fetches performed are now only interfered when accessing

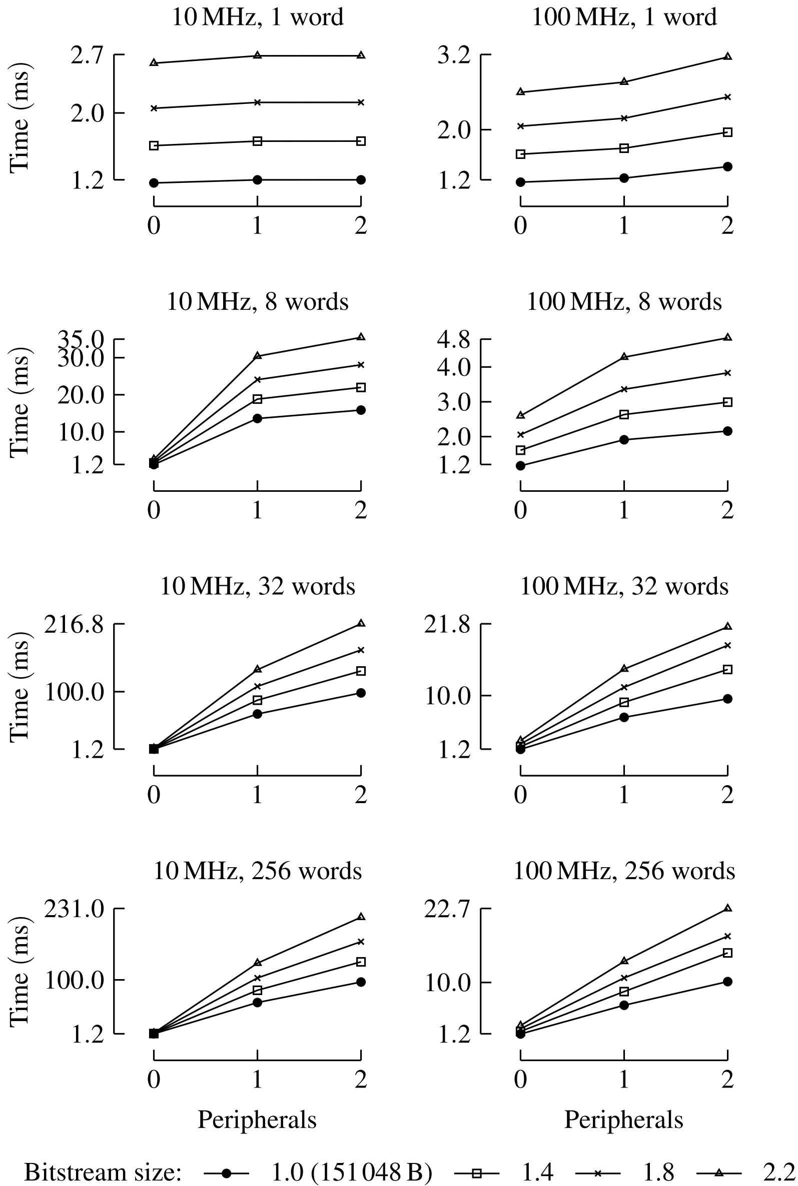


Figure 28 – Results for the interconnect contention experiment. Reconfiguration time for different normalized bitstream sizes with interference inflicted by peripherals performing DMA. Different configurations of peripheral's AXI transaction burst length, threads number, and peripheral operating frequency.

the DDR as the other peripherals are not using sharing the interconnect with the PCAP.

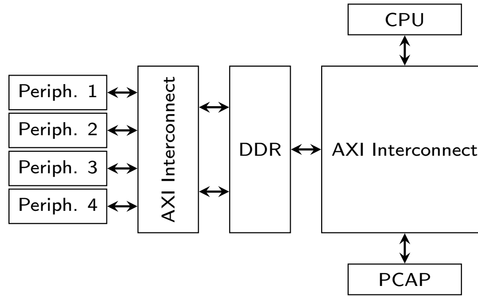


Figure 29 – Hardware architecture of the second experiment. Threads start DMA transactions in peripherals connected to the AXI interconnect isolated from PCAP but still share the DDR memory.

The results presented in Figure 30 show that, for higher frequencies, the reconfiguration time increases as the number of peripherals performing DMA grows. The smaller increase compared to the previous experiment can be explained by the DDR memory controller operating at a higher frequency (533 MHz) and being able to theoretically reach a throughput of 4270 MB/s (XILINX, 2014) while the interconnect in the previous experiment operates at 222 MHz. The biggest reconfiguration time increase happened with peripherals running at 100 MHz with their burst lengths configured to 256 for which the reconfiguration time grew 56 %. For 10 MHz there is only a small increase in reconfiguration time as the number of peripherals increase. In this experiment architecture, the burst length does not show a clear trend on how it interferes with the reconfiguration time. Moreover, it was not possible to identify the precise cause of the decrease of reconfiguration time in the transition from three to four peripherals at 100 MHz and 1 word burst length.

6.3 PABX SOC

Components that compose a digital PABX system were implemented using the proposed reconfiguration mechanisms to evaluate our approach in an industrial application with strict timing constraints. The system consists of a commutation matrix that switches connections amongst different input and output data channels. These channels are connected tone generators, tone detectors, and to phone lines through ADCs and Digital-to-Analog Converters

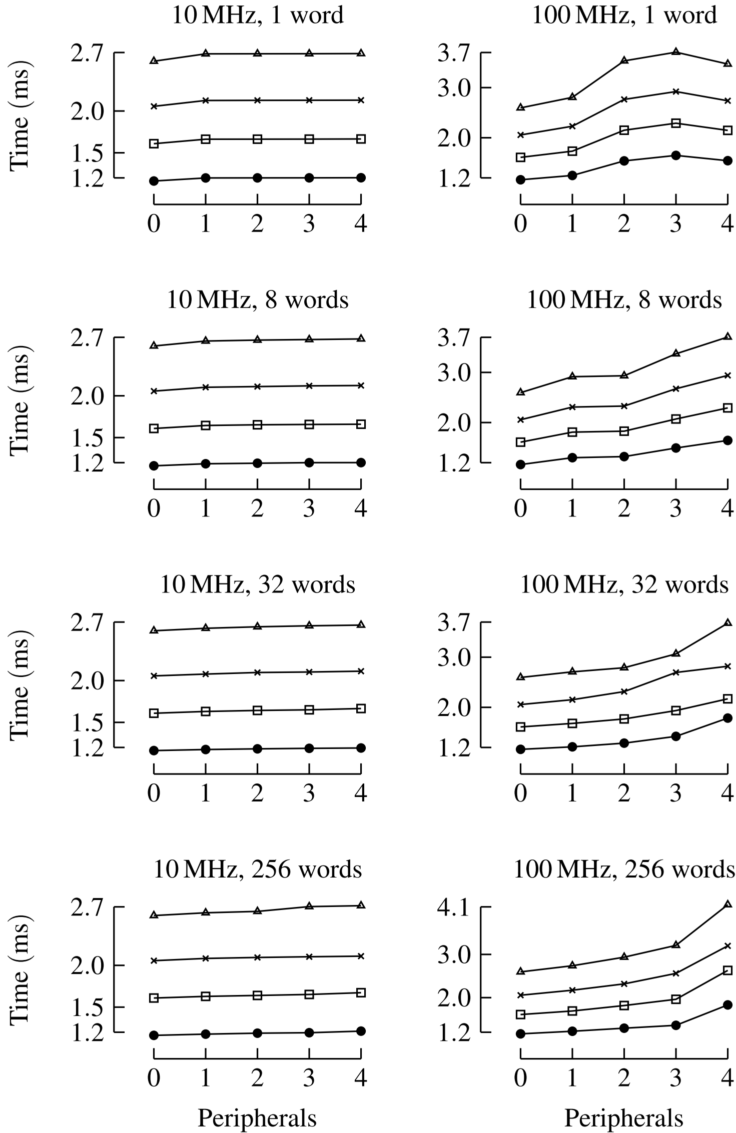


Figure 30 – Results for the memory contention experiment. Reconfiguration time for different normalized bitstream sizes with interference inflicted by peripherals performing DMA. Different configurations of peripheral's AXI transaction burst length, threads number, and peripheral operating frequency.

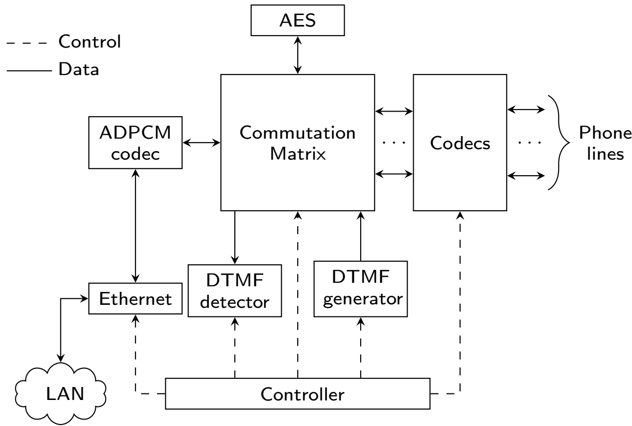
(DACs). The system also supports the transmission of phone call data through Ethernet to a Local Area Network (LAN).

A high-level diagram of the PABX system is shown in Figure 31a. The codecs interfacing the phone lines present trade-offs between compression ratio and complexity thus presenting a favorable scenario for dynamic partial reconfiguration. 4-bit Adaptive Differential Pulse-Code Modulation (ADPCM) phone line samples encapsulated in encrypted packets are received from the network. An Advanced Encryption Standard (AES) core first decrypts the packets using the 128-bit AES algorithm. The resulting data is then decoded to 16-bit Pulse-Code Modulation (PCM) samples and sent to a Dual-Tone Multi-Frequency (DTMF) detector. The DTMF detector uses the Goertzel algorithm to check if a sample frame contains particular tones. Once a tone is detected, the system controller is notified. The Ethernet MAC has fixed hardware implementation while the controller is implemented only in software. Both hardware and software implementations were generated from a higher level description of the DTMF detector, the ADPCM codec, and the AES core.

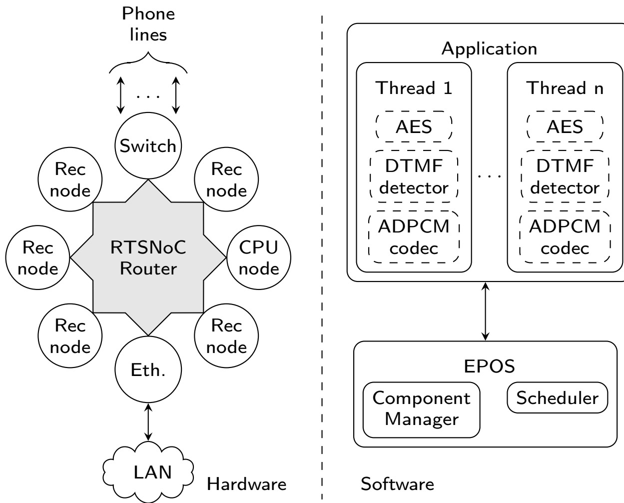
Figure 31b presents the mapping of the application on the EPOS-SOC. The commutation matrix was replaced by an RTSNoC router while the controller is implemented in the *CPU node*. Each phone line holding a conversation is a thread in EPOS; they are dynamically created according to the system load. Threads have the necessary components to perform the phone data processing, and all the components can be deployed in both hardware and software domains. Underlying the application, EPOS provides all the infrastructure mentioned in the previous chapters.

6.3.1 Reconfiguration policy

The codecs interfacing the phone lines present a data compression ratio that scales with the codec complexity (GUIDE; HERSENT; PETIT, 2002) and the resources necessary to implement it: the smaller the codec's data compression ratio, higher the bandwidth occupied by the call. As the available bandwidth limits the number of additional calls that can be carried by the PABX, it must be managed without exhausting other system resources according to system load. By sensing the bandwidth used by the codecs and the usage of hardware resources, the PABX can better adapt itself to the system load without compromising the ability to answer future calls. Such scenario is favorable for real-time dynamic reconfiguration as it is not possible to know how many calls will be held at a given time during system synthesis. Suppose a call was established initially using codec Internet Low Bitrate Codec (iLBC). If there are no available hardware resources at a given time, and a higher



(a) High level diagram.



(b) SoC implementation.

Figure 31 – PABX telephony system.

priority service is scheduled to execute in hardware, codec iLBC might be exchanged by codec G.711, which presents smaller data compression ratio. The exchange will free hardware resources, and the operating system must be able to exchange codecs without missing deadlines otherwise, the missing codec generates line noise, degrading voice quality. Table 2 illustrates the previously described reconfiguration policy used in the PABX, it focuses on balancing the bandwidth and hardware resources usage.

Table 2 – PABX reconfiguration policy.

Free bandwidth	Free hardware resources	Next calls	Reconfigure
High	High	Use iLBC	Nothing
High	Low	Use G.711	iLBC into G.711
Low	Low	Use software codecs	Nothing
Low	High	Use iLBC	G.711 into iLBC

6.3.2 Resources usage

We measured the resource utilization of the component's communication infrastructure that allows them to be deployed in hardware and communicate with software. It consists of the proxy and agent circuitry which allows hardware components to dispatch and receive method calls as presented in Chapter 3. The values in Table 3 were gathered from proxies and agents that support a single method with a 4 B argument and a 4 B return value. The estimated area is the arithmetic mean of the amount of each particular resource, weighted by its total amount available on the target device. This resulting value estimates the FPGA area required and can be used as an area estimation metric. They show a low resource utilization compared to the total resources available in the XC7Z020 SoC. Therefore, even if a component has an interface that consists of several methods, most of the hardware resources are available for implementing the component's functionality.

Our PABX case study makes use of three different reconfigurable components: ADPCM codec, DTMF detector and AES core. In this experiment, only one of the two Cortex-A9 cores is enabled as multi-core issues are not being addressed. Moreover, both L1 and L2 caches are disabled during experiment execution and bitstreams are stored in the DDR memory. The sum of the bitstream size of the three components correspond to almost 1 MB, as each of them has a partial bitstream for each rec node, the total amount of memory

Table 3 – Hardware resources utilisation for the communication infrastructure in the XC7Z020 SoC supporting the invocation of a method with a 4 B argument and a 4 B return value.

	Flip-flops	LUTs	Estimated area (%)
Proxy	311	87	0.2
Agent	348	102	0.3
XC7Z020	106 400	53 200	100

necessary to store them is nearly 3 MB. Considering that there is 512 MB of memory, the partial bitstreams occupy less than 0.6 % of the available memory.

6.3.3 Reconfiguration time

For each component, the execution time of each operation shown in Figure 17 and Figure 18 was measured. The number of processor cycles needed to execute each operation was collected using the processor's cycle count register (ARM, 2012). Next, the number of cycles was divided by the processor operating clock frequency, 667 MHz to obtain the execution time. Table 4 shows the gathered results; the first row presents the component name followed by the number of bytes of each partial bitstream containing the component's hardware implementation.

The major part of the reconfiguration process is spent on the `load_comp()` method. Although FPGA reconfiguration interfaces have improved the partial reconfiguration of coarse-grained components like ours takes much longer than simple software operations. It is important to observe that the ADPCM codec does not hold an internal state, thus its `save_state()` and `restore_state()` methods return almost immediately. Regarding the AES, its internal state consists of the encryption key it is using. For components that hold more complex internal states such as the DTMF detector, the number of cycles spent getting and setting the component state might rise. The DTMF detector has a buffer that can store hundreds of tone samples, when the buffer is filled the DTMF detection algorithm can be issued. All the tone samples held internally must be saved before reconfiguration and later restored. For the three components, the operations executed during the reconfiguration process have constant execution times and are thus, deterministic. Therefore, they can be easily incorporated in the operating system's idle thread when using a scheduler following a hard real-time policy or soft real-time policy such as the

Table 4 – Execution time of each function call in the reconfiguration process of different reconfigurable components.

	ADPCM codec (269 840 B)		DTMF detector (241 384 B)		AES (430 196 B)	
Operation	Time (μs)	Share (%)	Time (μs)	Share (%)	Time (μs)	Share (%)
`alloc_hw_res()`	10.40	0.50	10.40	0.35	10.40	0.03
`load_comp()`	2075.06	98.45	1855.29	63.15	3306.18	99.47
`lock()`	0.66	0.03	0.66	0.02	0.66	0.02
`save_state()`	0.01	0.00	1007.82	34.30	6.10	0.18
`restore_state()`	0.01	0.00	63.20	2.15	0.15	0.00
`set_domain()`	0.01	0.00	0.01	0.00	0.01	0.00
`unlock()`	0.42	0.02	0.42	0.01	0.42	0.01
Total	2086.57	100.00	2937.80	100.00	3323.92	100.00

presented PABX.

As previously mentioned, the state handling and rebinding during reconfiguration, depicted in Figure 18, must be executed atomically and depends on having enough slack time available for performing all operations. Of the three studied components, the DTMF detector is clearly the one that needs more time to execute the last reconfiguration step. For instance, adding `lock()`, `save_state()`, `restore_state()`, `set_domain()` and `unlock()` execution times, a slack time of 1.07 ms is necessary. For multimedia systems like the PABX, slack times within this dimension are quite reasonable considering that audio sampling rates are a few kHz and the processors executing the operating system run at hundreds of MHz.

6.4 NON-FUNCTIONAL TRADE-OFFS ANALYSIS

The last experiment realized was to investigate trade-offs between different components implementations that might be used to dictate reconfiguration policies adapting the system's non-functional characteristics. For instance, if there are two implementations of a given component, one that executes a given operation two times faster than the other but consumes twice the amount of energy. Suppose that initially the system is attached to an external power supply and energy consumption is not a concern: the faster implementation will be deployed. Nevertheless, if the external power supply is detached and the system now depends only on its internal batteries to function, the implementation that consumes less power might be preferred. This experiment evaluates two components, the first implements the Fast Fourier Transform Transform (FFT) algorithm and the second the natural exponential function. Despite providing general reconfiguration guidelines is out of this work's scope, we aim to show with this evaluation which kinds of trade-offs could be explored. We are not specifying when to reconfigure but showing insights that can be helpful when defining the reconfiguration policy for a given application.

6.4.1 Fast Fourier Transform

The FFT component API can compute the Discrete Fourier Transform (DFT) of an array of samples and its inverse. Behind the API, there are three implementations using the Cooley-Tukey algorithm. Two of them are software implementations, `sw_flt_fft()` and `sw_dbl_fft()`, based on the Ne10 library (DEVELOPERS, 2015) using a mixed Radix-2/Radix-4 algo-

rithm. `sw_flt_fft()` and `sw_dbl_fft()` operate on float samples and double samples respectively. The hardware version, `hw_flt_fft()`, relies on Xilinx LogiCORE FFT IP (XILINX, 2012a) that pipelines several Radix-2 butterfly processing engines to offer continuous data processing. Table 5 presents the hardware resources necessary to implement the Xilinx LogiCORE FFT IP. A DMA engine moves data in and out of the FFT IP through Zynq's Accelerator Coherency Port (ACP). ACP provides to hardware accelerators access to the DDR memory with L2 cache coherency.

Table 5 – Hardware resources to implement the FFT in the XC7Z020 SoC.

Resource	Estimation	Available	Utilization (%)
FF	15 476	106 400	15
LUT	9812	53 200	18
Memory LUT	1408	17 400	8
BRAM	14	140	10
DSP48	36	220	16

The direct transform execution time of all implementations was measured. The FFTs were performed on 1024 double-precision floating-point samples, which were typecasted to single-precision floating-point before being used in `sw_flt_fft()` and `hw_flt_fft()`. Implementing floating-point operations on an FPGA can be expensive regarding the resources required. Xilinx FFT IP converts the incoming floating-point samples to fixed-point then utilizes a fixed-point FFT internally and finally converts the results back to floating-point to achieve similar noise performance to a full floating-point FFT, with significantly fewer resources. The operations were repeated 1000 times to measure the average execution time. The CPU clock frequency is 666.666 MHz while FFT IP clock frequency is 150 MHz. The time measurements were made using the Zynq's 64-bit Global Timer (XILINX, 2015d).

The results are presented in the first column of Table 6. `hw_flt_fft()` execution time is 94 % smaller compared to `sw_dbl_fft()` when processing 1024 samples. The performance boost can be explained by the parallel implementation of the FFT butterflies coupled with the DMA engine to move to and from the IP. Moreover, the L2 cache coherency speeds up DMA fetching of samples lowering DMA memory access time.

Our following analysis was the energy consumption of Zynq SoC while executing each FFT implementation. It is not possible to measure Zynq's energy consumption in ZedBoard, only energy consumed by the whole board. To overcome this issue, the energy consumed by the board peripherals by

Table 6 – FFT implementations characteristics.

Implementation	Time (µs)	Energy (µJ)	RMSD	MAPE
`sw_dbl_fft()`	701.817	833.759	0.0	0.0
`sw_flt_fft()`	592.728	697.048	1.564×10^{-4}	9.412×10^{-5}
`hw_flt_fft()`	40.389	57.675	1.565×10^{-4}	9.423×10^{-5}

putting Zynq in low power mode and measuring the board energy consumption was measured. The gathered value, 3.132 W, was subtracted from all other measurements to obtain only Zynq energy consumption. The second column of Table 6 presents the measured consumed energy values for ZedBoard executing the three FFT implementations in a loop. The energy was calculated by multiplying the measured power with the execution time. While software implementations do not utilize the reconfigurable resources, the FPGA consumes only static power. As `hw_flt_fft()` depends on a hardware IP, its dynamic consumption accounted in the power measurement. Nevertheless, `hw_flt_fft()` is much faster than `sw_dbl_fft()` and `sw_flt_fft()` resulting in a smaller energy consumption. Compared to `sw_dbl_fft()`, `hw_flt_fft()` consumes 93 % less energy.

Moreover, the accuracy of `sw_flt_fft()` and `hw_flt_fft()` was compared with the higher precision FFT implementation, `sw_dbl_fft()`. To estimate accuracy the Root-Mean-Square Deviation (RMSD) and the Mean Absolute Percent Error (MAPE) of the results data sets was calculated taking `sw_dbl_fft()` results as the reference data set. The third and fourth columns of Table 6 present the calculated results. `hw_flt_fft()` is slightly less accurate than `sw_flt_fft()` due to its fixed-point processing phase nevertheless, both are considerably less accurate than `sw_dbl_fft()`.

`hw_flt_fft()` is arguably the implementation that better performs both in execution time and energy consumption but, it depends on hardware resources that might not always be available (other components might be occupying the reconfigurable fabric). In such cases, `sw_flt_fft()` might be used if its lower precision compared to `sw_dbl_fft()` is tolerable by the application due its to its smaller energy consumption.

6.4.2 Natural Exponential

Table 8 presents the profiling of five different implementations of the natural exponential function. `hw_exp()` utilizes Xilinx's single precision

Floating-Point Operator IP (XILINX, 2012b) clocked at 100 MHz on the FPGA. The FPGA resources necessary to implement the Floating-Point Operator IP are presented in Table 7. The function argument is written to a memory mapped register by the CPU, processed by the IP and read back. `exp()` and `expf()` are GNU's libm (FOUNDATION, 2014) implementation, they operate on double and single precision floating-point arguments respectively. `fastexp()` and `fasterexp()` are approximate implementations present in fastapprox library (MINEIRO, 2014), both operate on single precision arguments.

Table 7 – Hardware resources needed to implement the exponental function in the XC7Z020 SoC.

Resource	Estimation	Available	Utilization (%)
FF	2199	106 400	2
LUT	1762	53 200	3
Memory LUT	171	17 400	1
BRAM	3	140	2
DSP48	7	220	3

Each implementation processed 1000 arguments ranging from -32.0 to 32.0. The results presented in the first column of Table 8 show that the faster and approximated implementation is three times faster than `exp()`, the only implementation operating on double precision arguments. The communication overhead between software and hardware overwhelms the execution time of exponential operation itself. Such small granularity operations are faster in software when compared to its hardware counterparts running at smaller clock frequencies.

Table 8 – Natural exponential implementations characteristics.

Implementation	Time (µs)	Energy (µJ)	RMSD	MAPE
`exp()`	0.357	0.450	0.0	0.0
`expf()`	0.345	0.435	4.13×10^{6}	3.116×10^{-7}
`fastexp()`	0.191	0.241	2.28×10^{8}	2.280×10^{-5}
`fasterexp()`	0.117	0.152	1.50×10^{11}	1.530×10^{-2}
`hw_exp()`	1.280	1.567	4.13×10^{6}	3.108×10^{-7}

The accuracy of all natural exponentiation implementations was evaluated by comparing them with the most accurate implementation, `exp()`, operating with double precision floating-point inputs and outputs. Table 8 contains the RMSD and the MAPE of each implementation. As they are based on approximated calculations, `fastexp()` and `fasterexp()` present a bigger RMSD and MAPE compared to other version, being thus less accurate. `hw_exp()` does not fully comply with IEEE Standard for Floating-Point Arithmetic (IEEE..., 2008) as the deviations provide a better trade-off between resources against functionality (XILINX, 2012b). That is why its MAPE is not equal to `expf()` MAPE.

The energy consumption of the chosen natural exponential implementation is inversely proportional to its accuracy except for `hw_exp()`; it underperforms all other implementations in energy consumption and execution time. However, it still might be a valuable implementation option when the CPU is under heavy load or with a higher priority task scheduled, and the operating system wishes to transfer part of this load to the FPGA.

6.4.3 Discussion

A reconfiguration policy for a system using an FFT can prioritize its deployment in hardware as it is faster (94 % decrease in execution time) and consumes less energy (93 % less). Nevertheless, it can switch to software whenever the hardware resources are required by critical tasks in the application. As for the natural exponential, it has multiple software implementations in which the energy consumption and execution time decrease as the precision decreases. In a battery-operated system, these multiple levels can allow the reconfiguration policy, for example, to use a less precise implementation every time the battery charge reaches a certain level to increase the battery life-span.

7 CONCLUSION

Static partitioning of functionalities in embedded systems leads to suboptimal choices of implementations for the components deployed in target applications. As partitioning choices are made in design time, it is not possible to quantify or explore the variations suffered by the system and its environment during runtime. Future embedded system applications will need to reason on its current state and dynamically leverage hardware and software resources through self-adaptive architectures able to react to unpredictable requirements and workloads.

Due to theirs inherent parallelism, and energy efficiency, FPGAs are versatile substrates able to deliver implementations with different trade-offs when compared to software-only implementations. Its flexibility allows applications to explore different partitioning of components during runtime by means of dynamic reconfiguration. Nevertheless, current runtime support systems for FPGA usage do not provide uniform interfaces for reconfigurable abstractions allowing the exploration of such dynamic capabilities without the supervision of the application developer. To cope with dynamic and complex application scenarios, reconfiguration details should not be managed by the application but by underlying software layers with minimum interference on the rest of the system.

This work presented a transparent framework for reconfigurable computing geared towards the application programmer. Reconfigurable components interfaces may be realized through many different implementations, ranging from high-quality software versions to software approximations, cloud offloaders, and hardware accelerators. While the syntax and semantics of reconfigurable components interface is preserved across the different implementations, the system may at any time pick any of the implementations that suits better for the current execution context. The framework manages the whole reconfiguration process and ensures that it can be used in critical systems without interfering in its time constraints. With our reconfiguration mechanism, a task set schedulable on a reconfigurable fabric large enough to accommodate at the same time all hardware components used by its tasks, will still be schedulable on a smaller reconfigurable fabric where only some components can be simultaneously instantiated in hardware. The remaining components are momentarily deployed in software without compromising to the tasks requirements since all activities pertaining reconfiguration are performed within the slack time and made aware of I/O interference.

Our work presented a quantitative analysis of the interference on FPGA reconfiguration execution time generated by system peripherals performing

I/O operations. Reconfiguration time rises significantly due to hardware resource sharing and if not properly isolated from interference sources, can disrupt the timing constraints of other activities performed by the system. Experiments showed that reconfiguration time can rise up to 8800 % when other peripherals are active. The analysis considered peripherals synthesized in an FPGA platform with two variable characteristics: peripheral operating frequency and transaction burst length.

To show the feasibility of isolating the reconfiguration process on system idle time, we evaluated the reconfiguration process using a partial implementation of a PABX system in EPOSSoC, a flexible platform for SoC implementation in FPGAs. We investigated the reconfiguration process of three reconfigurable components with a software and hardware implementation that integrate the PABX: an ADPCM codec, a DTMF detector and an AES cryptographic standard core. The time necessary to implement each step of component reconfiguration is reasonably small and fits the idle time available in applications such as the PABX. For instance, the idle time necessary to perform the reconfiguration atomic operation was of 1.07 ms for the DTMF detector which was the component with the largest internal state.

We demonstrated our approach with a set of components implementing mathematical functions, and showed how the framework can help applications trade-off quality for energy efficiency and performance. When compared with their highest quality implementation, the components we studied used less energy and executed faster in exchange for a small output quality degradation. For the FFT component it is possible to trade implementation precision for energy efficiency (up to 93 % savings) and performance (94 % decrease in execution time) when there are available hardware resources. As for the natural exponential, only with software implementations it is possible to reduce execution time in 72 % and the energy consumption in 66 % if the application can tolerate a degradation in precision.

7.1 LIMITATIONS

This work does not delivers general guidelines for when to perform a reconfiguration or what components should be reconfigured or even which implementation should be used. We believe that such questions can be answered with a cross-layer sensing approach where sensors are placed in different levels of the embedded system stack, i.e., from hardware devices up to software layers. This sensing approach combined with a set of policies for adaptation tailored to each application allow a holistic reasoning and acting upon the system.

Our current framework currently only comprises RMI as a communication mechanism for components in different domains which can have a limited throughput. Applications that demand the rapid flow of massive amounts of data might be better served by memory sharing mechanisms to deal with the communication between hardware and software components. Another approach would be the integration of DMA-based mechanisms into the framework as a more adequate solution for dataflow applications.

7.2 FUTURE WORK

As future work, we will investigate how our framework might help to cope with multiple design objectives such as dependability, efficiency, and critical operation by providing a transparent approach to FPGA reconfiguration. We will focus the usage of sensors (e.g. intra-chip sensors, PMU data, software counters) and policies to guide the reconfiguration process and to choose when and which components need to be reconfigured based on the system requirements. We plan on investigating how to transparently integrate the sensors and how to express the reconfiguration policies in a flexible and extensible way dictated by the user and smoothly incorporated into the framework infrastructure. Finally, we will expand our current work on the component reconfiguration policy to account for different system optimization goals at runtime such as energy savings, mitigation of silicon aging, and critical performance.

Subsequent works will also focus on evaluating monitoring mechanisms used to infer the usage of I/O channels in the system used to guarantee a reconfiguration process without I/O interference. The evaluation will lead to heuristics to reduce I/O usage by other peripherals prior to FPGA reconfiguration in order to avoid the sharing of hardware resources to mitigate interference.

BIBLIOGRAPHY

ABEL, N. Design and implementation of an object-oriented framework for dynamic partial reconfiguration. In: **Proc. International Conference on Field Programmable Logic and Applications (FPL'10)**. [S.l.: s.n.], 2010. p. 240–243. ISSN 1946-147X.

AHMADINIA, A. et al. Task scheduling for heterogeneous reconfigurable computers. In: **Proc. Symposium on Integrated Circuits and Systems Design (SBCCI'04)**. [S.l.: s.n.], 2004. p. 22–27.

AL., C. B. et. The erlangen slot machine: Increasing flexibility in FPGA-based reconfigurable platforms. In: **Proc. International Conference on Field-Programmable Technology (FPT'05)**. [S.l.: s.n.], 2005. p. 37–42.

AL., D. A. et. Achieving programming model abstractions for reconfigurable computing. **IEEE Transactions on Very Large Scale Integration Systems**, v. 16, n. 1, p. 34–44, jan. 2008. ISSN 1063-8210.

AL., D. G. et. Operating system for runtime reconfigurable multiprocessor systems. **International Journal of Reconfigurable Computing**, p. 16, 2011.

AL., R. B. et. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In: **Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'06)**. [S.l.: s.n.], 2006. p. 259–264.

AL., X. I. et. R3TOS: A novel reliable reconfigurable real-time operating system for highly adaptive, efficient, and dependable computing on FPGAs. **IEEE Transactions on Computers**, v. 62, n. 8, p. 1542–1556, ago. 2013. ISSN 0018-9340.

AL., Y. W. et. SPREAD: A streaming-based partially reconfigurable architecture and programming model. **IEEE Transactions on Very Large Scale Integration Systems**, v. 21, n. 12, p. 2179–2192, 2013. ISSN 1063-8210.

ALTERA. **Altera Configuration Handbook**. [S.l.], 10 2008.

ANSEL, J. et al. PetaBricks: A language and compiler for algorithmic choice. **ACM SIGPLAN Notices**, ACM, New York, NY, USA, v. 44, p. 38–49, 2009. ISSN 0362-1340.

ARM. **AMBA Specification (Rev 2.0)**. [S.l.], 1999.

ARM. **CoreLink QoS-301 Network Interconnect Advanced Quality of Service**. [S.l.], 9 2011.

ARM. **Cortex-A9: Technical Reference Manual**. [S.l.], 2012.

AVNET. **(Zynq Evaluation and Development) Hardware User's Guide**. [S.l.], 1 2014.

BAEK, W.; CHILIMBI, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. **SIGPLAN Notices**, v. 45, p. 198–209, June 2010. ISSN 0362-1340.

BAK, S. et al. Real-time control of I/O COTS peripherals for embedded systems. In: **Proc. Real-Time Systems Symposium (RTSS'09)**. [S.l.: s.n.], 2009. p. 193–203. ISSN 1052-8725.

BEREJUCK, M. D. **Network-on-Chip with latency predictability for Real-Time Systems**. 195 p. Thesis (Ph.D.) — Federal University of Santa Catarina, Florianópolis, 2015.

BEREJUCK, M. D.; FRöHLICH, A. A. Evaluation of silicon consumption for a connectionless network-on-chip. **International Journal of Advanced Studies in Computer Science and Engineering**, v. 3(11), p. 1–11, 2014.

BERGAMASCHI, R. A.; COHN, J. The A to Z of SoCs. In: **Proc. IEEE/ACM International Conference on Computer Aided Design (ICCAD)'02.** [S.l.: s.n.], 2002. p. 791–798. ISSN 1092-3152.

CANCIAN, R. L. **Um Modelo Evolucionário Multiobjetivo para Exploração do Espaço de Projeto em Sistemas Embarcados**. Thesis (Ph.D.) — Federal University of Santa Catarina, 2011.

CHO, Y.-S.; CHOI, E.-J.; CHO, K.-R. Modeling and analysis of the system bus latency on the SoC platform. In: **Proc. International Workshop on System-level Interconnect Prediction (SLIP'06)**. New York, NY, USA: ACM, 2006. (SLIP '06), p. 67–74. ISBN 1-59593-255-0.

CLEMENTE, J. A.; RESANO, J.; MOZOS, D. An approach to manage reconfigurations and reduce area cost in hard real-time reconfigurable systems. **ACM Transactions on Embedded Computing Systems**, ACM, New York, NY, USA, v. 13, n. 4, p. 90:1–90:24, mar. 2014. ISSN 1539-9087.

COMPTON, K.; HAUCK, S. Reconfigurable computing: A survey of systems and software. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 34, n. 2, p. 171–210, jun. 2002. ISSN 0360-0300.

CZARNECKI, K.; EISENECKER, U. W. **Generative Programming: Methods, Tools, and Applications**. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 0-201-30977-7.

DALLY, W.; TOWLES, B. **Principles and Practices of Interconnection Networks**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 0122007514.

DEVELOPERS, P. N. **Project Ne10**. 2015. [Online] Available: https://github.com/projectNe10/Ne10.

DYNAMICALLY Reconfigurable Hardware/Software Mobile Agents. **Design Automation for Embedded Systems**, Springer US, v. 18, n. 1-2, 2014. ISSN 0929-5585.

ESTRIN, G. Organization of computer systems: The fixed plus variable structure computer. In: **Proc. Western Joint IRE-AIEE-ACM Computer Conference**. New York, NY, USA: ACM, 1960. p. 33–40.

FALAKI, H. **Automating Personalized Battery Management on Smartphones**. Thesis (Ph.D.) — UCLA, 2012.

FOUNDATION, F. S. **GNU C Library**. 2014. [Online] Available: http://www.gnu.org/software/libc/.

FRöHLICH, A. A. **Application-Oriented Operating Systems**. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001. (GMD Research Series, 17).

FRöHLICH, A. A.; SCHRöDER-PREIKSCHAT, W. Scenario adapters: Efficiently adapting components. In: **Proc. World Multiconference on Systemics, Cybernetics and Informatics**. Orlando, USA: [s.n.], 2000.

GRACIOLI, G. **ELUS: Projeto e Implementação de um Mecanismo de Reconfiguração Dinâmica de Software para Sistemas Profundamente Embarcados**. 98 p. Dissertation (M.Sc.) — Federal University of Santa Catarina, Florianópolis, 2009.

GRACIOLI, G. et al. A survey on cache management mechanisms for real-time embedded systems. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 48, n. 2, p. 32:1–32:36, nov 2015. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/2830555>.

GRACIOLI, G. et al. Implementation and evaluation of global and partitioned scheduling in a real-time OS. **Real-Time Systems**, v. 49, n. 6, 2013. ISSN 0922-6443.

GRACIOLI, G.; FRöHLICH, A. A. ELUS: A dynamic software reconfiguration infrastructure for embedded systems. In: **Proc. International Conference on Telecommunications (ICT'10)**. [S.l.: s.n.], 2010. p. 981–988.

GRACIOLI, G.; FRöHLICH, A. A. An embedded operating system API for monitoring hardware events in multicore processors. In: **Proc. Workshop on Hardware-Support for Parallel Program Correctness**. Porto Alegre, Brazil.: [s.n.], 2011.

GUIDE, D.; HERSENT, O.; PETIT, J. P. **IP Telephony**. First. USA: Prentice Hall, 2002.

HARTENSTEIN, R. Reconfigurable computing: From FPGAs to hardware/software codesign. In: . New York, NY: Springer New York, 2011. chap. The Relevance of Reconfigurable Computing, p. 7–34. ISBN 978-1-4614-0061-5.

HOELLER, L. F. W. A. J.; FRöHLICH, A. A. A hierarchical approach for power management on mobile embedded systems. In: **Prc. IFIP Working Conference on Distributed and Parallel Embedded Systems**. Braga, Portugal: [s.n.], 2006. p. 265–274. ISBN 0-387-39361-7.

HOFFMANN, H. et al. Dynamic knobs for responsive power-aware computing. **SIGARCH Computer Architecture News**, ACM, New York, NY, USA, v. 39, p. 199–212, 2011. ISSN 0163-5964.

IEEE Standard for Floating-Point Arithmetic. **IEEE Std 754-2008**, p. 1–70, Aug 2008.

ISMAIL, A.; SHANNON, L. FUSE: Front-end user framework for O/S abstraction of hardware accelerators. In: **IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'11)**. [S.l.: s.n.], 2011. p. 170–177.

JUNIOR, A. S. H. **Gerenciamento do Consumo de Energia Dirigido pela Aplicação em Sistemas Embarcados**. Dissertation (M.Sc.) — Federal University of Santa Catarina, Florianópolis, 2007.

KIM, S.; IM, C.; HA, S. Schedule-aware performance estimation of communication architecture for efficient design space exploration. **IEEE**

Transactions on Very Large Scale Integration Systems, v. 13, n. 5, p. 539–552, maio 2005. ISSN 1063-8210.

LACHENMANN, A. et al. Meeting lifetime goals with energy levels. In: **Conference on Embedded Networked Sensor Systems (SenSys'07)**. [S.l.: s.n.], 2007. ISBN 978-1-59593-763-6.

LEE, E. A.; SESHIA, S. A. **Introduction to Embedded Systems - A Cyber Physical Systems Approach**. 2. ed. [S.l.: s.n.], 2015. ISBN 978-1-312-42740-2.

LI, Y. et al. Dynamically reconfigurable hardware with a novel scheduling strategy in energy-harvesting sensor networks. **IEEE Sensors Journal**, v. 13, n. 5, p. 2032–2038, maio 2013. ISSN 1530-437X.

LIMITED eCosCentric. **Embedded Configurable Operating System (eCos)**. 2016. [Online] Available: http://ecos.sourceware.org.

LUBBERS, E.; PLATZNER, M. ReconOS: Multithreaded programming for reconfigurable computers. **ACM Transactions on Embedded Computing Systems**, ACM, New York, NY, USA, v. 9, n. 1, p. 8:1–8:33, out. 2009. ISSN 1539-9087.

MARCONDES, H. **Uma Arquitetura de Componentes Híbridos de Hardware e Software para Sistemas Embarcados**. 92 p. Dissertation (M.Sc.) — Federal University of Santa Catarina, Florianópolis, 2009.

MARCONDES, H.; FRöHLICH, A. A. Analysis, architectures and modelling of embedded systems: Proc. IFIP international embedded systems symposium (IESS'09). In: ____. [S.l.]: Springer, 2009. chap. A Hybrid Hardware and Software Component Architecture for Embedded System Design, p. 259–270. ISBN 978-3-642-04284-3.

MARTINS, V. M. G. et al. A TMR strategy with enhanced dependability features based on a partial reconfiguration flow. In: **Proc. IEEE Computer Society Annual Symposium on VLSI**. [S.l.: s.n.], 2015. p. 161–166. ISSN 2159-3469.

MARWEDEL, P. **Embedded System Design**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN 1402076908.

MICHELI, G. D. et al. Networks on chips: from research to products. In: **Proc. Design Automation Conference (DAC'10)**. [S.l.: s.n.], 2010. p. 300–305.

MINEIRO, P. **fastapprox software library**. 2014. [Online] Available: https://code.google.com/p/fastapprox/.

MüCK, T.; FRöHLICH, A. Towards unified design of hardware and software components using C++. **IEEE Transactions on Computers**, v. 63, n. 11, p. 2880–2893, nov. 2013. ISSN 0018-9340.

MüCK, T. R. **Projeto Unificado de Componentes em Hardware e Software para Sistemas Embarcados**. 137 p. Dissertation (M.Sc.) — Federal University of Santa Catarina, Florianópolis, 2013.

OSTROWSKI, K. et al. Programming with live distributed objects. In: **Proc. European Conference on Object-Oriented Programming (ECOOP'08)**. [S.l.: s.n.], 2008. p. 463–489.

PANT, A.; GUPTA, P.; SCHAAR, M. van der. AppAdapt: Opportunistic application adaptation to compensate hardware variation. **IEEE Transactions on Very Large Scale Integration Systems**, v. 20, n. 11, p. 1986–1996, nov. 2012. ISSN 1063-8210.

PASRICHA, S.; DUTT, N. **On-Chip Communication Architectures: System on Chip Interconnect**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN 012373892X, 9780123738929.

PELLIZZONI, R.; CACCAMO, M. Impact of peripheral-processor interference on WCET analysis of real-time embedded systems. **IEEE Transactions on Computers**, v. 59, n. 3, p. 400–415, mar. 2010. ISSN 0018-9340.

POLPETA, F. V. **Uma Estratégia para a Geração de Sistemas Embutidos baseada na Metodologia Projeto de Sistemas Orientados à Aplicação**. Dissertation (M.Sc.) — Federal University of Santa Catarina, Florianópolis, 2006.

POLPETA, F. V.; FRöHLICH, A. A. On the automatic generation of soc-based embedded systems. In: **Proc. IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)'05**. [S.l.: s.n.], 2005. ISBN 0-7803-9402-x.

POP, P. Embedded systems design: Optimization challenges. In: **Proc. Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)'05**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 16–16.

RAHIMI, A. et al. Task scheduling strategies to mitigate hardware variability in embedded shared memory clusters. In: **Proc. Design Automation Conference (DAC)'15**. New York, NY, USA: ACM, 2015. (DAC'15), p. 152:1–152:6. ISBN 978-1-4503-3520-1.

REIS, J. G.; FRöHLICH, A. A.; HOELLER, A. J. On the FPGA dynamic partial reconfiguration interference on real-time systems. In: **Proc. Brazilian Symposium on Computing Systems Engineering (SBESC'15)**. [S.l.: s.n.], 2015. p. 110–115.

REIS, J. G.; FRöHLICH, A. A.; WANNER, L. A Framework for Dynamic Real-Time Reconfiguration. In: **Euromicro Conference on Digital System Design (DSD'15)**. [S.l.: s.n.], 2015.

REIS, J. G.; WANNER, L. F.; FRöHLICH, A. A. X-Ware: Mutant computing substrates. In: **Proc. IEEE International Symposium on Rapid System Prototyping (RSP'15)**. Amsterdam: [s.n.], 2015.

REIS, T. de A. **Suporte de Sistema Operacional para Reconfiguração Dinâmica de Componentes de Hardware para Sistemas Embarcados**. 65 p. Dissertation (M.Sc.) — Federal University of Santa Catarina, Florianópolis, 2010.

REIS, T. de A.; FRöHLICH, A. A. Operating system support for difference-based partial hardware reconfiguration. In: **Proc. International Symposium on Rapid System Prototyping (RSP'09)**. [S.l.: s.n.], 2009. p. 75–80. ISSN 1074-6005.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. **IEEE Design Test of Computers**, v. 18, n. 6, p. 23–33, nov. 2001. ISSN 0740-7475.

SANTOS, D. M. **API Multiplataforma para Aplicações Multimídia Embarcadas**. 84 p. Dissertation (M.Sc.) — Federal University of Santa Catarina, Florianópolis, 2010.

SARMA, S.; DUTT, N. FPGA emulation and prototyping of a cyberphysical-system-on-chip (cpsoc). In: **Proc. IEEE International Symposium on Rapid System Prototyping (RSP'14)**. [S.l.: s.n.], 2014. p. 121–127. ISSN 2150-5500.

SORBER, J. et al. Eon: a language and runtime system for perpetual systems. In: **Conference on Embedded Networked Sensor Systems (SenSys'07)**. [S.l.: s.n.], 2007. ISBN 978-1-59593-763-6.

SPRUNT, B. The basics of performance-monitoring hardware. **IEEE Micro**, v. 22, n. 4, p. 64–71, jul. 2002. ISSN 0272-1732.

STEIGER, C.; WALDER, H.; PLATZNER, M. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. **IEEE Transactions on Computers**, v. 53, n. 11, p. 1393–1407, nov. 2004. ISSN 0018-9340.

VOSS, M. J.; EIGEMANN, R. High-level adaptive program optimization with ADAPT. **SIGPLAN Notices**, ACM, New York, NY, USA, v. 36, n. 7, p. 93–102, jun. 2001. ISSN 0362-1340.

WALDER, H.; PLATZNER, M. Reconfigurable hardware operating systems: From design concepts to realizations. In: **Proc. International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA'03)**. [S.l.]: CSREA Press, 2003. p. 284–287.

WANNER, L.; SRIVASTAVA, M. ViRUS: Virtual function replacement under stress. In: **Proc. USENIX Conference on Power-Aware Computing and Systems (HotPower'14)**. [S.l.]: USENIX, 2014. (HotPower'14).

WEISSEL, A.; BEUTEL, B.; BELLOSA, F. Cooperative I/O: A novel I/O semantics for energy-aware applications. **SIGOPS Operating Systems Review**, ACM, New York, NY, USA, v. 36, n. SI, p. 117–129, dez. 2002. ISSN 0163-5980.

WILHELM, R.; AL. et. The worst-case execution-time problem-overview of methods and survey of tools. **ACM Transactions of Embedded Computing Systems**, ACM, New York, NY, USA, v. 7, p. 36:1–36:53, maio 2008. ISSN 1539-9087.

XILINX. **LogiCORE IP Fast Fourier Transform**. [S.l.], 7 2012.

XILINX. **LogiCORE IP Floating-Point Operator**. [S.l.], 1 2012.

XILINX. **Xilinx Software Development Kit (SDK) User Guide: System Performance Analysis**. [S.l.], 11 2014.

XILINX. **7 Series FPGAs Configuration: User Guide**. [S.l.], 6 2015.

XILINX. **AXI HWICAP v3.0: LogiCORE IP Product Guide**. [S.l.], 11 2015.

XILINX. **Vivado Design Suite User Guide: Partial Reconfiguration**. [S.l.], 4 2015.

XILINX. **Zynq-7000 All Programmable SoC: Technical Reference Manual**. [S.l.], 2 2015.