Run-time Reconfigurable RTOS for Reconfigurable Systems-on-Chip

Dissertation

A thesis submited to the
Faculty of Computer Science, Electrical Engineering and Mathematics
of the
University of Paderborn
and to the
Graduate Program in Electrical Engineering (PPGEE)
of the
Federal University of Rio Grande do Sul

in partial fulfillment of the requirements for the degree of
Dr. rer. nat.
and
Dr. in Electrical Engineering

Marcelo Götz

November, 2007

Supervisors:

Prof. Dr. rer. nat. Franz J. Rammig, University of Paderborn, Germany

Prof. Dr.-Ing. Carlos E. Pereira, Federal University of Rio Grande do Sul, Brazil

Public examination in Paderborn, Germany

Additional members of examination committee:

Prof. Dr. Marco Platzner

Prof. Dr. Urlich Rückert

Dr. Mario Porrmann

Date: April 23, 2007

Public examination in Porto Alegre, Brazil

Additional members of examination committee:

Prof. Dr. Flávio Rech Wagner

Prof. Dr. Luigi Carro

Prof. Dr. Altamiro Amadeu Susin

Prof. Dr. Leandro Buss Becker

Prof. Dr. Fernando Gehm Moraes

Date: October 11, 2007

Acknowledgements

This PhD thesis was carried out, in his great majority, at the Department of Computer Science, Electrical Engineering and Mathematics of the University of Paderborn, during my time as a fellow of the working group of Prof. Dr. Franz J. Rammig. Due to a formal agreement between Federal University of Rio Grande do Sul (UFRGS) and University of Paderborn, I had the opportunity to receive a bi-national doctoral degree. Therefore, I had to accomplished, in addition, the PhD Graduate Program in Electrical Engineering of UFRGS. So, I hereby acknowledge, without mention everyone personally, all persons involved in making this agreement possible.

I wish to, first, express my deep gratitude to my supervisor Prof. Dr. Franz Josef Rammig for supervising, advising and tutoring me during my whole PhD journey. His understanding, patience, knowhow and constant encouragements were essential to bring me to the success of finishing this work. In the same way, I wish to thank specially my co-supervisor, Prof. Dr. Carlos Eduardo Pereira, for his equal support, even before starting this PhD thesis. The many fruitful talks, in which he shared with me his knowledge and experience, had strong influence in my decision of being a researcher and starting this PhD.

I am also glad of having being part of Prof. Rammig's working group, of having the opportunity to work on different research projects, to advise Bachelor and Master Students, and to supervise students during labs of his lectures. Furthermore, I am glad of having many chances for fruitful talks, discussions, brainstorming, that I had with my colleagues in the great atmosphere of the working group. All these facts together, have positively contributed to the success of this work.

My thanks go also to both examination committees: in Germany composed by Prof. Dr. Marco Platzner, Prof. Dr. Urlich Rückert, Dr. Mario Porrmann; and in Brazil composed by Prof. Dr. Flávio Rech Wagner, Prof. Dr. Luigi Carro, Prof. Dr. Altamiro Amadeu Susin, Prof. Dr. Leandro Buss Becker, Prof. Dr. Fernando Gehm Moraes. They, together with my both supervisors, spent their times reading this thesis and provided me a positive feedback during the PhD-defense.

I wish also to thank all my former colleagues, which helped be somehow during my PhD research. Their different points of view, wide range of interests and knowledge, were very important to improve the level of my work. My particular thanks go to Dr. Achim Rettberg, for spending his time in listening and discussing my research topic, and for offering his partnership and friendship, which was always pushing me forward in several moments. In the same way, I wish to thank Florian Dittmann, who shared with me his great knowledge in Reconfigurable Computing. With these two colleagues, I had several publications, which were extremely important in my research. Thank you both for your cooperation.

Further, I whish to extend my thankfulness to all my other colleagues: Dr. Sabina Rips, Tales Heimfarth for several and fruitful discussions, Yuhong Zhao, Katharina Stahl, Peter Janacik, Timo Kersten, Simon Oberthuer, Norma Montealegre, Dr. Carsten Böke, Dr. Martin Kardos, Dr. Klaus Danne, Dr. Stefan Ihmor, and Prof. Dr. Christophe Bobda. I wish to thank specially Dalimir Orfanus, which with his friendship gave me strong support in my last part of my PhD journey.

I must not forget my working students, Stefan Finke and MSc. Tao Xie, which helped me in implementing an important part of my concepts. Without them, this thesis would take longer to be finished.

I will never forget my friend Arne Rolf Spiller, who had crucial importance in my first years in Germany. With his unconditional friendship and support, I surpassed my self on several circumstances. He and his parents, Erwin and Ursula, are my second family now. Thank you for being so receptive and lovely.

Finally, I wish to thank my sister Fabiana and my loving mother Antônia Elvira, which pushed me forward in my studies, since I started in the school until today. Last, but not least, my lovely thanks go to Adriana Aparecida Paz, my girlfriend, who never abandoned me, even having an ocean between us. She was very important for the success of this work, due to his patience and support, and especially for her comprehensiveness and loveliness.

November, 2007.

Abstract

Embedded systems are massively present in our lives and they are becoming omnipresent. This has demanded strong efforts in research for providing new solutions for the challenges faced in the design of such systems. For instance, the requirements of high computational performance and flexibility of the contemporary embedded systems are continuously increasing. A single architecture must be able to support, in certain cases, different kind of applications with different requirements which can start asynchronously and dynamically (changing environments). Reconfigurable computing seems to be a potential paradigm for these scenarios as it can provide flexibility and high computational performance for modern embedded systems. Of especial interest are those architectures where a microprocessor is tightly connected with a reconfigurable hardware (hybrid platform), constituting a so called reconfigurable System-on-Chip (RSoC). However, the complexity in designing such systems rises. Therefore, the usage of an Operating System (OS) is essential to provide the necessary abstraction of the computational resources in reconfigurable computing. Moreover, due to the intrinsic overhead caused by the reconfiguration activities and the potential sharing of computational resources the necessity for support provided by an OS is unquestionable. Nevertheless, embedded system platforms lack in computational resources. This fact requires a careful design of an OS for such a system, since it also consumes its resources.

Along with the application tasks, the OS can profit from a RSoC based architecture by reconfiguring itself over this hybrid platform. Thereby, the OS can make use of the remaining resources that are not currently required by the application for its execution. Within this context, this work presents the design of proper methodologies, strategies, hardware and design support for a proper management of dynamic reconfiguration activities of a Real-Time Operating System (RTOS) running on a RSoC based platform. The intention thereby, is to promote the self-reconfiguration of the RTOS services on this hybrid platform, so that the computational resources of this execution platform are used in an efficient way.

Zusammenfassung

Eingebettete Systeme haben eine starke Präsenz in unseren alltäglichen Leben bekommen, in vielen Bereichen sind sie allgegenwärtig geworden. Dieses ist eine Herausforderung für die Forschung im Bereich solcher Systeme. Ständig müssen neue adäquate Lösungen gefunden werden. Durch die zunehmenden Anforderungen nimmt die Leistung und Flexibilität bei eingebetteten Systemen ständig zu. Zum Beispiel, eine einzelne Architektur muss in der Lage sein, in bestimmten Fällen, mehrere Applikationen mit verschiedenen Anforderungen zu unterstützen, die asynchron und dynamisch ablaufen können (dynamische Umgebungen). Rekonfigurierbare Rechensysteme scheinen ein potentielles Paradigma für diese Szenarien zu sein, weil sie Flexibilität und hohe Rechenleistung für moderne eingebettete Systeme liefern können. Von besonderem Interesse sind jene Architekturen, wo ein Mikroprozessor mit rekonfigurierbarer Hardware fest verbunden ist (hybride Plattform). Eine solche hybride Plattform nennt man rekonfigurierbares System-on-Chip (RSoC). Jedoch nimmt die Komplexität in solchen Systemen ständig zu. Deshalb ist die Anwendung eines Betriebssystems (OSs) wesentlich, um eine notwendige Abstraktion von den vorhandenen Ressourcen in rekonfigurierbaren Rechensystemen zu ermöglichen. Weiterhin, ist durch die gemeinsame Nutzung von Ressourcen einer solchen Architektur und deren Verwaltung in Bezug auf die Rekonfiguration, der Einsatz eines OS zwingend notwendig. Dennoch sind die Ressourcen in eingebetteten Systemen begrenzt. Deshalb muss beim Entwurf eines OS für ein solches System sorgfältig vorgegangen werden, da das OS an sich schon Ressourcen verbraucht.

Zusammen mit den Applikationen kann das OS auch von den RSoC Architekturen profitieren dadurch, dass das OS sich selbst auf der hybriden Plattform rekonfigurieren kann. Somit kann das OS die übrigen Ressourcen nutzen, die nicht gegenwärtig von der Applikation benutzt werden. In diesen Rahmen präsentiert die vorliegende Arbeit den Entwurf von geeigneten Methodologien, Strategien, Hardware und Entwurfsunterstützungen für eine geeignete Verwaltung von dynamischen Rekonfigurierungsaktivitäten eines Echtzeitbetriebssystems (RTOSs), das auf einer RSoC basierten Plattform läuft. Die Intention dabei ist es die Selbst-Rekonfiguration der RTOS Dienste auf einer hybriden Plattform zu ermöglichen, wodurch die vorhandenen Ressourcen der Plattform effektiv ausgenutzt werden können.

Resumo

Sistemas embarcados estão cada vez mais presentes em nossas vidas e estão se tornando onipresentes. Este fato tem demandado grandes esforços em pesquisa para criação de propostas e soluções para os desafios gerados no desenvolvimento destes sistemas. Por exemplo, uma arquitetura moderna de sistemas embarcados requer alto poder de computação e também grande flexibilidade, e a demanda por estes requisitos tem crescido constantemente. Uma única arquitetura deve executar, em certos casos, diferentes aplicações com diferentes requisitos e com inicio de execução indeterminado, caracterizando desta maneira um ambiente dinâmico. A computação reconfigurável aparece como um paradigma promissor para estes casos pois consegue prover alto poder de computação juntamente com flexibilidade requeridas pelos sistemas embarcados modernos. Especialmente interessantes são arquiteturas baseadas em System-on-Chip reconfiguráveis (RSoC), nas quais um microprocessador está fortemente conectado a um hardware reconfigurável (plataforma híbrida). Porém a complexidade no desenvolvimento destes tipos de sistemas cresce, tornando o uso de um sistema operacional (SO) indispensável. Entretanto, uma plataforma de execução de um sistema embarcado sofre pela escassez de recursos. Este fato exige um cuidado especial no desenvolvimento de um SO uma vez que este também usa os recursos desta plataforma.

Juntamente com as tarefas da aplicação, o SO também pode tirar proveito de uma plataforma baseada em RSoC onde este é capaz de se auto reconfigurar sobre esta plataforma híbrida. Deste modo, o SO pode usar os recursos computacionais, correntemente não requeridos pelas aplicações, para a sua execução. Dentro deste contexto, este trabalho apresenta o design de metodologias, estratégias e suporte em hardware e software para o gerenciamento apropriado das atividades de reconfigurações dinâmicas de um sistema operacional de tempo-real (RTOS), que é executado em uma plataforma baseada em RSoC. A intenção com isto é a de proporcionar ao RTOS meios com os quais este é capaz de se auto reconfigurar nesta arquitetura híbrida com a intenção de atingir um uso mais eficiente dos recursos computacionais desta plataforma de execução.

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

$\mathcal{S}$	Set of OS services
s_i	A generic OS service
$\mathcal{S}^\star$	Subset of $\mathcal{S}$ representing the OS services that will undergo a reconfiguration
$s_i^\star$	A generic OS service that will undergo a reconfiguration
$\mathcal{S}^s$	Set of software services that will be changed by another software service ($\mathcal{S}^s \subset \mathcal{S}^\star$)
$\mathcal{S}^h$	Set of hardware services that will be changed by another hardware service ($\mathcal{S}^h \subset \mathcal{S}^\star$)
$\mathcal{S}^w$	Set of services that will change the execution domain ($\mathcal{S}^w \subset \mathcal{S}^\star$)
$\mathcal{T}s$	Set of services located in software
$\mathcal{T}h$	Set of services located in hardware
$\mathbf{C}_1$	Cost set of software services
$\mathbf{C}_2$	Cost set of hardware services
$\mathbf{C}$	Cost set of all OS services
$c_{i,j}$	Cost of a service s_i assigned to execution domain j ($j = 1$: CPU; $j = 2$: FPGA)
A_i	Cost used by service s_i when assigned to FPGA ($c_{i,2} = A_i$)

U_i	Cost used by service s_i when assigned to CPU ($c_{i,1} = U_i$)
c_i	Hardware and software costs of a service s_i ($c_i = \{U_i, A_i\}$)
l_i	Sum of U_i and A_i ($l_i = c_{i,1} + c_{i,2}$)
A	Total cost of OS services assigned to FPGA
U	Total cost of OS services assigned to CPU
B	Balance between FPGA and CPU costs
w_1	Tendency of application resource utilization in CPU
w_2	Tendency of application resource utilization in FPGA
A_{max}	Maximum FPGA area cost available for OS services
U_{max}	Maximum CPU workload cost available for OS services
δ	Balancing constraint: maximum allowed unbalance
$\mathbf{X}_1$	Assignment solution for software services
$\mathbf{X}_2$	Assignment solution for hardware services
$\mathbf{X}$	Assignment solution for all OS services
$x_{i,j}$	Assignment solution of a service s_i at execution domain j
x_i	Assignment solution for a service s_i ($x_i = \{x_{i,1}, x_{i,2}\}$)
R	Complete system reconfiguration cost
$\mathbf{R}_i$	Relocation matrix cost (3 x 3) of a component s_i
$r_{i,j}$	Cost to relocate a service from domain i to domain j
z_i	Assignment difference of a service s_i between two different assignment solutions
$\mathbf{Z}$	Assignment difference set of services between two different assignment set solutions
C^{α}	Communication cost between two OS services, both located in software
C^{β}	Communication cost between two OS services, both located in hardware

C^{γ} Communication cost between two OS services, each one located in a different execution domain

$\kappa(u, v)$ Communication costs between two OS services, u and v ($\kappa = \{C^{\alpha}, C^{\beta}, C^{\gamma}\}$)

(λ_1, λ_2) Maximum resulted component cost that is allowed when clustering two OS services

$\mathcal{J}$ Set of jobs needed to perform a system reconfiguration

J_i A generic reconfiguration job associated to service s_i

J_i^a Configuration phase of a reconfiguration job J_i

J_i^b Migration phase of a reconfiguration job J_i

M^s "Migration" time of a component when changed by another component version at CPU

M^h "Migration" time of a component when changed by another component version at FPGA

M^w Migration time of a component when relocated between CPU and FPGA

Q^s Configuration (programming) time of a component at CPU

Q^h Configuration (programming) time of a component at FPGA

E^s Execution time of a component at CPU

E^h Execution time of a component at FPGA

P_i Period of service s_i

D_i Relative deadline of service s_i

$d_{i,k}$ Absolute deadline of the kth instance of service s_i

$\hat{d}_i$ Absolute deadline of migration job J_i^b

$a_{i,k}$ Arrival time of the kth instance of service s_i

$\hat{a}_i$ Arrival time of migration job J_i^b

$b_{i,k}$ Starting (beginning) time of the kth instance of service s_i

$\hat{b}_i$ Starting (beginning) time of migration job J_i^b

$f_{i,k}$	Finishing time of the kth instance of service s_i
$\hat{f}_i$	Finishing time of migration job J_i^b
η	Computation ratio ($0 \leq \eta \leq 1$) of a migration job
σ	Time distance between arrival times of a service and its related migration job J^b

CHAPTER 1

Introduction

The presence of embedded systems is massive in our lives. It is possible to identify its usage in a great variety of products: cellular phones, Personal Digital Assistance devices (PDAs), household appliances (e.g., washing machines, microwave ovens, DVD players), vehicles, airplanes, missiles, medical equipments, etc.; only to cite some of them. Some numbers show that more than 99% of the microprocessors produced nowadays are devoted to embedded system platforms [1].

The design of embedded systems is complex and involves many interdisciplinary research areas, from high level modeling and simulation, through software, hardware and platform design to the hardware and software synthesis and testing [2]. Unlike a general-purpose computer, such as a Personal Computer (PC), the design of an embedded system is considerably more complex. Usually, when designing such system one needs to take into account requirements such as memory and power consumption, real-time behavior, short time to market, etc.

On the other hand, with technology advances, most of embedded system components can be incorporated into a single chip, leading to the so called System-on-Chip (SoC) [3]. One SoC may contain one or more CPUs, memory, peripherals and dedicated hardware components (e.g., coprocessors) specifically developed for a target application in order to meat the performance required by the application. These solutions, however, are usually static, meaning that adaptations and/or modifications due to application changes are not adequately supported.

A side effect of this integration is that contemporary embedded systems are increas-

ingly incorporating more and more functionalities, requiring thereby higher computation performance. Moreover, these systems are becoming multipurpose, since they need to support more than one application which may be different in their nature. For instance, modern PDA or mobile phones are capable to play movies, connect to the internet and make telephone calls among other activities. In addition, the algorithm complexity tends to increase in many application domains such as signal, image and processing control. Therefore, modern embedded systems are increasingly requiring more computational performance and flexibility due to the growing complexity and dynamics of the application domain.

In order to be able to handle this tradeoff between flexibility and high performance, an execution platform combining reconfigurable hardware and SoC becomes an attractive solution. This new platform is usually referred as Reconfigurable System-on-Chip (RSoC). A single RSoC based platform may provide different facilities. For instance, a product can change or upgrade its functionality by e.g., at initialization phase, loading different configuration data. This enables, further, to upgrade a product that has been already assembled, or even to correct some error identified in this product after delivering it to the market. Additionally, a RSoC based system may support run-time reconfiguration. This may even allow the creation of more complex systems, with capability to dynamically adapt to system changes.

1.1. Motivation

The duty of an Operating System (OS) is to provide the necessary abstraction of the hardware platform and provide services to the application, like for instance, message passing, shared resources management, etc. Given that the computational power and size (in terms of components integration) of nowadays SoC are rapidly increasing, the utilization of an OS is also gaining in importance. An OS easies programming activities by providing well defined interfaces to the underlying execution platform, hiding thereby low level details from programmers. Furthermore, it enables the portability, reusability and protection among applications. Actually, the reasons of using an OS for a SoC are not different from those for running an OS on any system [4].

The demand of an OS is more emphasized in the case of platforms based on RSoC. Due to the aggregated overhead caused by the reconfiguration activities, and management of the shared reconfigurable hardware (dynamic reconfiguration), the support given by an OS is highly desired. The OS executing on such platforms needs to provide suitable methods and infrastructure for managing and using the resources in an efficient manner.

The design of a SoC architecture is application dependent. Due to the necessity to reduce costs, power consumption, etc, it is required to provide an architecture only with those components that a specific application will require. Since an OS also uses resources from the execution platform, it is also desirable to have a modular OS. Thus, the software

designer may tailor the OS to provide only the required services that an application needs. These solutions, however, are usually static.

Since applications of modern embedded systems do present dynamic behavior (e.g., the PDA example mentioned above), imposing thereby dynamic requirements to the OS, those static solutions are no longer efficient. For such scenarios, the presence of an OS capable to be dynamically reconfigured is highly desirable, in order to provide only the current facilities required by the application and, thereby, using the resources in an efficient manner.

1.2. Thesis Goals

A typical RSoC architecture includes a CPU and reconfigurable hardware as main computational resources, which are shared between application and OS activities. Towards an optimal usage of the available resources, the RTOS should be able to reconfigure itself over the underlying hybrid architecture.

In this direction, this thesis aims to provide methodologies, strategies, mechanisms, as well as hardware and design support, which aggregates self-reconfiguration capabilities to an embedded operating system. By this means, the used operating system can configure itself over the hybrid architecture in order to use the computational resources that are currently not being used by application programs, and similarly, freeing resources currently demanded.

Figure 1.1 summarizes the idea of this thesis. An extra component in the system, called RRM (Run-time Reconfiguration Manager), monitors application requirements along with the current occupation of the execution platform. So, by continuously analyzing the overall resource utilization it coordinates the configuration of the OS services over the hybrid architecture.

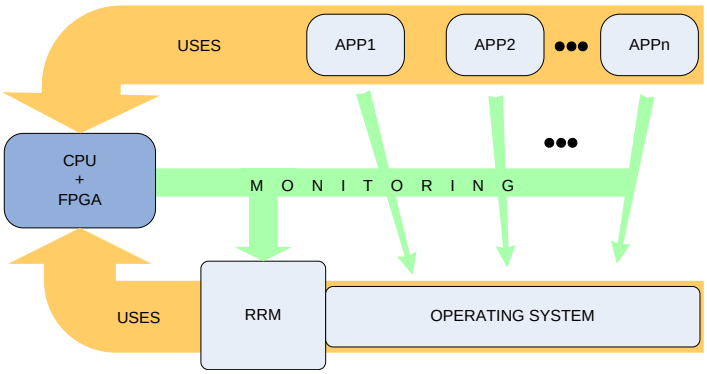


Figure 1.1.: Reconfigurable OS for RSoC: Overview.

1.3. Thesis Contributions

The main contribution of this thesis is the development of strategies and methodologies comprising an OS extension that can be aggregated to a real-time operating system running under a hybrid execution platform enabling, thereby, an effective usage of the computational resources by application tasks as well as by OS services. Such OS is well adequate for given support to the modern and future embedded systems, relying on execution platforms with high flexibility and performance.

All main parts of this thesis have been published in several conferences, which demonstrate the recognition of this work by the research community. Moreover, it indicates the relevance of the investigations carried out in the scope of this thesis research.

The raw proposal and concept of a reconfigurable OS for RSoC has been published in [5]. Then, in [6] a more sound proposal for an execution platform is presented along with heuristic algorithms for allocation of OS services over this hybrid platform. Later on, in [7, 8] those algorithms were improved in order to decrease the reconfiguration overhead. A further extension to the allocation algorithms, in order to take into account the communication costs of OS components depending on their allocations, has been presented in [9].

Appropriated model and strategies used to assure a deterministic reconfiguration of OS services on the hybrid platform were firstly presented in [10] and then extended to cover all aspects of this thesis in [11]. In addition, a framework and additional infrastructure to support the design of relocatable components was presented in [12], [13] and [14].

The use of the proposed OS adaptation mechanisms and strategies in self-optimizing systems is presented in [15, 16]. Furthermore, an overview of the overall proposal was published in [17] and recently accepted for journal publication [18].

1.4. Thesis Outline

This thesis is organized as follows:

Chapter 2 provides theoretical background information, necessary for a clear understanding of the topics discussed in the subsequent chapters.

Chapter 3 summarizes relevant related work. It includes a survey on (re)configurable operating systems based solely in software, filtering those ones intended for embedded systems domain. Then, several aspects related to reconfigurable computing and the proper and adequate support of an operating system for such systems is largely discussed. Comparisons between those works and the subject of this thesis are left to the end of the chapter.

Chapter 4 discusses in more details the ideas briefly introduced in Section 1.2. Herein,

target applications are carefully specified. In addition, the target OS, which will be used in the validation phase of this work, as well as the reasons for choosing it, are presented. Furthermore, the execution platform is introduced in this chapter along with the related design assistance requirements.

Chapter 5 presents the models adopted for the execution platform, the OS components, and the reconfiguration activities. Additionally, the main problems that the Run-time Reconfiguration Management (RRM) needs to solve are here formulated.

Chapter 6 introduces the methodologies and strategies adopted by the RRM to deterministically manage the allocation and reconfiguration of the OS services over the hybrid architecture. These strategies are mainly based on low complexity heuristic algorithms proposed to solve NP-Hard problems faced by RRM, which further need to be executed concurrently with the normal system operation.

Chapter 7 evaluates each heuristic algorithm presented in the previous chapter. For that purpose, the MATLAB tool was used. Results are analyzed according to algorithm efficiency, pointing out where further research need to be spent to solve some open questions.

Chapter 8 describes the support made available to the programmer to enable the usage of the proposed system, covering thereby two aspects. First, the automatic generation of hardware-software interface using the IFS [19] (Interface Synthesis) tool is explained. Second, a framework for generation of hybrid services/tasks that enables their run-time relocation between CPU and FPGA is presented.

Chapter 9 shows a case study used to validate the proposed reconfiguration strategies, the design support proposed, and to analyze the effectiveness of the underlying architecture in carrying out the reconfiguration activities. As a target OS service, an encryption algorithm was selected, which is an OS service (for both, embedded and non embedded systems) increasingly demanded by safety and security applications.

Chapter 10 gives a synopsis of the work presented in this thesis. Furthermore, it signals directions on which further work can be conducted.

CHAPTER 2

Background

This chapter gives the reader the necessary background information for a complete understanding of the technical discussions in the following chapters. Fundamentals of Reconfigurable Computing and the mainly used hardware technologies are presented.

2.1. Embedded System Design

Traditional embedded systems, those existing until middle of nineties, could be specified as a system comprising a microcontroller, memory, analog devices and some I/O signals. Nowadays, due to technology advances and also with the increasingly requirements from applications supported by contemporary embedded systems, they became more complex, which have a direct impact on the design of such systems.

Unlike a general-purpose computer, such as a Personal Computer (PC), the design of an embedded system is considerably more complex. It is an interdisciplinary activity, involving many research areas. It goes from abstract level modeling and simulation, through software, hardware and platform design to the hardware and software synthesis and testing [2].

Besides low cost and tight time to market, other constraints, like for instance the limited amount of memory available, low power consumption requirement, etc, make the design of such a system a challenge. Furthermore, the fact that a single "standard" execution platform for an embedded system does not exists (differently, for instance, from the situation of PC market) increases even more the degrees of freedom by searching complete

solution for an embedded system.

Contemporary embedded systems are further integrating more and more functionalities and requiring, therefore, higher computation performance. With technological advances most of system components can be incorporated into a single silicon die leading to a so called System-on-Chip (SoC) [3]. One SoC may contain one or more CPUs, memory, peripherals and dedicated hardware components (e.g., coprocessors) specifically developed for a target application. A SoC offers more advantages and benefits to system designers such as for instance, higher performance, lower power consumption, and higher reliability, if compared with the case where a system is build by assembling various chips and components on a circuit board.

Another trend in the development of execution platforms for embedded system is towards multiprocessor architectures. Due to constraints in power consumption and increasing demand in performance, some solutions in integrating multiple processors in a single die are being currently investigated. Examples are Multi-Processors System-on-Chip (MPSoC) and Network-on-Chip (NOC) systems [3].

Even for normal PCs this trend is observed (for instance, the Dual Core processor from Intel [20]), or even in graphic cards used inside such architectures. Recently, NVIDIA company released the G8 graphic chips where multithreading is used as based environment for the application design (for more details see [21] and [22]).

Another example is the Cell Multiprocessor [23], which was developed to attend the demands of the game/multimedia industry. The Cell multiprocessor combines in a single die one 64bit based microprocessor with eight cooperative processors, all connected together through a high bandwidth on-chip coherent bus.

Hardware/Software Codesign

A typical SoC architecture is based on hardware and software components, and the decision of which parts will be developed in hardware and which in software is one of the most important parts by designing an embedded system execution platform [24]. Hardware/Software codesign defines some methodologies and strategies for designing heterogeneous (hardware and software) systems. The goal is to find an efficient solution for a system which meets the specified requirements and constraints. In this process, it is important to have a unified model environment, which allows the co-simulation and co-verification of hardware and software parts together. A considerable amount of work has been done in this area: [25], [26], [27] and [28].

One key part of the hardware-software codesing, shown in Figure 2.1, is the hardware-software partition and the correspondent interface synthesis. The partition phase is not a trivial task. Even though there are some tools which do this in an automatic way, this activity is done also manually [29]. Nowadays, the partition of the system in hardware and software is influenced mainly by the designer experience and also by the availability

of previously designed and used architectures (also known as architecture templates). Moreover, due to the market pressure, there is a need to deliver a product as soon as possible, which also decreases the development costs. This short time-to-market also emphasizes the reusability of hardware and software components in the design of new embedded system architectures.

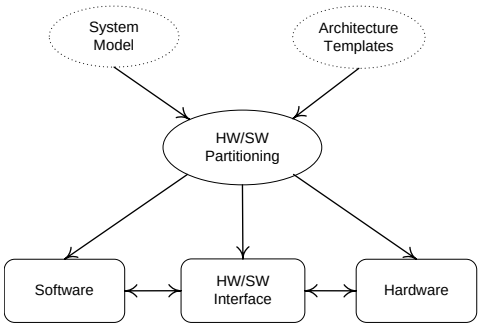


Figure 2.1.: HW/SW partitioning scheme.

Another important aspect of hardware/software codesign is the generation of an interface between hardware and software components. This phase is commonly left to the end part of the design flow, mostly due to the lack of properly abstraction of HW/SW interfaces, which makes difficult the dialog between software and hardware design teams. This problem is highlighted even more in new architectures that are becoming multiple and heterogeneous multiprocessing. Therefore, a hardware/software interface codesing [30] is necessary in order to tackle the complexity of designing SoC architectures in the near feature.

2.2. Reconfigurable Computing Overview

As previously mentioned, the required application performance is achieved by implementing the computational intensive functionalities in a dedicated device, usually referenced as a hardware accelerator. If, for instance, a hard-wired technology is used then an Application Specific Integrated Circuit (ASIC) is designed and fabricated specially for the purpose of this specific application.

Although an ASIC could perform very well when executing the exact computation for which it was designed, it does not provide flexibility. In order to change the computation performed inside an ASIC, re-design and re-fabrication are necessary. A microprocessor, in an opposite way, offers great flexibility, since its functionalities are determined by the software instructions stored in the memory. In terms of efficiency, however, a microprocessor is by far inferior to an ASIC, since the computation is executed sequentially.

Reconfigurable systems may fill the gap between application-specific platforms based on custom hardware functions, and software programmable systems based on traditional

microprocessors [31]. The resulting system is one which can provide higher performance by implementing custom hardware functions in reconfigurable units, and still be flexible by reprogramming the hardware and/or a microprocessor (hybrid architectures). Figure 2.2 shows the trade off between performance and flexibility related to software, hardware and reconfigurable computing.

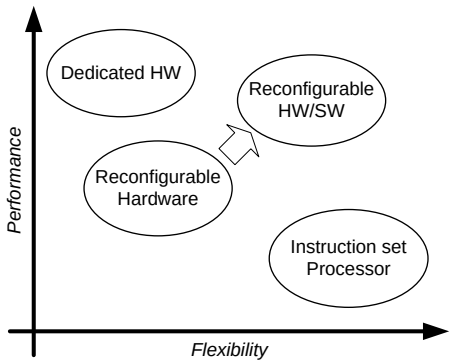


Figure 2.2.: Performance versus Flexibility tradeoff.

A reconfigurable hardware (RH) is very attractive for designing execution platforms for modern embedded systems. One motivation is the reduction of the Non-Recurring Engineering (NRE) costs, since the same hardware architecture may be used for more than one product. Even the same product can be provided by a company with several variations by reprogramming the hardware and software. Furthermore, the post-manufacturing programmability allows upgrades or corrections of problems when the product has been already finished.

Execution platforms based on RH may also be dynamically reconfigured, allowing the dynamic adaptation of the system to the run-time environments [32]. Flexibility can be added to the architecture by dynamically allocating different and dedicated processing operators within the reconfigurable device. It also allows a configuration larger than the available RH to be used (virtual hardware concept). This improves the resource utilization by sharing the hardware between various applications.

The Figure 2.3 illustrates an example scenario where computations A, B, C and D are accelerated by the RH [33]. After finishing the execution of A and B, the computation D can replace them in the RH. Depending on the hardware support, computation C may be kept running while the reconfiguration happens. Such platforms require, however, more attention on the management and control.

There are two main challenges by following this approach. First, the kernel candidates to be implemented in hardware need to be mutual exclusive in their execution (no concurrency). This is also true for hardware devices that comport more than one kernel. However, in this cases the mutual exclusivity need to be assured among the kernel sets. To find a solution for these problems, techniques known as temporal and

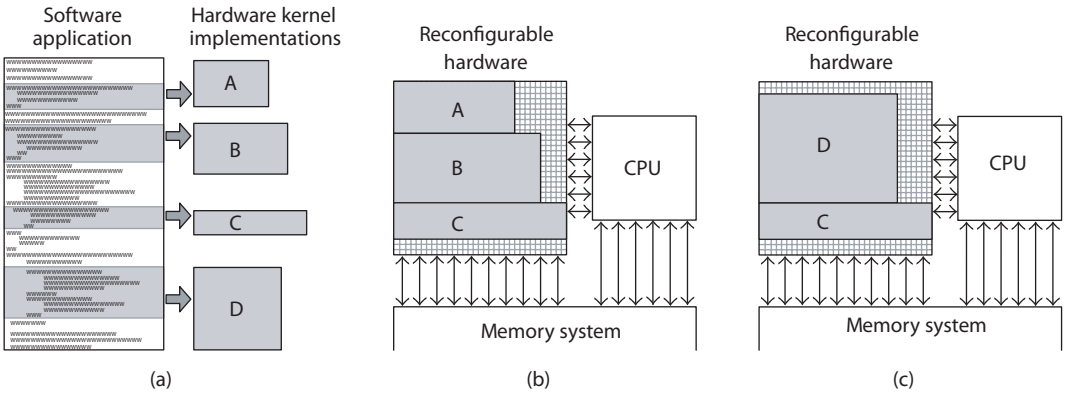


Figure 2.3.: Example of dynamic reconfiguration usage [33].

spatial partitioning [34, 35] are used. The second challenge is to provide a well designed infrastructure, which needs to provide efficient methods to, e.g., swap in and out the configurations on the RH, keep track of data exchanged between configurations, schedule the configurations, etc.

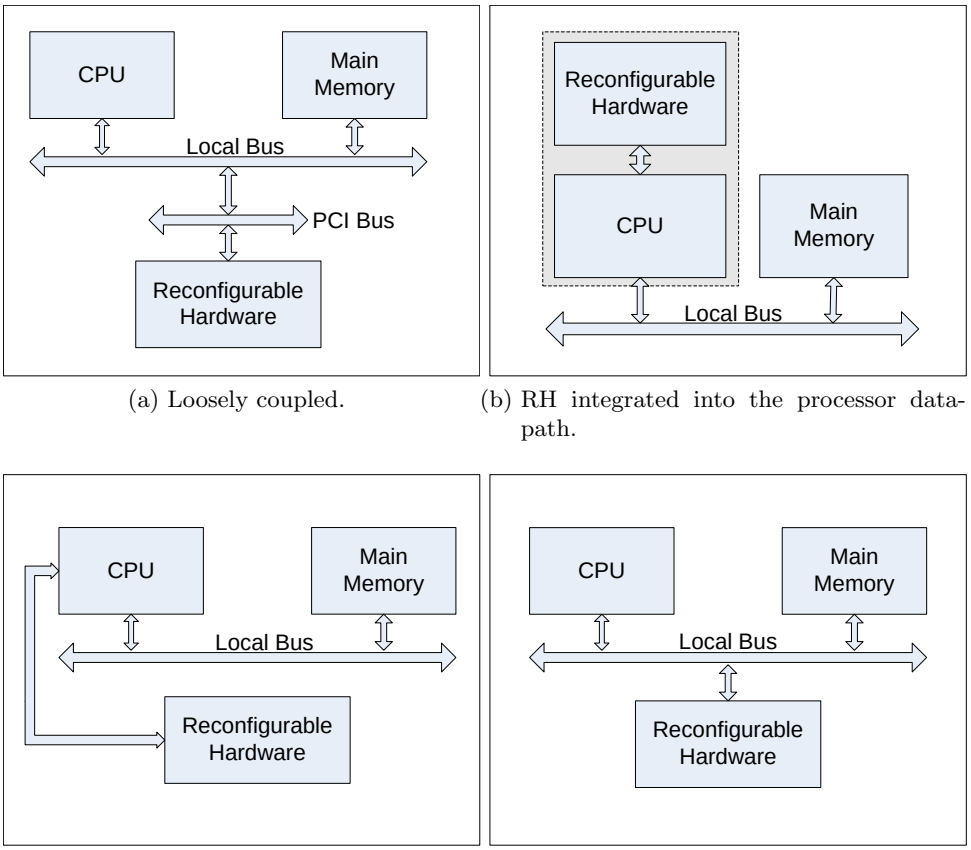
2.2.1. Coupling CPU and Reconfigurable Hardware

A reconfigurable architecture appears normally in form of a CPU connected with a RH. The coupling type has a strong influence in the communication costs between CPU and RH. Furthermore, certain connection types require that the CPU is specially designed to be connected with the RH (non standardization).

Figure 2.4 shows four different types of connections. In a really tightly coupled architecture (Figure 2.4b) the RH is deeply integrated with the CPU, so that it belongs to the processor data-path. The RH may also be connected with the CPU in the form of a classical coprocessor as presented in Figure 2.4c. In a middle connected architecture, the RH is plugged to the CPU through a local bus, as in Figure 2.4d. This can be seen also as an extended version of a coprocessor, since the RH may be accessed by the CPU using memory mapped technique. However, such architecture may also be used as an execution platform for a multiprocessing system. In loosely coupled architectures (Figure 2.4a) the RH is attached to the CPU through an external bus (like PCI) which is also an usual solution for a system architecture.

2.2.2. Reconfigurable System Design

Event though the usage of reconfigurable hardware in embedded systems is increasing [33], there is no standard method to design such system yet. Instead, there are proposals which are based on the extension of the traditional hardware/software codesign methods



(a) Loosely coupled.

(b) RH integrated into the processor datapath.

(c) RH as a classical coprocessor.

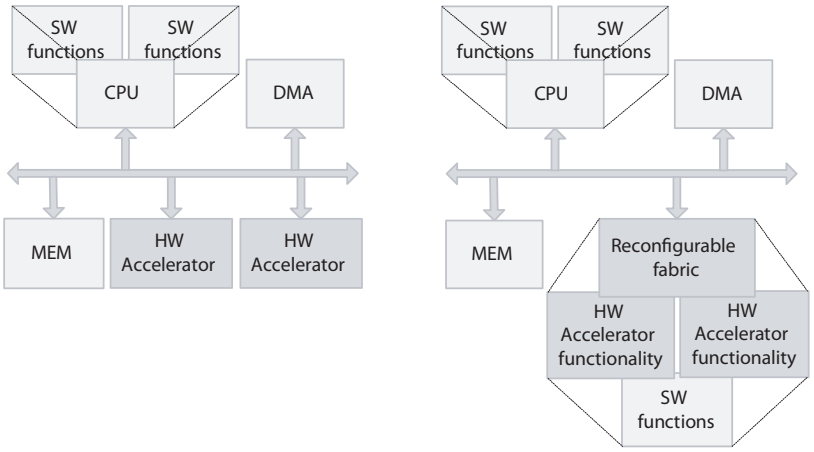
(d) RH attached at the local bus.

Figure 2.4.: CPU and RH: Coupling types.

[36, 37], and their differences concentrate on the hardware-software partition phase.

However, the design of such system is even more complex than the traditional one due to the necessity to evaluate the temporal and spatial utilization of the reconfigurable hardware (which is considered as a shared resource), configuration overhead, communication infrastructure, etc. Moreover, the new approaches rely on the usage of modeling languages, which allows the consideration of the reconfigurability aspects since the beginning of the design phase.

Figure 2.5 gives an idea about the differences at execution platform between SoC and RSoC. In Figure 2.5a HW accelerators are static components and in Figure 2.5b they are placed in the reconfigurable fabric according to its usage.

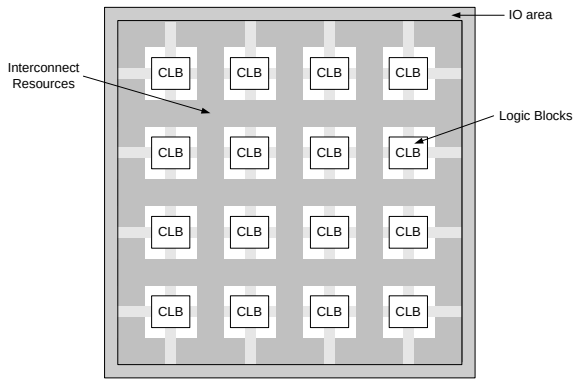


(a) (b)

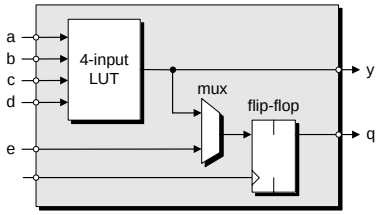
Figure 2.5.: Typical SoC architecture (a) and a RSoC architecture (b) [38].

2.3. Reconfigurable Hardware Technology

An execution platform for a reconfigurable computing application is based typically on a Field Programmable Gate Array (FPGA), which is a two-dimensional grid of configurable logic cells, called Configuration Logic Blocks (CLBs). These blocks are embedded in a general routing structure (also configurable) which allows their interconnections (inputs and outputs of each CLB). This architecture, shown in Figure 2.6a, supports a construction of a relatively arbitrary interconnection scheme between the logic blocks in the system.



(a) Basic FPGA architecture.



(b) Simplified Configurable Logic Block (CLB) [39].

Figure 2.6.: FPGA comprises a set of CLBs.

Each CLB can implement a distinct and limited logic function. A very simplified structure of a CLB is shown in Figure 2.6b which contains a 4-input Look-Up Table (LUT)[1], a Multiplexer (MUX) and a storage element (flip-flop). A real CLB available in the modern FPGAs usually comprises more than on of such logic blocks and more signals for controlling, allowing the implementation of complex logic circuits.

FPGAs currently represent the most popular and mature segment of RH technologies. Of special interest are the SRAM-based FPGAs, which are the state-of-the-art in FPGA technology. The configuration (CLBs and interconnect resources) are stored internally on a static RAM. Changing the content of the configuration RAM will also change the resulting circuit running on the FPGA.

2.3.1. Hybrid Architecture

Meanwhile there are FPGA fabrics which incorporate a processor core. Figure 2.7 shows the structure of a Virtex-II Pro FPGA used in this work. This FPGA (as well as Virtex4, from the same company) is equipped with up to four hardcores PPC405 processors, signifying also that RH and CPU are located in the same device. With these devices, it is possible to design sophisticated architectures, where even more than one RH component may be connected with the CPU. This provides more flexibility when designing Reconfigurable SoC (RSoC) execution platforms.

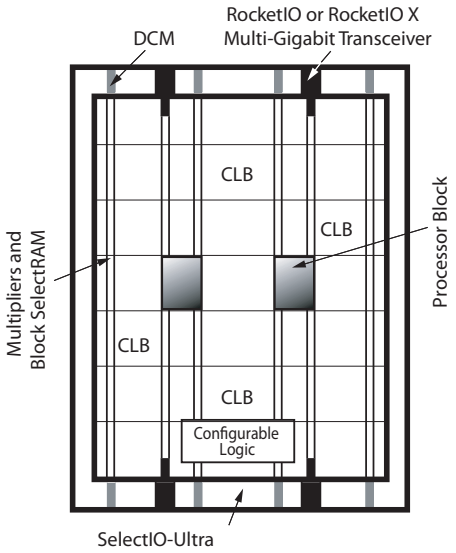


Figure 2.7.: Viertex-II Pro architecture overview [40].

If a processor core is needed and not available in a FPGA, like the Virtex and Spartan3 FPGA families, one can use a softcore processor. An example available from Xilinx

[1] A LUT implements a combinatorial logic by storing a function truth table.

company is the so called MicroBlaze, which is a softcore processor based on RISC architecture. This processor even allows the integration of accelerators, designed inside the FPGA, to the processor data-path [41].

In order to help the design of embedded systems based on such hybrid devices, a design tool called EDK (Embedded Development Kit) is made available by the FPGA Company. The architecture advised and supported by this tool, when using the hardcore PPC405, is shown in Figure 2.8. The processor block is connected to the other components through the CoreConnect^{TM}bus architecture, composed by two main buses: Processor Local Bus (PLB) and On-chip Peripheral Bus (OPB). Slow peripherals should be attached to the OPB bus in order to offload the PLB bus, which provides low latency access to peripherals requiring high performance.

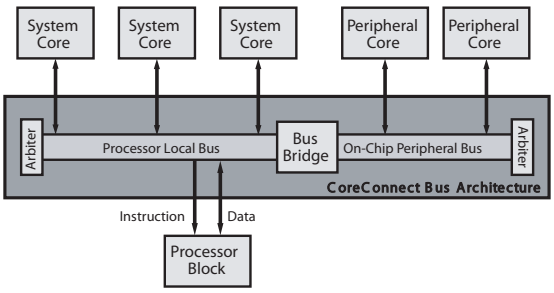


Figure 2.8.: CoreConnect block diagram [40].

Other FPGA manufactures provide similar solutions in their products as well. For instance, Altera offers the Stratix II[2] and Cyclone II[3] devices, which allow the usage of NIOS softcore processors. Also ATMEL have an architecture for dynamically reconfigurable SoC, called FPSLIC (AVR processor and FPGA) [42]. Nonetheless, the background information given in this work is concentrated on FPGAs manufactured by Xilinx Company. This device was chosen for implementation purpose, since not all FPGAs support a specific feature called dynamic partial reconfiguration (discussed in Section 2.3.3).

2.3.2. Configuration Techniques

A FPGA configuration is performed by downloading a bitstream[4] through a specific configuration port. This can be done offchip assisted by an extra component connected to the FPGA configuration port (for details please refer to [40]). Alternatively, the configuration data can be downloaded by an internal FPGA entity (not available in all FPGAs), called ICAP (Internal Configuration Access Port), which is mainly used for partial reconfiguration approaches (Section 2.3.3).

[2]Available at: http://www.altera.com/products/devices/stratix2/st2-index.jsp
[3]Available at: http://www.altera.com/products/devices/cyclone2/cy2-index.jsp
[4]A bitstream is a configuration data used to program the FPGA in a serial manner.

Furthermore, through the same port the present configuration on the FPGA can be read. This technique, called *Readback* operation, is used to verify the configuration on the FPGA and it is also used as a main technique to provide basic support for multitasking on FPGA (more details about this subject is given in Section 3.2.7.

2.3.3. Partial Reconfiguration Feature

Some FPGAs support partial reconfiguration. In these fabrics, part of the device can be reconfigured whilst the remaining part keeps its execution normally. This feature allows the implementation of multitasking systems based on reconfigurable devices. The FPGA is partially configured by downloading the related partial bitstream on the FPGA through the same port used for the case of a complete configuration. If the system wants to configure itself, the internal ICAP entity must be used.

Additionally, partial reconfiguration is a device dependent feature. Xilinx FPGAs are the few ones on the market with support to partial reconfiguration. Xilinx FPGAs with this feature goes from Virtex-4 devices to Spartan-3/E family. However, usually on these devices the partial reconfiguration is possible in a column-wise manner [43]. In these cases the device is divided into a number of columns. Each column spans vertically the chip (1D partition model) and it can be independently reconfigured without affecting the other columns (Figure 2.9a). Meanwhile, the release of new guidelines and design flows, like *Early Access Partial Reconfiguration* [44, 45] and other techniques from Xilinx shall allow a more flexible rectangle sizing (2D partition model) to be partial reconfigured on a device (Figure 2.9b).

An architecture designed to support partial reconfiguration is similar to the one presented in Figure 2.9. Usually, the FPGA device is divided in different parts. A static one related to the management of configuration data and the input/output data, and one or more parts which are reconfigured on-the-fly. The connections between the static and dynamic parts of the FPGA must be implemented by specific interconnect device resources. A known technique used is the so called *Busmacros* [43]. Therefore, all circuits implemented inside the dynamically reconfigurable parts of the FPGA must use the *Busmacros* wires to connect with other entities in the system.

The technical realization of such architectures are difficult due to the poor tool support and strong restrictiveness of the design guidelines [43]. Therefore, the partial reconfiguration feature is mostly used in academic research works. However, by recently improvement in the design support (like the [44] and [45] guidelines) and recently introduced design tools, which explicitly support a modular design approach (PlanAhead [46]), this feature is going to be used commercially as well. One evidence for that is the recently usage of partially reconfigurable FPGA based platforms for Software Defined Radio (SDR) SoC modems [47].

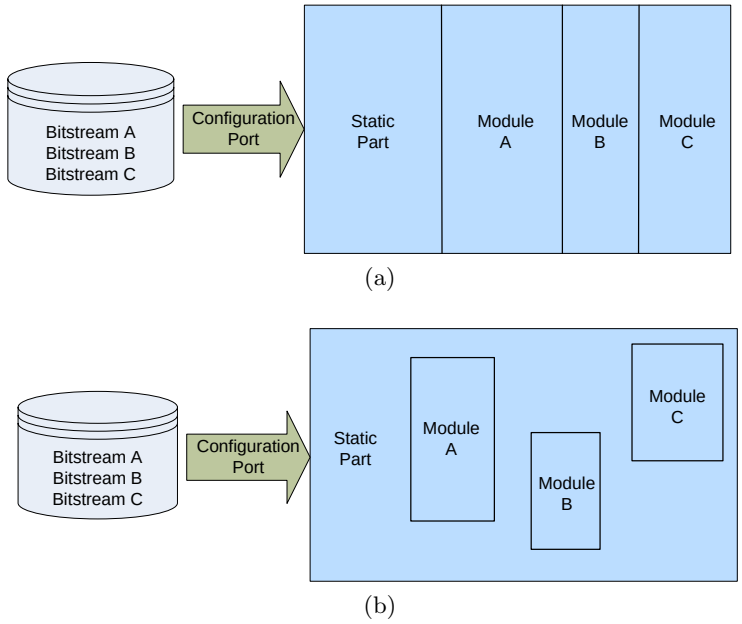


Figure 2.9.: Example of device partition for 1D (a) and 2D (b) partition models.

2.4. Chapter Conclusions

This chapter highlights the main features of reconfigurable computing and how it can contribute in improving the flexibility and performance for modern embedded system execution platforms. Additionally, the FPGA device has been introduced as being the mainstream technology in reconfigurable computing systems, keeping the focus on those FPGAs with support for partial reconfiguration. The investigations carried out in the scope of this work rely on this specific feature, which is still being improved by the FPGA vendors. Evidence for that are the availability of new tools and design flow guidelines (introduced in Section 2.3.3), which are intend to decrease the current difficult practicability of partial reconfigurable based platforms.

CHAPTER 3

Related Work Survey

This chapter starts giving a survey on reconfigurable operating systems conceived for software-only based architectures, concentrating on those ones envisioned for embedded systems domain. The intention thereby, is to figure out what are the trends and results already achieved in this research field.

In the sequence, important and relevant results related to operating system support *for* reconfigurable computing architectures, operating systems running *on* this architectures, and some new further approaches are presented. Along that, different aspects related to reconfigurable computing and the support required by an OS are covered, which concentrate on the interaction between CPU and reconfigurable hardware. Such a support may represent a simple driver as well as more sophisticated services that are required for managing the reconfigurable hardware in an abstract manner.

3.1. (Re)Configurable Operating Systems

An operating system with the ability to be reconfigured is not new and most OSs support it in one form or in another. The motivation for reconfiguration of an operating system is that by configuring it one can achieve customization of the OS to the application requirements, allowing system upgrade, etc. Moreover, dynamic reconfiguration can allow run-time adaptation of the OS to the current application requirements, and so achieving a better execution environment for this application.

Such operating systems can be in a first level classified into two categories, static and

dynamic. Beside that, reconfigurable OS can be classified in respect to the initiator of the reconfiguration [48], which could be triggered by:

Human This is often the case when the designer decides which components of an OS is going to be used for the target application and this feature is supported by most operating systems. Dynamic reconfiguration is also usual for those systems, even at boot time (by passing some parameters to the kernel) or at run-time by loading or unloading modules (e.g., Linux OS).

Application A reconfiguration that is triggered by an application appears only in dynamic case and this has received strong attention in the OS research community. Several proposals therefore have been proposed and in the following subsection some representative OSs of this category will be presented and discussed.

Operating System Also called Automatic or Self-Adaptation, OSs belonging to this category are those which the reconfiguration is initiated by the OS itself. Portability of the OS to different platforms at compile time may be seen as a feature of this kind of OS which does not represent a significant innovation, since most of the OSs provide this feature. The interesting approach is in dynamic scenarios, which require more intelligence from the OS.

Most of the investigations done for dynamic reconfigurable OS are concentrated on the cases where reconfiguration is triggered by the application, and in the majority, those approaches accept the idea that applications know better than the OS what their needs and requirements are. Therefore, usually the application makes requests to the OS to change its policies on its behalf. A self-reconfiguring OS, on the other hand, needs to have enough intelligence in order to gather sufficient and appropriated information from the running applications. Such information is needed to make a decision regarding what and when to reconfigure itself, in such a way to provide an optimal execution environment for each of the applications.

Following, some OSs representative of the categories cited above are listed and briefly discussed focusing on those ones intended for embedded system application domain.

3.1.1. Statically Reconfigurable OS

eCos The eCos (Embedded Cygnus Operating System) [49, 50] is an open-source embedded real-time configurable operating system currently supported by the RedHat company (since 1999). The eCos OS can be customized at source code level, thus offline reconfiguration. To help the designer by the reconfiguration phase a tool is made available to select the components which will compose the kernel source and compile them to generate the final OS footprint.

DREAMS DREAMS[1] [51, 52, 53, 54] is a library-based OS for embedded systems, which was

[1] DREAMS: Distributed Real-Time Extensible Application Management System.

primarily designed for offline configuration to allow customization of the OS to the target application. Currently, DREAMS also provides some dynamic reconfiguration which will be discussed in Section 3.1.4. DREAMS have similarities with the eCos OS in the sense that the operating system code is composed of components selected by the designer through a tool and finally compiled. The complete system has been designed following the object-oriented paradigm (using C++ for coding). DREAMS differs from eCos in the way that OS services are built as a run-time library, instead of a kernel (as in eCos). This is achieved by allowing the customization of the objects and selecting those objects that are really needed by the application. The customization features are based on pre-processor techniques and are incorporated into the Skeleton Customization Language (SCL). To help the designer in this process, a tool called TEReCS [55, 56] is provided which further supports the design of a communication system for distributed embedded applications.

Choices

Choices [57] is a customizable object-oriented operating systems developed at the University of Illinois and started in 1987. It is one of the first operating systems which configure the application and OS specific functionalities together. Choices allows a customization of the operating system by specializing classes in the various hierarchies, and by instantiating a specific set of objects. The application interface is a collection of kernel objects exported through the application/kernel protection layer.

3.1.2. Dynamically Reconfigurable OS: Application Triggered

EXOKERNEL

EXOKERNEL [58, 59] project is developed at the Massachusetts Institute of Technology and basically consists of a driver for the processor which supports context management, interrupt handling and functionalities for status information and functionality change. It follows the microkernel[2] concept where all other operating system extensions can be loaded as pre-compiled code into the kernel during run-time by dynamic linking (in a form of a library) at user level.

Kea

Kea [61, 62] is a kernel that has been designed for application extensibility and dynamic reconfiguration. The central unit of reconfiguration and extensibility is a *service* (encapsulated as an object). Kea provides means through which kernel services can be reconfigured, by using the notion of portals. A portal is represented as a proxy in a client's domain (application) and maps onto a system data structure, which links the proxy to a service method entry point in the kernel. In this way, portals act like agents of indirection for a service. By altering the destination of a portal, calls to a OS service can be transparently remapped to replacement services. Furthermore, by making this process available during application execution time, it is possible to implement extensible services.

[2] Microkernel is a small operating system core which provides only the basic functionalities (e.g., scheduling, message passing and timing), and all other functionalities are provided by means of modular extensions [60].

MMLite

MMLite [63] is an object-based, modular system architecture that provides a menu of components for use at compile-time, link-time, or run-time. The main goal of MMLite is to allow a system to be dynamically assembled into a full application system. The dynamic configuration is achieved through a unique mechanism called mutation which allows a transparent replacement of components whilst they are in use.

Mutation is the act of automically changing an ordinarily constant part of an object, e.g. a method implementation. Each mutation is performed by a thread which is called *mutator*. A *mutator* must translate the state of the object from the representation expected by the old implementation to the one expected by the new implementation. It must also coordinate with worker threads and other *mutators* through suitable synchronization mechanisms. Transition functions capture the translations that are applied to the object state and to the worker threads execution state. In order to limit the amount of metadata, execution transitions only happen between corresponding clean points in the old and new implementations.

Pebble

Pebble [64, 65] is a component based operating system designed to support also component-based applications for embedded systems domain. Pebble is based on three main concepts, which are (1) a minimal privileged mode nucleus, (2) a set of replaceable system services and application components running in distinct protection domain and (3) a code generator specialized for each possible cross-domain transfer.

As an operating system it adopts a microkernel architecture with a minimal privileged mode nucleus that is only responsible for switching between protection domains. The programming model is client/server where client components (applications) request services from system components (servers). Examples of system components are the interrupt dispatcher, scheduler, portal manager, device driver, file system, virtual memory, and so on. In Pebble, it is possible to dynamically load and to replace servers to fulfill applications requirements.

SPIN

SPIN [66, 67] is an operating system designed to allow applications to dynamically specialize the kernel in order to achieve a particular level of performance and functionality. A specialization can add new kernel services (by linking new code), as well as replace default policies or migrate applications function to kernel address space. In SPIN a specialization is called an extension and its behavior is defined through the execution model.

K42

K42 [68, 69, 70] is an open-source and scalable operating system kernel under development in IBM research center. It uses the object-oriented paradigm. Even though supporting extensibility, K42 is strong focusing on online upgrading of existing components (objects). For this purpose, well defined mechanisms and methods for reconfiguration of these objects have been defined [70]. One of those is the a *hot-swaping* mechanism based upon C++ virtual function tables. This includes the definition of safety points for an object in order to assure a safely upgrade of the object. The upgrade is even assisted by a dynamically created object which is interposed between the caller of the object and the object itself.

THINK

Think [71] is more a component-based framework which is used to build an operating system than an operating system itself. It provides the minimal abstraction of the underlying hardware and services in order to allow the designer to create extra operating system services following the component-based paradigm. One of the main goals of the THINK project is to make an operating system as flexible as possible in order to build dedicated and fully configurable operating system. Last investigations on THINK operating system [72] incorporate dynamic reconfiguration as an extra feature. Here again there is no definition about the initiator of the reconfiguration, either application or operating system is able to start a configuration. However, the last one is possible only with extra activities like self-monitoring which is not available in the THINK operating system.

3.1.3. Dynamically Reconfigurable OS: System Triggered

Synthetix

Synthetix [73, 74, 75] operating system, and its ancestor Synthesis [76] enhances the application performance by providing specialized implementations of operating system services which are generated on-the-fly. This is done through partial evaluation of application code and its actual input data in order to recompile at run-time condition statements part of the code. The entire procedure is executed transparently to the application. A commercial operating system [75] has used the techniques investigated in Synthetix. Synthetix only makes extensions at the top layers of the operating system services, since the services existing in the low layer are required to support a system reconfiguration. Moreover, it does not provide any means by which applications may control their own resources, because they are entirely controlled by the operating system.

VINO

VINO is an object-oriented operating system [77, 78] and it is one of the few operating systems which provides self-x capabilities, which comprises self-monitoring and self-adaptation. By using a specific framework, the operating system is able to gather information of the application in order to decide which extensions or modifications should be made in the operating system itself. This automatic adaptation has not been implemented in VINO. However, it has been investigated and concepts and methods have been proposed.

The information used to automatically adapt the system is provided by static and dynamic sources:

- The compiler generate profiles from the system as output, which is used to build a database.
- This database is augmented with online information gathered by periodically retrieving statistics maintained by each VINO module.
- Additionally, the system collects traces and logs from incoming requests and produced results.

All dynamic information is captured on-the-fly under regular system execution and used for analysis purposes. An adaptation is performed when the system detects that the resources required by an application is rising bigger than specified previously. Furthermore, the system shall be capable to detect some pattern activities which will guide the system in the decision of the most appropriated service that can satisfy the current application needs.

3.1.4. Towards Online Reconfigurable DREAMS

DREAMS operating system is being further designed towards online reconfiguration [79] and going towards microkernel architecture (which supports properly the modularization of the system). Currently, DREAMS provide means for the designer to give information from the application (e.g., resource required, time interval for resource usage, etc.) in a form of a profile. For each application more than one profile may be specified. This allows the specification of different QoS for each application depending on the resources required. At run-time, the system is able to choose the best combination of application profiles depending on current applications status and an overall system quality measurement. This mechanism is carried out by a Framework Resource Manager (FRM) [80] designed for this purpose.

3.1.5. Further Comments

Other well known OSs, like for instance, QNX [81], VxWorks [82], Linux [83], and uCLinux [84] (a lightweight version of Linux designed for processors without MMU), also present offline and online configurability. On those OSs, for instance, device drivers are able to be linked to the system at run-time. However, in these cases a configuration is decided and started by the system designer. There is no direct support from the system to help the application to decide when, why or which OS service should be loaded or unloaded.

		Initiator		
		Human	Application	Operating System
Time	*Static*	DREAMS(1st), Choices, eCos, et.al.	(not applied)	—
	Dynamic	uCLinux, OSE, VxWorks, et.al.	EXOKERNEL, Kea, MMLite, Pebble, SPIN, K42, THINK, et.al.	Synthetix, VINO, DREAMS(2nd)

Table 3.1.: OS classification according to the reconfiguration *Time* and *Initiator*.

The OSs listed above are summarized, in Table 3.1, according to the initiator of the reconfiguration activity and to the reconfiguration time. This list is only a sample of the numerous OSs found in the literature. Further details as well as further references can be found in the reviews given, for instance, in [85, 48, 86]. Nevertheless, the number of operating systems with self-x properties is not big at all. The main reason for this fact is the assumption made by the most designers, that the application have the right knowledge of its own needs, and therefore it is capable to decide what to change and when this change takes place. However, a system supporting different applications requires that each of them would have to be aware of the concurrently applications present on the system. In this scenario, the OS is the most appropriated entity in the system which can manage the resources being shared by the applications. Furthermore, this is the duty of an OS by its basic definition.

3.2. Operating System for Reconfigurable Computing

As it has been shown in previous chapter, Reconfigurable System-on-Chips are interesting for contemporary embedded system platforms. An increasing number of RTOSs developers provide support for such architectures. Well known commercial RTOSs (e.g., VxWorks, QNX, RTAI, eCos, etc) have being ported to the PowerPC-405 CPU available in FPGAs like Virtex-II Pro and Virtex-4 FX[3]. The majority of these commercial embedded systems do not support dynamic reconfiguration (particularly partial reconfiguration), which have its main attention in the academy research. One reason for this fact is that the configuration overhead is sometimes prohibiting by using the currently technology.

However, as this technology becomes mature, dynamic reconfiguration is starting to be used also commercially. One evidence for that is the recent usage of partially reconfigurable platform FPGAs for Software Defined Radio (SDR) modems SoC [47]. Furthermore, in the field of High Performance Computing (HPC), RC is starting to be incorporated in commercial systems. Enterprises like Cray, SGI and SRC Computers Inc. are examples of this trend. The Cray Inc. already sells Cray XD1 supercomputers including dual core AMD processors and Virtex-II Pro FPGAs.

An optimal scenario for which an operating system for RC can provide support is the one shown in Figure 3.1. In this case, a task pool comprising hardware tasks, software tasks and hybrid tasks are allocated by the operating system over the hybrid architecture. The programmer should not be aware about the location in which each task is going to be executed. Within this section, the research efforts toward the achievement of such an environment are presented.

In the next sections, the aggregation of services in order to support a RH in some available OSs (as well as novel ones) will be presented and discussed. It will begin,

[3] An updated list of OS ported for platforms based on Xilinx FPGA can be seen in www.xilinx.com.

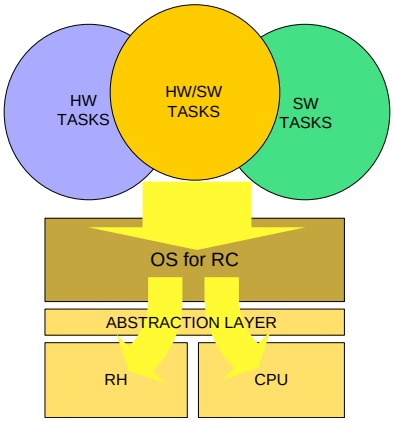


Figure 3.1.: Execution environment envisioned by the Reconfigurable Computing research community.

in the first subsection, focusing on low level support. Then, in the sequence, it will continue with proposals for more high level services that allow a proper abstraction of the underlying heterogeneous execution platform.

3.2.1. Low-level OS Support for Reconfigurable Hardware

Within this subsection, approaches dealing with the abstraction of RH resources and properly support by the OS in exchanging data between CPU and RH coprocessor are presented.

FPGA Reconfigurable Resources Abstraction

In the work presented in [84], dynamic reconfiguration capabilities are smoothly integrated into the embedded Linux version, uCLinux. In this system, the user is able to drive a reconfiguration by using normal Unix-like shell commands. Additionally, the configuration memory of the reconfigurable device is made accessible through this interface.

In the related platform, the uCLinux is running on a soft-core CPU (MicroBlaze). A reconfiguration is possible due to the usage of an internal entity present in the Virtex-II and Virtex-4 FPGA device series, called Internal Configuration Access Port (ICAP). By using a driver that wraps the ICAP entity, the OS provides to the user the capability to read/write the configuration memory of the FPGA in an abstract manner shielding, thus, the complexity of the underlying executing platform.

The execution platform used in this approach can be seen in Figure 3.2. The whole platform is implemented using a single device. Because the reconfiguration triggered by

the system is able to change the functionality of the device in which the system itself is running, it can be categorized as a self-reconfiguration system.

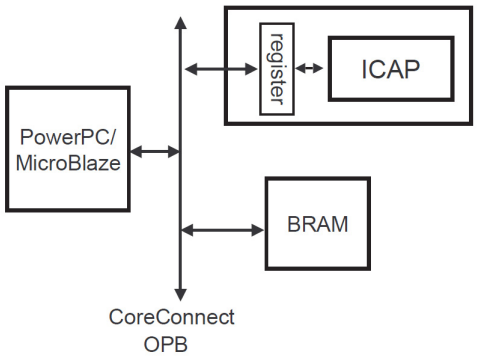


Figure 3.2.: Self-programming hardware architecture [87].

A similar work is also provided by [88], [87] and [89]. In these approaches the internal resources of the FPGA (e.g., LUTs, BRAMs, etc) are made accessible to the programmer through the Linux File System. Thus, by simply making use of standard Linux command shells, one can read, write or modify the FPGA configuration in a small grain manner.

Furthermore, in [87] a toolkit called XPART (Xilinx Partial Reconfiguration Toolkit) has been presented. This tool was built on top of the ICAP API (explained above) and provides functionalities that able the user to relocate entire modules (partial bitstreams) presented inside the FPGA.

Those works, however, do not directly focus on the technical difficulties related to the relocation of modules inside a FPGA considering its heterogeneity (some resources, like BRAM and multipliers are hardcore available in the FPGA). Furthermore, to shift a module inside a FPGA without generating it by the design tools, the bitstream read from the device (related to the module) need to be first manipulated. This requires deeply technical knowledge of the underlying reconfigurable hardware device.

Hardware-Software Interface

Another important aspect of an architecture based on a CPU and a coprocessor is the communication schemes between these two components. Moreover, if the operating system does not provide a proper abstraction of the hardware accelerator, the programmer needs to be aware of the details about the architecture, like for instance, memory pointers, sending/receiving data chunks, etc. Typically, for such platforms, the programmer is responsible for mapping the physical address space of the external device (hardware accelerator) into the virtual space of the software task, and copying the data to and from this device. This makes the design less portable requiring details of the underlying architecture.

In order to increase the abstraction, a virtualization layer is proposed in [90], which shifts the responsibility of exchanging data, between coprocessor and CPU, from the programmer to the operating system. The authors of this work propose the extension of the Virtual Memory Management (VMU) of the operating system by adding a small layer in software and in hardware (adding, therefore, an extra logic to the coprocessor). Figure 3.4 illustrates the idea, which is based on the utilization of a Dual Port Memory for its technical realization.

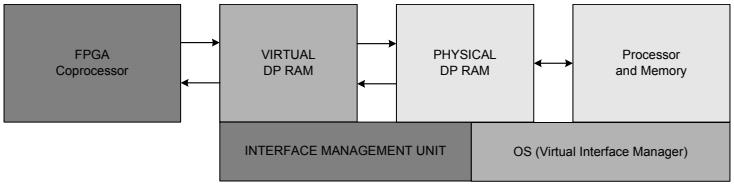


Figure 3.3.: Virtualized interface [90].

The software layer called Virtual Interface Manager (VIM) extends the VMU and it is responsible to translate the virtual address (from the software program to the coprocessor) to a physical one. This layer has the support of the Interface Management Unit (IMU), similar to the Memory Management Unit (MMU), which is able to generate interrupts to the OS when a data request from VIM is being processed. Again, the ideas proposed by the authors were also implemented in a system using embedded Linux as the target OS.

In their further research, presented in [91], the authors improve the IMU hardware support (calling it Virtual Memory Window - VMW) in order to allow the usage of virtual addresses also for the coprocessor. By this means, portability is also provided for hardware designs. Moreover, these new OS add-ons may provide support towards a hybrid multithreading system [92, 93]. In such a scenario, either a software or a hardware thread may initiate a communication. Thus, it differs from the traditional paradigm, where RH is used only as a coprocessor.

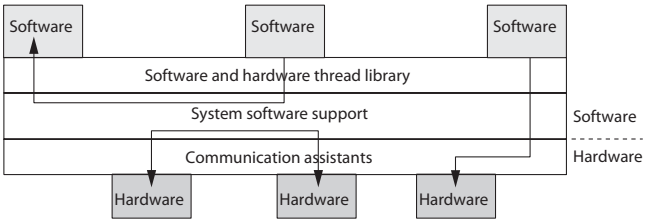


Figure 3.4.: Software and hardware threads connected through a virtualization layer [93].

Although the focus of these proposals are not to decrease the overhead caused by the communication costs between processor and coprocessor, same authors shown in [94] a concept which may hide the communication latency by prefetching memory accesses of software threads to the hardware accelerator.

3.2.2. Application Model

The majority of the authors model the input problem using a directed graph, where the nodes may represent either simple operations (e.g., ADD, MUL, etc) or more complex functions (like FFT, IDT, etc). In each way the nodes abstractly represent tasks. The edges connecting the nodes represent the task dependency and the correspondent weights (if used) denote the data transferred or data transfer costs, between them.

A task graph model has its similarity when modeling the application for a software only environment. A single task model used in hardware (hardware task), however, has more degrees of freedom than a software task. Aside from usual parameters, like (worst case) execution time, arrival time, deadline (for real-time scenarios), dependency, etc., a task in hardware also occupy some area of the FPGA, which implies in dealing with additional constraints such as shape and area size. Furthermore, in more accurate models, the time needed to configure a hardware task in FPGA may also be considered.

3.2.3. OS Services for Reconfigurable Hardware

One of the first proposals towards the development of an operating system for FPGA based reconfigurable computers was presented by Brebner in [95]. With the availability of the XC6200 FPGA from Xilinx, which allows configuration and data registers to be memory mapped, fast and partial reconfiguration were possible. Brebner highlights that these flexible reconfigurable hardware, seen as an additional computational resource (together with the CPU), need to be managed by the OS.

First introduced by [96], Brebner also presents the concept of hardware virtualization in order to produce a run-time resource allocation environment, which hides from the programmer the complexity and details of the underlying reconfigurable hardware.

Brebner proposes the division of the application into so called Swappable Logic Units (SLU). These units are defined as logic circuits that implement some functionality and are position-independent tasks. Thus, the duty of the operating system is to swap in and out these units on the FPGA. To accomplish this job, the OS uses some techniques already known from the OS theory. For instance, by placing a task (SLU for this case) into the FPGA it needs to first find free place for the incoming SLU. If the OS does not find enough space it uses the Least Recently Used (LRU) strategy to choose a SLU from the ones placed on the FPGA and remove it.

A more embracing and deeply study about the services that an OS should provide for a FPGA-based reconfigurable computer is presented in [97] and [98]. Although there is not a clear consensus about the terminology definition when referring to each OS service (as in software only case), it is possible to find close similarity for these definitions.

Following, the services that were most investigated in the referenced works are listed. The service definition used here may slightly differ depending on the author.

Partitioning This is the activity of partitioning the given task graph into sub-graphs, which will be placed later on the FPGA. Although this activity is usually done offline, due to the related overhead, there are some approaches where this is executed at run-time. The partitioning of a task graph into sub-graphs takes into consideration the physical partition of the reconfigurable hardware which may, in its turn, be also dynamically partitioned (for certain cases);

Placement This OS service is responsible to manage the reconfigurable hardware area. At the arrival of a new incoming task, it needs to allocate the necessary free resources to place this task on the device. For this purpose, it uses some strategies to decide where to allocate the task in order to, for instance, minimize the area fragmentation. Furthermore, depending on the capabilities of the reconfigurable hardware, some tasks may be swapped out from the device or even relocated (in order to free some space). This would require, however, techniques to preempt and resume a hardware task. Additionally, issues related to partitioning, like fragmentation of the device area (leading to not full utilization of its area) is also a concern;

Scheduler The decision about the order in which tasks must be executed is made by this service. Depending on some approaches, this service operates in close relation with the Placement service. In order to avoid a non feasible scheduling, in certain cases it is necessary to relocate other running tasks in the reconfigurable hardware. The policy used by the scheduler depends on the application area. For instance, the overall execution time could be the parameter to be minimized. In other cases, however, task deadlines need to be met (real-time systems). Moreover, depending on the device capabilities, preemption may or may not be allowed;

Communication Each task needs to read/write some amount of data. It is the duty of the OS to provide means for data exchange among tasks, as normal for a common OS. This has to be available for tasks being executed in the reconfigurable hardware and/or tasks running in software environment, as well as across hardware/software boundary.

Loader The low level support, responsible to actually configure the hardware, is carried out by this service. Here, the knowledge and support presented in the previous chapter are integrated;

3.2.4. RTOS issues in High Level Design

RTOS issues may also be considered at high level design phase of a system. In the project OVERSOC (see [99]) the application is modeled as a task graph and each task may be considered to be a software, hardware or hybrid task (able to run on both environments). By further modeling the architecture of a RSoC platform, this proposal provides a global methodology for a design space exploration of such system. In this environment the allo-

cation, placement and scheduling of the tasks over the platform are easily explored and evaluated. By this means, designers are able to tailor the RTOS towards the resources needed as well as estimate its performance. Furthermore, RTOS services, like hardware task preemption, task migration, message passing, reconfigurations activities etc, are smoothly integrated into the system model (see also [100]). Concerning system level design of a RSoC, the proposal of [36] is more embracing. From system specification to system implementation, the proposal is able to analyze and evaluate different system partitioning (in hardware and software) and also evaluate the run-time behavior of the system. In this work, however, the OS is not the focus of the system model.

In [101] a complete design flow for implementing an application (represented as a directed acyclic graph) is proposed. The flow is divided into three main stages: Application, Static and Dynamic stages. In the first stage, the application and the design constraints are specified. The static stage is responsible to perform a cost estimation for each task, which is used to decide the hardware/software partitioning and, finally, synthesize these tasks. The final stage (Dynamic Stage) is responsible for, at run-time, scheduling the previously synthesized tasks on the RH, supported by a dedicated microarchitecture.

3.2.5. Offline Approaches

In offline approaches, scheduling and placement algorithms are executed offline, imposing therefore a much lower overhead. Usually, offline strategies may use sophisticated models of tasks (e.g., [102, 103]). The authors in [102] use a three dimensional (3D) model for a hardware task: horizontal and vertical position on the FPGA area as two dimensions and time as a third dimension. This problem approximates the 3D packing problem and an optimal branch and bound algorithm is used to optimize the task placement over the FPGA device respecting temporal precedence constraints. In their work, the authors provide optimized solutions for this problem, providing as a result either a minimal FPGA size to solve a given problem or the minimal execution time of this given problem.

Some other approaches aim to minimize the reconfiguration overhead by allowing the exploration of task scheduling observing the configuration overhead associated. In [104] a framework to generate the scheduling of tasks that results on a minimized reconfiguration time is presented. It enables the exploration of different strategies in order to achieve different optimization goals concerning task scheduling and configuration (i.e., minimization of configuration overhead).

For scenarios where hybrid architectures (comprising CPU and FPGA) are considered and the system has not being yet partitioned, some authors prefer to use a procedure where the partitioning, placement and scheduling are executed together. This is the case of, e.g., [105]. In this work the application is specified as a task graph, where the edges inform the dependency between tasks. Additionally, for each task it is known its resource used if placed either in hardware or in software. A modified version of the Kernighan-

Lin/Fiduccia-Matheyes (KLFM) is used to solve the problem. The solution provides a near optimal minimal execution time for the given problem taken into consideration the heterogeneity of the FPGA device.

For a similar scenario, the work presented in [106] proposes the usage of Genetic Algorithms (GA) for modeling and solving the problem. In this approach CPUs, FPGAs and Buses (used to build the execution platform) are modeled and included into the problem. The schedule and mapping of the task graph on the underlying architecture is obtained by solving the GA algorithm.

In [107] a task set with asynchronous arrival and no dependencies are scheduled and placed on a FPGA device. In order to ease the placement, task shapes are manipulated by means of footprint transformation. For this case, tasks are assumed to be coarse grain. Complete IP (Intellectual Property) are examples for these tasks: FFT (Fast Fourier Transformation), Ethernet Sender, Ethernet Receiver etc. With these assumptions, this problem is close related to 2D bin-packing problem (e.g., best-fit, first-fit and bottom-left). In this work, heuristics algorithms used to minimize the overall execution time are presented.

These authors further incorporate these concepts on an OS prototype, called WURM - OS [108] which is an extension based on a standard real-time kernel for microprocessors. It provides support for loading, executing and removing tasks and services for inter task communication. Furthermore, hints for online scheduling and placement of tasks are also given.

3.2.6. Run-time Support

The work presented in [109] and [110] identify and discuss the fundamental services that an OS need to provide in order to properly manage reconfigurable resources focusing on those required for run-time execution. These, however, do not differ so much from those defined in Section 3.2.3. Further, in [98] an OS prototype for RC called OS4RC (Operating System for Reconfigurable Computing) is described. Additionally, an extended version of the OS has been incorporated into a complete system called ReConfigME (described in [111]).

In the works described above, the authors define a task as being implemented in hardware, which was previously synthesized having a square shape [98]. Additionally, all tasks have the same shape and are position independent on the RH. The RH, in its turn, can hold several of these tasks and provide means for connection among the tasks placed on the RH device.

In order to execute a given task graph on the RH device, the OS4RC provides the following basic services: Partitioning, Allocation, Placement and Routing. Depending on the task dependencies and the available free space on the RH, a group of tasks may be grouped together forming a partition (application module). This will require, further,

the Routing (for communication) service from the OS to enable the connection of the tasks that are inside an application module. The Allocation service searches for free space on the RH device for a given partition. It may happen that a replacement of some application modules will be required in order to provide the necessary space, or even a re-partition will be required. Figure 3.5 shows an example of a task graph allocated on a FPGA.

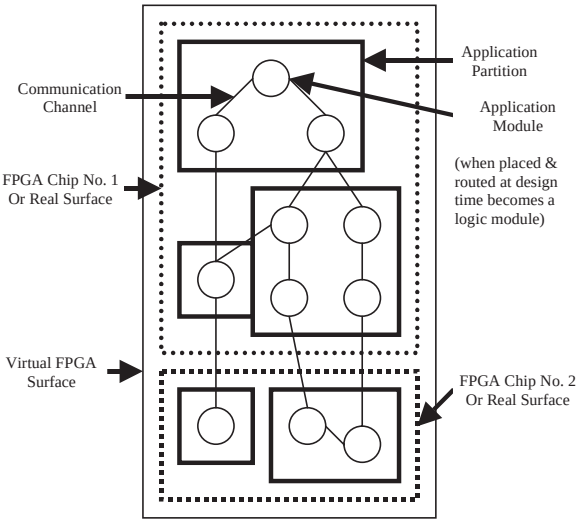


Figure 3.5.: A task graph on FPGA [98].

A rather different approach is presented in [112] and [113]. Here, a reconfigurable hardware (FPGA) is also used as the main computational resource. The reconfiguration of the FPGA and the execution of the high level services of the OS are executed by one CPU. The FPGA surface is partitioned in two basic parts: OS frame and User Area.

The User Area is divided into slots with homogeneous size. Each slot have the same width and spans the device height, as it can be seen in Figure 3.6. On each slot one user task can be placed. In [114] the same authors propose a relaxation of these dimensions constraints by allowing a user task to occupy a multiple number of consecutive slots. The user tasks are designed and synthesized in advance, and it is assumed that user tasks can be allocated to any slot. In addition, the design of each task must fit a well defined template (denoted as Task Communication Block - TCB), which specifies an interface that a task need to provide for communication purposes. A TCB also defines communication channels that allow the OS frame to have access to all slots on the FPGA.

The OS frame is the static part of the FPGA and provides communication services among slots and also establishes the connection with CPU, where task management services are executed. The work gives further hints that preemption and resumption of hardware tasks could be done via config/readback FPGA port.

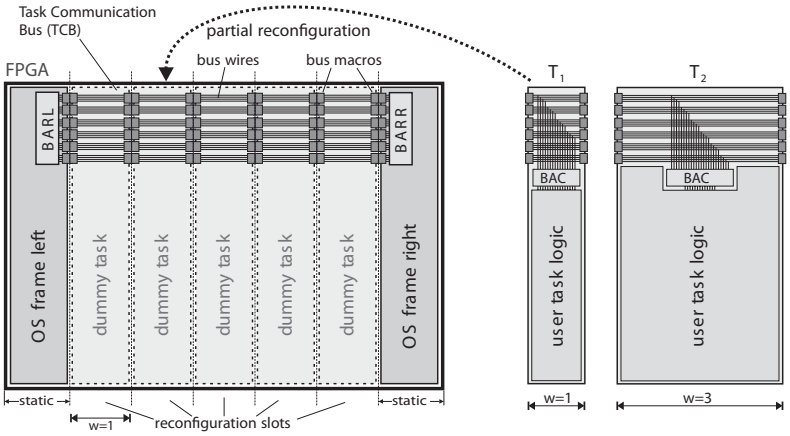


Figure 3.6.: OS frame and task communication block [114].

In [115] the approach is extended in order to allow the partitioning of the RH in a fixed number of slots, where each slot may have different widths. The focus of this work was the evaluation of certain scheduling policies well known from single processor scheduling. Both non-preemptive schedulers (First Come First Serve and Shortest Job First) and preemptive schedulers (Shortest Remaining Processing Time and Earliest Deadline First) were evaluated using different FPGA partitions (varying the number and the widths of the FPGA columns). The evaluation of the results provide some guidance to the designers with respect to the best way to execute FPGA partition based on the characteristics of the input tasks.

The same authors have extended the previous system by including real-time constraints for run-time scheduling and placement of 1D and 2D tasks. By considering tasks having arbitrary and synchronous arrival times some non-preemptive as well as preemptive scheduling and placement algorithms were proposed in [116] and [117], respectively. Additionally, an acceptance test is derived in order to meet deadlines of executing tasks.

Some other authors focus on the efficiency of the scheduler and placer OS services. For instance, in [118] the Scheduler and Placer were combined together into a single method. The work shows some improvements due to the reason that these two services have a strong influence in each other concerning efficiency. Also, in [119] and [120] these two methods are proposed to be implemented completely in hardware, thus, decreasing the overhead they introduce in the system.

Different strategies to execute hardware tasks on a slotted FPGA have also been proposed. For instance, the authors in [121] configure and execute tasks in slots based on priorities. The decision of which task will be configured in each time is made based upon priorities, which change dynamically during system execution. Higher priority tasks have preferred occupancy of the FPGA. The priority assignment policy follows an adaptive rule based on the reconfiguration-request-ratio and communication rate. Additionally,

in order to avoid hardware task starvation some design rules are specified in [121].

Also in [122] and [123] a proposal is presented for a completely reconfigurable computer running on FPGA only. The work focuses on scheduling and memory management for time constrained hardware tasks. The system provides means for running a set of periodic tasks over a FPGA and derives several feasibility analyses based on the resources shared among the tasks considering partial or full reconfiguration of the reconfigurable hardware.

3.2.7. Multitasking Issues

By allowing multitasking on reconfigurable devices one may expect to find the same issues present in multitasking scenario of software only environments. However, some of them may appear in a different manner or may represent a challenge for the implementation phase of the system.

Context switching

Task switching in a preemptive scenario is discussed in [124] and [125]. The authors specify the necessary hardware requirements which enable preemption and resumption of a hardware task. They further define the state extraction and reconstruction based on bitstream manipulation. However, they do not deal with partial reconfiguration. Bitstream manipulation is also proposed in [126], where detailed methods for context saving and restoring are explained for partial reconfigured devices.

Further approaches, requiring some extra circuit added to the user tasks, are proposed in [127]. In this work, context data transformation was also mentioned for allowing task state migration (e.g., between different execution environments).

Reconfiguration port exclusiveness

Most of the approaches used for multitasking on FPGA assume partial reconfigurable devices. However, they seldom respect the sequentially reconfiguration of the FPGA slots due to the mutual exclusive usage of the reconfiguration port. Generally this constraint is neglected because the execution time of a task is assumed to be much higher than the reconfiguration time. Nevertheless, for comparable reconfiguration and execution times of a single hardware task, this may decrease the system performance for time critical scenarios facing high reconfiguration rates. Therefore, the authors of [128] and [129] propose the application of techniques from single processor schedule theory for controlling the access of the FPGA reconfiguration port.

Reconfiguration latency Reconfiguration overhead has also been pointed out by some authors. For instance, in [130] the authors present an algorithm for run-time scheduling of partial FPGA reconfigurations in order to minimize the reconfiguration delay. By

noticing that an user task may appear more than one time in the input task graph, the reconfiguration overhead may be reduced by keeping it into the reconfigurable device for the next execution. Further approaches based on software techniques have being used to reduce or even to hide the reconfiguration latency, by configuring and executing tasks concurrently. Examples of these techniques are based on prefetching, caching and compression of configuration data [31].

Fragmentation

Memory fragmentation appears in software environment when pages are swapped in and out between main and secondary memories. The same concept is present in a device supporting partial reconfiguration. When a hardware task does not completely occupy the FPGA slot, some resources are not used, which characterizes an internal fragmentation. The external fragmentation turns up when the remaining free device area is split into several unconnected vertical stripes (for 1D task model) or rectangles (for 2D task model).

In [131] specific metrics for modeling this situation is proposed, which may improve the results of certain placement strategies. Task relocation and transformation were further approaches presented by some other authors (e.g., in [132] and [133]).

3.2.8. Dynamically Hybrid Architectures

In the work presented in [127] the allocation and scheduling of a dynamically changing set of tasks on a reconfigurable system is investigated. The execution platform comprises one processor and one FPGA. Each task is able to be executed in the reconfigurable device or in the processor. A Scheduler, which decides where a task should be placed (software or hardware), and a Placer (to manage the reconfigurable hardware resource) are the two main components of this system.

The work presented in [134] deals with run-time migration of a task across FPGA and CPU boundaries. The proposal introduces the concept of a tool used to design hardware and software tasks. Such a tool, should guarantee the same behavior of it in a later implementation, regardless of its execution environment (hardware or software). Furthermore, this feature easies the analysis of the system in high level, since it allows the preemption and resumption of tasks across different execution environments. However, neither proper analysis in identification of context data required by a preemption, nor further analyses using the proposed unified representation of a task was provided. In addition, a communication infrastructure is provided, which supports an uniform communication scheme among tasks placed on this hybrid architectures.

The same authors provide further details of the system in [135] and [136], and present an Operating System for Reconfigurable Systems (OS4RS).As a case study a multimedia application was chosen. A video decoder task was designed to be relocatable across

FPGA and CPU domains depending on the available resources and the performance required. Nevertheless, the design of such task was so that no context data transfer was required by the unique relocation point assumed.

Further conceptual analysis for dynamically hybrid systems is done in [137]. The authors consider the problem of allocation of dynamically arriving tasks over an architecture comprising CPU and FPGA. The tasks are capable to be executed in every one of these two environments. Additionally, tasks are able to be relocatable (relocation of tasks over the hybrid architecture by means of reconfiguration). To solve the problem, the authors propose some algorithms for scheduling, relocation management and derive an on-line admission check for the arriving tasks.

3.3. Further Approaches

This section will focus on other approaches where OS may appear in reconfigurable computing. Starting with the idea about the implementation of OS services in hardware by using static hardware (e.g., ASIC), the discussion will then cover some approaches where reconfigurable hardware (e.g., FPGA) is used to execute some OS activities. In addition, this section will discuss a different approach, where FPGA is not used as a coprocessor.

3.3.1. Hardware Accelerator for RTOS

Static approach

For time critical applications where hard real-time constraints are imposed, even the overhead caused by RTOS activities need to be considered, in order to guarantee the feasibility of the overall system. Solutions for this problem were already proposed, for instance, in [138] and [139] where a scheduler was proposed to be implemented in hardware (VLSI circuit) to offload the CPU workload. Thus, a deterministic timing behavior is achieved, which is a mandatory characteristic in hard real-time systems.

The FASTCHART project [140] goes into the same direction. The work proposes a custom CPU that could perform a context switch in one cycle with a kernel coprocessor called the RTU (Real Time Unit). Later on, the RTU was extracted from FASTCHART and implemented as an ASIC [141]. This allows its appliance to a broader range of applications by attaching this ASIC with a general purpose CPU using standard real-time buses.

Comparisons among RTOS-kernel implemented in ASIC, hardware/software co-designed and purely in software have being done in [142] and [143], which shows the improvement on the determinism, caused by shifting the RTOS services execution from CPU to hardware.

Configurable approach

The authors in [144] provide a hardware scheduler, which is able to dynamically change the schedule policy among three different ones (EDF, RM and Priority Based) in a deterministic and fast manner. On their work, dynamically reconfiguration is not used. Instead, three different policies are all implemented in a FPGA and the scheduler switches from one policy to another on-the-fly. This configurable scheduler is provided as an IP, which may be easily integrated in a SoC design.

In [145] and [146] the co-design of a RTOS is proposed. By using a specified framework the user can choose and evaluate the partition of the RTOS modules in hardware and software. The main goal of the work is to help the programmer to explore and find the most appropriated configuration of his system, taken into consideration RTOS activities. By using the framework, the programmer may achieve a higher efficiency in resource usage (compared to a non tailored RTOS) and a speed up in the application execution time. The components provided by the configurable RTOS include services like Deadlock Detection Unit, Dynamic Memory Management Unit among others.

3.3.2. Multithreading on Hybrid Architectures

An interesting approach which skips the common view of a FPGA as a coprocessor has first been presented in [147]. In this work the authors present a framework, which allows a unified model of programming for developers of Massively Parallel Processing (MPP) systems when using hybrid CPU/FPGA-based execution platforms. Assisted by a hardware/software co-designed RTOS, threads running in software and threads running in hardware have means of synchronization and control between each other when sharing the available resources.

The central idea of the proposal lies on the extension of the multithreading programming paradigm for those two different execution environments. A well designed RTOS for such environment shall provide usual services like mutex, semaphore, read-write locks etc, for all threads, independently on their location (software or hardware).

The former authors go further in their investigations and provide more details of their concepts and execution platform in [148] and [149]. The architecture is based on a hybrid FPGA (in this case the Virtex-II Pro from Xilinx), which enables the implementation of a complete system on a single chip. The abstract view of the system can be seen in Figure 3.7.

The architecture provides the standardization and transparency for the programmer either by designing a task for software or for hardware environment. For this purpose, the same API interface is made available for both kinds of threads, which actually compose the hardware and software thread interfaces. These interfaces make the connection to the hardware/software co-designed RTOS and threads. In addition, a proposal for a

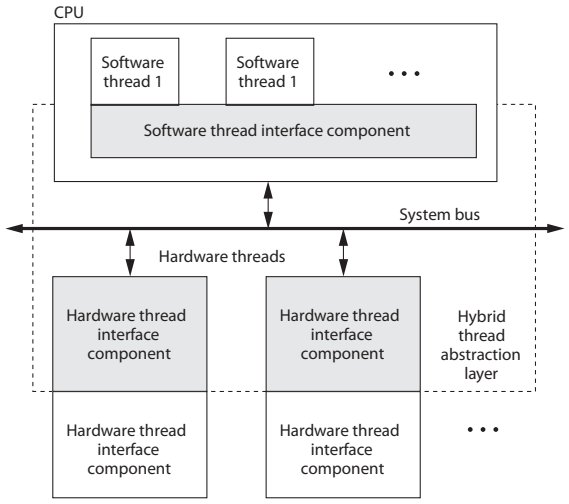


Figure 3.7.: Hybrid thread abstraction layer [148].

common programming language for hybrid threads is proposed in [150], which is based on a translation from C to VHDL language.

The authors propose further the (static) migration of OS functionalities from software to hardware with the intention to improve determinism and offload the CPU from the RTOS execution overhead (same intention of the works related in previous section). At the current status, thread management, thread scheduler, semaphore and thread interrupt controller are some RTOS services already available in hardware ([151] and [150]).

The proposed paradigm, where hardware threads are no longer simple coprocessors, changes the paradigm usually adopted in the reconfigurable computing research area [152]. In this new case, a hardware thread is able to initiate by itself a transaction or request services from the underlying OS and thus interact with the software threads.

Although this new computational model may provide advantages for the contemporary embedded systems platforms, features already investigated by the reconfigurable computing area could improve this scenario. For instance, threads could be dynamically reconfigured to allow dynamically arrival of hardware threads, achieving an equivalent scenario of software threads. Moreover, in a more flexible scenario, threads would be allowed to be dynamically preempted and resumed across execution environment boundary. This would allow the system for dynamically adaptation in order to satisfy the requirements of changing applications, and further use the available resources in a more efficient manner.

Another challenge faced for this paradigm during design phase, is how to partition the application in software and hardware when following such programming model, as noticed in [148]. For this purpose, a first design flow concept is presented in [152].

3.4. Chapter Conclusions

This chapter summarizes relevant works related to (re)configurable operating systems comprising those ones based solely on software, and those ones having support from hardware. Configurable OS can help the designer to provide only those OS services required by the application. Nevertheless, modern embedded systems are becoming multi purpose (one single device needs to execute several and different kinds of applications). With this increasing system complexity a static solution for an operating system is no longer enough. Therefore, the operating system must be (dynamically) reconfigured. Nevertheless, for embedded systems the usage of an operating system introduces an overhead which need to be considered.

3.4.1. Correlation With This Thesis

The architecture investigated in this thesis will rely on the capabilities investigated in [84], namely the ability to partially reconfigure the system using (in the case of a Virtex-II Pro device) the ICAP entity. However, it is not intended to freely relocate the modules on to the device surface. Instead, it will be previously partitioned into a number of slots, in which circuits may be placed by means of partial reconfiguration. Furthermore, the hardcore microcontroller is going to be used, available inside this device and further analysis in higher level will additionally be investigated.

In [127] the authors remain on the conceptual level and do not present further implementation solutions. Similarly, the results presented in [134] and [137] cannot be used in the present thesis, since neither strategies nor methodologies for a deterministic reconfiguration and relocation of components over the hybrid architecture are provided. Additionally, the authors do not consider the reconfiguration overhead of the software and hardware tasks. Moreover, the challenge related to the relocation of tasks across hybrid execution domains are not tackled in their analyses.

The work presented in [148, 152] follows a paradigm where an OS service, regardless its location (either software or hardware), is able to start a transaction with another component (application task or OS service) of the system. Furthermore, application tasks and OS services are statically allocated over the hybrid architecture (offline decision).

The work presented in this thesis differs from the previous work essentially in two aspects. First, the system is able to relocate OS services at run-time. Thus, the design of an infrastructure for dynamic relocatable OS services is necessary. Second, the relocatable OS services are not able to start a communication transaction. Instead, they are activated only when an application task requires its service, and consequently, do not communicate with other OS services.

3.4.2. Additional Comments

The works related to reconfigurable computing area were spent mostly in the investigation on how an operating system can provide efficient support and abstraction of a heterogeneous execution platform to the executing application. However, making the operating system also profit from the reconfigurability and from the high computational performance of such platforms is a promising combination.

The next chapters will introduce a proposal of a run-time reconfigurable RTOS intended to support modern and future embedded systems. This thesis has thereby, the intention to contribute to DREAMS OS, towards aggregation of enough abstraction and seamless integration of reconfigurable hardware in the execution platform. In addition, further methodologies and strategies, which enable the operating system to reconfigure itself in a stand-alone manner (without using DREAMS‘s FRM), are also investigated and proposed.

CHAPTER 4

Run-time Reconfigurable RTOS

This chapter presents conceptual ideas and gives an overview of the proposed real-time reconfigurable operating system devoted to RSoC systems. The OS is depicted in its main parts and briefly explained accompanying the delineation of target applications and target RTOS. Additionally, the hardware architecture is presented.

4.1. System Overview

Figure 4.1 gives an overview of the proposed system highlighting the main components that were investigated inside this work. Both, application and OS, use computational resources from the execution platform, which are provided by the presence of FPGA and CPU in the hardware platform. The application uses the OS services through a specific API, which is monitored by the Runtime Reconfiguration Manager (RRM). Based on the analyses made by the RRM, the reconfiguration of the OS components over the hybrid execution platform may be necessary in order to achieve better resource utilization. This reconfiguration is represented by a run-time relocation of OS services over this architecture. For a better understanding of the whole system, its main components are explained in the following subsections.

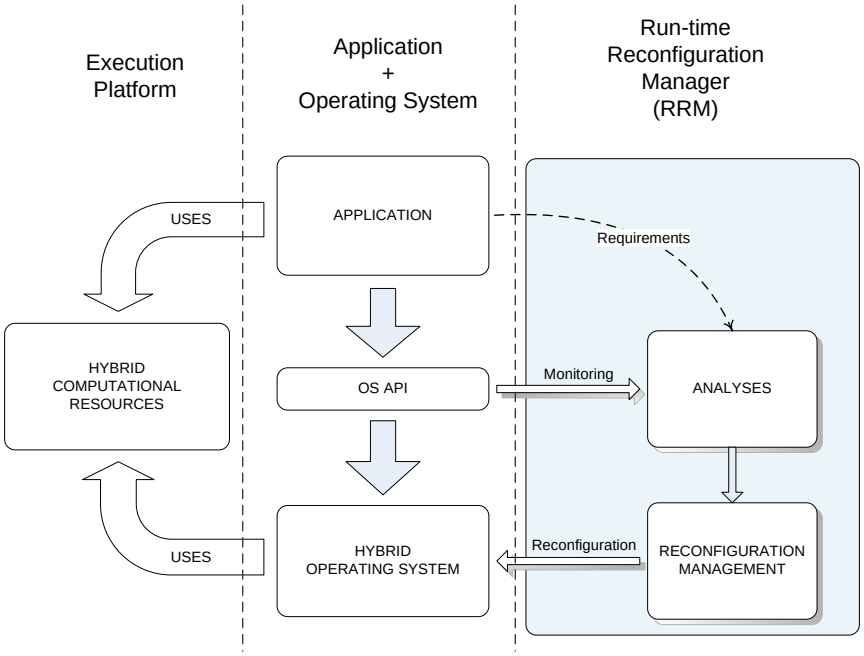


Figure 4.1.: System overview.

4.1.1. Target Applications

The research work presented here focuses on embedded systems whose computational resources requirements cannot be precisely specified, for each instant of time in its lifetime, at design phase. In these cases, each application may be started and/or finished asynchronously. Additionally, one single application may be designed to attend different Quality-of-Services (QoSs), which leads to a varying demand of system resources in time. These situations characterize a dynamic changing environment.

In such systems, due to the changing environment, the requirement demanded by the applications may change during system execution. Therefore, it is not possible, at design phase, to specify an optimal set of computational resources that the system will require during its lifetime. Examples of such embedded systems are modern mobile phones and PDAs with support for different kind of applications: telephone call, music player, movie players, camera, video recorder, text editor, gaming, etc.

In these scenarios, generally soft real-time requirements are necessary to be considered (e.g., multimedia applications), which is also the type of time constraints considered inside this thesis work. Furthermore, applications are represented by a set of tasks that are considered to have periodic behavior. This is a model suitable and generally used for real-time embedded systems.

Tasks may use the CPU as well as the RH (Reconfigurable Hardware) as computational

resources. However, they are mainly software based and only computation intensive software tasks may use the RH. An application specific task that usually profit from a hardware implementation is, for instance, a FFT (Fast Fourier Transformation) computation, which is usually needed for multimedia applications.

4.1.2. Target RTOS

The target RTOS architecture follows the microkernel concept, where application and operating system services are seen as components running on top of a small layer that provides basic functionalities. Figure 4.2 shows abstractly this architecture, where each application task, as well as each OS service, is considered as a component.

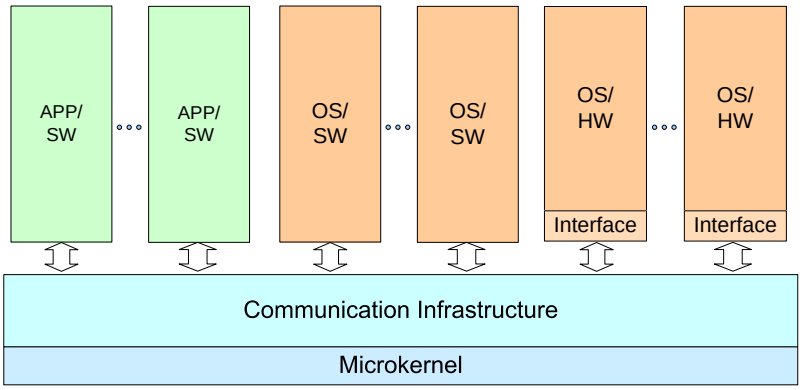


Figure 4.2.: Proposed microkernel based architecture.

The target RTOS DREAMS has been chosen for many reasons. First, because DREAMS supports the concept of components (as presented in the Section 3.1.1). Second, DREAMS also follows the microkernel concept and, if necessary, provides memory protection among the components [153]. Third, the availability of detailed information concerning OS internals, due to the fact that this OS is a result from an in-house research, makes development using DREAMS easier. Last, but not least, DREAMS has a port to the PPC405 processor available in the Virtex-II Pro FPGA, which is the target device used within this work.

The OS reconfiguration is achieved by adding/removing, at run-time, its services and/or relocating them during system execution. For each one of these relocatable services, two different implementation versions are made available (technically synthesized before system starts): one in software and another in hardware. At run-time, these services can be attached and/or detached from the system. For hardware components this relies on the FPGA partial reconfiguration capabilities. For software component versions, known techniques for dynamic linking of software objects are used.

The communication layer, shown in Figure 4.2, provides the necessary support to allow

the communication among components running over the hybrid architecture in an efficient manner. Additionally, this layer provides enough transparency to the application which should not be aware about the location (either FPGA or CPU) of OS services.

At this point a question arrives concerning which OS service is considered to be relocatable and which one shall belong to the application, since both use computational resources of the system. The definition and separation between OS services and application components is usually not clear and difficult to be made (as usual for microkernel architectures). In this work high level OS services are considered for relocation. Examples for this classification are encryption algorithms (which will be used in the case study presented in Chapter 9) and communication protocol stack (e.g., TCP/IP stack protocol [154]). In summary, relocatable OS services, considered for this research, are those services that can be used by more than one application in the system. The remaining OS services may belong to the microkernel layer (e.g., internal communication, synchronization) or stay as fixed components outside this layer.

At the beginning of this thesis research, the dynamic reconfiguration of basic OS services (like scheduling, synchronization mechanisms, communication, etc.) was also considered as possible. However, the mechanisms developed to dynamically reconfigure the high level OS services require the continuously availability of these basic services. Due to this necessity, basic OS services are static and not relocatable.

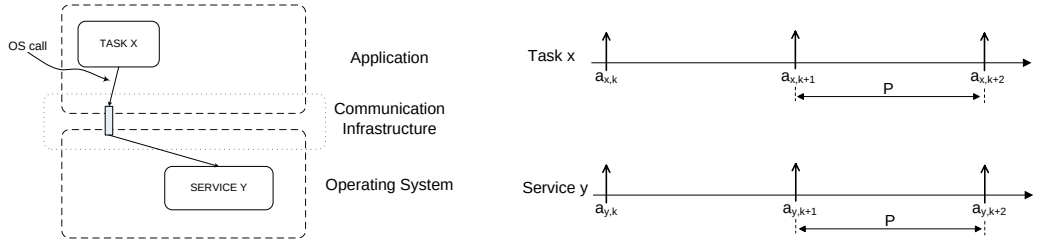
4.1.3. Instrumented OS API

The observation of the OS services usage is an important and necessary feature for a dynamically adaptable OS. In this work, observation is done by instrumenting the OS API in order to provide enough information about OS services (concerning computational resource usage), which is used by the analysis and allocation system parts. With an instrumented OS API it is possible to measure the execution time and frequency of service, either software or hardware services, which are used to estimate the computational resource usage.

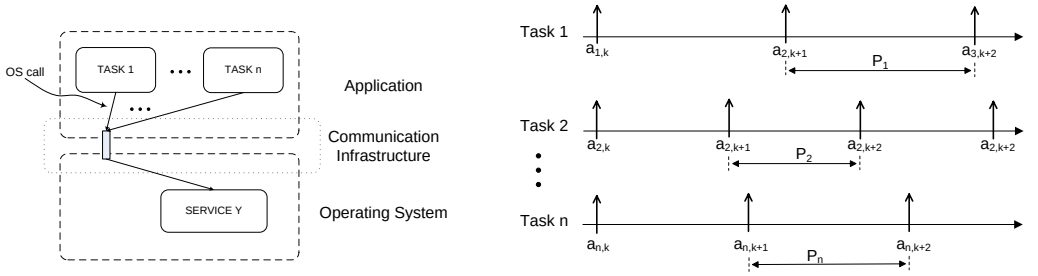
Since applications are modeled as a periodic task set, it can be assumed that OS services will also present a periodic behavior, when called by these application tasks. Nevertheless, each OS service may experience different call patterns, depending on the number of application tasks and their temporal definitions. Different scenarios can be seen in Figure 4.3.

For a service s_y called by only one single periodic task T_x, the period of this service can be precisely determined as being the same of the caller task: $P_x = P_y$. This situation is illustrated in Figure 4.3a. Please note that $a_{x,k}$ (arrival time of instance k of Task T_x) is not necessary equal to $a_{y,k}$ (instant of time when service s_y is called by task T_x in its instance k).

For a service shared among several periodic tasks, its behavior will be still periodic.



(a) OS service s_y being called by a single task T_x.



(b) OS service s_y being called by several tasks.

Figure 4.3.: Call patterns experienced by an OS service.

However, its execution pattern will be determined by the combination of the periods of the tasks using this service. In Figure 4.3b a service s_y is called by a task set $\mathcal{T} = \{T_1, T_2, \ \ldots \ , T_n\}$.

Generally, if for any interval $I = [t, \ t + I)$, the arrival times set $\mathcal{A}_i$ of Task T_i can be determined by Equation 4.1, and so the correspondent arrival times set $\mathcal{A}_{y,i}$ of Service s_y (due to caller task T_i) determined by equation 4.2 (where φ_i denotes the time distance between a_i and a_y, i) , then the arrival times experienced by service S_y in the case of multiple tasks can be determined by Equation 4.3.

$$\mathcal{A}_i(I) = \{a_{i,k} \mid a_{i,k} = kP_i, \ k \geq 0, \ a_{i,k} \ \in \ [t, t + I)\} \tag{4.1}$$

$$\mathcal{A}_{y,i}(I) = \{a_{i,k} \mid a_{i,k} = kP_i \ + \ \varphi_i, \ k \geq 0, \ a_{i,k} \ \in \ [t, t + I)\} \tag{4.2}$$

$$\mathcal{A}_y(I) = \bigcup_{i=1}^{n} \mathcal{A}_{y,i}(I) \tag{4.3}$$

Each task $T_i \in \ \mathcal{T}$ is periodic, determined by a specific period P_i. The behavior of s_y, however, is not periodic in a first glance. This means, that depending on the execution rates of each task T_i, and the interval I selected, no period P_y for service s_y can be

explicitly detected. However, at every hyperperiod H of the task set $\mathcal{T}$, the pattern observed for service s_y in the interval $I^\star$ (where $I^\star = H$) is repeated.

Since the number of OS services is assumed to change dynamically and asynchronously during system execution, the run-time identification of the hyperperiod of the current task set is not a trivial job. Moreover, due to the different combinations of the tasks periods, the hyperperiod can be very large.

In the case of hard real-time systems, worst case scenarios must be considered in the analyses to assure system feasibility. Hence, Equation 4.3 would need to be precisely determined, and the worst case execution time would be considered as the nominal execution time. If a periodic behavior of an event would be required for simplification of the analyses, then the minimal interarrival time would need to be used. In the literature, some other approaches may be used to model such scenarios, like for instance in [155] where a sophisticated arrival function is used to describe a bursty arrival of a method invocation.

For a soft real-time case, one may relax this requirement and use, for instance, averages values as a simplified metric to estimate the arrival rate and execution time. Note that more sophisticated models for this scenarios are available depending on the target application [156]. However, within this work each OS service is considered to present a periodic behavior and these metrics for soft real-time events are used. Nevertheless, a system improvement in this topic is foreseen as a future work, which is further commented in Chapter 10.

Thus, the arrival rate (period) P_y of a service s_y is estimated in the following manner:

$$P_y(I) = \bar{p_y} = \frac{I}{N_y(I)} \tag{4.4}$$

Where I is a predetermined interval ($I = [t, t + I)$) and $N_y(I)$ is the amount of calls to service s_y inside the interval I.

The execution time of the service s_y is assumed as being the average of the execution times measured by the monitoring system. Thus, the execution time E_y of a service s_y is estimated in the following manner:

$$E_y(I) = \bar{e_y} = \frac{\sum_t^{t+I} L_y}{N_y(I)} \tag{4.5}$$

Where L_y is the accumulated execution time of all service calls made inside an interval I and completely finished.

4.1.4. Run-time Reconfiguration Manager - RRM

OS Service Assignment

In this stage the system decides to which execution environment (CPU or FPGA) each OS service will be assigned. This decision is made based upon the computational resources used by each of the OS service in the system. For a software component this is the percentage of the CPU required and for a hardware component this is the amount of FPGA area that it requires (static information gathered after its synthesis process). Additionally, by an arrival of an application into the system, its requirements (OS services needed) are also known by the assignment algorithms.

The assignment decision is driven mainly by the computational resources required by the application. As usual for an embedded execution platform, resources are rare. The OS (which is wanted by the benefits it provides), introduces an overhead in the system. Therefore, the intention is always to direct the remaining resources from the platform to the OS and leave the resources wanted by the application available to it. In summary, the RTOS tries to adapt itself to the remaining computational resources of the hybrid execution platform.

Reconfiguration Management

This module is responsible for the reconfiguration management of each OS service over the hybrid architecture, and it incorporates the low level support necessary to partial reconfigure the FPGA and to dynamically link the software components.

A system reconfiguration is necessary always when the OS service assignment algorithms decides to change the allocation of the components over the platform. This can be caused by the dynamics of the running applications or when an application arrives/leaves the system. Furthermore, since it is assumed that applications are time constrained (not necessarily all of them) the system should carry out the reconfiguration activities in a deterministic manner. In an ideal case, the *blackout*[1] time should be zero or minimized [157]. Therefore, in this approach, deterministic mechanisms known from real-time scheduling theory are adapted and used for this purpose.

4.2. Hardware Architecture

For validation purposes a target architecture based on Virtex-II Pro FPGA, from Xilinx, has been selected. By using its partial reconfiguration capability, hardware components may be reconfigured on the FPGA at run-time whilst the remaining part of the device keeps on executing. Additionally, the availability of an embedded hardcore general

[1] Blackout is the time interval during which the system do not react due to the reconfiguration.

processor embedded in this FPGA device, features it with the necessary heterogeneity for accomplishment of the requirements of a hybrid reconfigurable architecture.

As noticed before, DREAMS has a port to the PPC405 processor available in the Virtex-II Pro FPGA. Nonetheless, support for dynamically reconfiguration was not supported by this RTOS, which actually is one of the contributions of this thesis.

The execution platform proposed can be seen in the Figure 4.4a, where details were omitted for a better understanding. The FPGA surface is spitted into static and reconfigurable parts. The static part embraces the CPU system, where the RRM and the OS services in software are executed. The reconfigurable part is divided into n slots and each slot provides an OS service framework (Figure 4.4b). Additionally, each slot can be dynamic reconfigured by means of partial reconfiguration. In this particular, the proposed approach is similar to those found in the literature, which were discussed in Chapters 2 and 3.

Each subsystem (software and hardware) has its own local bus for communication. These buses are connected through a dedicated bridge to provide communication across CPU and FPGA components. Further details of the execution platform are given in Chapter 9.

The local memory is used to support the communication between local components, and the global shared memory is used to perform the communication with components running on the CPU. The local controller is used to manage the access to the local memory and the global controller, which together with its counterpart in software, performs the communication infrastructure layer mentioned previously in Section 4.1.2.

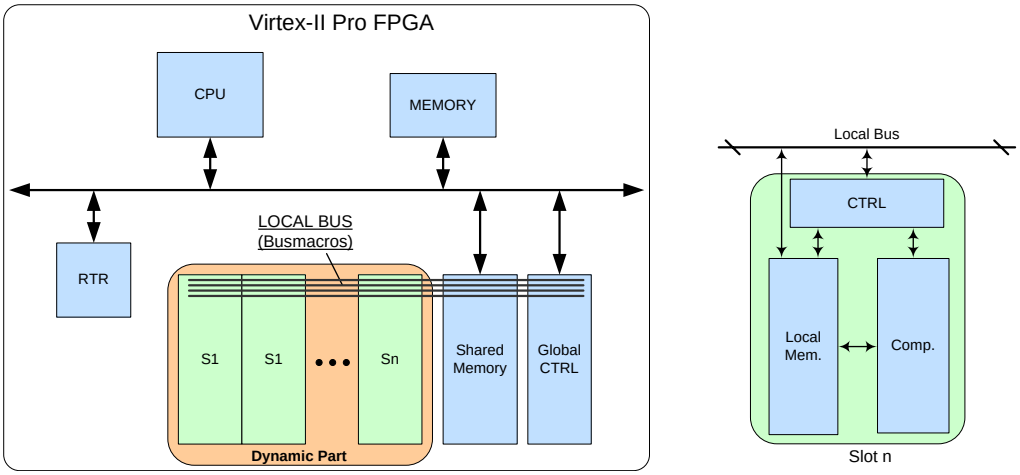
The slots are connected using *Busmacros*. In order to program the FPGA slots, the reconfiguration port is used, which may be local (by using the ICAP Xilinx entity) or an external Run-Time Reconfiguration (RTR) controller.

4.3. Design Support

In order to make the system accessible for further developments, a proper support in the design phase needs to be made available. Related to this topic, the following points are handled in this thesis work:

HW/SW Interface The generation of the interface between hardware and software is an important aspect in such reconfigurable architectures. Since hardware components may be relocated, removed or inserted dynamically into the FPGA slots, there is a need to define in which way these reconfigurable components can establish a communication channel with the static part of the system.

Relocatable Tasks Since a dynamic migration of components between hardware and software domains are necessary for this approach, appropriate mechanisms are required for its achievement. If preemption and resumption of a component across



(a) Architecture overview.

(b) OS service slot template.

Figure 4.4.: System architecture.

CPU-FPGA boundaries are required, there is a need to assure a safe context translation and migration of this component. Therefore, a suitable environment for designing of such relocatable tasks is necessary, along with a run-time environment to carry out these activities on-the-fly.

4.4. Chapter Conclusions

In this chapter a proposal for a real-time reconfigurable operating system was presented, which is based upon reconfiguration of its services over the hybrid architecture. The intention is to promote a flexible utilization of the computational resources, both CPU and FPGA, by the operating system services and application tasks. Thus, an efficient usage of the execution platform can be achieved, even if the system requirements dynamically changes .

For its achievement, the system needs to monitor the resource utilization and, by using appropriated analyses, decide which OS services shall be relocated. Moreover, reconfiguration activities need to be performed in a deterministic manner, since the system must guarantee soft-real time constraints.

A preliminary investigation towards pattern identification of operating system service usage has been presented. Nonetheless, as already noticed, these analyses need to be improved for a better identification of their usage profiles. This investigation is planned as a future work.

CHAPTER 5

Modeling & Problem Formulation

This chapter introduces the model costs for OS services, communication among services and reconfiguration activities. These models allow a suitable formulation of the main problems investigated in this thesis.

5.1. Component Assignment

The problem of assigning OS components over two execution environments can be seen as a typical assignment problem. Indeed, this problem can be formulated as an Integer Linear Program (ILP), or more specifically, as a Binary Integer Program (BIP), which characterizes the component assignment problem as a NP-Hard one.

Let $\mathcal{S}$ be a set which denotes the relocatable OS services (hereafter, *component* and *service* will be used as interchangeable terms):

$$\mathcal{S} = \{s_i,\ i = 1, ..., n\}. \tag{5.1}$$

Since the computational resources are intended to be managed between operating system and application, they have to be measured. Therefore, every component i has an estimated cost $c_{i,j}$, which represents the percentage of computational resource from the execution environment used by this component. On the FPGA ($j = 2$) it represents the area needed by the component: A_i. On the CPU ($j = 1$) it represents the processor load: U_i. Thus, for each component i that is relocatable, there are two different costs

associated:

$$c_{i,j} : \begin{cases} c_{i,1} = U_i & \text{denotes the processor work load required by component } i \\ c_{i,2} = A_i & \text{denotes the FPGA area required by component } i \end{cases} \tag{5.2}$$

Within this thesis, the component costs of a component i may also be denoted in a shorter form:

$$c_i = \{U_i, A_i\} \tag{5.3}$$

The assignment of a component, to either CPU or FPGA, is represented by the variable $x_{i,j}$. It is said that $x_{i,j} = 1$ if the component i is assigned to the execution environment j and $x_{i,j} = 0$ otherwise:

$$x_{i,j} : \begin{cases} x_{i,j} = 1 & \text{if component } i \text{ is located on environment } j \\ x_{i,j} = 0 & \text{otherwise} \end{cases} \tag{5.4}$$

For legibility reasons, the notation above may also appear in the following form:

$$x_i = \begin{cases} \{1, 0\} & \text{if component } i \text{ is assigned to the CPU, or} \\ \{0, 1\} & \text{if component } i \text{ is assigned to the FPGA.} \end{cases} \tag{5.5}$$

Since only one distinct version of a specific component (either $j = 1$ or $j = 2$) is allowed to be used at the same time, the component i can be assigned only to one of the execution environments. Thus, the following equation must hold:

$$\sum_{j=1}^{2} x_{i,j} = 1 \quad \text{for every} \quad i = 1, ..., n. \tag{5.6}$$

5.1.1. Constraints Definition

The computational resources that the operating system can use are bounded and depend on the resources currently required by the application. These limits imposed to the operating system are denoted by:

U_{max} Maximum CPU workload;

A_{max} Maximum FPGA area.

Thus, the total FPGA area (A) and the total CPU workload (U) used by the hardware and software components, respectively, can not exceed their maximums. These derive two constraints to the BIP problem:

$$U = \sum_{i=1}^{n} x_{i,1} c_{i,1} \leq U_{max} \tag{5.7}$$

$$A = \sum_{i=1}^{n} x_{i,2} c_{i,2} \leq A_{max} \tag{5.8}$$

Besides these constrains, an additional one is defined in order to maintain a balanced utilization of the CPU and FPGA resources. This is necessary for two main reasons. First, it has to be avoided that one of the execution environments would have its usage near to the maximum. This condition improves the performance of the reconfiguration algorithms which require enough CPU workload and FPGA area for their feasibility (discussed further in Section 6.3).

A second reason is the need to influence the migration of operating system components to the execution environment which are going to be less used by the application. For instance, if the application is going to require more FPGA resources, due to the necessity to execute high performance tasks, the operating system shall migrate to the CPU. In an opposite way, when the CPU is mostly used by the application, the operating system may take advantage of the FPGA for its execution.

This balancing is denoted by B and is constrained by a value δ:

$$B = |w_1 U - w_2 A| \leq \delta \tag{5.9}$$

Where δ is the maximum allowed unbalanced resource utilization and the weights w_1 and w_2 are used to proper direct the migration of the components of the operating system as described above. Hence, the weights w_1 and w_2 are used to reflect the tendency of resource utilization by the application. So, it is possible to guide the assignment of the OS services, which shall use the resources currently discarded by the application.

5.1.2. Objective Function Definition

An operating system should use computational resources from an embedded system execution platform as less as possible. In this way, more resources are available to execute application tasks. Therefore, the objective function used to minimize the overall

resource utilization is defined as:

$$min\{\sum_{j=1}^{2}\sum_{i=1}^{n} c_{i,j}x_{i,j}\} \tag{5.10}$$

The solution for Equation 5.10, subjected to the constraints defined in 5.7, 5.8 and 5.9, are the assignment variables $x_{i,j}$ which are defined as being a specific system configuration: $\Gamma = \{x_{i,j}\}$.

5.1.3. Allocation Example

For a better understanding of the OS service allocation problem, a simple problem involving two relocatable services is shown in Table 5.1. Assuming a situation where two OS services present the estimated costs given in Table 5.1a, there are four different assignments of these two services to the two execution domains. If the goal is to achieve the minimal overall computational resources usage, the solution presented in Table 5.1e is the best one. Note that, in this example, the constraints defined in Equations 5.7, 5.8 and 5.9 are not considered.

5.2. Reconfiguration Costs

Due to the application dynamics OS requirements may change. This is represented by changes in $c_{i,j}$ and the size of $\mathcal{S}$ ($|\mathcal{S} = n|$). Moreover, the available computational resources that are available to allocate the demanded OS components may also change. Therefore, the assignment decision needs to be continuously checked. This implies that, depending on the result of the new OS service assignment, some of the OS components may need to be relocated (reconfigured) by means of migration. In other words, a service may migrate from software to hardware or vice-versa. Additionally, services may be replaced by new ones (in order to use less/more resources). In this case, a reconfiguration of a service in the same execution environment occurs (hereafter also called migration).

The reconfiguration cost of every OS component represents the time required to completely migrate a component from one execution environment to the other one. Therefore, to every possible component migration, a correspondent cost is specified. If o is the source environment of a component and p is its destination, the value $r_{o,p}$ denotes the migration cost. So, for each component i a migration cost matrix $\mathbf{R}_i$ is defined. Each entry in this matrix denotes one possible migration cost of the component.

Since a new component may arrive into the system, or a service that is no longer required may leave, one can incorporate these services inclusions and exclusions costs in order to

	CPU	FPGA
$c_{1,j}$	5	10
$c_{2,j}$	15	10

(a) Component costs specification.

	CPU	FPGA
$x_{1,j}$	1	0
$x_{2,j}$	1	0
$\sum$	20	
B	20	

(b) All components in software.

	CPU	FPGA
$x_{1,j}$	0	1
$x_{2,j}$	0	1
$\sum$	20	
B	20	

(c) All components in hardware.

	CPU	FPGA
$x_{1,j}$	0	1
$x_{2,j}$	1	0
$\sum$	25	
B	5	

(d) s_1 in hardware and s_2 in software.

	CPU	FPGA
$x_{1,j}$	1	0
$x_{2,j}$	0	1
$\sum$	15	
B	5	

(e) s_1 in software and s_2 in hardware.

Table 5.1.: Different allocation possibilities for two services, and the respective overall cost and balance values.

get a more precise information about the overall reconfiguration costs. This situation can be modeled by defining a third virtual environment (which can be seen as a memory pool of OS components). For instance, $r^i_{3,p}$ represents the cost for aggregating a new service i into the system. Thus, the migration cost matrix has a size of 3×3.

The cost to migrate a component i in the system can be easily calculated using the current and new allocation of this component. Let $R_i = \{r^i_{j,j'}\}$, where j and j' are the current and new execution environment of component i. If $x_i = \{x_{i,1}, x_{i,2}, x_{i,3}\}$ and $x'_i = \{x'_{i,1}, x'_{i,2}, x'_{i,3}\}$ are the current and new assignment of the component i, then its reconfiguration cost is calculated by the equation below:

$$R_i = \begin{bmatrix} x_{i,1} & x_{i,2} & x_{i,3} \end{bmatrix} \begin{bmatrix} 0 & r'_{1,2} & r'_{1,3} \\ r'_{2,1} & 0 & r'_{2,3} \\ r'_{3,1} & r'_{3,2} & 0 \end{bmatrix} \begin{bmatrix} x'_{i,1} \\ x'_{i,2} \\ x'_{i,3} \end{bmatrix} = x_i^T \mathbf{R}_i x'_i \tag{5.11}$$

Where $r_{i,i} = 0$, $\{x_{i,1} + x_{i,2} + x_{i,3}\} = 1$ and $\{x'_{i,1} + x'_{i,2} + x'_{i,3}\} = 1$.

The complete reconfiguration cost R (overall reconfiguration time) of the system is de-

fined as:

$$R = \sum_{i=1}^{n} x_i^T \mathbf{R}_i x_i' \tag{5.12}$$

5.2.1. Temporal Specification

As usual for a dynamic reconfigurable system, reconfiguration activities represent an extra overhead for the system. Depending on the system specifications, this overhead may be prohibitive if not properly tackled. For real-time systems these activities must be specially taken into account in order to guarantee temporal determinism.

Therefore, the activity r^i to reconfigure a component is divided into two distinct phases, called *Programming* and *Migration* phases. During *Programming*, the component is programmed on the FPGA (by downloading the corresponding partial bitstream), or the object code is placed in the main memory (if its software version is going to be used).

Once having the FPGA programmed with the service's bitstream, or the CPU software linked with the service's object code, the effective migration can start (*Migration* phase). This phase comprises the activity of transferring the internal data of a service between two execution environments, and afterwards its activation.

Table 5.2 presents the notations used to represent different migration costs. Based on the implementation results, the time to execute *Migration* can be considered to be the same for a task migrating from software to hardware or vice-versa (parameter M^w). More details on technical realization concerning migration of a running component across hardware and software will be given in Chapter 8.

Table 5.2.: Time costs related to each migration case.

Parameter	Description
Q_i^s	Programming time in software
Q_i^h	Programming time in hardware
M_i^s	"Migration" time from software to software
M_i^h	"Migration" time from hardware to hardware
M_i^w	Migration time between hardware/software

Furthermore, to proper tackle reconfiguration activities together with the executing OS services, an extension of the component cost model is required. Actually, their temporal parameters need to be specified. As mentioned in Section 4.1.1, in a typical embedded real-time system the application may be modeled as a set of periodic activities. Even

sporadic tasks can also be modeled as periodic tasks through the assumption that their minimum interarrival time being the period. Hence, it is assumed that the applications are represented by a set of periodic tasks and that OS services present also a periodic behavior. Thus, each OS service is characterized by temporal parameters (e.g., period, deadline, etc.). The temporal specification of a service, related to its periodic behavior is presented in Table 5.3.

Table 5.3.: Service definition related to its periodic execution.

Parameter	Description
E_i^s	Execution time of service i in software
E_i^h	Execution time of service i in hardware
P_i	Period of service i
D_i	Relative deadline of service i
$d_{i,k}$	Absolute deadline of the kth instance of service i
$a_{i,k}$	Arrival time of the kth instance of service i
$b_{i,k}$	Starting time of the kth instance of service i
$f_{i,k}$	Finishing time of the kth instance of service i

5.3. Communication Costs

Additionally to the computational resources that each component requires, it is also desirable to take into account costs related to communication among the components. Observing the underlying execution platform used in this work, these costs depend on the allocation of the components over the architecture. For instance, if two communicating components are located in the same environment, the related cost may be lower than the case where each would be placed on a different execution environment.

For the purpose of communication cost modeling, the proposed architecture shown in Figure 5.1, already introduced in the previous chapter, provides enough abstraction for a proper communication cost modeling. There, communication channels available are highlighted. It can be seen that software components can exchange data among themselves using the *SW BUS*. Similarly, hardware components can use *HW BUS* for this purpose. To exchange data across the hardware/software boundary, a bridge connecting *SW BUS* and *HW BUS* need to be used, aggregating more cost for this communication. This would be more in evidence if the two architecture parts would be implemented in different devices.

In order to model each possible communication between each pair of components, the system has been modeled using a undirected weighted graph $G = (\mathcal{V}, \mathcal{E})$. Each edge in $\mathcal{E}$ connecting two components u and v is weighted by $\kappa(u, v)$, which represents all possible

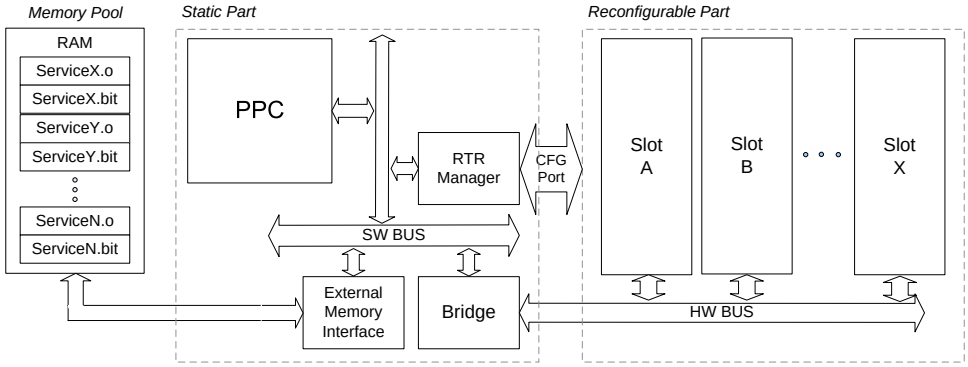


Figure 5.1.: System architecture highlighting the communication channels.

communication costs between u and v. Note that κ depends on two main factors: a static one, related to the architecture (time to deliver a message), and a factor related to the amount of data exchanged between two components, which is dynamic and application dependent. As each component may be located in one of two different execution environments, $\kappa(u, v)$ represents three different communication costs ($\kappa(u, v) = \{C^{\alpha}, C^{\beta}, C^{\gamma}\}$):

- C^{α}, when both are in SW domain;
- C^{β}, when both are in HW domain;
- C^{γ}, when each component is located in a different domain.

Figure 5.2 shows a sample of such a graph. Grey nodes in the graph may be seen as OS services primitives (API) that are made available for application tasks. Note that such nodes do not have allocation costs.

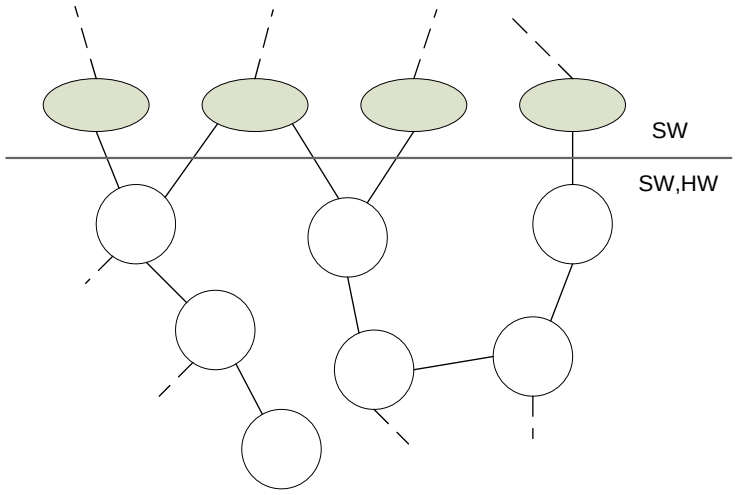


Figure 5.2.: Sample of an OS component graph.

5.4. Chapter Conclusions

The main problems to be dealt by RRM were discussed in this chapter. It can be seen that those problems are NP-Hard ones, which requires the development of heuristics for their solutions. Moreover, these heuristics need to be executed at run-time, which demands the development of algorithms that present low time complexity.

Furthermore, a model of the reconfiguration activities was introduced, which consider the proposed underlying execution platform. Only by having all activities modeled by its temporal characteristics, it is possible to tackle them in a deterministic manner. Such a complete and embracing model of reconfiguration activity is not usually adopted, as noticed from the works investigated in Chapter 3.

CHAPTER 6

Run-Time Methods

Up to now, the main problems have being properly introduced and modeled, which allows this chapter to present the solutions proposed to solve them.

First, an allocation algorithm, considering only the computational resource required by each component, is introduced. Then, an attempt to indirectly reduce the overall reconfiguration cost is presented. Furthermore, an algorithm extension is proposed, with the aim to reduce the communication costs among OS components. By the development of these algorithms, efforts were spent in order to achieve a low complexity due to the fact that they need to be applied at run-time, during system execution.

Afterwards, the methods designed to handle the reconfiguration activities are presented, considering thereby time constraints. Here, the reconfiguration activity model is further extended enabling an efficient management. For the reconfiguration of a single OS component, methods from real-time scheduling theory are adapted and applied for this specific problem. Furthermore, since in a system reconfiguration situation, more than one component may suffer a reconfiguration, it is also required to decide the order in which these components will be reconfigured.

6.1. OS Service Allocation

The assignment problem is solved by using a greedy based heuristic algorithm. It decides at run-time where to place each OS component taking into consideration its current cost and the remaining available computational resources. As it has been explained

in the previous chapter, the system has to allocate the OS components to a limited FPGA area (A_{max}) and limited CPU processor workload (U_{max}). The heuristic tries to minimize the objective cost function (Equation 5.10) subjected to system constraints (Equations 5.7, 5.8 and 5.9).

The allocation algorithm is composed of two phases. The first one creates two clusters (FPGA and CPU component sets), representing the assignment of components to either CPU or FPGA. The second phase improves the first solution towards the balance constraint δ.

6.1.1. OS Service Assignment Phase

In the first phase, shown in Algorithm 1, the algorithm starts selecting the component that has the smallest cost among those currently needed by the application, and assigns it to the corresponding execution environment. It then selects the component that has the smallest cost among the remaining (unassigned) ones so that it tries to keep the usage of CPU resource U equal to the FPGA resource A. This selection process is repeated until all components have been assigned. The algorithm terminates by checking if the CPU and FPGA resources usage constraint are fulfilled: $U \leq U_{max}$ and $A \leq A_{max}$. If so, the algorithm returns a valid assignment solution, or an error otherwise. It can be seen that this algorithm has a polynomial complexity of $O(n^2)$, since there is only one *for* loop (line 7) which produces n searches in a list of (maximum) n elements.

6.1.2. OS Service Assignment Example

To better understand how the proposed algorithm works, an example is presented here. Assuming three components, which are able to be located either in CPU or in FPGA, and having respective estimated costs as shown in Table 6.1, the algorithm will take three steps to provide the assignment solution. At the beginning, the component having smallest cost among them is selected, which in this case is s_2 in its CPU version ($j = 1$). Thus, in the first step (Table 6.2a), the assignment decision for this component is made: $x_2 = \{1, 0\}$. For the second step (Table 6.2b), the component having the smallest cost, from those available for FPGA, is selected (since the result from first step implies in $U > A$). Hence, component s_1 in its FPGA version ($j = 2$) is selected: $x_1 = \{0, 1\}$. This leads to $U = 3$ and $A = 5$, which determines the search for the smallest cost in the remaining unselected components of CPU versions. As, in this case only one component remains unselected, the choose lies on service s_3: $x_3 = \{1, 0\}$ (Table 6.2c). The resulted overall resource utilization is $\sum = U + A = 23$, with a Balance $B = |U - A| = 13$.

Algorithm 1 Service assignment heuristic.

$C_1 \leftarrow \{c_{i,1}\}$ Set of components available for CPU ($j = 1$)
$C_2 \leftarrow \{c_{i,2}\}$ Set of components available for FPGA ($j = 2$)
$X_1 \leftarrow \{x_{i,1}\}$ Assignment of CPU components
$X_2 \leftarrow \{x_{i,2}\}$ Assignment of FPGA components
$C \leftarrow C_1 \cup C_2$; $X1 \leftarrow X_1 \cup X_2$; $U \leftarrow 0$; $A \leftarrow 0$
$n \leftarrow$ number of components
for $k \leftarrow 1$ to n **do**
    if $U \leq A$ **then**
        Find an unassigned component i among $\{c_{i,1}\}$ so that it has the smallest cost.
        Assign this component to CPU: $x_i \leftarrow \{1, 0\}$
    else
        Find an unassigned component i among $\{c_{i,2}\}$ so that it has the smallest cost.
        Assign this component to FPGA: $x_i \leftarrow \{0, 1\}$
    end if
    $U \leftarrow C_1 X_1^T$
    $A \leftarrow C_2 X_2^T$
    if $U > U_{max}$ or $A > A_{max}$ **then**
        Exit with error: "Not feasible"
    end if
end for
return $X1$

	$c_{1,j}$	$c_{2,j}$	$c_{3,j}$
CPU	10	3	15
FPGA	5	10	12

Table 6.1.: Example of three components and their respective costs.

6.1.3. Balance B Improvement Phase

In the second phase the balancing B is improved in order to meet the δ constraint which was not considered in the first phase. It is based on Kernighan-Lin algorithm [158] and it aims to obtain a better balancing B than the first one by swapping pairs of components between CPU and FPGA. The algorithm receives as input the first assignment solution $X1$ which has $nc_1 = \sum_{i=1}^{n} x_{i,1}$ components assigned to CPU and $nc_2 = \sum_{i=1}^{n} x_{i,2}$ components assigned to FPGA. The maximum number of pairs that are possible to be swapped is defined as: $max\_pairs = min(nc_1, nc_2)$.

By moving a component i that was, for instance, previously assigned to CPU ($x_i = \{1, 0\}$), to FPGA ($x'_i = \{0, 1\}$) a new balancing B is achieved: $B_{new} = |B_{current} - l_i|$, where $l_i = \{c_{i,1} + c_{i,2}\}$. Similarly, by moving a component i from FPGA to CPU, the new balancing B will be: $B_{new} = |B_{current} + l_i|$. Thus, swapping a pair of components

	$x_{1,j}$	$x_{2,j}$	$x_{3,j}$	$\sum$
CPU		1		3
FPGA		0		0

(a) Step 1.

	$x_{1,j}$	$x_{2,j}$	$x_{3,j}$	$\sum$
CPU	0	1		3
FPGA	1	0		5

(b) Step 2.

	$x_{1,j}$	$x_{2,j}$	$x_{3,j}$	$\sum$
CPU	0	1	1	18
FPGA	1	0	0	5

(c) Step 3.

Table 6.2.: Assignment algorithm applied in the example presented in Table 6.1.

o, p the new balancing B is defined as:

$$B_{new} = \begin{cases} |B_{current} - l_o + l_p|, & \text{if } x_o = \{1,0\} \text{ and } x_p = \{0,1\} \\ |B_{current} + l_o - l_p|, & \text{if } x_o = \{0,1\} \text{ and } x_p = \{1,0\} \end{cases} \tag{6.1}$$

Additionally, G_{op} is defined as the gain obtained in the balancing B by swapping a pair o and p of components: $G_{op} = B_{current} - B_{new}$. A gain above 0 means an improvement obtained in the balancing B. Observing Equation 6.1, it follows that:

$$G_{op} = \begin{cases} l_o - l_p, & \text{if } x_o = \{1,0\} \text{ and } x_p = \{0,1\} \\ l_p - l_o, & \text{if } x_o = \{0,1\} \text{ and } x_p = \{1,0\} \end{cases} \tag{6.2}$$

The balance improvement algorithms is shown in Algorithm 2. Before starting, it first build a set $\mathcal{M}$ of component pairs, respecting the following: For each pair $\{o, p\} \in \mathcal{M}$ it must hold either $x_o = \{1,0\}$ and $x_p = \{0,1\}$; or $x_o = \{0,1\}$ and $x_p = \{1,0\}$.

Afterwards, it starts trying to swap all possible pairs and storing the obtained gain by every try. It then chooses the one that provides the greatest gain. If this gain is lower than or equal to zero, no swapping is able to provide an improvement in the balancing B and the algorithm stops. Otherwise, the pair that provides the greatest gain is swapped and locked (no longer a candidate to be swapped).

The above process is repeated until all pairs have been locked or no improvement can be obtained by any interchange or if δ constraint has been fulfilled. The algorithm terminates by returning the new assignment solution that provides a better (or at least an equal) balancing B than the solution provided by the first phase. The overall complexity of the balancing improvement algorithm is (worst case) $O(m^2)$, where m is the maximum number of pairs. The complexity of building the set $\mathcal{M}$ is $O(m^2)$, in its worst case, and the search (line 10) made inside the loop (line 9) has also this complexity.

Algorithm 2 Heuristic for balancing B improvement.

1: $X_1^{init} \leftarrow \{x_{i,1}\}$ Initial assignment of CPU components
2: $X_2^{init} \leftarrow \{x_{i,2}\}$ Initial assignment of FPGA components
3: $X^{init} \leftarrow X_1^{init} \cup X_2^{init}$
4: $X^{new} \leftarrow X^{init}$
5: $B^{init} \leftarrow |U^{init} - A^{init}|$
6: $B^{new} \leftarrow B^{init}$
7: Build the set $\mathcal{M}$
8: $m \leftarrow |\mathcal{M}|$ maximum number of pairs
9: **for** $k \leftarrow 1$ to m **do**
10: Find the pair $\{o, p\} \in \mathcal{M}$ so that o and p are unlocked and G_{op} is maximal
11: **if** $G_{op} > 0$ **then**
12: Swap o and p changing the current assignment: $X^{new} \leftarrow (X^{new}$ with o and p swapped)
13: $B^{new} \leftarrow B^{new} - G_{op}$
14: Lock o and p
15: **end if**
16: **if** $G_{op} \leq 0$ OR $B^{new} < \delta$ OR all pairs are locked **then**
17: **break**
18: **end if**
19: **end for**
20: **return** X^{new}

6.1.4. Balance B Improvement Example

Again, the algorithm can be better understood by using an example. For this case let the input be the result achieved by the assignment phase when applied on the given example. Table 6.3c shows a complete example of an OS service set allocation. In Table 6.3a the OS service set input is presented, along with their respective costs and l parameter. Table 6.3b shows the assignment result when applying Algorithm 1 on this input example. Finally, Table 6.3c presents the balancing B improvement when applying Algorithm 2 on the previous result.

For this case, $\mathcal{M}$ comprises two services pairs, which are $\{o, p\} = \{s_1, s_2\}$ and $\{o, p\} = \{s_1, s_3\}$. The second pair delivers the best gain in comparison to the first one. Hence, changing the assignment of s_1 and s_3 from $x_1; x_3 = \{0, 1\}; \{1, 0\}$ to $x_1; x_3 = \{1, 0\}; \{0, 1\}$ a better balancing B is achieved. However, this swap implies in a small increase in the overall resource utilization. This situation is shown in Table 6.3c.

	$c_{1,j}$	$c_{2,j}$	$c_{3,j}$
CPU	10	3	15
FPGA	5	10	12
l_i	15	13	27

(a) Example of three components and their respective costs.

	$x_{1,j}$	$x_{2,j}$	$x_{3,j}$	$\sum$
CPU	0	1	1	$U = 18$
FPGA	1	0	0	$A = 5$
	s_1	s_2	s_3	

(b) Assignment result after Algorithm 1.

	$x_{1,j}$	$x_{2,j}$	$x_{3,j}$	$\sum$
CPU	1	1	0	$U = 13$
FPGA	0	0	1	$A = 12$
	s_1	s_2	s_3	

(c) Assignment result after Algorithm 2.

Table 6.3.: Example of a complete OS service allocation.

6.1.5. Reconfiguration Cost Reduction

Since reconfiguration activities represent an overhead for the system, it is desired to reduce it. For this purpose the assignment algorithm, in its second phase, was slightly modified in order to decrease the number of components to be reconfigured. This action indirectly decreases the overall reconfiguration cost.

Assuming that Γ and Γ' are the current and new system configuration, let the assignment difference z_i of a component i be denoted by $diff(x_i, x'_i)$. Where $x_i \in \Gamma$ and $x'_i \in \Gamma'$. Thus, $diff$ is defined as follows:

$$z_i = \begin{cases} 1 & : \texttt{if } \{x_i\} \neq \{x'_i\} \\ 0 & : \texttt{otherwise} \end{cases}$$

The new algorithm phase is shown in Algorithm 3. The original algorithm is modified in the following manner. Instead of immediately changing and locking the position of the pair o and p after a gain below 0 was found (line 12 of Algorithm 2), this is done based on some rules. If, at least, one of the components from the pair keeps its position in relation to the current system configuration (line 12 of Algorithm 3), the pair swap is allowed.

In addition, the component that preservers its position (or both) is locked (no longer a candidate to be swapped). However, if both components of the pair change their positions in relation to the current system configuration, no swap occurs. Moreover, just one component (which provides the smaller reconfiguration cost) is locked. This lock is necessary, otherwise the algorithm would not terminate. This process is then repeated until all pairs have been locked or no improvement can be obtained by any interchange, similarly as in the original algorithm phase.

By applying those rules, the algorithm tries to reduce the number of components needed to be reconfigured. Also note that the algorithm does not terminate if the δ constraint is fulfilled. This enforces the search for more components (pairs) that could be kept in its current allocation solution. In respect to its complexity, it is the same than the original algorithm.

Algorithm 3 Improved heuristic for balancing B.

1: $X_1^{init} \leftarrow \{x_{i,1}\}$ Initial assignment of CPU components
2: $X_2^{init} \leftarrow \{x_{i,2}\}$ Initial assignment of FPGA components
3: $X^{init} \leftarrow X_1^{init} \cup X_2^{init}$; $X^{new} = X^{init}$
4: $B^{init} \leftarrow |U^{init} - A^{init}|$; $B^{new} = B^{init}$
5: $X^{orig} \leftarrow$ *Current System Configuration* Γ
6: $m \leftarrow max\_pairs$ maximum number of pairs
7: **for** $k \leftarrow 1$ to m **do**
8: Find the pair o, p $(x_o = \{1,0\}; x_p = \{0,1\}$ or $x_o = \{0,1\}; x_p = \{1,0\})$ so that o and p are unlocked and G_{op} is maximal
9: **if** $G_{op} > 0$ **then**
10: Swap o and p and test it $\Rightarrow X^{try} = (X^{new}$ with o and p swapped)
11: $Z = diff(X^{orig}, X^{try})$
12: **if** $z_o = 0$ OR $z_p = 0$ **then**
13: Update the new configuration $\Rightarrow X^{new} = X^{try}$
14: $B^{new} = B^{new} - G_{op}$
15: **if** $z_o = 0$ **then** Lock o **end if**
16: **if** $z_p = 0$ **then** Lock p **end if**
17: **else**
18: **if** $x_o^T R_o x_o' < x_p^T R_p x_p'$ **then** Lock o **else** Lock p **end if**
19: **end if**
20: **end if**
21: **if** $G_{op} \leq 0$ OR all pairs are locked **then**
22: **break**
23: **end if**
24: **end for**
25: **return** X^{new}

6.2. Communication-aware Allocation Algorithm

In order to take into account the communication costs when deciding the assignment of each OS component, an additional phase is applied, preceding the appliance of the former algorithms explained above. The idea of this phase is to group together those components that present lower communication costs when located at same execution domain. This idea is based on the observation of the architecture presented in Section 5.3, where two

communicating components may use the local bus (when both are placed on the same environment), or use the bus across the environmentes.

The algorithm executed in this phase is based on a clustering process and delivers new components set, which are then used as input to the assignment algorithm. In this case not only single components are assigned to CPU or FPGA, but also meta-components (cluster of components) instead. The previous algorithm for balancing B improvement (Algorithms 2 or 3) is slightly modified, in which a pair of components is allowed to change their location only if this change will improve the balancing and concurrently reduce the communication costs inside the execution environment. This is necessary, because the balancing B improvement algorithm do not handle components clusters.

Previous chapter introduced the communication cost model for two components u and v assigned to the proposed architecture: $\kappa(u,v) = \{C^{\alpha},\ C^{\beta},\ C^{\gamma}\}$. Where C^{α}, C^{β} and C^{γ} denote the communication cost between two components located both in software, both in hardware and each one in different environment, respectively. To measure the connection degree between these two communicating components, two further metrics were specified. The first one is the pl (*local preference*):

$$pl = \frac{2C_{\gamma}}{2C^{\gamma} + C^{\alpha} + C^{\beta}} \tag{6.3}$$

This is calculated using CM ($pl = f(C^{\alpha}, C^{\beta}, C^{\gamma})$). The metric pl compares the communication cost between two components when both are placed in the same execution domain in comparison with the case where each of them are placed in different execution domains.

The second metric is defined as pg (*global preference*) which is pl multiplied by a global factor (see Equation 6.4). This metric enables the comparison of all local preferences with each other when doing the clustering process.

$$pg = (\frac{C^{\gamma}}{\max_V \{C^{\gamma}\}})pl \tag{6.4}$$

Clustering components

The clustering procedure is iteratively executed in k passes. In each pass, the algorithm starts searching for the largest global preference value, pg', among all edges and tries to cluster two components, o and p, respecting two conditions:

- Components o and p have not been clustered yet (considering the current pass only).
- $(c_{o,1} + c_{p,1},\ c_{o,2} + c_{p,2}) \leq (\lambda_1,\ \lambda_2)$, where λ_1 and λ_2 are the maximum component costs allowed when performing the combination of o and p. This criterium is used

to avoid the efficiency deprecation of the assignment algorithm when the allocation costs of the formed components increase.

If two components form a cluster, then they are combined and the search is executed again. This method is repeated until no additional components are free for clustering, which characterizes the end of a pass. The resulting clustered graph is then used as input for the subsequent pass. This is repeated until all k passes have being executed. The number k of passes can be adjusted previously, before system starts, which may be dependable on the granularity and homogeneity of the component costs. Nevertheless, in Chapter 7, evaluations made for this algorithm indicates how to correctly choose the number k of passes.

Figure 6.1 shows an example of two components, $T1$ and $T2$, being clustered, which generates a new component T^*. For this case, $\kappa(T^*, T3)$ is generated as follows: $\kappa(T^*, T3) = \kappa(T1, T3) + \kappa(T2, T3)$. Thus, pg^* is calculated as: $pg^* = f(\kappa(T^*, T3))$. Note that communication costs, C^α and C^β, between $T1$ and $T2$ (from the example), are no longer considered for pg evaluation. Nevertheless, they are stored and used during the balance improvement algorithm.

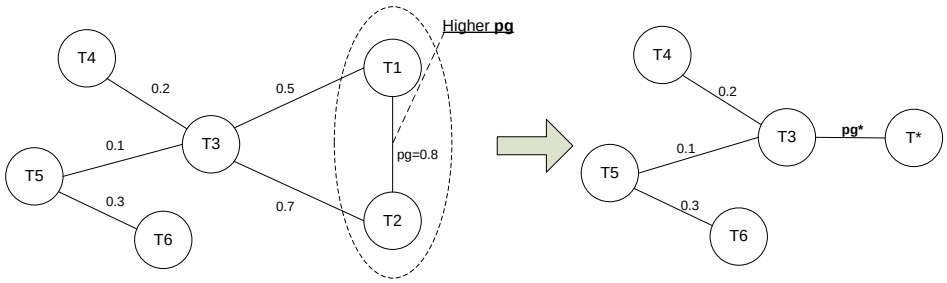


Figure 6.1.: Example of two components being clustered.

6.3. Handling Reconfiguration Activities

As it has been introduced in Section 4.1.4, whenever the current allocation of OS components does not fulfill the specified constrains, which is caused by application dynamics, a reconfiguration needs to take place. In the proposed system, this action is triggered when an unbalanced situation is detected: $|w_1 U - w_2 A| > \delta$. Furthermore, if new OS services are going to be included into the system (e.g., if a new application have been started), a new set $\mathcal{S}' = \{s'_i,\ i = 1, ..., n'\}$ of services is identified. In each case, a new assignment solution needs to be computed again, resulting in a new system configuration Γ'.

To bring the system from the original Γ configuration to a new Γ', a system reconfiguration needs to take place. This means that a certain number of services must be relocated

on the hybrid architecture, new services may be inserted, or even some services may be deleted if no longer required.

In Section 5.2 it has been shown that reconfiguration costs may be relatively high, which may require a large *blackout* time to reconfigure all components at once. This fact may be prohibited for some systems, especially those having temporal constraints. Therefore, the strategies developed in this thesis aims to achieve a near zero *blackout* time.

In order to find a reasonable solution for this problem, the reconfiguration activities carried out over the proposed platform were further specified. In the following analysis, at the first moment only current active services, that will undergo a reconfiguration, are going to be considered. Afterwards, new components inserted into the system (e.g., due to the arrival of new applications) are taken into account. So, let a subset $\mathcal{S}^*$ of the current active service set $\mathcal{S}$ be defined, to denote the OS components that will undergo a reconfiguration (hereafter also called migration):

$$\mathcal{S}^* = \{s_i^\star,\ i = 1, ..., m\} \tag{6.5}$$

Where m is the number of components to be reconfigured. Please note that $\mathcal{S}^* \subseteq \mathcal{S}$ and $m \leq n$, where n is the number of currently active services present in the system.

Now, the system reconfiguration problem can be formulated in two sub-problems:

P1 The reconfiguration activities of a single component s_i^* need to be performed in a deterministic way, concurrently with the system activities. By this means, a component reconfiguration can be accomplished without stopping system execution.

P2 Since m components are going to be reconfigured, the order in which these components are going to be reconfigured need to be carefully determined. If after each component reconfiguration constraints A_{max} and U_{max} are not violated, no system stop is required.

Hence, solving P1 and P2, the system can be reconfigured concurrently with system execution. In the subsequent sections, the solution for both P1 and P2 sub-problems are presented.

6.4. OS Component Reconfiguration

The reconfiguration of a component implies in the execution of respective *Programming* and *Migration* phases (introduced in Section 5.2.1). Since the arrival times of these activities are triggered by asynchronous events (system reconfiguration), they can be modeled as aperiodic jobs. Furthermore, if these jobs can be executed concurrently with the running services, without causing the violation of any deadline, a deterministic and zero *blackout* time reconfiguration can be accomplished.

For a more precisely model of these reconfiguration activities, let them be denoted by a set $\mathcal{J}$:

$$\mathcal{J} = \{J_i(J_i^a, J_i^b),\ i = 1, ..., m\}. \tag{6.6}$$

Where J^a and J^b represents the phases *Programming* and *Migration*, respectively. The execution times of these phases have been already introduced in Table 5.2. For a job J^a this time may be either Q^s or Q^h, depending on which direction a component is being migrated. Similarly, J^b represents one of three possible migration cases: M^s, M^h or M^w.

This scenario characterizes a situation where aperiodic jobs need to be carried out together with real-time periodic tasks. To cope with such scenarios, very efficient and well known strategies from real-time scheduling theory can be here applied. The basic idea of these approaches is the inclusion of a new periodic task (called server) into the system to reserve some bandwidth (CPU workload) to serve the aperiodic activities (jobs). Thereby, the response time of these jobs can be improved with the guarantee that deadlines of periodic tasks will not be violated. A more comprehensive and detailed explanation of this technique is given in [159, 160].

Among different types of servers, the analysis relies on Total Bandwidth Server (TBS) [159] due to the following reasons:

- The Earliest Deadline First (EDF) is assumed as the policy adopted by the system scheduler;
- Among other servers, it is one of the most efficient service mechanism in terms of performance/cost ratio [160].

According to the literature, TBS assigns a deadline d_k for an aperiodic job k arriving in the system at time a_k in the following manner:

$$d_k = \max(a_k, d_{k-1}) + \frac{C_k}{U_s} \tag{6.7}$$

Where d_{k-1} represents the deadline of the aperiodic job that has arrived before job k; U_s is the server bandwidth and C_k is the execution time requested by the aperiodic job. Deadline d_{k-1} is 0 if job k is the first one, or if all pending aperiodic jobs that arrived before job k have already been finished.

6.4.1. Applying Total Bandwidth Server

Preliminary Statements

The TBS server is used here as a technique to perform the migration of a component being executed in one execution environment to the other one. Depending on the system support, appropriated preemption/resumption facilities may or may not be available for migration purposes. Thus, in some cases the migration need to occur only when the service is not active or, it may be performed in the middle of the service computation. Nonetheless, in both cases some amount of context data may be required to be transferred across domains.

Even though there are some methods that allow preemption of hardware tasks as *read-back* and *scan-path* (e.g., [127]), they can not be used for a preemption and posterior resumption of a running component across hardware and software domains. Without proper support during design phase, which needs to assure safe points of preemption/resumption of a component and enough information about the reconfiguration costs, its accomplishment would be rather difficult. Therefore, a framework for designing relocatable tasks across hardware and software execution domains was developed and will be presented in Chapter 8. The proposed framework, together with a specific run-time environment, allows the preemption of a task being executed in software and a posterior resumption in hardware (and vice-versa).

Furthermore, it is worth to mention that due to the usage of a server, which is allocated to CPU, the reconfiguration activities are going to be performed sequentially. However, this fact does not represent a degradation of the hardware parallelism, since the availability of a single configuration port on today's FPGA serializes its programming operation.

Total Bandwidth Server Applied

The first phase J^a of a component reconfiguration is only a preparation of the execution environment to the second phase, where the migration is actually performed. Hence, no extra synchronization with the running tasks or services is required for its accomplishment. Therefore, the appliance of the deadline assignment rule from TBS to schedule J^a is straightforward. Differently, the mapping of J^b to the TBS rule is not that trivial and require thereby more attention.

In the following, the deadline assignment rule from TBS is applied on J^b under different situations. As a result, the minimal bandwidth U_s for TBS server is derived so that a migration phase can be safely performed. These following analyses will start with the assumption that no preemption is used for migration purposes. Then, in the sequence, the case where preemption is allowed will be derived from previous results.

6.4.2. Deriving Migration Conditions

The execution of a job J^b, related to service i, can only start in time intervals where no instance of s_i is active. Thus, considering that the service has a periodic behavior, the execution of J^b has to happen between two consecutive instances of this service. Moreover, this has to be performed respecting the deadlines of this service.

If $b_{i,k}$ and $f_{i,k}$ represent the starting and finishing execution times of a kth instance of service s_i, let $\hat{b}_i$ and $\hat{f}_i$ be the starting and finishing execution times of J^b. The conditions above can be formulated in **C1** and **C2**:

C1 $\hat{b}_i \ni [b_{i,k}; f_{i,k}]$: Job J_i^b cannot start if an instance of a service s_i has been started or if this instance has not been finished yet.

C2 $b_{i,k} \ni [\hat{b}_i; \hat{f}_i]$: Once job J_i^b has been started, it cannot be preempted by service s_i.

Condition **C1** arises due to the absence of preemption support for migration. It avoids that J_i^b starts to transfer the internal state of an instance k of service s_i if this instance k has been started and still not finished. Condition **C2**, in its turn, guarantees that a service s_i will be started only after the finishing the execution of J_i^b. Note that a job J_i^b may still be preempted by a service other than s_i.

In the following subsections every migration case is analyzed observing the conditions stated above.

6.4.3. Software to Hardware Migration

In order to facilitate the following analyses, let $\hat{a}_i$ and $\hat{d}_i$ denote the arrival and deadline times of a job J_i^b, respectively. To ensure that J_i^b will not start before instance $s_{i,k}$, J_i^b is released only when $s_{i,k}$ has been finished: $\hat{a}_i \geq f_{i,k}$.

In order to provide as much time as possible to execute J_i^b, the maximum time interval between two consecutive instances of a service s_i could be chosen. This happens in a specific instance k^* where the lateness of s_i is minimal: $\hat{a}_i^* = f_{i,k^*} | (d_{i,k^*} - f_{i,k^*}) = \min_k(d_{i,k} - f_{i,k})$. Figure 6.2 shows such a situation.

However, to generally find the specific instance k^* when a periodic task set is scheduled under EDF is not trivial, and its solution would incur in the usage of more complex algorithms, which is not desired. Furthermore, a non wanted delay in the complete reconfiguration time would be included due to the shifting of $\hat{a}_i$ from $f_{i,k}$ to f_{i,k^*}, which actually depends on the current hyperperiod of the service set.

Nevertheless, it is possible to control the arrival time $\hat{a}_i$ to be the same as the finishing time $f_{i,k}$, which can be, in a worst case, equal to the deadline $d_{i,k}$. Therefore, in the analysis this case is considered in order to provide a result that can be applied to every instance k. Hence, under this condition a minimal server bandwidth can be derived,

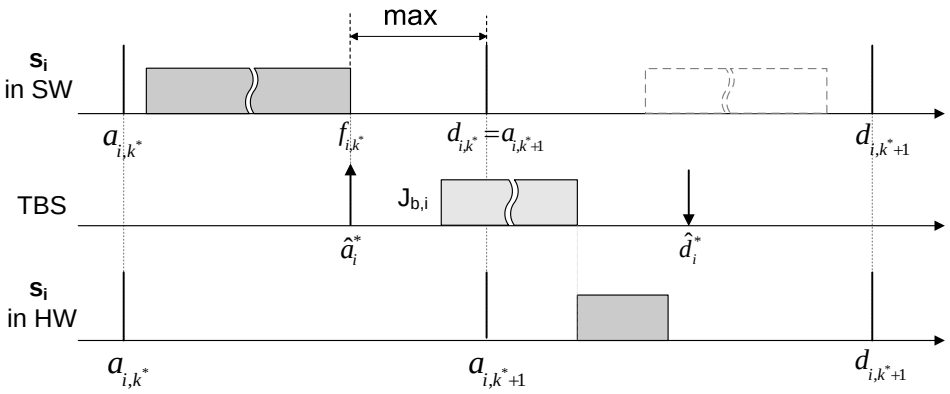


Figure 6.2.: Optimal arrival time $\hat{a}_i$ for J_i^b.

which will guarantee the non violation of conditions **C1** and **C2**.

Figure 6.3 shows this situation, where the arrival of J_i^b occurs at the arrival time of the next service instance $a_{i,k+1}$. Note that the relative deadline D_i is assumed to be equal to the period P_i.

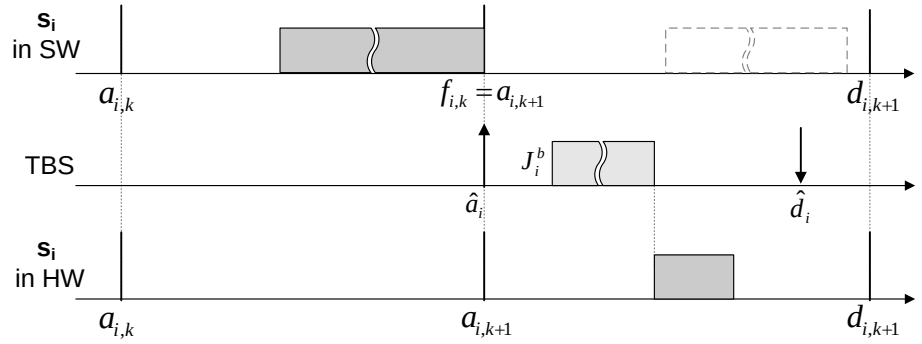


Figure 6.3.: Worst-case arrival time $\hat{a}_i$ for J_i^b.

Under EDF scheduling, the running task is always the one with the smallest absolute deadline among the ready ones. Thus, it can be ensured that J_i^b will not be preempted by $s_{i,k+1}$ if $\hat{d}_i \leq d_{i,k+1}$.

As the software component s_i will be moved to hardware, its CPU utilization factor U_i is released and can be added to the server bandwidth. Moreover, J_i^b phase needs to be finished, in the worst-case, at $d_{i,k+1} - E_i^h$ to allow the next instance of the service i (running afterwards in hardware) to finish before or at its deadline. Hence, the deadline $d_{b,i}$ assigned to J_i^b using TBS is:

$$\hat{d}_i = \hat{a}_i + \frac{M_i^w}{U_s + U_i} \leq d_{i,k+1} - E_i^h \tag{6.8}$$

Noting that $d_{i,k+1} = a_{i,k+1} + P_i$ and $a_{i,k+1} = \hat{a}_i$, the Equation 6.8 can be rewritten in order to derive the minimal server bandwidth:

$$U_s \geq \frac{M^{w,i}}{P_i - E_i^h} - U_i \tag{6.9}$$

Equation 6.9 shows that the bandwidth required to migrate service s_i is equal to $\frac{M_i^w}{P_i - E_i^h}$. Hence, as the server uses the workload U_i released by s_i, U_s needs to be at least equal to the difference necessary to achieve the requested bandwidth. Although the consideration of U_i as being part of the server bandwidth, already in the instance k, seams to be intuitively correct, a formal prove that it can be securely made is given in Appendix D.

If the condition expressed in Equation 6.9 is fulfilled under all services that will undergo a migration from software to hardware, it can be guaranteed that conditions **C1** and **C2** will be satisfied when EDF and TBS are used.

6.4.4. Hardware to Software Migration

For this migration case, the arrival time of job J_i^b can be defined more precisely since a service executing in hardware profits from the true parallelism of this environment. Additionally, the server needs to provide additionally the workload that will be reclaimed by s_i when running in software. Hence, the remaining bandwidth $(U_s - U_i)$ needs to be big enough to support the migration of J_i^b. The deadline assigned to J_i^b by TBS is then defined as:

$$\hat{d}_i = a_{i,k} + E_i^h + \frac{M_i^w}{U_s - U_i} \leq d_{i,k+1} \tag{6.10}$$

Noting that $d_{i,k+1} = a_{i,k+1} + P_i$ and that $a_{i,k+1} = a_{i,k} + P_i$, the equation above can be rewritten in order to derive the minimal server bandwidth required:

$$U_s \geq \frac{M_i^w}{2P_i - E_i^h} + U_i \tag{6.11}$$

6.4.5. Software Service Reconfiguration

Similar analysis as done above can be made for the case where a service is going to be replaced by another service, but only in software domain. In this case, using the deadline

assignment rule from TBS:

$$\hat{d}_i = \hat{a}_i + \frac{M_i^s}{U_s + U_i - U_i^{new}} \leq d_{i,k+1} \tag{6.12}$$

Where U_i^{new} is the new processor load used by service s_i after migration.

Rearranging Equation 6.12 and solving it for U_s, the minimal bandwidth required is:

$$U_s \geq \frac{M_i^s}{P_i} + U_i^{new} - U_i \tag{6.13}$$

Note that the period P_i is not changed, only the execution time: $E_i^{s,new}$, which refers to the new service that replaces the older one. It this case, the bandwidth requested by J_i^b is $\frac{M_i^s}{P_i}$ and the processor load released is $U_i - U_i^{new}$.

6.4.6. Hardware Service Reconfiguration

Similarly, it may be necessary that a hardware service need to be replaced by another one, also in hardware domain. In this case, it can be concluded that $\hat{a}_i = a_{i,k} + E_i^h$, since no preemption occurs in a hardware service execution. In addition, the deadline to finish J_i^b can be precisely defined as: $\hat{d}_i = d_{i,k+1} - E_i^{h,new}$. Note that here also the period P_i is the same and only the execution time of the service s_i is different. Thus, the assigned deadline to job J_i^b is:

$$a_{i,k} + E_i^h + \frac{M_i^h}{U_s} \leq d_{i,k+1} - E_{hw,i}^{new} \tag{6.14}$$

As the temporal distance between $a_{i,k}$ and $d_{i,k+a}$ is $2P_i$, the minimal server bandwidth for this case is:

$$U_s \geq \frac{M_i^h}{2P_i - E_i^{h,new} - E_i^h} \tag{6.15}$$

6.4.7. Migrating by Preempting

If a migration may be performed by preempting and resuming a service s_i from one execution environment to the other one, a speed up in the whole system reconfiguration may be achieved. This happens due to the fact that a migration is completed inside the same period of a service execution.

Software to Hardware

Differently from the first scenario, the condition **C1** is no longer necessary, since now the arrival of job J_i^b is allowed to be even in the middle of the execution of the service s_i. The **C2** however, is still necessary to avoid that a service s_i would start when the related job J_i^b is under execution.

For this new scenario analyses, additional notations are introduced. In Figure 6.4 these are further illustrated. Lets assume that job J_i^b arrives shifted from $a_{i,k}$ in a time distance of σ so that service s_i is stopped. In this moment, certain amount of computation of s_i would have been executed already. Let ηE_i^s be this amount, so that $0 \leq \eta \leq 1$. Thus, it follows that the remaining computation that will be executed in hardware afterwards is equal to $(1-\eta)E_i^h$.

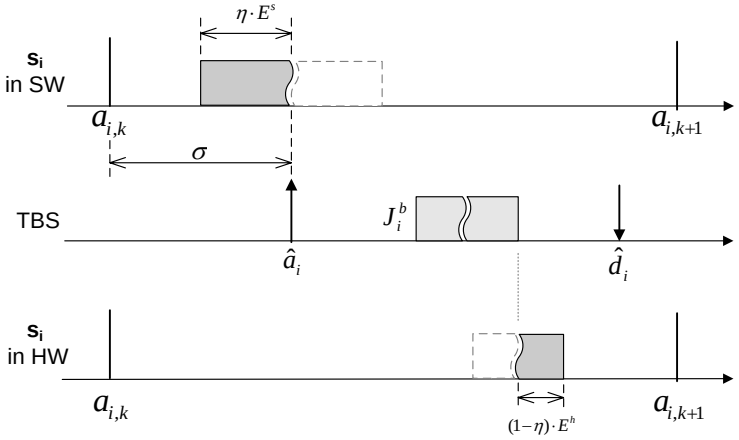


Figure 6.4.: Migrating by preempting: software to hardware.

The value η depends on σ and it is, therefore, defined for each instance k of a service s_i:

$$\eta_k = \begin{cases} 0 & : \text{ if } a_{i,k} + \sigma_k < b_{i,k} \\ 1 & : \text{ if } a_{i,k} + \sigma_k > f_{i,k} \\ \frac{a_{i,k} - s_{i,k} + \sigma_k}{E_i^s} & : \text{ otherwise} \end{cases} \tag{6.16}$$

For this case, deadline $d_{i,k}$ assigned to job J_i^b, arriving at $a_{i,k} + \sigma$ using TBS need to be smaller than $P_i - (1-\eta)E_i^h$ in order to guarantee the achievement of the deadline $d_{i,k}$ of service s_i. Hence, applying TBS rule for J_i^b, similarly as in the scenarios explained in above sections, the deadline $\hat{d}_i$ is defined by:

$$\hat{d}_i = \hat{a}_i + \sigma + \frac{M_i^w}{U_s + (1-\eta)U_i} \leq d_{i,k+1} - (1-\eta)E_i^h \tag{6.17}$$

Hence, as already noticed in the previous analyses, $d_{i,k+1} = a_{i,k+1} + P_i$ and $a_{i,k+1} = \hat{a}_i$, the Equation 6.17 can be rewritten to solve U_s as follows:

$$U_s(\eta, \sigma) \geq \frac{M_i^w}{P_i - (1-\eta)E_i^h} - \frac{(1-\eta)E_i^s}{P_i} \tag{6.18}$$

It has to be noticed that Equation 6.18 is defined only for $0 \leq \eta < 1$, since when $\eta = 1$, service s_i has already finished its execution and the migration will be performed in the next instance. Additionally, if job J_i^b arrives in the beginning of a instance so that $\sigma = 0$, which also implies in $\eta = 0$, the scenario will be identical to the case analyzed in Section 6.4.3. Indeed, making $\sigma = 0$ and $\eta = 0$ in Equation 6.18, this becames identical to Equation 6.9.

For this migration case, the minimal bandwidth required for TBS is no longer possible to be determined alone, but it depends on values σ and η instead. Thus, it is not trivial to specify a minimal U_s, even for a worst-case scenario. Nevertheless, it is noticeable that the bigger σ is the smaller the temporal distance available for a migration will be. As a consequence, depending on the arrival time $\hat{a}_i$, the minimal server bandwidth U_s required may not be feasible. However, as the work presented in this thesis envisioned soft real-time systems, a reasonable procedure would be to shift the arrival $\hat{a}_i$ to the finishing time $f_{i,k}$.

Hardware to Software

Based on the analyses made above, and on the scenarios where preemption were not allowed, the examination of a migration case of a service going from hardware to software is deductive. For clarification, Figure 6.5 shows such a case.

Hence, by similar analyses, it follows that the deadline di, k assigned for job J_i^b, arriving at $a_{i,k} + \sigma$ is equal to:

$$\hat{d}_i = a_{i,k} + \eta E_i^h + \frac{M_i^w}{U_s - (1-\eta)U_i} \leq d_{i,k+1} \tag{6.19}$$

Noting that $d_{i,k+1} = a_{i,k+1} + P_i$ and that $a_{i,k+1} = a_{i,k} + P_i$, the equation above can be rewritten to solve U_s:

$$U_s(\eta, \sigma) \geq \frac{M_i^w}{2P_i - \eta E_i^h} + (1-\eta)U_i \tag{6.20}$$

Also for this case, the arrival time $\hat{a}_i$ of job J_i^b can be shifted to $f_{i,k}$, which derives the same results where preemption is not used.

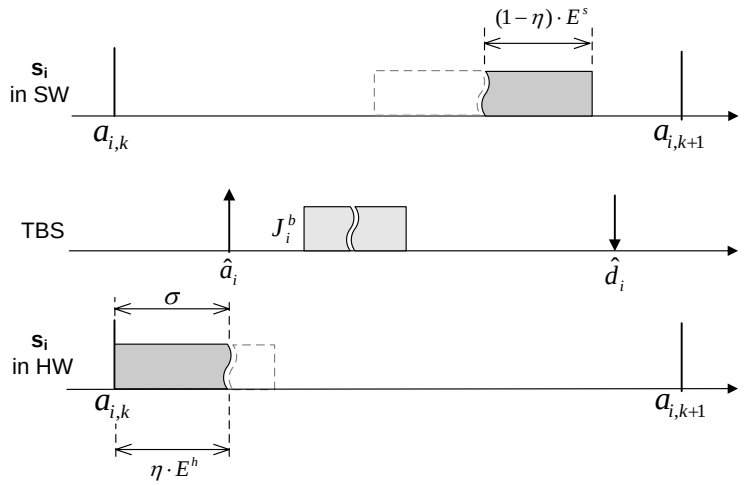


Figure 6.5.: Migrating by preempting: hardware to software.

6.5. Schedulability Analysis

All analyses made in the above sections were based on the proper assignment of arrival time and deadline of aperiodic reconfiguration activities, and the establishment of conditions for definition of the server bandwidth. These analyses were made in order to properly represent the conditions imposed in different explained scenarios.

As a typical scenario is assumed, where services are periodic and scheduled using the EDF policy, a standard schedulability analysis can be applied. Additionally, in the approach here presented no modification on deadlines or arrival times of any periodic service was performed. As a consequence, if after every migration step the sum of all processor utilizations (used by every software component) plus the server bandwidth does not exceed a maximum U_{max}, the feasibility of the schedule is guaranteed. Beside that, there is also a hardware constraint A_{max} (maximum FPGA area available). Thus, it follows that the FPGA needs to have available the area requested by each service being transferred to it. These conditions implies that before every service migration either enough FPGA area is available, or enough CPU workload. Furthermore, after each service migration, the new resources available need to be enough for further migrations. Only in this way, it can be guaranteed that the system will have be valid in terms of schedulability.

Defining $\mathcal{T}s$ and $\mathcal{T}h$ as the services running in software and hardware, respectively, after one migration a different configuration is achieved: $\mathcal{T}s^*$ and $\mathcal{T}h^*$. Thus, if for every component migration the following conditions are fulfilled, the schedulability are

guaranteed:

$$\sum_{i \in \mathcal{T}s^*} U_i \leq U_{max} \; ; \; \sum_{i \in \mathcal{T}s^*} A_i \leq A_{max} \tag{6.21}$$

In order to find a feasible schedule for every task migration, the service subset $\mathcal{S}^\star$ (defined in Section 6.3) needs to be previously sorted in a proper order. In other words, the sorting of services that will suffer a reconfiguration characterizes a schedule problem under resources constraints.

6.6. OS Components Scheduling

Within this section, a heuristic algorithm is proposed for scheduling the order in which components are going to be reconfigured. The components need to be sorted in such a way that after each component migration, the constraints specified in Equation 6.21 are not violated.

In order to tackle this problem in a proper manner, the components that will undergo a reconfiguration in their own execution environment are separated from the ones that will change it. Given the set $\mathcal{S}^\star$, three new subsets are defined:

$\mathcal{S}^s$ Services that will be reconfigured within software;

$\mathcal{S}^h$ Services that will be reconfigured within hardware;

$\mathcal{S}^w$ Services that will be relocated between hardware and software.

Please note that $\mathcal{S}^s \cup \mathcal{S}^h \cup \mathcal{S}^w = \mathcal{S}^\star$.

From Equation 6.21 it can be concluded that the lower $\sum U_i$ and $\sum A_i$ are, the higher are the chances in finding a feasible schedule. Therefore, the services from subset $\mathcal{S}^s$ will be scheduled first only if this will leads to a reduction in the final CPU workload, otherwise, they will be scheduled at the end. The same rule is applied on the components from $\mathcal{S}^h$, in respect to FPGA area. Hence, the problem is reduced by finding a schedule for the service set $\mathcal{S}^w$.

Due to the same reasons, new OS services that may arrive, are inserted (configured) into the system only after accomplishing the reconfiguration of the complete set $\mathcal{S}^\star$.

If $|\mathcal{S}^w| = x$, the feasible solutions S_f is a subset of $x!$ possible scheduling solutions (permutations of components in $\mathcal{S}^w$). To solve this problem, Bratley's algorithm [161] could be applied, in which the search space for a valid schedule is reduced. Nevertheless, the worst-case complexity of this algorithm is still $O(x \cdot x!)$ (NP-Hard problem), as $x!$ paths of length x have to be analyzed. Because of that, a heuristic algorithm was developed to schedule the order of components that will undergo a reconfiguration.

The scheduling problem was broken down in two different problems. First, only components that must migrate from CPU to FPGA are properly sorted, generating in this way a so called *Partial Schedule*. The same procedure is applied on components that must migrate from FPGA to CPU. Second, the complete schedule is generated by properly merging both *Partial Schedules*.

6.6.1. Partial Schedule

The basic idea of this heuristic is the use of component costs ($c_{i,j}$) as a criteria for searching a solution in the tree of all possible schedules. Looking at the components that need to leave the CPU, the strategy is the following: try to relocate the component with the highest software cost and with the smallest hardware cost first. In this way, after this component reconfiguration, the CPU workload will be decreased as much as possible, and in the same way, the FPGA area used will be increased as little as possible. Similarly, the same strategy is applied to the components that need to leave the FPGA. Consequently, two partial schedules are generated using the strategy explained above.

Let $Sa = \{sa_1, ..., sa_p\}$ and $Sb = \{sb_1, ..., sb_q\}$ be the components that need to leave the CPU and FPGA, respectively, so that $Sa \cup Sb = \mathcal{S}^w$ and $Sa \cap Sb = \emptyset$. Let $Ia = \{i_1, ..., i_p\}$ be the index array that represents Sa sorted by decreasing software costs, so that $\{c_{i_1,1} \geq c_{i_2,1} \geq ... \geq c_{i_p,1}\}$. Similarly, $Ja = \{j_1, ..., j_p\}$ is defined as the index array that represents Sa sorted by increasing hardware costs: $\{c_{j_1,2} \leq c_{j_2,2} \leq ... \leq c_{j_p,2}\}$.

The partial schedule, PS, shown in Algorithm 4 starts comparing the first two indexes of Ia and Ja ($k = 1$). If no match (same index in both arrays) is found, it expands the search ($k = 2$) on the first two components of Ia and Ja (total of four components). Hence, the search is done gradually (line 5) until a match is found. If this is the case, the index is removed from both arrays, the schedule is updated and the search restarts on the remaining arrays. Note that a match is always found, since the same elements from Ia are also presented in Jb. Hence, the algorithm will always terminate.

It can be seen that for every k value the algorithm calculates, in worst-case, $2k - 1$ comparisons. Thus, for a worst-case scenario when searching for a match, where the search is done over the whole array ($k = p = |Ia| = |Ja| = |Sa|$), the total number of comparisons will be $1+3+5+...+(2p-1) = \sum_{i=1}^{p}(2i-1) = p^2$ (which is the maximum number of combinations that can be done between two arrays of size p). Therefore, the complete partial schedule algorithm has a complexity of $O((n-1)n^2)$, since for every index match found, the search will be applied again on a reduced index array.

If Ib and Jb are defined as the index arrays that represent Sb sorted by decreasing hardware costs and sorted by increasing software costs, respectively, the same partial algorithm is applied on Sb using these two index arrays. Both partial schedules generated, $PS(Ia, Ja)$ and $PS(Ib, Jb)$, are then used to produce the complete schedule.

Algorithm 4 Partial schedule.

```
1: I_cur ← Ia;   J_cur ← Ja
2: PSa ← ∅
3: while |I_cur| ≠ 0 do
4:     match ← 0
5:     for k ← 1, |Ia| do
6:         for i ← 1, (k − 1) do
7:             if I_cur[i] = J_cur[k] then
8:                 match ← 1
9:                 break
10:            end if
11:        end for
12:        if match = 1 then
13:            break
14:        end if
15:        for j ← 1, (k) do
16:            if I_cur[k] = J_cur[j] then
17:                match ← 1
18:                break
19:            end if
20:        end for
21:        if match = 1 then
22:            break
23:        end if
24:    end for
25:    PSa ← PSa + i                          ▷ Update the schedule
26:    I_cur ← I_cur − i;   J_cur ← J_cur − j
27: end while
```

6.6.2. Complete Schedule

The whole schedule WS is generated selecting the components from the partial schedules, similar to an interleaving manner. The algorithm, presented in Algorithm 5, starts selecting one component from every partial schedule. However, it select first from the larger (in size) partial schedule. The selection of the remaining components is done respecting the following rule: The relation between the number of components that have been selected from $PS(Ia, Ja)$ over the ones that have been selected from $PS(Ib, Jb)$ should be similar to the relation between the lengths of $|PS(Ia, Ja)|$ and $|PS(Ib, Jb)|$. The complexity of this algorithm is $O(n)$, since it has only one *while* loop with simple operations.

After obtaining the complete scheduling, a check is needed to verify if this solution is a feasible one. This is done by evaluating the $\sum U_i$ and $\sum A_i$ after every component

Algorithm 5 Whole schedule.

1: $WS \leftarrow \emptyset$
2: **if** $|PSa| \geq |PSb|$ **then**
3: $\quad PSa_{tmp} \leftarrow PSa$
4: $\quad PSb_{tmp} \leftarrow PSb$
5: **else**
6: $\quad PSa_{tmp} \leftarrow PSb$
7: $\quad PSb_{tmp} \leftarrow PSa$
8: **end if**
9: $R \leftarrow PSa_{tmp}/PSb_{tmp}$
10: $WS \leftarrow WS + PSa_{tmp}[1]$
11: $PSa_{tmp} \leftarrow PSa_{tmp} - PSa_{tmp}[1]$
12: $WS \leftarrow WS + PSb_{tmp}[1]$
13: $PSb_{tmp} \leftarrow PSb_{tmp} - PSb_{tmp}[1]$
14: $r \leftarrow 1$
15: $s \leftarrow 1$
16: **while** ($|PSa_{tmp}| \neq 0$ **and** $|PSb_{tmp}| \neq 0$) **do**
17: $\quad$ **if** $r/s \leq R$ **then**
18: $\quad\quad WS \leftarrow WS + PSa_{tmp}[1]$
19: $\quad\quad PSa_{tmp} \leftarrow PSa_{tmp} - PSa_{tmp}[1]$
20: $\quad\quad r \leftarrow r + 1$
21: $\quad$ **else**
22: $\quad\quad WS \leftarrow WS + PSb_{tmp}[1]$
23: $\quad\quad PSb_{tmp} \leftarrow PSb_{tmp} - PSb_{tmp}[1]$
24: $\quad\quad s \leftarrow s + 1$
25: $\quad$ **end if**
26: **end while**
27: $WS \leftarrow WS + PSa_{tmp} + PSb_{tmp}$ ▷ If one array gets empty

reconfiguration, obeying thereby the scheduled obtained previously. If one system constraint is not fulfilled during one of the steps of evaluation, the solution is said to be not feasible and the heuristic does not find a feasible solution.

6.7. Chapter Conclusions

This chapter presented the main methodologies and strategies developed and available inside the RRM system. These comprise heuristic algorithms for allocation of OS services on the hybrid architecture (observing thereby different goals), a well defined strategy to reconfigure each single OS service by means of relocation between CPU and FPGA, and heuristics for sorting the components to be reconfigure.

The strategies developed to carry out the reconfiguration activities allow the achievement of a zero *blackout* time. This is accomplished by several meanings. First, the FPGA reconfiguration time is hidden completely, which is usually the most time consuming operation in a reconfigurable computing system based on this technology. Second, by using techniques from real-time scheduling theory, the effective relocation of the service across CPU and FPGA is performed in a deterministic way. Last, but not least, a heuristic was developed to assure the correct order in which the components should undergo a reconfiguration.

Additionally, it has to be noticed that the communication-aware allocation algorithm here developed is more appropriated for a scenario were OS services are able to call further services, as modeled in Figure 5.2. For its application in the system here proposed, different scheduling strategies would need to be investigated, since in this case, for instance, dependencies among services would need to be considered. Hence, this algorithm is considered as an additional feature of RRM, and further work in this direction will be discussed in Chapter 10.

All heuristic algorithms were developed observing the requirement to present a low computation complexity and good efficiency, since they all need be executed at runtime. Therefore, in the next chapter, simulations will be performed with each of these heuristics for evaluation purposes.

CHAPTER 7

Methods Evaluation

This chapter presents evaluations performed with the proposed heuristic algorithms by means of simulations. Thereby, the performance and efficiency of proposed methods and strategies in the previous chapter are evaluated.

The MATLAB tool was chosen for this purposes since it has the necessary features and facilities for the required analyses as well as an environment, which promotes system modeling. For instance, due to the availability of solvers for Linear Integer Programming and Binary Integer Programming, results from the proposed heuristics algorithms could be compared with optimal solutions.

7.1. OS Components Allocation

In this section, the proposed heuristic for allocation of OS components is evaluated. First, the assignment and balancing heuristics are tested. Then, in the sequence, the modified balancing algorithm is evaluated concerning its efficiency in reducing the number of components being reconfigured.

7.1.1. OS Components Assignment

The testbed created for evaluating the OS component assignment algorithm has the following configuration. A set $\mathcal{S}$, having size of $n = 20$ was randomly generated. So,

each component has varying random costs in the range: $1\% \leq c_{i,1} \leq 15\%$ and $5\% \leq c_{i,2} \leq 25\%$. The maximum FPGA resource was defined to be 100% ($A_{max} = 100$), as well as for CPU ($U_{max} = 100$). The components assignment were calculated for every system using the Binary Integer Programming solver from MATLAB (which delivers an optimal solution), and the heuristic algorithms.The average value of total cost ($U + A$) and the absolute difference cost ($|w_1 U - w_2 A|$) were compared for different values of δ (the resource usage balancing restriction): $0.5, 1, 2, 3, 4, 5, 10, 20, 30, 40, 50$ and 60. These average values were achieved by running the experiment 100 times.

The solutions provided by the assignment heuristic were very similar to the optimal one, if δ constraint has values around 10% ($\delta \approx 10\%$). Concerning fulfillment of δ constraint (see Figure 7.1, Heuristic-1), it can be said that, the smaller the δ the poor the results given by this heuristic. This was expected to be so, since the assignment heuristic does not consider the balancing restriction.

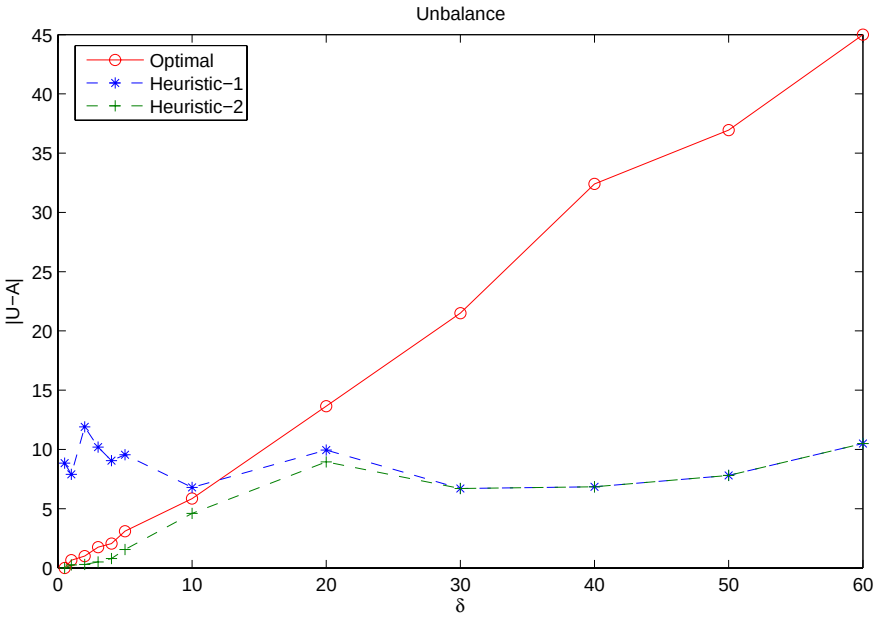


Figure 7.1.: Unbalance average for different δ constraints.

7.2. Balancing Heuristic

The application of the balancing heuristic over the solution provided by the previous one deliver a better balancing B. This can be seen in the Figure 7.1, Heuristic-2. However, an increase in the total assignment cost was verified for cases where the balancing heuristic has achieved an improvement in the balancing B ($\delta \lessapprox 10\%$). Nevertheless, the total

cost assignment achieved by this heuristic algorithm were quite satisfactory: maximum of 15% bigger if compared with the optimal case (see Figure 7.2, Heuristic-2).

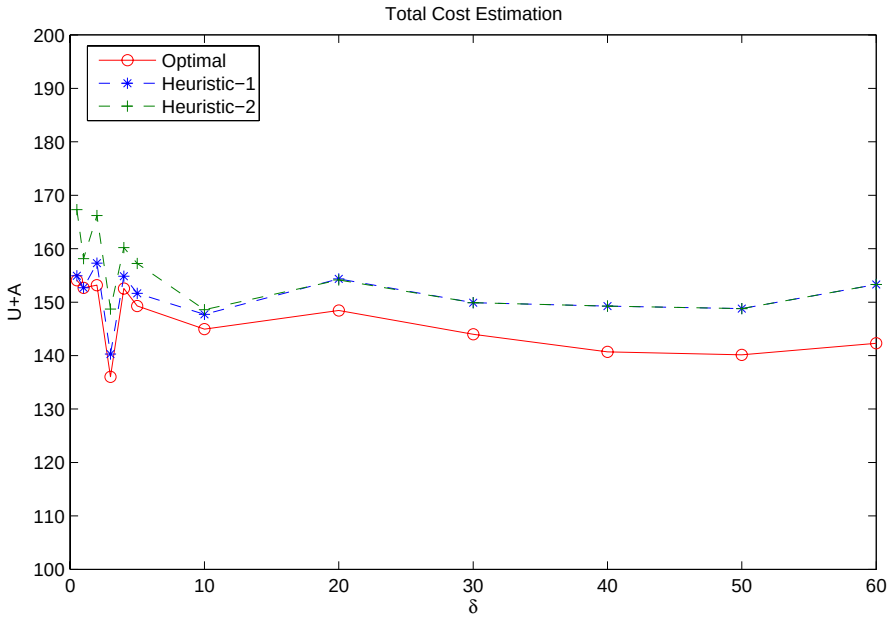


Figure 7.2.: Total cost assignment average for different δ constraints.

7.2.1. Reconfiguration Cost Reduction

Similarly to the evaluation made above, randomly systems were generated here. For each case Algorithm 1 was applied and the results were used as input for two balancing heuristics: Algorithm 2 and Algorithm 3, generating therefore two different assignment solutions: Γ_1 and Γ_2. The assignment solutions Γ'_1 and Γ'_2 given by the same algorithms applied to the subsequent system generated, simulates the next system configuration, and therefore the need for system reconfiguration. The amount of components having different allocations, which requires reconfiguration, between two subsequent assignment generated was counted.

By comparing the results between those two kinds of balancing heuristics, it can be evaluated the gain obtained by the Algorithm 3 towards reduction of reconfiguration costs. As already mentioned in previous chapter, the reconfiguration costs reduction is achieved indirectly by trying to reduce the number of components being reconfigured. The results achieved by the heuristics in these scenarios are presented in Figure 7.3. The number of components required to be reconfigured when using the original balancing algorithm is represented by the curve *Heuristic-2a*, and by *Heuristic-2b* the results provided by the modified balancing heuristic.

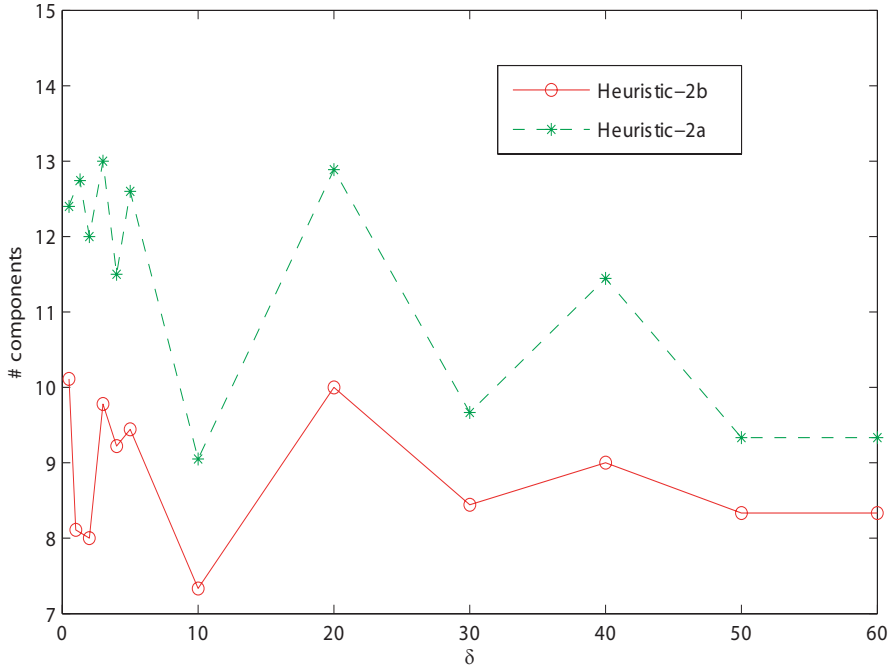


Figure 7.3.: Number of components being reconfigured for different δ constraints. Heuristic-2a: original balancing algorithm. Heuristic-2b: modified balancing algorithm.

In addition, the balancing achieved using the modified algorithm was also evaluated. Figure 7.4 shows the results achieved by the original algorithm (*Heuristic-2a*) and the modified one (*Heuristic-2b*). The improvement made in both case were satisfactory. Note that the results achieved by Heuristic-2b, concerning balance improvement, are quite under the constraint δ. This is due to the fact that in Heuristic-2b it still searches for more pairs to be swapped (even with δ constraint being fulfilled) in order to reduce the number of reconfigurations.

In order to analyze the influence of the modified algorithm in the overall resource utilization, the results between these two algorithms were drawn together and presented in Figure 7.5. It can be seen that the modified balancing algorithm presents a pay-off in the overall resource utilization, especially for values of δ bigger than 5%.

The modified balancing algorithm did provide reduction in the number of components being reconfigured with an acceptable incremental in the total resource utilization cost. Additionally, depending on the individual component costs, a significant reduction of the reconfiguration costs may be achieved. This fact should guide the designer by the decision of its usage or not in the target system.

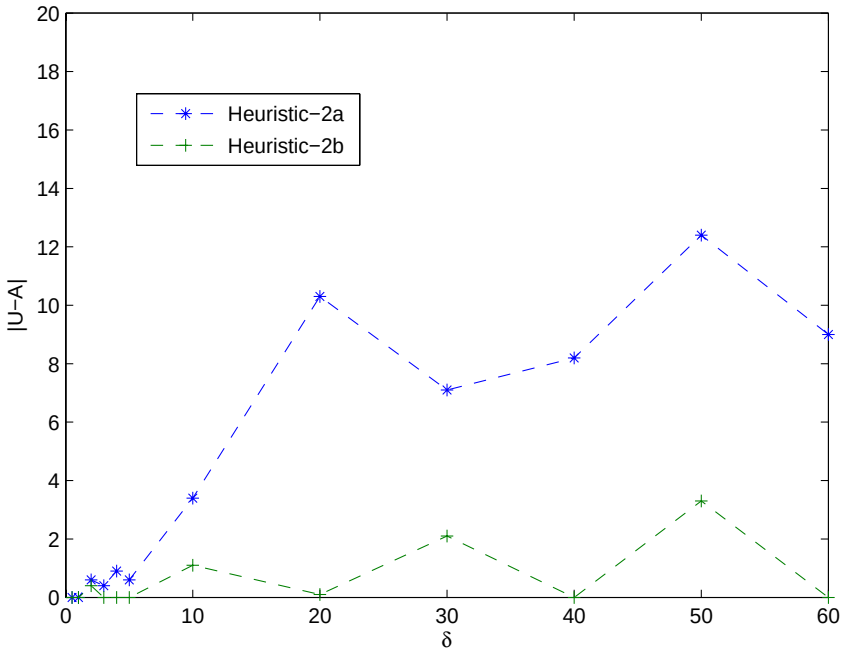


Figure 7.4.: Unbalance average for different δ constraints.

7.3. Components Reconfiguration Scheduling

Since for each scheduling problem, more than one feasible solution may exist, the efficiency of the proposed heuristic was measured by means of its capacity in providing one feasible solution among the existing ones. Therefore, all possible feasible solutions were generated calculating all possible components permutations. Due to computation complexity in this case, the system size was limited in 8 components, which may produce a maximum of 8! possible solutions.

For this experiment, random system pairs (representing the current and new system configuration) were randomly generated and both, Algorithm 4 and Algorithm 5 for partial and complete scheduling, respectively, were applied on them. Every system pair generated was tested under different system constraints: $U = A = [100\%,\ 90\%,\ 80\%,\ 70\%,\ 60\%\ and\ 50\%]$. For each pair, the component costs were generated observing the following range: $1\% \leq c_{i,1} \leq 15\%$ and $5\% \leq c_{i,2} \leq 25\%$ for the first system, and $5\% \leq c_{i,1} \leq 25\%$ and $1\% \leq c_{i,2} \leq 15\%$ for the second one. The intention thereby, is to cause a considerable number of components to be reconfigured. For each system pair, the experiment was repeated 100 times.

The dashed line at Figure 7.6 shows the percentage of cases where at least on feasible scheduling exists, by probing all scheduling combinations. The solid line shows the per-

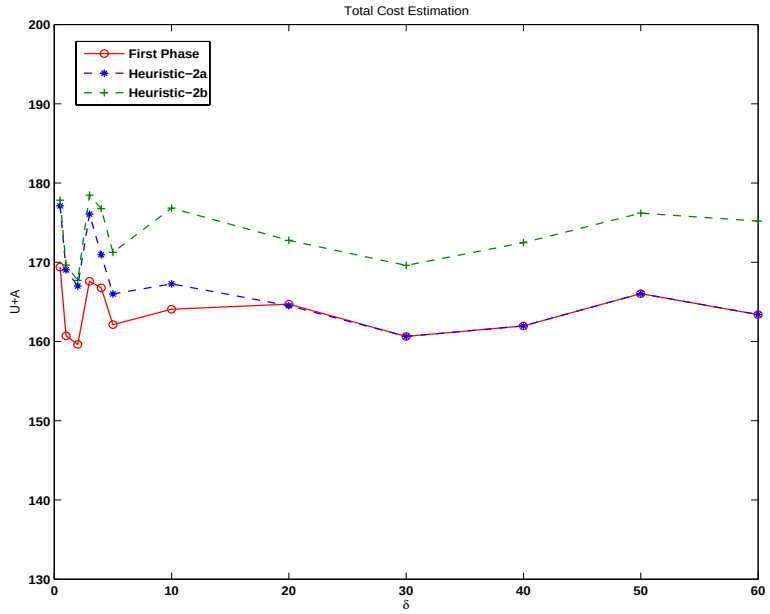


Figure 7.5.: Payoff in $U + A$ due to the balancing algorithm modified for reconfiguration costs reduction.

centage of cases when the heuristic did find at least one feasible scheduling, considering only the cases where at least one feasible solution exists. Based on the results shown in Figure 7.6, it can be concluded that the smaller U_{max} and A_{max} constraints are, the poorer the efficiency of the heuristic algorithm is. Nevertheless, the efficiency of the heuristic algorithm decreases slower than the chance of existing at least one solution, by decreasing U_{max} and A_{max} constraints values. For instance, when $U = A = 70\%$, in 40% of the cases, at least one feasible solution exists, and in around 80% of this cases, the heuristic did provide a feasible solution.

7.4. Communication costs reduction

To evaluate the proposed communication-aware allocation algorithm, the same environment used in the evaluation of the allocation algorithm was created. Additionally, it was communication graphs of size $n = 20$ were randomly generated respecting the following relation: $C^{\alpha} < C^{\beta} < C^{\gamma}$, which reflects the characteristics of the proposed architecture (communication costs across execution domains are the most expensive ones).

The results achieved by the allocation algorithm were compared considering in the input, first the pure generated systems, and second using the clustered system (generated as

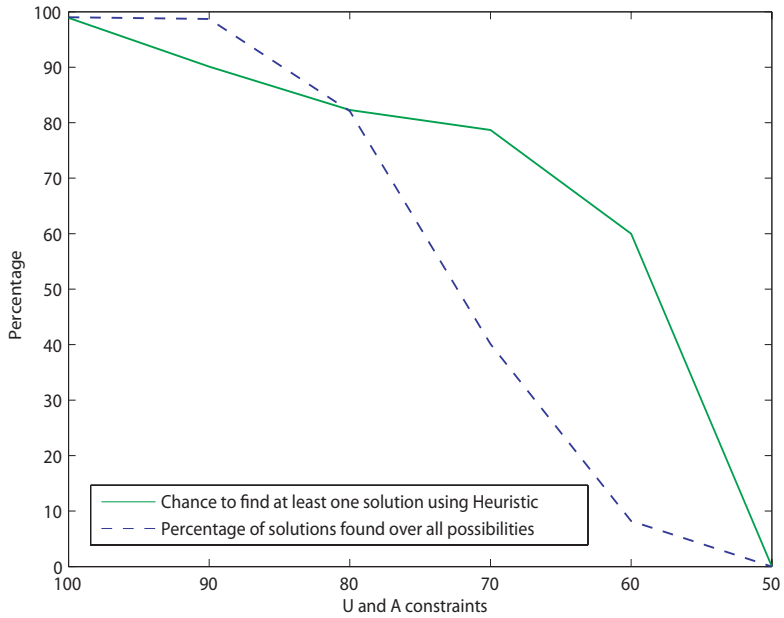


Figure 7.6.: Component reconfiguration scheduling: heuristic algorithm evaluation.

described in Section 6.2).

Figure 7.7 presents the communication cost increment obtained in each execution domain (HW: Hardware, SW: Software) and also between them (SW-HW: across CPU-FPGA). The lines in the graphic indicate, in percentage, the amount of communication cost increased when comparing the assignment results when clustering process was used against the results when it was not used. Hence, a negative percentage value indicates that a reduction of the communication costs was achieved when clustering process was used. The evaluations were performed for different number of passes in the cluster process (denoted by *#folding* in Figure 7.7).

The results obtained show that the amount of communication costs among the hardware and among hardware-software components were reduced when the proposed clustering heuristic were applied. On the other hand, the communication cost among software components suffers from a little increase. This tradeoff is caused due to the fact that the clustering heuristic promotes a balance among communication costs inside and across execution domains.

The results show further that the maximum gain obtained in reducing the communication costs stays around 10% in average. This limit comes from the fact that the clustering algorithm avoids to merge two components that overcomes the constraints λ_1, and λ_2 (introduced in Section 6.2). These constraints avoid the efficiency deprecation of the

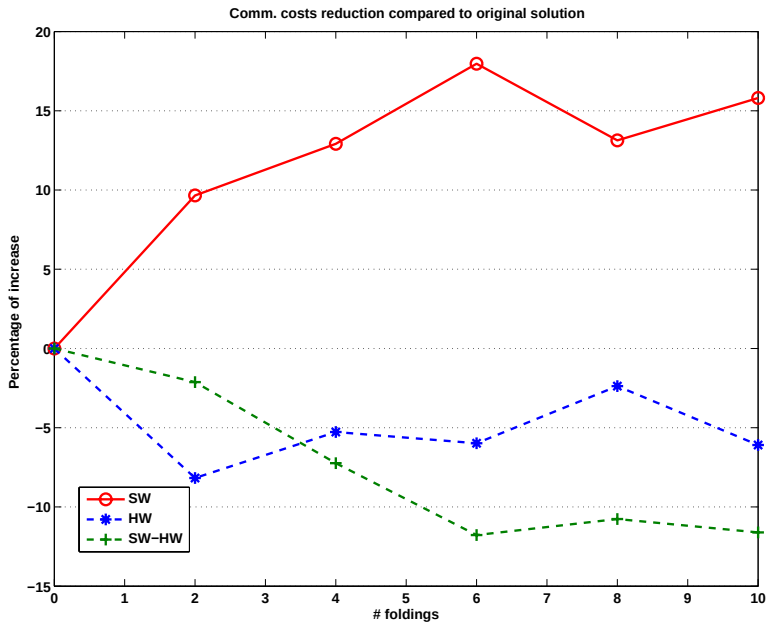


Figure 7.7.: Evaluation results comparison.

assignment algorithm due a non homogeneity distribution of the component cost values.

Additionally, it can be observed in Figure 7.8 that there exists an increment in the overall resource utilization (payoff) when clustering process is applied. This increment, however, was always under the value of 10%, which is acceptable.

In order to examine the influence of clustering heuristic for different relations of C^{α}, C^{β} and C^{γ}, three additional evaluations were conducted, using the same testbed explained above. So, three different relations among the component costs were respected for $\overline{\kappa}$ (where $\overline{\kappa}$ is the average of the generated communication costs):

Case 1 $\overline{\kappa} = \{C^{\alpha},\ 2C^{\alpha},\ 3C^{\alpha}\}$

Case 2 $\overline{\kappa} = \{C^{\alpha},\ 2C^{\alpha},\ 8C^{\alpha}\}$

Case 3 $\overline{\kappa} = \{C^{\alpha},\ 2C^{\alpha},\ 15C^{\alpha}\}$

The results obtained in these three evaluations provided the same patterns shown in Figure 7.7 and Figure 7.8. For further details, the graphics obtained in these experiments can be seen in Appendix A.

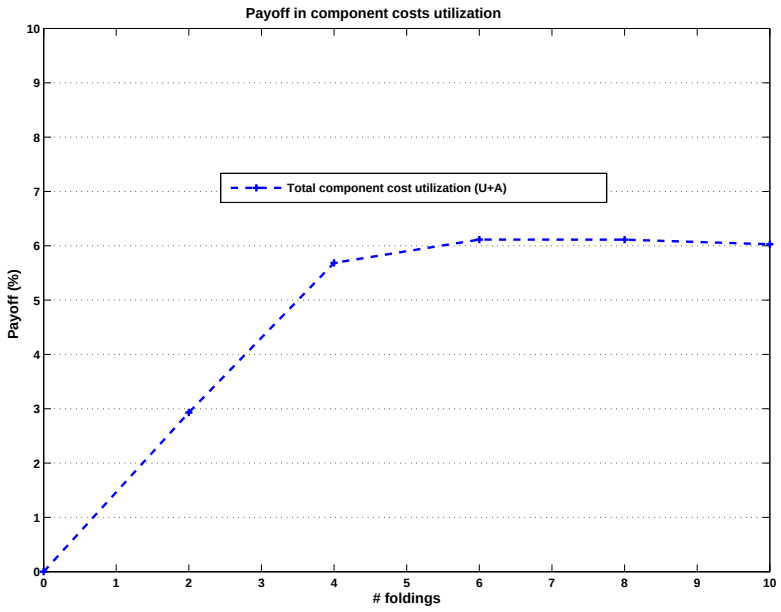


Figure 7.8.: Payoff in overall resource usage due to clustering process.

7.5. Chapter Conclusions

Within this chapter heuristics algorithms proposed in the previous chapter were evaluated by means of simulation and comparison with the optimal solutions provided by MATLAB tool.

OS services allocation algorithms, comprising heuristics for assignment and different versions of balancing improvement, did show satisfactory performances under different conditions.

The heuristic for sorting the components under reconfiguration was tested. The results show that its efficiency is quite satisfactory, taking into consideration the hardness of the problem, which increases rapidly ($O(n \cdot n!)$) in the number n of components undergoing a reconfiguration.

The communication-aware allocation algorithm was also evaluated, even though it cannot be directly applied in the proposed RRM system. Therefore, its efficiency was limited for the gains observed in the reduction of the communication costs among the communicating components being located in the hybrid architecture. It is worth to mention that this specific strategy has not the intention to minimize the communication costs. If so, a multi-objective optimization problem would be characterized and, therefore, an appropriated heuristic would be required.

CHAPTER 8

Design Support

An important issue in choosing a RTOS, especially in the development of embedded systems, is the available support for designing, monitoring, debugging, etc. This depends further on the hardware architecture used as execution platform. For instance, depending on which CPU is going to be used, a specific cross-compiler may be required. For the execution platform used in this work, which is based on the Virtex-II Pro FPGA device, the main implementation support is already provided. This comprises the EDK and ISE tools from Xilinx Company (see Section 2.3.1). Additionally, VHDL code for hardware parts and C/C++ for the software parts are used.

With those tools it is possible to generate a complete embedded system, comprising CPU plus the reconfigurable hardware, and the correspondent executables. Nevertheless, the system proposed in this work is based on dynamic relocation of components both intra-execution domain as well as inter-execution domain. This fact imposes some challenges that cannot be easily solved only by making use of compilation tools. In this work the solution for this problem is based in providing support in higher level, when designing relocatable OS components (which can be used also for relocatable application tasks design). Therefore, within this chapter two main approaches covering these aspects are proposed.

First, an automatic interface generation strategy, for connecting software components to dynamically reconfigurable hardware components, is introduced. Second, a framework that allows the design of dynamically relocatable components, across CPU and FPGA, is presented.

Since the design support presented here is not limited for generation of OS services, but also for application tasks, hereafter the term task is used to generically denote a relocatable component.

8.1. Hardware-Software Interface Synthesis

Among different communication schemes for coupling hardware and software, the traditional method, Memory Mapped I/O, was selected. It is suggested and supported by Xilinx tools along with the CoreConnect bus architecture, from IBM. This means that each IP attached to this bus can be accessible from software side, by read/write actions in specific memory regions. In the case of a design based on a Virtex-II Pro FPGA, such an interface is shown in Figure 8.1, where the OPB bus is assumed.

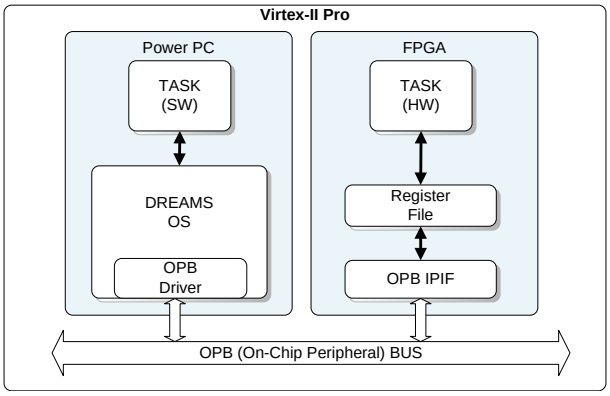


Figure 8.1.: Virtex-II Pro hardware/software interface.

The EDK tool support for HW-SW interface generation, such as in Figure 8.1, is only basic. In software side, it restricts itself in providing low level read/write functionalities to access a certain number of software registers (a generic OPB Driver), previously informed in the design entry. For hardware side, an address decoder is generated, called OPB IPIF (IP Interface) [162], which is responsible to solve the write/read commands between OPB bus and a set of hardware registers (Register File). Hence, the generation of a basic infrastructure to interconnect OPB Driver with Register File is automated and supported by EDK tool.

Since no further support is provided for HW-SW interface generation, it is responsibility of the designer to integrate the Register File with the hardware task, and the OPB Driver with the remaining software part (e.g., DREAMS OS). This limitation is due to the fact that EDK interface generation tool do have neither appropriated knowledge of hardware task internals, nor sufficient knowledge of software side (target OS)internals. Thus, in this thesis, an extension of this basic interface support, both in HW and SW sides, was created.

8.1.1. OS Driver Extension

A feature included is the automatic generation of a mapping between registers present in hardware, the Register File (from now on called physical registers), and registers present in software (called virtual registers). Figure 8.2 shows a typical interface design situation, where a single physical register may concatenate bit fields, each one having different meaning. Usually, these bits need to be accessed individually in the software side. In the example, a physical register *Reg1* contain the following distinct bit fields: ACK, RDY and DATA1. Likewise, *Reg2* contain two other data: DATA2 and DATA3. On the software side, these five bit fields are split into five virtual registers.

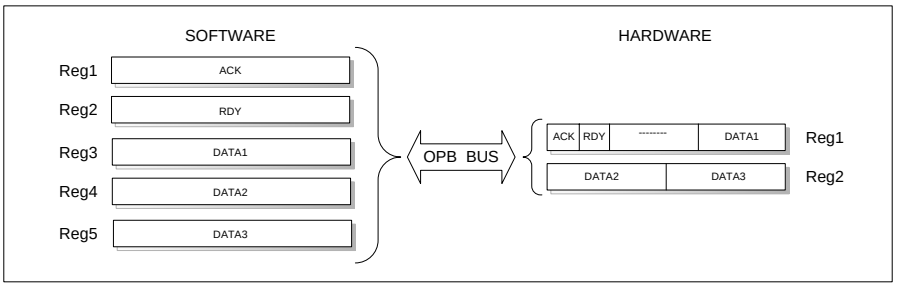


Figure 8.2.: Mapping between physical and virtual registers.

The interface generation creates, in addition, the necessary functionalities in software side in order to directly access those virtual registers. Thereby, the direction (read/write) of each register is observed and solved automatically by these functions. Moreover, these functionalities are all automatically generated for DREAMS OS. Thus, the software component, running on this OS, can make direct access to these virtual registers.

With this feature incorporated into the interface generation tool, the designer can generate an interface for any IP, which automatically maps n physical registers to a set of m virtual registers. Furthermore, these registers can be directly accessed by the SW threads running in DREAMS.

8.1.2. Software Interface for Reconfigurable IPs

The proposal of this thesis is based on dynamic reconfiguration of components running in hardware. Since memory mapped I/O is the technique used to connect IPs to software domain, one may think on using a flexible memory mapping management, which needs to support a dynamic number of IPs. Moreover, each IP may require a different address range in the memory system address.

Alternatively, it was decided to specify a fixed address range for each IP connected to the system. Reason for this choice is the limitation of the modular design flow [43] technique, which advises to statically partition the FPGA surface in fixed number of slots. In each

slot an IP can be reconfigured. Following these guidelines, for each physical slot a physical base address is assigned in the CPU memory address. This assignment is static and is made in the hardware architecture design phase. In a first glance, one may think that this fact imposes a limit in the number of physical register available for an IP (slot). However, in the following it is shown how this limitation is overcome.

In order to allow an arbitrary amount of data to be exchanged using a fixed number of physical registers, a communication protocol was defined. This protocol defines five virtual registers, REG_ID, DATA, READY, CMD, and START, which enables to read-/write in any of the registers of Register File. REG_ID is used to identify the register in the Register File. DATA contains the data that will be copied to, or from, the target register, depending on the content of the register CMD. This register specifies the action that will be performed with the register identified by REG_ID. Up to now, only WRITE and READ commands have been implemented. The remaining registers are used to start/stop and to provide the handshake during data exchanging. These five virtual registers are mapped into three physical registers, which do not belong to the IP Register File.

The adoption of the above strategy has some further advantages. First, the data exchange scheme based upon a fixed number of physical registers overcomes the problem of a fixed and limited IP memory range. This can become a problem depending on the number of IPs attached to the system, or if the address range of the I/O bus system is not big enough to embrace all IPs. Second, the overhead caused by the protocol could be considered as a drawback in this strategy. However, due to the relative small number of *busmacros* wires, which is a resource typically used in system based on partial reconfiguration (as discussed in Section 2.3.3), the IP address decoding capability is already limited.

8.1.3. Integration into IFS Tool

The Interface Syntheses (IFS) Design Flow is the result of a research conducted at the University of Paderborn, in the scope of a Phd work [19]. It is intended for an automatic generation of interfaces, enabling the connection of different type of components. Furthermore, it has an adequate level of abstraction that allows it to cover different media types and different communication protocols, in both connected endpoints. In cooperation with the authors, IFS was extended in order to automatically generate an interface, which connects an arbitrary reconfigurable IP to DREAMS OS, following the strategies explained above.

To allow the creation of the data structures expected from OS driver, which need to be DREAMS conform, C++ classes templates give the structure of the possible registers that can be formed. For further information, in Appendix B the class diagram of the registers templates is presented. This information, along with adequate and sufficient knowledge of the underlying hardware architecture, was included into IFS tool. The IP

Register File specification is entered into IFS using its GUI (Graphical User Interface). Based on these inputs, and using proper mechanisms, IFS generates an appropriate interface for the specified IP, which is in accordance with the strategies specified above.

8.1.4. Further Extension for DREAMS

After interface generation, the designer starts the integration of the virtual register access functions with the remaining software code in DREAMS. Since a protocol is used to communication with the IP, access patterns when using those functions can be observed. For instance, to write a data in one register of the Register File, according to the specified protocol, following steps are necessary: (1) wait until READY is set to OK (channel is idle), (2) write the ID of the related register into REG_ID, (3) write the data into DATA, (4) write the command for writing into CMD, and (5) set up the START. If more registers are sent in sequence, then this process needs to be repeated accordingly. Therefore, it would be adequate to provide those kinds of patterns as functionalities, increasing the abstraction of this low-level register access. A further advantage of this approach is that, further changes in low level access (e.g., the protocol improvement by including burst transfers support) are hidden.

In the current implementation, this layer is incorporated into DREAMS in the form of a library, which allows a seamless integration of the IP functionalities in a more abstract manner. Figure 8.3 depicts the required interaction and data exchange as previously explained, using as an example a sample code of *Relocation IP Manager*, which is responsible to manage the location of the task, either in FPGA or in CPU.

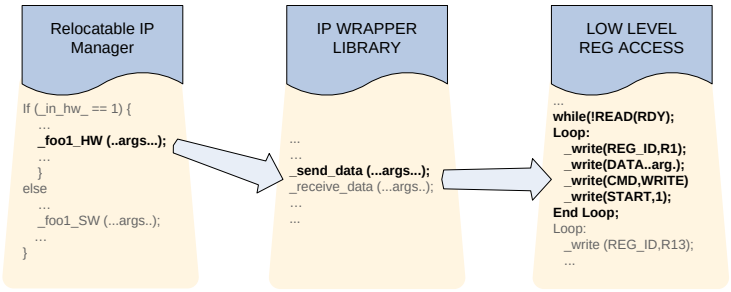


Figure 8.3.: A method *_foo1_HW* build upon library calls.

8.2. Relocatable Tasks Design

Task migration, in the software only execution domain, has been analyzed by several researchers in the past and techniques are almost consolidated. The requirement for task migration appears in distributed system scenarios. In these cases, a task may be

dynamically relocated over the several computation units in order to, for instance, allow an adequate adaptation of the computational load.

Inside the same execution unit, the relocation of a complete task was required in the early times, when fixed or dynamic partitioning of the physical memory were the current techniques applied for the memory management. Nowadays, virtual memory schemes based on paging and segmentation are superior. These new methods allow the division of the task code in several pages, which are then freely placed on the memory frames [60].

For tasks running on FPGA this technique cannot be used, and therefore several authors propose different approaches to handle the relocation of tasks into the RH in order to overcome the fragmentation of the RH area. Some examples have been already analyzed in Section 3.2.7.

When multitasking is the paradigm adopted, then an additional feature that may be required is the support for task preemption. Standard and fast techniques are available for software tasks and are widely used. For hardware tasks, there existing approaches based on different techniques. Hence, as long as the preemption and resumption remains inside the same execution domain (intra-domain migration), it is not a problem. Nevertheless, the migration type demanded here needs to happen across different execution domains (inter-domain migration), whose computation paradigms are completely different: sequential versus spatial computation. This by far imposes a big challenge to accomplish.

As it has been already noticed in Section 3.4.1, this problem has not been completely addressed in literature. Even though that some researchers have also identified that task relocation/migration across domains are necessary in order to provide the required flexibility for an execution platform, they are limited in the system level analysis. Hence, no solutions for designing such relocatable tasks have been yet proposed.

8.2.1. Unified Task Representation

When stopping a computation being performed, for instance, by the FPGA, which will be resumed later on in the CPU, context data has to be transferred across execution domains. So, a mapping between these two different contexts needs to be identified. In addition, it has to be assured that the preemption/resumption point of a computation needs to be equivalent in both CPU and FPGA. In other words, a match between two different implementations of a computation (CPU and FPGA) is necessary.

If both FPGA and CPU tasks are designed independently, the identification of these match points is not trivial and sometimes impossible, since the used computation paradigms are completely different. Therefore, the approach used in this work is based on the identification of these match points in early stages of a task design. For this purpose, the task needs to be represented in a unified manner, where those match points can be specified.

Then afterwards, this task can be translated to hardware and software implementations in a proper way so that these match points are kept.

In this thesis a state transition graph, similar to the one introduced in [134], is used as a unified task representation. Figure 8.4 shows an example of such a graph, where each state represents a computation block performed inside a task. The number of states and their granularity is defined by the user.

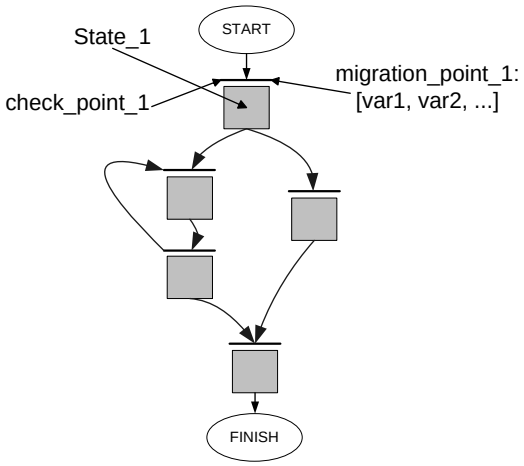


Figure 8.4.: State transition graph representation of a relocatable task.

The major goal by using this representation is the ability to establish the matching points between compiled and synthesized task versions, for CPU and FPGA respectively. Thus, each state transition specifies a *switching point* that represents a possible relocation point. Nevertheless, the state transition possibilities showed in the graph help the user in the specification of the context data, which is associated to each matching point only (hereafter called *migration point*). Thus, *migration points* and *states* are paired one to one.

The arcs in the graph (state transition) represent the possible Switching Points for a task. Thus, each arc carries the context information, which needs to be transferred into the related migration, allowing the correct execution of the remaining computation in the target domain.

In order to allow an automated code generation, the information given by the user, represented using the proposed unified form, should be expressed in a formal manner. By the time of writing this thesis, the specification of a proved language for a formal description of switching points and context data was in its first steps and has not been finished. Thus, the tool for automatic code generation in this case was not completed. Even though, an informal specification exists, along with templates for code generation, strategies for a hybrid task generation and a run-time infrastructure to perform task relocation. These all together belongs to the framework for relocatable task design and will be presented in the following sections.

8.2.2. A Framework for Relocatable Task Design

The complete framework design flow consists of three steps, which are applied after having the task represented using a state transition graph as explained above. The end result of the framework is the hardware and software versions of the task, along with a migration manager component. The latter is incorporated into the target operating system as a driver, offering the necessary interface to make the usage of services related to a task migration (e.g., preempt/resume, save/restore context, etc.). The complete design flow of a relocatable task embraced by the framework is depicted in Figure 8.5 and will be explained in the following.

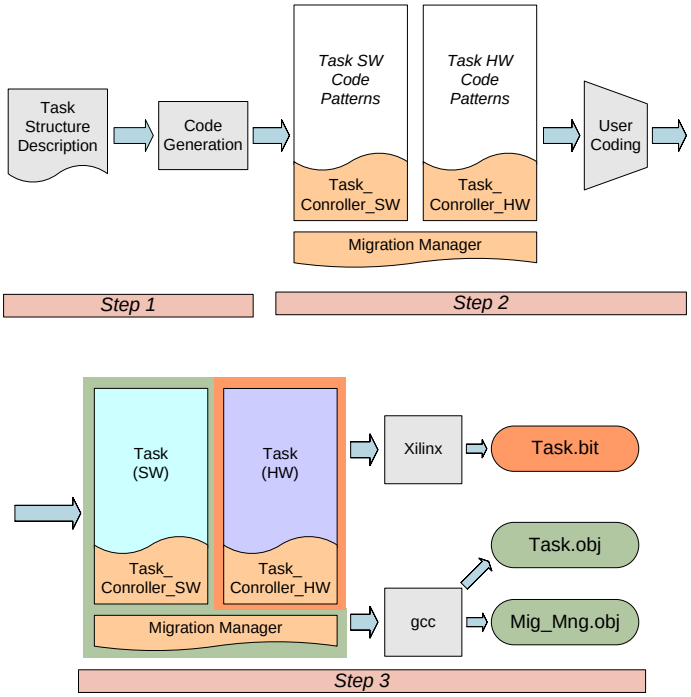
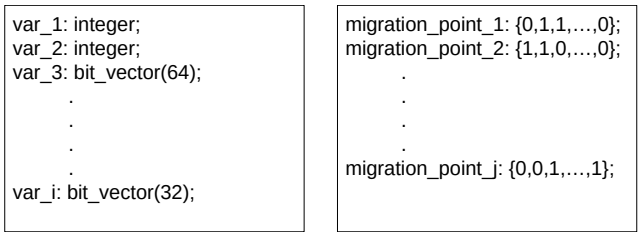


Figure 8.5.: Design flow supported by the framework.

Context Data Representation

In the first step switching points and context data are identified for each migration point using the unified task representation, which are afterwards specified using an informal description, called *Task Structure Description* (TSD). Figure 8.6 shows an TSD example, which comprises two segments: *Context Data Definition* and *Migration Points Definition*.

Instead of defining the context for each migration point individually, it was chosen to define a set of variables comprising all context data candidates. Then, for each migration



(a) Context Data Definition. (b) Migration Points Definition

Figure 8.6.: Informal description of TSD.

point a sub set from this context set is specified, since the migration points may probably share some common context data.

Each entry in the *Context Data Definition* set (Figure 8.6a) has a form of `var_i:variable;`, where *var_i* is a variable specified for the context data and *variable* is a unified type specification. Some commonly used are, for instance, integer, bit, bit-vector, etc. But when they are translated by the code generation tool, to C and VHDL, the type of one variable may have two different versions. For example, the bit-vector type has straightforward representation in VHDL. However, in C a possible representation may be the use of integer arrays.

In the second segment (Figure 8.6b), each migration point is specified by its context data set, which is a subset of the previous one. The syntax of each entry in this segment has a form of `migration_point_j:{...x_i...};`, where `x_i` is 1 if variable `var_i` (specified in *Context Data Definition*) needs to be transferred in this *migration point*, and 0 otherwise.

Code Generation

Based on TSD, the framework generates: code patterns for task coding; two *Task Controllers* (one for each execution domain) that directly interfaces the task with the runtime system; and the *Migration Manager* entity that is responsible to manage the migration of a task and, at the same time, provide an appropriated interface to DREAMS OS.

The *Task Controller* is an interface between *Migration Manager* and the task to be migrated. It is responsible for extraction and/or restoration of related context data, and to drive specific signals to the task (e.g., when task preemption is required). The code generated for *Task Controller* is synthesized and compiled together with the hardware and software task code, respectively. Hence, *Task Controller* belongs to the task component.

Furthermore, the *Task Controller* keeps a Register File, where context data extracted is stored, or from which the context data are taken to be restored back. In the case of a hardware task, the methodology explained previously in Section 8.1 is applied to

generate the appropriate interface to *Migration Manager*.

The *Migration Manager* entity is responsible to coordinate the context data transfer when a migration is being performed, communicating therefore with both *Task Controllers*. Moreover, this component offers proper functionalities (a driver) to DREAMS OS in order to manage a task migration. Examples of these functionalities are *get_context*, *set_context*, *task_preemt*, and *task_resume*.

Since the behavior of both *Task Controller* and *Migration Manager* are well defined, depending only on the specification given by TSD, they are stored into the code generation repository as parametrizable templates.

Task Coding

In order to assure a correct communication between *Task Controller* and the task itself, the latter must be coded following predefined code patterns and coding rules. In the following, the procedure for hardware task coding is explained, while the application of the same procedure for a software task is straightforward.

Figure 8.7 shows the architecture skeleton of a hardware task generated by the Framework. The VHDL code generated comprises a **entity** - **architecture** pair, along with **signals** for context data representation. Each *Task Process* is one VHDL **process** used for hardware task coding. It has to be noticed that the proper mapping of states (defined in the state transition graph) to VHDL **processes** is the responsibility of the programmer. For instance, one single state, specified in the graph, may be implemented using more than one **process**. Therefore, *State Synchronization Controller* is used to resolve the *Task Out State* signal, which indicates which state the task execution is. *Task Controller* needs this information in order to assure a correct context data saving and/or restoring.

A hardware task may be implemented using several *Task Processes*. However, each **process** that make use of context data signals must be created upon a pattern shown in Listing 8.1. It can be seen that signal *enable*, driven by *Task Controller*, is used to indicate when a task is under normal execution (not under migration or suspension). All meaningful task computation needs to be done only when this signal is asserted.

In order to support the programmer in solving *Task_State_Out* signal, another VHDL **process** pattern, presented in Listing 8.2, is provided by the framework. It allows the correct integration of *State Synchronization Controller* with the hardware task.

Additionally, the programmer needs to assure that when a task is enabled for computation, it can use only those signals specified for the migration point right before this state. Only they have been maintained at the migration point and have thus valid values.

For information completeness, further details of *Task Controller* and *Migration Manager* are given in Appendix C.

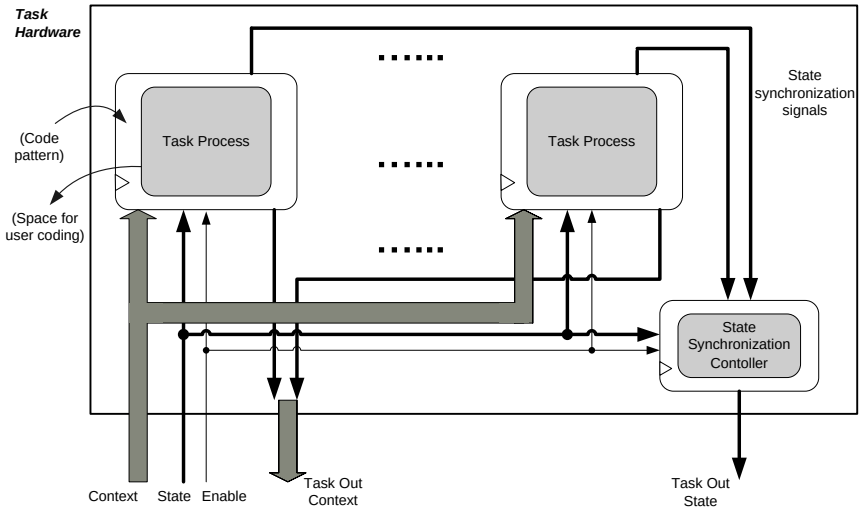


Figure 8.7.: Hardware task overview.

Binary Files Generation

In Step 3 of the the hardware and software task versions are generated, together with the *Migration Mananger*. For a hardware task, a partial bitstream is generated following the guidelines provided by Xilinx in [43]. For the software task and for the *Migration Mananger* the *gcc* compiles is used.

8.3. Chapter Conclusions

In this chapter, the necessary design support for the proposed architecture was presented. This support comprises two aspects: the provision of an automatic interface generation between a hardware component and DREAMS OS; and a framework that supports programmers in designing relocatable components for the hybrid architecture.

It has to be noticed that the framework is proposed with the intention to support the designer in the development phase of a relocatable task. Additionally, it generates the necessary infrastructure to enable the migration at run-time. By this means, the strategies provided by RRM (explained in the last chapters) can make use of this infrastructure to decide at run-time about the allocation of each relocatable component.

The number of switching points defined by the designer of a hardware task influences directly the migration flexibility. Indeed, as higher the number of switching points a higher degree of flexibility is achieved. Since each switching point represents a matching point between the computation in software and hardware, one may think in a first glance that this may represent a serialization of the hardware execution. This is true for an

Listing 8.1: VHDL code sample for Task Process template.

```
...
-- << USER CODE/
-- define your signals and variables here
-- USER CODE >>

-------------------------------------------------------
--         Code pattern for Task Process             --
-------------------------------------------------------
process(CLK)
  -- << USER CODE/
  -- define your variables here
  -- USER CODE >>
begin
  if(CLK='1'and CLK'event) then
    if(enable ='0') then
    -- << USER CODE/
    -- reset your own variables and signals
    -- USER CODE >>
    else
      -- << USER CODE/
      -- All the computations go here
      -- USER CODE >>
    end if;
  end if;
end process;
-------------------------------------------------------

-- NEXT PROCESSES GO IN THE FOLLOWING ---------------
...
```

algorithm which presents a high degree of parallelism. However, there is always a limit in which an algorithm can be parallelized, either because of its nature or because of the available hardware area available for its implementation. Moreover, there is no limit of parallelism that can be used when implementing a single computation block (state), since a task computation can be preempted only at switching points. Hence, the designer needs to find a proper compromise between flexibility and efficiency when searching for a optimal mapping of the algorithm in a state transition graph.

It is worth to mention each switching points should also be selected so that the amount of related context data would be relatively low. This will result in the generation of a small *Task_Controller* (i.e., less FPGA area) and will decrease the amount of time necessary for migration (i.e., faster switching time).

Furthermore, knowing the amount of migration data required in each state transition,

Listing 8.2: VHDL code sample for State Synchronization Controller template.

```
...
-------------------------------------------------------
-- Code pattern for State Synchronization Controller --
-------------------------------------------------------
process(clk)
  -- << USER CODE/
  -- define your variables here
  -- USER CODE >>
  begin
    if(clk='1'and clk'event) then
      if(enable ='0') then
        state_out <=0;
        -- << USER CODE/
        -- reset your own variables and signals
        -- USER CODE >>
      else
        -- << USER CODE/
        -- logic for determining Task_Out_State
        -- USER CODE >>
      end if;
    end if;
end process;
-------------------------------------------------------
...
```

and additionally the characteristics of the execution platform, it is possible to estimate the time necessary for task relocation. This is important since it gives the necessary information for the reconfiguration management strategies, as explained in Section 6.4.

CHAPTER 9

Case Study

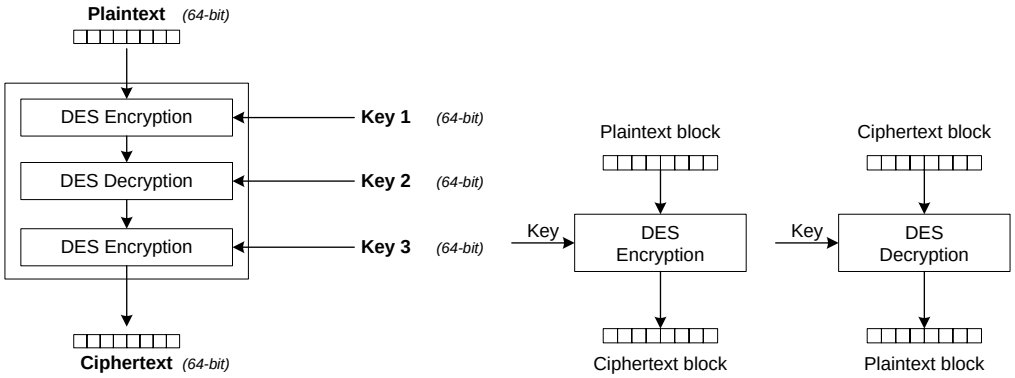
This chapter presents a case study that has been carried out to validate the core methods developed in this thesis. Nevertheless, the focus is given in the evaluation of the framework in designing a relocatable OS service, and the practicability in relocating a component between FPGA and CPU along with related strategies.

9.1. Target OS Service

An encryption algorithm has been selected as target OS service, since it is being increasingly demanded by safety and security applications (e.g., when using a smart phone to access a bank account). Although a hardware implementation of such an algorithm may provide better performance compared to its software counterpart, the latter is still commonly deployed in computer systems. Furthermore, encryption algorithms are considered an OS service, for both desktop and embedded scenarios [163].

Triple-DES

Among different kind of encryption algorithms, a Triple-DES (shortly TDES) was chosen, since it is one of the most known and widely used one. The TDES, depicted in Figure 9.1, is based upon a basic encryption-decryption block, called DES (Data Encryption Standard), which is an algorithm for a block cipher operating with a 64-bit key on 64-bit plaintext blocks (in this case).



(a) Triple-DES encryption process.

(b) DES cipher block for encryption and decryption.

Figure 9.1.: Triple-DES and its basic DES cipher block.

DES Block Cipher and its Operation Mode

A single block cipher maps n-bit plaintext blocks to n-bit ciphertext blocks, where n is equal to 64 in this case. Since each block cipher operates on blocks of fixed length, but plaintext messages to be encrypted can be of any length, the input data (plaintext) need to be partitioned into 64-bit blocks. From now on, there are different ways in which the block ciphers are employed on these blocks, and they are called *operation modes*. Among these modes, the CBC mode (Cipher-Block Chaining) has been chosen, which is the most commonly mode used.

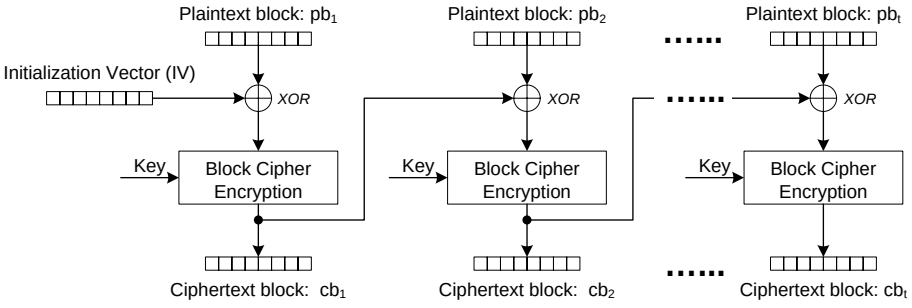


Figure 9.2.: CBC mode for block ciphers.

Figure 9.2 presents the DES configured for the CBC mode used to encrypt the input, a plaintext. If the plaintext pb is divided into t blocks, so that $pb = pb_1 \dots pb_t$, after the encryption a number of t ciphertext blocks, so that $cb = cb_1 \dots cb_t$, will be generated as the output (the cb ciphertext).

In the CBC mode the encryption works as follows. Every incoming plaintext block pb_j is

XORed with the previous encrypted ciphertext block cb_{j-1} before being applied to the block cipher algorithm. For the first block pb_1 an Initialization Vector (IV) is used. In this way, since all encryptions are chained, each ciphertext depends on all the plaintext blocks before it. Hence, an intrinsic sequential operation is required.

In Figure 9.1a it can be seen that the final output cipher block cb_j is not achieved by applying only one time the DES encryption. Instead, each input pb_j is streamed through a chain of DES encryption/decryption blocks, using thereby three different keys (one for each DES phase). Hence, the encryption procedure of an input plaintext using TDES can be described as stated in Algorithm 6. Please note that for a plaintext with a length bigger than the input specified (64-bit for this case), it needs to be partitioned in a certain number of blocks (controlled by *block_count* in line 17) and each of them needs to be sequentially fed when requested (line 11).

Algorithm 6 Triple-DES pseudo algorithm.

```
1:  define variables:
    key_1, key_2, key_3, plain_block, cipher_block, nblock, Init_Vector and block_count;
2:  Begin
3:      read_in(key_1);                                              ▷ State 1
4:      read_in(key_2);                                              ▷ State 1
5:      read_in(key_3);                                              ▷ State 1
6:      read_in(nblock);                                             ▷ State 1
7:      read_in(Init_Vector); \\IV                                   ▷ State 1
8:      block_count ← nblock;                                        ▷ State 1
9:      cipher_block ← Init_Vector;                                  ▷ State 1
10:     while block_count > 0 do
11:         read_in(plain_block);                                    ▷ State 2
12:         plain_block ← plain_block XOR cipher_block;              ▷ State 2
13:         cipher_block ← DES_Encryption(key_1,plain_block);        ▷ State 2
14:         cipher_block ← DES_Decryption(key_2,cipher_block);       ▷ State 3
15:         cipher_block ← DES_Encryption(key_3,cipher_block);       ▷ State 4
16:         send_out(cipher_block);                                  ▷ State 5
17:         block_count ← block_count − 1;                           ▷ State 5
18:     end while
19: End
```

9.2. Relocatable Triple-DES

A possible representation of the TDES algorithm using the unified task representation, is shown in Figure 9.3. Each DES block is mapped to a single state (States 2,3 and 4), since there is only one hardware instantiation of the DES encryption-decryption algorithm. Moreover, two additional states were included: State 1 and State 5. The

fist is responsible for gathering the initial data (three keys, Initial Vector, and block numbers),and the latter is responsible to send out the ciphertext block and to decrement *block_count*. The initialization phase of the algorithm is not represented in this figure. The variable *nblock* is used to indicate how many blocks of plaintext will be applied into the TDES.

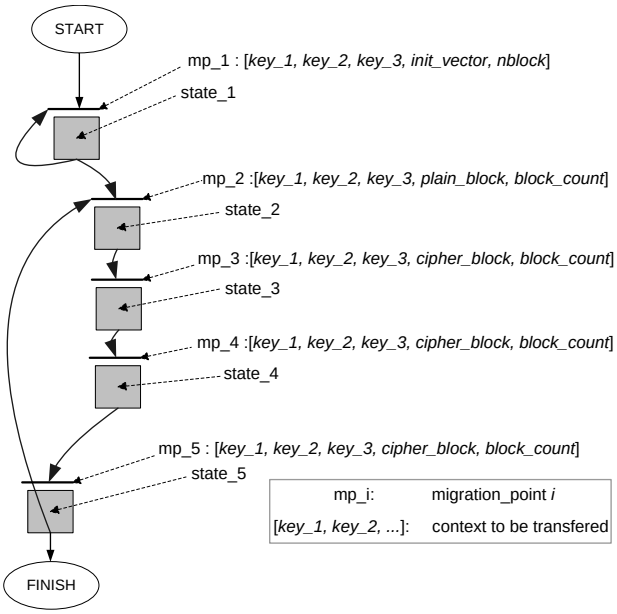


Figure 9.3.: State transition graph corresponding to Algorithm 6.

Figure 9.3 shows further the context data identified, which later was specified in the TSD format, in order to allow the system to be generated using the framework. The plaintext is read in State 1, thus *plain_block* data should be transferred in the case of a migration between States 1 and 2. The variable *block_count* is used to count how many blocks of plaintext are still waiting in the input. Moreover, due to the fact that a state transition exists, from State 5 to State 1, all keys used by TDES must be transferred in every migration case. If this transition would not exist (the case of a single TDES pass), after each DES block one key less would be necessary to be transferred.

It is worth to mention that the chosen state graph presented chosen for this case study is only one possibility among several others. One looking for a more coarse grain approach may combine all three DES encryption-decryption states in a single one, so that a pipeline approach in hardware may be used. However, this will certainly leads in an increase of the FPGA area needed for its implementation. Another possibility is to go towards a more fine grained approach, so that the DES decryption-decryption block could be depicted allowing the specification of further states.

9.3. Testbed Set Up

Determinism is an important aspect of a real-time system. Therefore, it is not a good practice to allow an application task to perform its activity with an arbitrary amount of input data at once, since this may lead to an unbounded execution time. This may complicate the scheduling analysis. For same cases, an approach used is to split the entire computation in time, using therefore a periodic task. Hence, the whole computation is accomplished after executing a bunch of instances of this task. So, by using this strategy, it is possible to consider this task into the scheduling analyses.

This strategy was also followed in this case study. An application task was created so that in its initialization phase it divides the data to be encrypted in several blocks, and in each of the subsequent instances of this periodic task, one block is encrypted.

By adopting such a strategy, the TDES OS service will also present a periodic behavior, which perfectly fits in the migration cases analyzed in the Section 6.4. Hence, the RRM is able to relocate an OS service from software to hardware (or vice-versa), either in between two consecutive instances of a service execution (before State 1), or preempting it. Furthermore, this strategy allows the usage of this service by other application tasks, which characterizes it even more as an OS service.

Basic Execution Platform

The execution platform built for the purpose of this case study is presented in Figure 9.4. Some further details are omitted for legibility reasons. The architecture was constructed using the Virtex-II Pro device XC2VP7, on which one PowerPC 405 hardcore microprocessor was used. The system was set up so that the microprocessor was clocked internally at 200MHz and the busses at 100MHz.

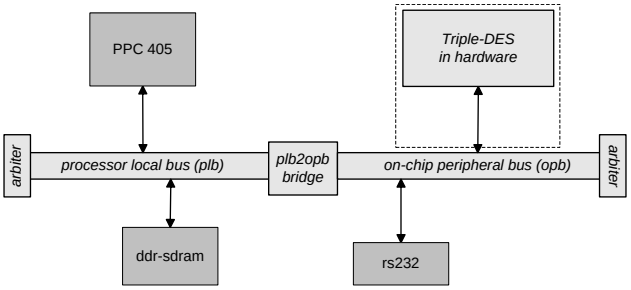


Figure 9.4.: Basic hardware platform.

The Triple-DES OS service was designed and implemented using the framework and the interface generation as explained in previous chapters. The hardware version was attached to the OPB bus of the system and the software version placed into the external memory, along with the remaining software system that is executed by the PPC405

microprocessor.

For this case study a static instantiation of the architecture was used, instead of partially configuring the FPGA. Alternatively, the partial reconfiguration was emulated by providing inside the system two dummy stubs *load_sw_task();* and *load_hw_task()*. Nonetheless, this fact does not compromise the results achieved by this case study, whose time cost can be estimated by using the results presented in related work.

An additional hardware component was added to the platform, a serial interface. This is merely used to enable an interface to the user so that the system can be started, stopped, and debugged.

Along with the software version of Triple-DES, three other threads were created. One is a periodic application thread that, in each instance, calls the Triple-DES OS service getting back the results. A second thread was responsible to manage the migration/relocation of the OS service, communication therefore with the *Migration Manager*. The third thread was responsible for controlling the UART in order to provide the interface with the user.

9.4. Quantitative Results

FPGA and Memory Costs

Table 9.1 presents the FPGA area (in LUT units) and the memory (in bytes) consumed by the main system parts, according to the report given by the EDK tool. The *DES-block_HW*, implemented following the templates given by the framework, consumes 2320 LUTs (23% of the total FPGA resources) and *Task_Controller_HW* uses 2108 LUTs (21% of the total amount).

HARDWARE	LUT	Utilization
DES-Block_HW	2320	23%
Task_Controller_HW	2108	21%
Entire FPGA	9856	100%
SOFTWARE	Memory Utilization (bytes)	
DES-Block_SW + Task_Controller_SW	9794	
Migration Manager	18236	

Table 9.1.: FPGA and memory utilization.

The majority of the FPGA resources required to implement *Task_Controller_HW* was related to the migration controller, rather then the task itself. Thus, the resources used by *Task_Controller_HW* is less related with the resources used by the task itself.

The values presented in Table 9.1 are resulted from a first prototype of the framework.

Neither the system design nor its implementation has received special care in the code optimization. Therefore, less FPGA resource utilization for the migration system can be foreseen. Moreover, taking into consideration that a great flexibility is achieved from the run-time relocation capability, this resource requirement increase is acceptable.

Reconfiguration Time

The FPGA reconfiguration time related to the hardware component generated (*DES-block_HW* plus *Task_Controller_HW*) was estimated using the results provided in [164]. This work was chosen due to its close similarity with the architecture used in this case study. According to it, the reconfiguration time T_d of a partial bitstream is given approximately by:

$$T_d \approx f_d \times \left(\frac{1}{t} + \frac{1}{w_d}\right) \tag{9.1}$$

Where f_d is the equivalent number of frames in the bitstream, t is the rate at which a frame is transferred from memory to the OPB peripheral, and w_d is the rate at which a frame is written into the ICAP entity.

Using the results shown in Table 9.1, the partial bitstream corresponds to $2320 + 2108 = 4428$ LUTs. From the official information given by Xilinx Company in [40], the XC2VP7 FPGA device (the same used in this case study and in the related work) has a total of 1320 frames and 9856 LUTs. Hence, it can be concluded that 4428 LUTs represent roughly 593 frames in this device. Using $t = 7.86$ frames/ms and $w_d = 117$ frames/ms, since the buses frequency are the same as in [164], the reconfiguration time of the OS service component would be:

$$T_d \approx 593 \times (\frac{1}{7.86} + \frac{1}{117.0}) = 81ms \tag{9.2}$$

Following the model proposed in this thesis for a reconfiguration activity, (see Chapters 5 and 6), this is the execution time Q^h of job J^a (*Programming* phase).

Migration Time

The time spent to transfer the context date during one component relocation was not estimated, but measured using specific physical outputs available in the development board used for this testbed. The value measured corresponds to the complete time spent since the moment in which the component recognizes the signal sent from *Migration Manager* for preemption, until finishing the complete reconstruction of the context data in the target execution domain. The average values measured are presented in Table 9.2.

SW ⇒ HW	HW ⇒ SW
$10\mu s$	$8\mu s$

Table 9.2.: Context data transfer: average time measured.

The migration times observed are rather the same for each one of the possible preemption points. This is due to the fact that the amount of context data specified in Figure 9.3 is the same, even though the different variables in each case. These values even incorporate the overhead caused by the usage of the communication protocol (specified in Section 8.1.2).

9.5. Chapter Conclusions

In this chapter a case study was performed, in which the main mechanisms for enabling the incorporation of flexibility together with efficiency could be analyzed. As a target OS service, a specific encryption algorithm was chosen, which fits well in the scope of the work proposed in this thesis.

Based on the quantitative results, especially on those related to the resource utilization, it could be identified that a relative big amount of resources are spent only for the management of the relocation activities. This is more evident when looking to the FPGA area utilization. Nonetheless, one can argue that this resource is traded in favor of the great flexibility obtained from the capability to relocate hybrid components at run-time.

Reconfiguration and migration times need to be considered separately for a reasonable analysis. In certain systems the FPGA reconfiguration time, which stays in the range of dozens of milliseconds, may represent an obstacle for some target applications, especially when it needs to happen in a relatively short time. In the system proposed in this thesis, this time is hidden by executing it concurrently with the application.

These results are caused by, first, properly choosing the migration points, so that in each migration case, a low amount of data is required to be transferred. A second reason is that the context data do not contain any data processed by the component. In this case study, plaintext and ciphertext are not included into the context data. Instead, the application task transmits and receives the plaintext and ciphertext, respectively, using the same communication channel, regardless the allocation of the OS service (hardware or software).

This approach is beneficial, since transparency is achieved for the application tasks that do not need to be aware of the execution environment of the required OS service. However, this solution may impose an obstacle for some applications requiring very high performance. Alternatively, the hardware architecture could be incremented with

another bus, interconnecting the hardware components and the main memory (where the data processed would be stored), so that both software and hardware components may have access to them, without loosing communication efficiency.

CHAPTER 10

Conclusion & Outlook

10.1. Summary

This thesis work presents a set of methodologies, strategies, and mechanisms that, when incorporated into a reconfigurable RTOS, promotes its self-reconfiguration over the execution platform. The main goal, thereby, is to enable an efficient utilization of the hybrid computational resources, shared among application tasks and OS services. By allowing a flexible management of the OS services, the proposed system is capable to assign those operating system services to the computational resources not currently used by application tasks.

The focus of this thesis is on embedded systems requiring flexibility, high computation capability, and soft-real time constraints. Some contemporary products in the marked already provide such characteristics, like modern mobile phones or PDAs. In such devices, applications may enter or leave the system dynamically, characterizing in this way a changing environment.

In order to achieve the proposed goals, two research fields were combined: reconfigurable computing (RC) and self-reconfigurable operating systems (self-x OS). Therefore, an extensively survey of relevant works from both areas was made. Thereby, it was identified that an OS can also profit from a reconfigurable architecture along with the application, instead of only providing support to the application.

Following the tendency towards fine-grained RC architectures, this work bases its execution platform on a FPGA device. Its reconfigurable capabilities, especially the partial

reconfiguration, make it well suited for the purposes of this work. Based on the results presented in Chapter 2, it can be concluded that even though the technical realization of the partial reconfiguration of a FPGA is still a problem, some tendencies were identified towards improvement of tools and support from the side of the FPGA suppliers.

System Overview

The main system parts can be summarized in Figure 10.1, which provides an overview of the whole scenario. In this figure, it is shown that both OS and application use the computational resources of the hybrid architecture for their execution.

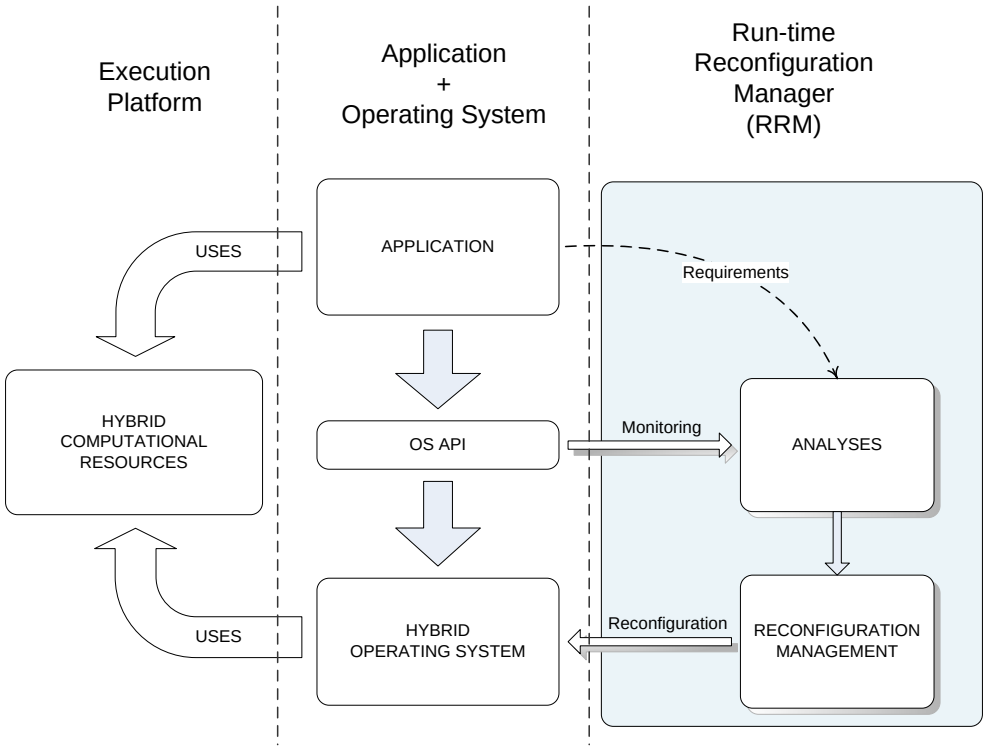


Figure 10.1.: System overview.

The main contributions of this thesis concentrate in the RRM part that provides several heuristic algorithms necessary to tackle NP-Hard problems identified in the proposal, methodologies to conduct the reconfiguration activities in deterministic manner, and a preliminary proposal towards run-time profiling of OS services. Besides that, proper models of the reconfigurable components along with an extended model for enabling an efficient management of reconfiguration activities were proposed.

The *Analyses* subsystem acquires dynamic information about the usage of OS services by the application through API calls monitoring. Here, a preliminary analysis regarding

the recognition of usage profiles of the OS services was performed. However, for the analyses made in the thesis, a simplified version of those metrics was used. Additionally, this subsystem receives off-line information from the possible applications of the system, regarding their requirements. These are used when starting the application in the system, giving the information about new OS services that may be required. Using this information, the *Analyses* decides where each OS service needs to be placed, either on FPGA or on CPU.

The *Reconfiguration Management* administrates the reconfiguration activities, comprising the relocation/reconfiguration of each single component and the ordering of components undergoing a reconfiguration, respecting thereby time constraints. To accomplish this administration, a proper model of these activities was provided that allows the appliance of specific techniques from real-time scheduling theory.

Besides, the relocation of hybrid components across FPGA/CPU boundaries was deeply investigated in this thesis. As a result, a novel framework, comprising a run-time execution environment and a design flow that guides the programmer by designing relocatable components, was developed and implemented. Furthermore, the interface between software and dynamically reconfigurable hardware components was automatically generated by taking advantage of a previous developed Interface Synthesis (IFS) tool in [19]. For this purpose, the IFS tool was properly augmented with the information of the platform (target OS adopted, FPGA internals information, etc.), so that desired interfaces could be automatically generated for the underlying architecture.

In order to promote a verification of main contributions in the practice, a study was conducted by setting up a basic architecture and choosing therefore an appropriated target OS service, which was an encryption algorithm. Furthermore, this component was implemented using the proposed framework allowing, hence, its dynamic relocation across CPU/FPGA. The tests performed have enabled the identification of some topics that may be further explored in a future work.

10.2. Outlook

As pointed out in Chapter 4, run-time profiling of OS service usage by the application may improve the quality of the decision related to the placement of OS services. By identifying patterns in which a service may be called, and furthermore, by identifying relation among these services, the allocation result may be more efficient. For instance, using the near past information concerning the usage of a certain service, the system would be able to infer, for the near future, the pattern expected by other services. Furthermore, these results may even avoid unnecessary reconfiguration activities that cannot be foreseen by a system that uses only current information.

Another aspect that may have direct impact, in the system presented by this thesis, is the frequency in which the system is required to reconfigure. This depends on which

specific target embedded system the proposed techniques are going to be used. In the case of a system where the reconfiguration is directly influenced by human inputs (mobile phone where the user randomly starts/stops applications), the reconfiguration frequency cannot be precisely determined. So, as a future work, different case studies should be investigated in order to estimate the system reconfiguration frequency. This information may be used to avoid unnecessary reconfigurations and also to avoid the system to become inoperable (a big reconfiguration frequency).

Concerning a hardware service of an operating system, some comments are worth to be made. First, the *area vs. speed* trade-off, intrinsically related to a hardware implementation of an algorithm, could be further exploited in order to make available more versions of the same service to the allocation algorithms, and so possibly achieving better results. This would have nevertheless direct impact in the framework proposed in this thesis.

Additionally, as long as enough space is available in the FPGA, replication of OS services may also be considered. This could increase the reliability of the system, or even allow heavy loaded services to be supported by the system.

The flexibility incorporated into the operating system by means of reconfiguration, as proposed in this thesis, could be expanded in order to promote the relocation and reconfiguration of applications tasks. Considering that the system architecture is based on microkernel concept, this approach should not be a problem, since both application tasks and operating system services are seen as components connected to the microkernel level. Nevertheless, most of proposed heuristic algorithms must be adapted in order to consider the dynamic relocation of application tasks.

It is worth to mention that more focus could be incorporated into the strategies and methods presented here concerning power consumption, since this is one of the most important factors that influence the design of an embedded system, along with performance and flexibility.

In spite of these future research directions, a question may also be opened concerning the appropriateness of using a FPGA as a mainstream technology for the execution platform envisioned within this thesis. Further efforts should be spent in investigations towards evaluation of, for instance, platforms based on coarse-grained reconfigurable architectures.

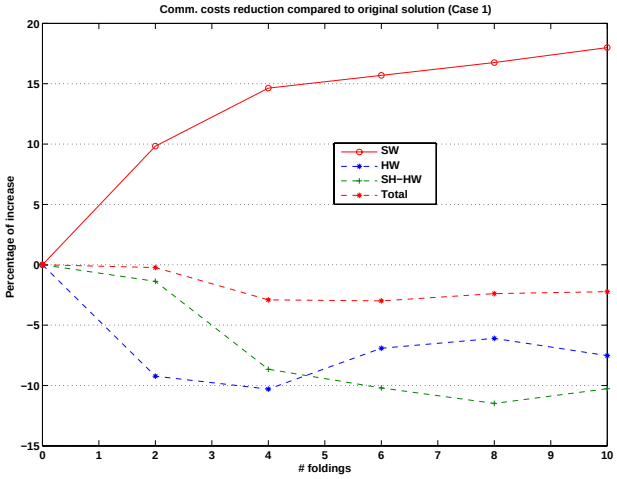
APPENDIX **A**

Further Evaluation Results

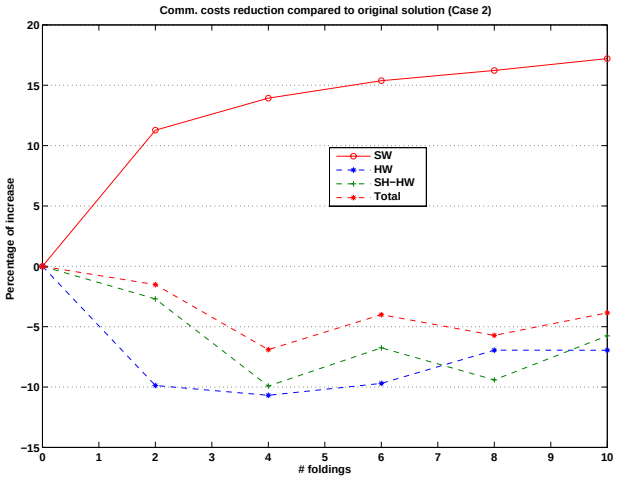
Additional evaluations were performed with the communication-aware allocation algorithm. For this experiment, three different relations of C^{α}, C^{β} and C^{γ} were used to generate the input graphs.

Figures A.1a, A.1b and A.1c show the gain in the communication cost reduction in each case.The related payoff in the overall resource usage utilization is shown in Figures A.2a, A.2b and A.2c.

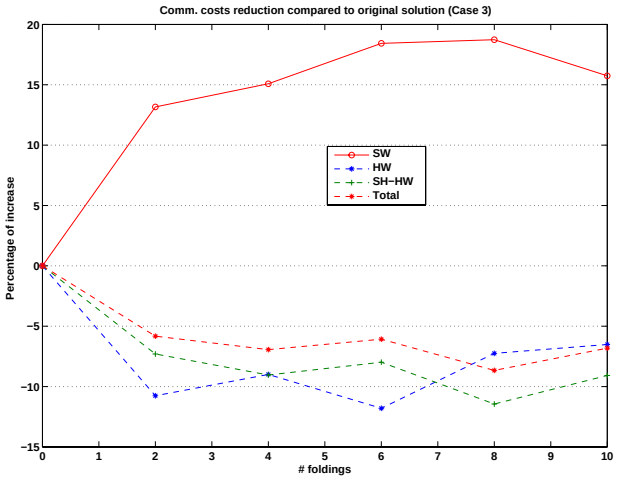
It can be seen that there is no significant difference in the gains obtained in Case 2 and Case 3 when compared to the gain obtained in Case 1. Thus, for larger differences in communication costs between inter and across domains, the algorithm do not achieve better results, even when the number of passes is increased.



(a) Case 1: $\overline{\kappa} = \{C^{\alpha},\ 2C^{\alpha},\ 3C^{\alpha}\}$

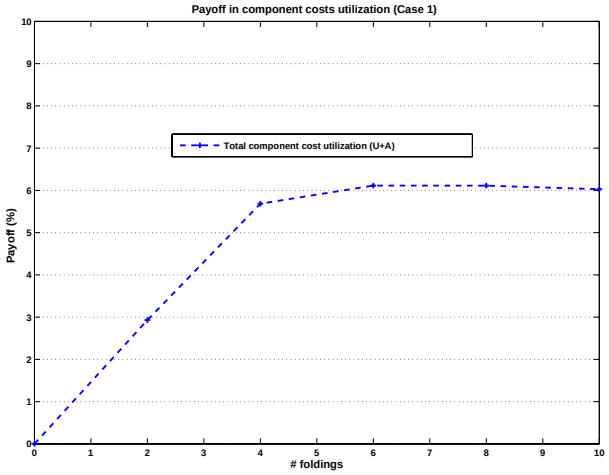


(b) Case 2: $\overline{\kappa} = \{C^{\alpha},\ 2C^{\alpha},\ 9C^{\alpha}\}$

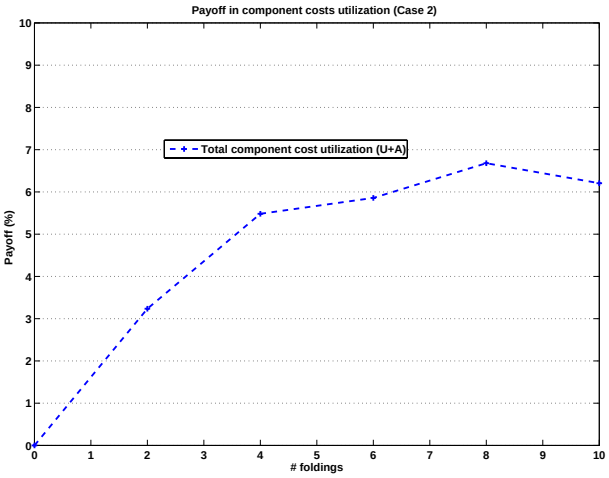


(c) Case 3: $\overline{\kappa} = \{C^{\alpha},\ 2C^{\alpha},\ 15C^{\alpha}\}$

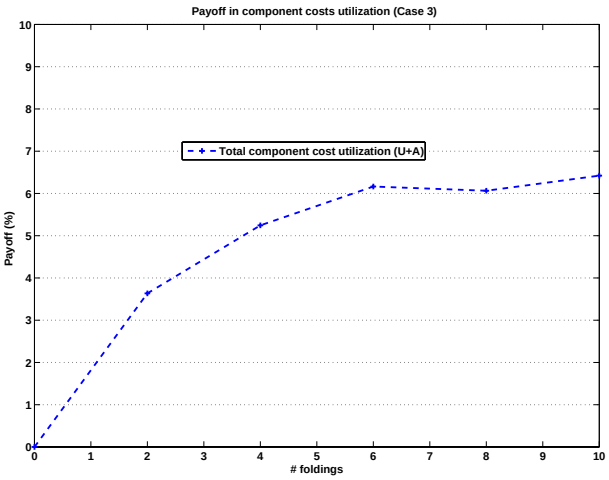
Figure A.1.: Comparison among communication costs reduction for three different situations.



(a) Case 1: $\overline{\kappa} = \{C^{\alpha},\ 2C^{\alpha},\ 3C^{\alpha}\}$



(b) Case 2: $\overline{\kappa} = \{C^{\alpha},\ 2C^{\alpha},\ 9C^{\alpha}\}$



(c) Case 3: $\overline{\kappa} = \{C^{\alpha},\ 2C^{\alpha},\ 15C^{\alpha}\}$

Figure A.2.: Comparison among payoffs in overall resource utilization for three different situations.

APPENDIX B

HW/SW Interface Generation

Figure B.1 illustrates the class diagram of possible interface registers used in the automatic interface generation process for DREAMS OS.

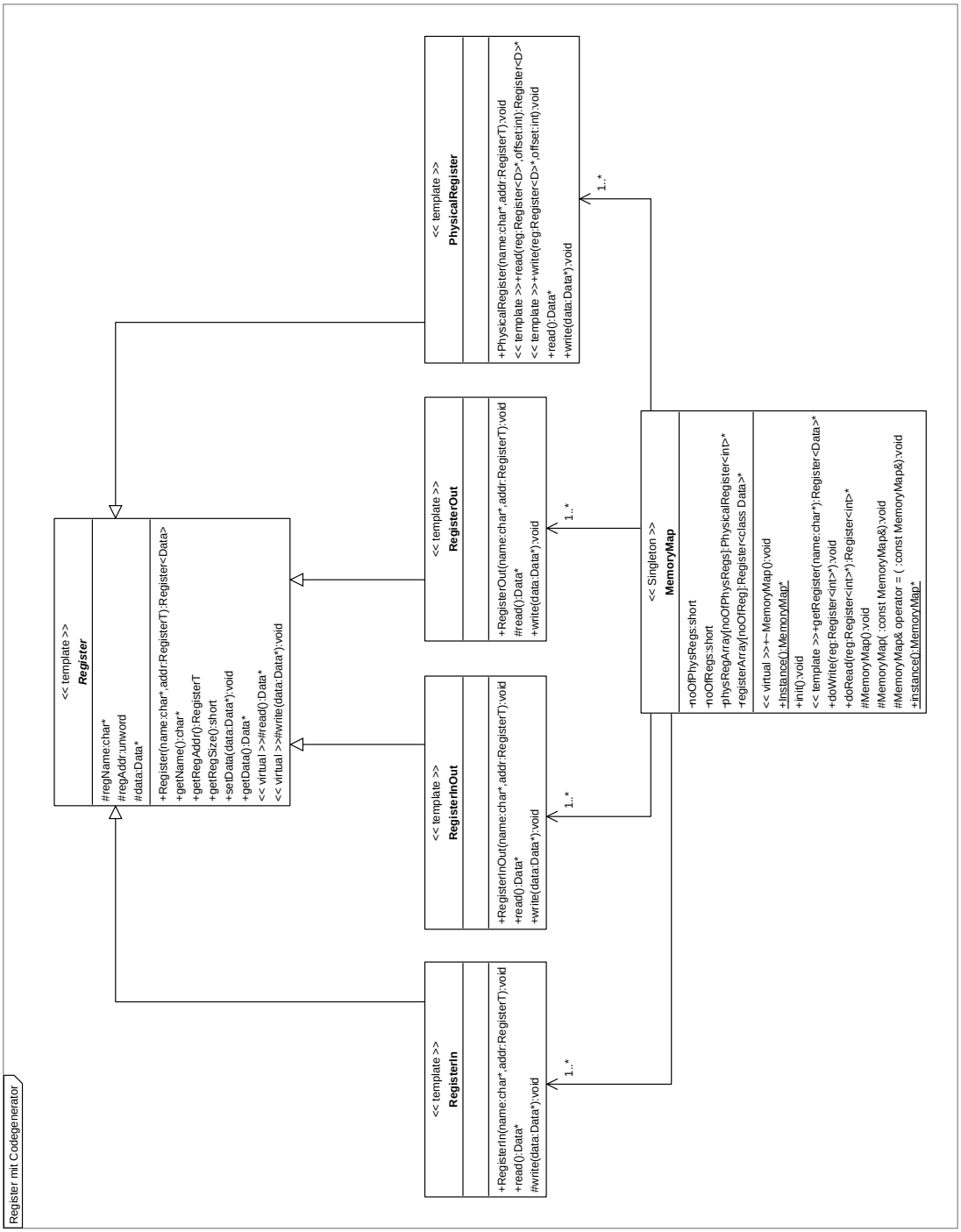


Figure B.1.: Class Diagram of possible interface registers.

APPENDIX C

Hardware/Software Task Design

C.1. Hardware Task Controller Template

Figure C.1 shows the template of a hardware task controller. The signals located at the upper part of this figure are connected to the hardware task, and the signals at the lower part are connect with *Migration Manager*.

C.2. Sequence Graphs for Two Migration Cases

The following two sequence graphs present the actions performed in two different migration cases, specifying concrete interface-callings that are performed across them. Figure C.2 shows a migration case from CPU to FPGA, and Figure C.3 the opposite case.

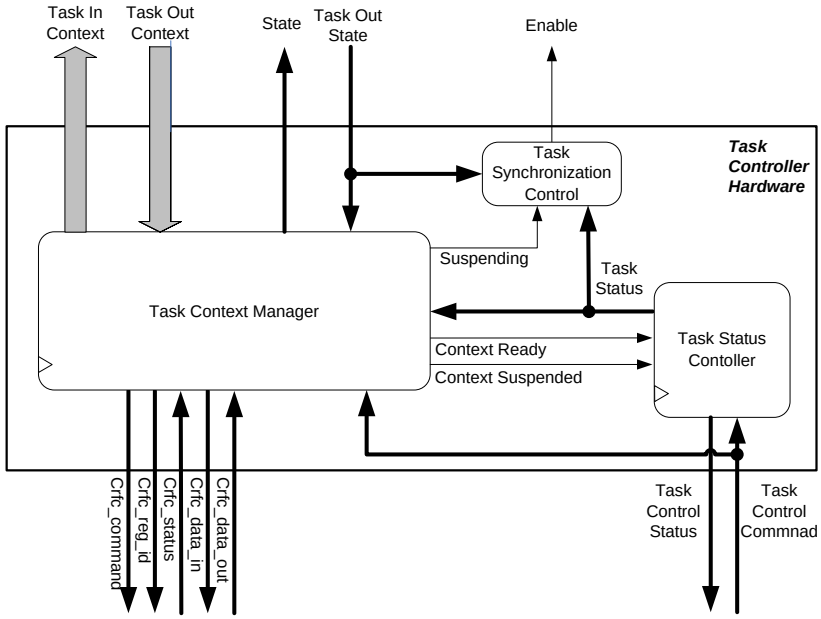


Figure C.1.: The controller template for a hardware task.

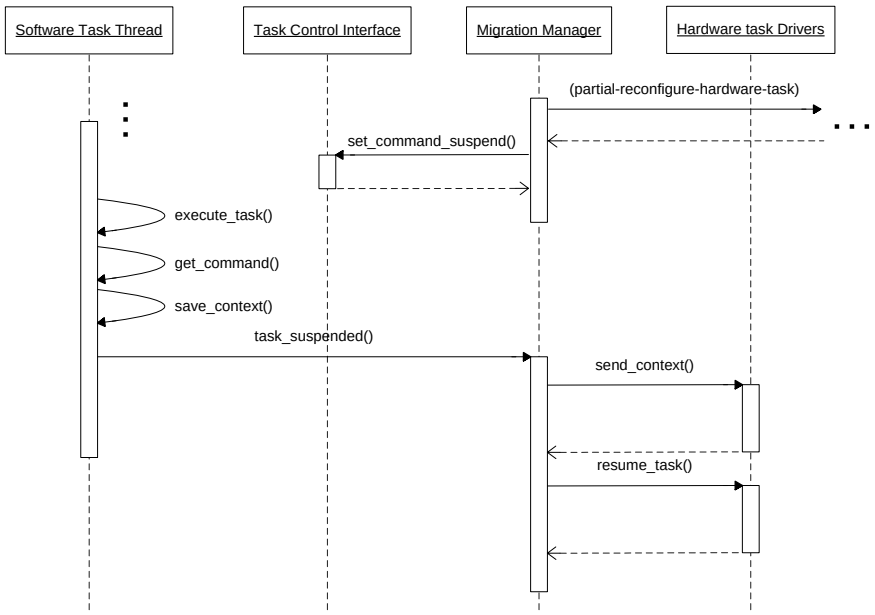


Figure C.2.: Sequence graph specifying the task migration from software to hardware.

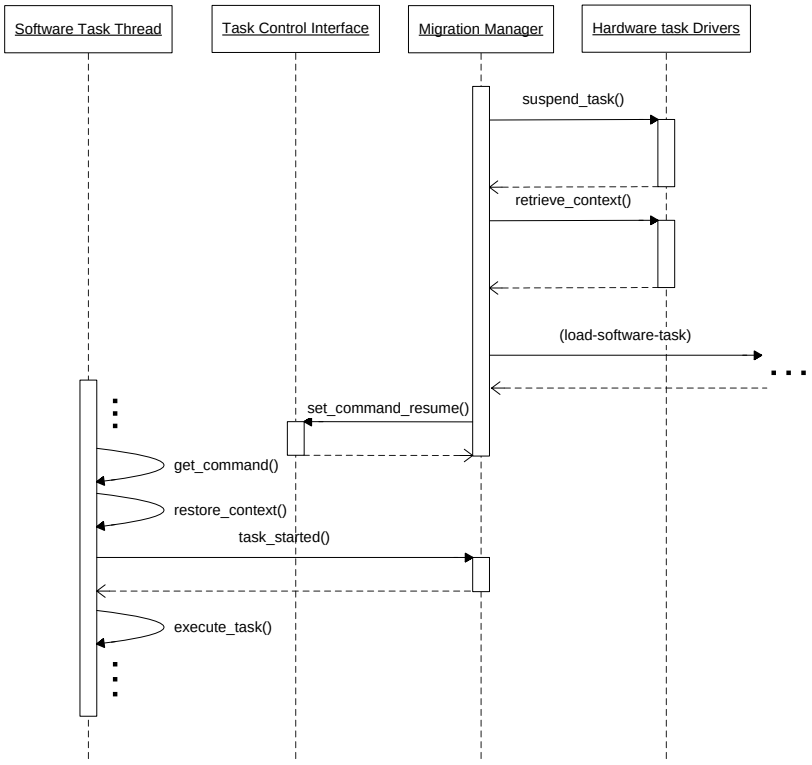


Figure C.3.: Sequence graph specifying the task migration from hardware to software.

APPENDIX D

TBS Server Bandwidth Estimation

This appendix presents the prove that the bandwidth U_i released by a component s_i, which is undergoing a migration from software to hardware, can be added to the TBS server bandwidth U_s, to schedule this migration.

Figure D.1 shows the scenario that is considered, similar to the one faced in Section 6.4.3, highliting the scheduling of the service that undergo a migration and the TBS server.

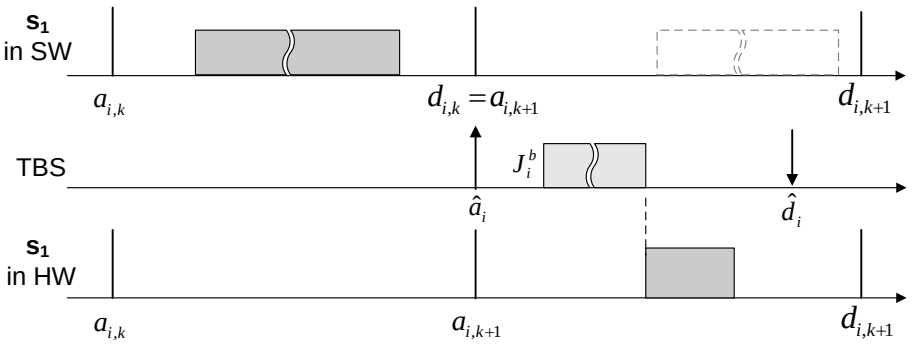


Figure D.1.: Relocation from software to hardware: An Example.

The whole system, considering all n OS services together with TBS, is schedulable under

EDF if:

$$U_s + \sum_{i=1}^{n} \frac{E_i^s}{P_i} \leq K \qquad \text{were } K \leq 1, \tag{D.1}$$

where E_i^s is the execution time of a service s_i in software, P_i is the period of this service, and K is the CPU workload available for both OS service set and TBS. Please note further that $U_i = E_i^s/P_i$.

Without loss of generality, lets assume that s_1 is the service undergoing a relocation, and let Equation D.1 shown above be rewritten as:

$$U_s + \frac{E_1^s}{P_1} + \sum_{i=2}^{n} \frac{E_i^s}{P_i} \leq K. \tag{D.2}$$

Looking to the equation above, it can be concluded that the workload U_i can be seen as the bandwidth of a second TBS server (hereafter called *second server*). Moreover, assuming that at time $a_{1,k+1}$ no job is waiting to be executed by *second server*, since service s_1 from that time on is no longer scheduled in software (see Figure D.1), its related job queue is free. Thus, its bandwidth U_i can be used to carry out further aperiodic jobs. In this scenario, let the aperiodic job J_1^b represent the migration activity, and M_1 its correspondent execution time.

Applying TBS

Due to conditions already introduced in Section 6.4.2, the deadline $\hat{d}_1$ assigned to the incoming job J_1^b at time $a_{1,k+1}$ need to be smaller than the deadline $d_{1,k+1}$ of service s_i. Furthermore, the maximum value assigned to this deadline must be $d_{1,k+1} - E_1^h$, which assures that the deadline of the $(k+1)$th instance of service s_1 will be respected.

Instead of letting job J_1^b be completely executed by TBS, let M_1 be split into two parts, ρM_1 and $(1-\rho)M_1$, where $0 \leq \rho \leq 1$, so that each part is carried out by a different server. Letting $\hat{d}_{s1}$ be the deadline assigned by TBS when scheduling $(1-\rho)M_1$, and respecting the conditions explained above, it follows that

$$\hat{d}_{s1} = a_{1,k+1} + \frac{(1-\rho)M_1}{U_s} \leq d_{1,k+1} - E_1^h, \tag{D.3}$$

which can be rewriting to solve U_s. Thus, knowing additionally that $d_{1,k+1} = a_{1,k+1} + P_1$:

$$U_s \geq \frac{(1-\rho)M_1}{P_1 - E_1^h}. \tag{D.4}$$

Similarly, let $\hat{d}_{s2}$ be the deadline assigned by *second server* when scheduling the remaining part ρM_1:

$$\hat{d}_{s2} = a_{1,k+1} + \frac{\rho M_1}{U_1} \leq d_{1,k+1} - E_1^h. \tag{D.5}$$

Solving Equation D.5 to ρ, it follows that

$$\rho \leq \frac{U_1}{M_1}\left(P_1 - E_1^h\right). \tag{D.6}$$

Now, lets *second server* carry out the maximum amount of M_1. For this case ρ must be the maximum:

$$\rho = \frac{U_1}{M_1}\left(P_1 - E_1^h\right). \tag{D.7}$$

By making now a simple substitution of Equation D.7 in Equation D.8, it follows that

$$U_s \geq \frac{M_1}{P_1 - E_1^h} - U_1, \tag{D.8}$$

which is essentially the same as the Equation 6.9 derived in Section 6.4.3.

References

[1] MASSA, A.; BARR, M. **Programming Embedded Systems**. 1.ed. Sebastopol, CA, USA: O'Reilly, 2006.

[2] MARWEDEL, P. **Embedded System Design**. 1.ed. Boston, Dordrecht, London: Kluwer Academic Publishers, 2003.

[3] JERRAYA, A. A. Long Term Trends for Embedded System Design. In: EUROMICRO SYSTEMS ON DIGITAL SYSTEM DESIGN (DSD), 7., 2004, Rennes, France. **Proceedings...** Washington: IEEE Computer Society, 2004. p.20–26.

[4] ENGEL, F. *et al*. Operating Systems on SoCs: a good idea? In: EMBEDDED REAL-TIME SYSTEMS IMPLEMENTATION (ERTSI) WORKSHOP, 2004, Lisbon, Porgutal. **Proceedings...** [S.l.: s.n.], 2004.

[5] GÖTZ, M. Dynamic Hardware-Software Codesign of a Reconfigurable Real-Time Operating System. In: CONFERENCE ON RECONFIGURABLE COMPUTING AND FPGAS (RECONFIG), 1., 2004, Colima, Mexico. **Proceedings...** Mexico: Mexican Society of Computer Science, 2004. p.330–339.

[6] GÖTZ, M.; RETTBERG, A.; PEREIRA, C. E. Towards Run-time Partitioning of a Real Time Operating System for Reconfigurable Systems on Chip. In: INTERNATIONAL EMBEDDED SYSTEMS SYMPOSIUM (IESS), 1., 2005, Manaus, Brazil. **Proceedings...** [S.l.: s.n.], 2005. p.255–266.

[7] GÖTZ, M.; RETTBERG, A.; PEREIRA, C. E. A Run-time Partitioning Algorithm for RTOS on Reconfigurable Hardware. In: INTERNATIONAL CONFERENCE IN EMBEDDED AND UBIQUITOUS COMPUTING (EUC), 2005, Nagasaki, Japan. **Proceedings...** Berlin: Springer, 2005. p.469–478.

[8] GÖTZ, M.; RETTBERG, A.; PEREIRA, C. E. Run-Time Reconfigurable Real-Time Operting System For Hybrid Execution Platforms. In: IFAC SYMPOSIUM ON INFORMATION CONTROL PROBLEMS IN MANUFACTURING (INCOM), 12., 2006, Saint-Etienne, France. **Proceedings...** Oxford: Elsevier, 2006. v.I - Information Systems, p.81–86.

[9] GÖTZ, M.; RETTBERG, A.; PEREIRA, C. E. Communication-aware Component Allocation Algorithm for a Hybrid Architecture. In: IFIP WORKING CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS (DIPES), 5., 2006, Braga, Porgutal. **Proceedings...** Boston: Springer, 2006. p.175–184.

[10] GÖTZ, M.; DITTMANN, F. Scheduling Reconfiguration Activities of Run-time Reconfigurable RTOS Using an Aperiodic Task Server. In: WORKSHOP ON APPLIED RECONFIGURABLE COMPUTING (ARC), 2., 2006, Delft, The Netherlands. **Proceedings...** Berlin: Springer, 2006. p.255–261.

[11] GÖTZ, M.; DITTMANN, F.; PEREIRA, C. E. Deterministic Mechanism for Run-Time Reconfiguration Activities in an RTOS. In: INTERNATIONAL IEEE CONFERENCE ON INDUSTRIAL INFORMATICS (INDIN), 4., 2006, Singapore. **Proceedings...** Washington: IEEE Computer Society, 2006. p.693–698.

[12] GÖTZ, M.; XIE, T.; DITTMANN, F. Dynamic Relocation of Hybrid Tasks: a complete design flow. In: INTERNATIONAL WORKSHOP ON RECONFIGURABLE COMMUNICATION-CENTRIC SYSTEMS-ON-CHIP (RECOSOC), 3., 2007, Montpellier, France. **Proceedings...** Montpellier: University of Montpellier II, 2007. p.31–38.

[13] XIE, T. **A Programming Framework Enabling Runtime Task Migrations Between CPU and FPGA**. 2007. Master Thesis (Master of Computer Science) — Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany.

[14] FINKE, S. **Schnittstellensynthese für HW/SW-migrierbare Dienste eines Realzeitbetriebssystems**. 2006. Bachelor Work (Bachelor of Computer Science) — Heinz Nixdorf Institut, Universität Paderborn, Paderborn, Germany.

[15] RAMMIG, F. J. *et al*. Real-time Operating Systems for Self-coordinating Embedded Systems. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT AND COMPONENT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING (ISORC), 9., 2006, Gyeongju, Korea. **Proceedings...** Washington: IEEE Computer Society, 2006. p.382–392.

[16] RAMMIG, F. J. *et al*. Real-time Operating Systems for Self-coordinating Embedded Systems. In: DAGSTUHL-WORKSHOP MBEES: MODELLBASIERTE ENTWICKLUNG EINGEBETTETER SYSTEME II, 2006, Wadern, Germany. **Tagungsband...** Germany: TU Braunschweig, 2006. p.95–104.

[17] GÖTZ, M.; DITTMANN, F. Reconfigurable Microkernel-based RTOS: mechanisms and methods for run-time reconfiguration. In: INTERNATIONAL CONFERENCE ON RECONFIGURABLE COMPUTING AND FPGAS (RECONFIG), 3., 2006, San Luis Potosi. **Proceedings...** Washington: IEEE Computer Society, 2006. p.12–19.

[18] GÖTZ, M. *et al*. Run-time Reconfigurable RTOS for Reconfigurable Systems-on-Chip. **Journal of Embedded Computing**, [S.l.], 2007. To Appear.

[19] IHMOR, S. **Modelling and Automated Synthesis of Reconfigurable Interfaces**. 2006. PhD Thesis (Dr. rer. nat.) — Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany.

[20] GOCHMAN, S. *et al*. Introduction to Intel Core Duo Processor Architecture. **Intel Technology**, [S.l.], v.10, n.2, p.89–97, May 2006.

[21] GPGPU.ORG. **General-Purpose computation on GPUs (GPGPU)**. Available at <http://www.gpgpu.org>. Accessed in: jan 2007.

[22] NVIDIA. **G8 Graphics Devices**. Avaliable at <http://www.nvidia.com>. Accessed in: nov 2006.

[23] KAHLE, J. A. *et al*. Introduction to the Cell Multiprocessor. **IBM Journal of Research and Development**, Riverton, NJ, USA, v.49, n.4/5, p.589–604, 2005.

[24] ERNST, R. Codesign of Embedded Systems: status and trends. **IEEE Design and Test of Computers**, [S.l.].

[25] BUCHENRIEDER, K. **Hardware/Software Codesign an Annotated Bibliography**. 1.ed. Chicago, USA: IT Press Hartenstein, 1994.

[26] HARDT, W. **HW/SW-Codesign auf Basis von C-Programmen unter Performanz-Gesichtspukten**. 1996. PhD Thesis (Dr. rer. nat.) — Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany.

[27] HARDT, W.; CAMPOSANO, R. Specification Analysis for HW/SW-Partitioning. In: GI/ITG WORKSHOP IN ANWENDUNG FORMALER METHODEN FüR DEN HARDWARE-ENTWURF, 3., 1994, Passau, Germany. **Proceedings...** Aachen: Shaker Verlag, 1994. p.1–10.

[28] KAMDEM, R.; NJIWOUA, P. Galois Lattice Approach to Hardware/Software Partitioning. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS (PDPTA), 1999, Las Vegas, USA. **Proceedings...** Nevada: CSREA Press, 1999. p.3029–3036.

[29] BALARIN, F. *et al*. **Hardware-software co-design of embedded systems: the polis approach**. 1.ed. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[30] JERRAYA, A. A.; WOLF, W. Hardware/Software Interface Codesign for Embedded Systems. **Computer**, Los Alamitos, CA, USA, v.38, n.2, p.63–69, 2005.

[31] COMPTON, K.; HAUCK, S. Reconfigurable Computing: a survey of systems and software. **ACM Computing Surveys**, New York, NY, USA, v.34, n.2, p.171–210, 2002.

[32] LACH, J.; BAZARGAN, K. Editorial: special issue on dynamically adaptable embedded systems. **ACM Transactions on Embedded Computing Systems**, New York, NY, USA, v.3, n.2, p.233–236, 2004.

[33] GARCIA, P. *et al*. An Overview of Reconfigurable Hardware in Embedded Systems. **EURASIP Journal on Embedded Systems**, New York, NY, United States, v.2006, p.1–19, 2006.

[34] PLESSL, C.; PLATZNER, M. Virtualization of Hardware - Introduction and Survey. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS (ERSA), 4., 2004, Las Vegas, USA. **Proceedings...** Nevada: CSREA Press, 2004. p.63–69.

[35] BOBDA, C. **Synthesis of Dataflow Graphs for Reconfigurable Systems using Temporal Partitioning and Temporal Placement**. 2003. PhD Thesis (Dr. rer. nat.) — Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany.

[36] VOROS, N. S.; MASSELOS, K. **System Level Design of Reconfigurable Systems-on-Chip**. 1.ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[37] ADRIATIC Project IST-2000-30049 Deliverable D2.2. **Definition of ADRIATIC High-Level Hardware/Software Co-Design Methodology for Reconfigurable SoCs**. Available at: <http://www.imec.be/adriatic>. Accessed in: aug 2006.

[38] QU, Y.; SOININEN, J.-P. SystemC-based Design Methodology for Reconfigurable System-on-Chip. In: EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN (DSD), 8., 2005, Porto, Portugal. **Proceedings...** Washington: IEEE Computer Society, 2005. p.364–371.

[39] MAXFIELD, C. **The Design Warrior's Guide to FPGAs**. 1.ed. Burlington, MA, USA: Newnes, 2004.

[40] Virtex-II Pro and Virtex-II Pro X Platform FPGAs: complete data sheet (v.4.5). San Jose, CA, USA: Xilinx Inc., 2005.

[41] XAPP529 (v1.3): connecting customized ip to the microblaze soft processor using the fast simplex link (fsl) channel. San Jose, CA, USA: Xilinx Inc., 2004.

[42] ATMEL. **FPSLIC (AVR with FPGA)**. Available at: <http://www.atmel.com/products/FPSLIC/>. Accessed in: dec 2006.

[43] XAPP290 (v1.2): two flows for partial reconfiguration: module based or difference based. San Jose, CA, USA: Xilinx Inc., 2004.

[44] Early Access Partial Reconfiguration User Guide (v1.1). San Jose, CA, USA: Xilinx Inc., 2006.

[45] LYSAGHT, P. *et al*. Enhanced Architectures, Design Methodologies and CAD Tools for Dynamic Reconfiguration on XILINX FPGAS. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 16., 2006, Madrid, Spain. **Proceedings...** Washington: IEEE Computer Society, 2006. p.1–6.

[46] DORAIRAJ, N.; SHIFLET, E.; GOOSMAN, M. PlanAhead Software as a Platform for Partial Reconfiguration. **Xcell Journal**, San Jose, CA, USA, p.68–71, February 2005.

[47] UHM, M. Software-Defined Radio: the new architectural paradigm - reduce system power and cost with a shared resources soc. **DSP magazine**, San Jose, CA, USA, v.1, n.1, p.40–42, October 2005.

[48] DENYS, G.; PIESSENS, F.; MATTHIJS, F. A Survey of Customizability in Operating Systems Research. **ACM Computing Surveys**, New York, NY, USA, v.34, n.4, p.450–468, 2002.

[49] MASSA, A. J. **Embedded Software Development with eCos**. 1.ed. Upper Saddle River, NJ, USA: Prentice Hall, 2002.

[50] REDHAT. **eCos Operating System**. Available at: <http:// sources.redhat.com/ecos/>. Accessed in: jun 2006.

[51] DITZE, C. DReaMS - Concepts of a Distributed Real-Time Management System. In: IFIP / IFAC WORKSHOP ON REAL-TIME PROGRAMMING (WRTP), 20., 1995, Fort Lauderdale, Florida. **Proceedings...** The Netherlands: Elsevier, 1995.

[52] DITZE, C. A Customizable Library to Support Software Synthesis for Embedded Applications and Micro-kernel Systems. In: ACM SIGOPS EUROPEAN WORKSHOP ON SUPPORT FOR COMPOSING DISTRIBUTED APPLICATIONS, 8., 1998, Sintra, Portugal. **Proceedings...** New York: ACM Press, 1998. p.88–95.

[53] DITZE, C. A Step towards Operating System Synthesis. In: ANNUAL AUSTRALIAN CONFERENCE ON PARALLEL AND REAL-TIME SYSTEMS (PART), 5., 1998, Adelaide, Australia. **Proceedings...** USA: IEEE, 1998.

[54] DITZE, C. **Towards Operating System Synthesis**. 1999. PhD Thesis (Dr. rer. nat.) — Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany.

[55] BÖKE, C. Combining Two Customization Approaches: extending the customization tool terecs for software synthesis of real-time execution platforms. In: WORKSHOP ON ARCHITECTURES OF EMBEDDED SYSTEMS (AES), 2000, Karlsruhe, Germany. **Proceedings...** [S.l.: s.n.], 2000.

[56] BÖKE, C. **Automatic Configuration of Real-Time Operating Systems and Real-Time Communication Systems for Distributed Embedded Applications**. 2003. PhD Thesis (Dr. rer. nat.) — Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany.

[57] CAMPBELL, R. H. *et al*. Designing and Implementing Choices: an object-oriented system in c++. **Communications of the ACM**, New York, NY, USA, v.36, n.9, p.117–126, 1993.

[58] ENGLER, D. R.; KAASHOEK, M. F.; O'TOOLE, J. Exokernel: an operating system architecture for application-level resource management. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP), 15., 1995, Copper Mountain Resort, USA. **Proceedings...** New York: ACM Press, 1995. p.251–266.

[59] KAASHOEK, M. F. *et al*. Application Performance and Flexibility on Exokernel Systems. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP), 15., 1997, Saint Malo, France. **Proceedings...** New York: ACM Press, 1997. p.52–65.

[60] STALLINGS, W. **Operating Systems: internals and design principles**. 4.ed. Upper Saddle River, NJ, USA: Prentice-Hall Int., 2001.

[61] VEITCH, A. C.; HUTCHINSON, N. C. Kea - A Dynamically Extensible and Configurable Operating System Kernel. In: INTERNATIONAL CONFERENCE ON CONFIGURABLE DISTRIBUTED SYSTEMS - ICCDS, 3., 1996, Annapolis, USA. **Proceedings...** Washington: IEEE Computer Society, 1996. p.236–242.

[62] VEITCH, A. C. **A Dynamically Reconfigurable and Extensible Operating System**. 1998. PhD Thesis (Doctor of Philosophy) — University of British Columbia, Vancouver, Canada.

[63] HELANDER, J.; FORIN, A. MMLite: a highly componentized system architecture. In: ACM SIGOPS EUROPEAN WORKSHOP ON SUPPORT FOR COMPOSING DISTRIBUTED APPLICATIONS, 8., 1998, Sintra, Portugal. **Proceedings...** New York: ACM Press, 1998. p.96–103.

[64] GABBER, E. *et al*. The Pebble Component-Based Operating System. In: USENIX ANNUAL TECHNICAL CONFERENCE, 24., 1999, Monterey, USA. **Proceedings...** Berkley: USENIX Association, 1999. p.267–282.

[65] MAGOUTIS, K. *et al*. Building Appliances Out of Components Using Pebble. In: ACM SIGOPS EUROPEAN WORKSHOP, 9., 2000, Kolding, Denmark. **Proceedings...** New York: ACM Press, 2000. p.211–216.

[66] BERSHAD, B. N. *et al*. Extensibility Safety and Performance in the SPIN Operating System. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP), 15., 1995, Copper Mountain, USA. **Proceedings...** New York: ACM Press, 1995. p.267–283.

[67] PARDYAK, P.; BERSHAD, B. N. Dynamic Binding for an Extensible System. In: USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION (OSDI), 2., 1996, Seattle, USA. **Proceedings...** New York: ACM Press, 1996. p.201–212.

[68] SOULES, C. A. N. *et al*. System Support for Online Reconfiguration. In: USENIX ANNUAL TECHNICAL CONFERENCE, 28., 2003, San Antonio, USA. **Proceedings...** Berkley: USENIX Association, 2003. p.141–154.

[69] APPAVOO, J. *et al*. Experience with K42, an Open-Source, Linux-Compatible, Scalable Operating-System Kernel. **IBM Systems Journal**, Riverton, NJ, USA, v.44, n.2, p.427–440, 2005.

[70] BAUMANN, A.; APPAVOO, J. Improving Dynamic Update for Operating Systems. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP), 20., 2005, Brighton, UK. **Proceedings...** New York: ACM Press, 2005. p.1–11.

[71] FASSINO, J.-P. *et al*. THINK: a software framework for component-based operating system kernels. In: USENIX ANNUAL TECHNICAL CONFERENCE, 27., 2002, Berkeley, USA. **Proceedings...** Berkley: USENIX Association, 2002. p.73–86.

[72] POLAKOVIC, J. **Dynamic Reconfiguration in THINK: design and implementation.** 2004. Diplomarbeit (Diplom-Informatik) — Universität Karlsruhe, Karlsruhe, Germany.

[73] COWAN, C. *et al*. Fast Concurrent Dynamic Linking for an Adaptive Operating System. In: INTERNATIONAL CONFERENCE ON CONFIGURABLE DISTRIBUTED SYSTEMS (ICCDS), 3., 1996, Annapolis, USA. **Proceedings...** Washington: IEEE Computer Society, 1996. p.108–115.

[74] COWAN, C. *et al*. Specialization Classes: an object framework for specialization. In: INTERNATIONAL WORKSHOP ON OBJECT ORIENTATION IN OPERATING SYSTEMS (IWOOOS), 5., 1996, Seattle, USA. **Proceedings...** Washington: IEEE Computer Society, 1996. p.72–77.

[75] PU, C. *et al*. Optimistic Incremental Specialization: streamlining a commercial operating system. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 15., 1995, Copper Mountain Resort, USA. **Proceedings...** New York: ACM Press, 1995. p.314–321.

[76] MASSALIN, H. **Synthesis: an efficient implementation of fundamental operating system services**. 1992. PhD Thesis (Doctor of Philosophy) — Columbia University, New York, NY, USA.

[77] SELTZER, M. I. *et al*. Dealing with Disaster: surviving misbehaved kernel extensions. In: USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION (OSDI), 2., 1996, Seattle, Washington, United States. **Proceedings...** New York: ACM Press, 1996. p.213–227.

[78] SELTZER, M.; SMALL, C. Self-Monitoring and Self-Adapting Operating Systems. In: WORKSHOP ON HOT TOPICS IN OPERATING SYSTEMS (HOTOS-VI), 6., 1997, Cape Cod, USA. **Proceedings...** Washington: IEEE Computer Society, 1997. p.124–129.

[79] OBERTHÜR, S.; BÖKE, C.; GRIESE, B. Dynamic Online Reconfiguration for Customizable and Self-Optimizing Operating Systems. In: ACM INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE (EMSOFT), 5., 2005, Jersey City, USA. **Proceedings...** New York: ACM Press, 2005. p.335–338.

[80] BOEKE, C.; OBERTHUER, S. Flexible Resource Management - A Framework for Self-Optimizing Real-Time Systems. In: CONFERENCE ON DISTRIBUTED AND PARALLEL EMBEDDED SYSTEMS (DIPES), 18., 2004, Toulouse, France. **Proceedings...** The Netherlands: Kluwer Academic Publishers, 2004. p.177–186.

[81] HILDEBRAND, D. An Architectural Overview of QNX. In: WORKSHOP ON MICRO-KERNELS AND OTHER KERNEL ARCHITECTURES, 1., 1992, Berkeley, USA. **Proceedings...** Berkley: USENIX Association, 1992. p.113–126.

[82] WIND RIVER. **VxWorks**. Available at <http://www.windriver.com>. Accessed in: jun 2006.

[83] CARD, R.; DUMAS, È.; MÈVEL, F. **The Linux Kernel Book**. 1.ed. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[84] WILLIAMS, J. W.; BERGMANN, N. Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-on-Chip. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS (ERSA), 4., 2004, Las Vegas, USA. **Proceedings...** USA: CSREA Press, 2004. p.163–169.

[85] FRIEDRICH, L. F. *et al*. A Survey of Configurable, Component-Based Operating Systems for Embedded Applications. **IEEE Micro**, Los Alamitos, CA, USA, v.21, n.3, p.54–68, 2001.

[86] TOURNIER, J.-C. **A Survey of Configurable Operating Systems**. Albuquerque, NM, USA: University of New Mexico, 2005.

[87] BLODGET, B. *et al*. A Self-Reconfiguring Platform. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 13., 2003, Lisbon, Portugal. **Proceedings...** Berlin: Springer, 2003. p.565–574.

[88] DONLIN, A. *et al*. A Virtual File System for Dynamically Reconfigurable FPGAs. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 14., 2004, Leuven, Belgium. **Proceedings...** Berlin: Springer, 2004. p.1127–1129.

[89] BLODGET, B.; MCMILLAN, S.; LYSAGHT, P. A Lightweight Approach for Embedded Reconfiguration of FPGAs. In: DESIGN, AUTOMATION AND TEST IN EUROPE (DATE), 2003, Munich, Germany. **Proceedings...** Washington: IEEE Computer Society, 2003. p.10399–10401.

[90] VULETIĆ, M. *et al*. Operating System Support for Interface Virtualisation of Reconfigurable Coprocessors. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE (DATE), 2004, Paris, France. **Proceedings...** Washington: IEEE Computer Society, 2004. p.748–749.

[91] VULETIĆ, M.; POZZI, L.; IENNE, P. Virtual Memory Window for Application-Specific Reconfigurable Coprocessors. In: DESIGN AUTOMATION CONFERENCE (DAC), 41., 2004, San Diego, USA. **Proceedings...** New York: ACM Press, 2004. p.948–953.

[92] VULETIĆ, M.; POZZI, L.; IENNE, P. Programming Transparency and Portable Hardware Interfacing: towards general-purpose reconfigurable computing. In: IEEE INTERNATIONAL CONFERENCE ON APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS (ASAP), 15., 2004, Galveston, USA. **Proceedings...** Washington: IEEE Computer Society, 2004. p.339–351.

[93] VULETIĆ, M.; POZZI, L.; IENNE, P. Seamless Hardware-Software Integration in Reconfigurable Computing Systems. **IEEE Design & Test of Computers**, Los Alamitos, CA, USA, v.22, n.2, p.102–113, 2005.

[94] VULETIĆ, M.; POZZI, L.; IENNE, P. Dynamic Prefetching in the Virtual Memory Window of Portable Reconfigurable Coprocessors. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 14., 2004, Leuven, Belgium. **Proceedings...** Berlin: Springer, 2004. p.596–605.

[95] BREBNER, G. J. A Virtual Hardware Operating System for the Xilinx XC6200. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 6., 1996, London, UK. **Proceedings...** Berlin: Springer, 1996. p.327–336.

[96] LING, X.-P.; AMANO, H. WASMII: a data driven computer on a virtual hardware. In: IEEE WORKSHOP ON FPGAS FOR CUSTOM COMPUTING MACHINES, 1., 1993, Napa Valley, USA. **Proceedings...** Washington: IEEE Computer Society, 1993. p.33–42.

[97] DIESSEL, O.; WIGLEY, G. **Opportunities for Operating Systems Research in Reconfigurable Computing**. Mawson Lakes, Australia: Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, 1999.

[98] WIGLEY, G.; KEARNEY, D. The Development of an Operating System for Reconfigurable Computing. In: IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE

CUSTOM COMPUTING MACHINES (FCCM), 9., 2001, Napa Valley, USA. **Proceedings...** Washington: IEEE Computer Society, 2001. p.249–250.

[99] VERDIER, F. *et al*. Exploring RTOS Issues with a High-Level Model of a Reconfigurable SoC Platform. In: INTERNATIONAL WORKSHOP ON RECONFIGURABLE COMMUNICATION-CENTRIC SYSTEMS-ON-CHIP (RECOSOC), 1., 2005, Montpellier, France. **Proceedings...** Montpellier: University of Montpellier II, 2005. p.71–78.

[100] SEGARD, A.; VERDIER, F. SOC and RTOS: managing ips and tasks communications. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 14., 2004, Leuven, Belgium. **Proceedings...** Berlin: Springer, 2004. p.710–718.

[101] NOGUERA, J.; BADIA, R. M. Multitasking on Reconfigurable Architectures: microarchitecture support and dynamic scheduling. **ACM Transactions on Embedded Computing Systems**, New York, NY, USA, v.3, n.2, p.385–406, 2004.

[102] FEKETE, S.; KÖHLER, E.; TEICH, J. Optimal FPGA Module Placement with Temporal Precedence Constraints. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE (DATE), 2001, Munich, Germany. **Proceedings...** Washington: IEEE Computer Society, 2001. p.658–667.

[103] TABERO, J. *et al*. Task Placement Heuristic Based on 3D-Adjacency and Look-Ahead in Reconfigurable Systems. In: CONFERENCE ON ASIA SOUTH PACIFIC DESIGN AUTOMATION (ASP-DAC), 2006., 2006, Yokohama, Japan. **Proceedings...** New York: ACM Press, 2006. p.396–401.

[104] MAESTRE, R. *et al*. A Framework for Reconfigurable Computing: task scheduling and context management. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Piscataway, NJ, USA, v.9, n.6, p.858–873, 2001.

[105] BANERJEE, S.; BOZORGZADEH, E.; DUTT, N. Physically-Aware HW-SW Partitioning for Reconfigurable Architectures With Partial Dynamic Reconfiguration. In: DESIGN AUTOMATION CONFERENCE (DAC), 42., 2005, San Diego, USA. **Proceedings...** New York: ACM Press, 2005. p.335–340.

[106] MEI, B.; SCHAUMONT, P.; VERNALDE, S. A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems. In: PRORISC WORKSHOP ON CIRCUITS, SYSTEMS AND SIGNAL PROCESSING, 11., 2000, Veldhoven, Netherlands. **Proceedings...** [S.l.: s.n.], 2000.

[107] WALDER, H.; PLATZNER, M. Non-preemptive Multitasking on FPGAs: task placement and footprint transform. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS (ERSA), 2., 2002, Las Vegas, USA. **Proceedings...** Nevada: CSREA Press, 2002. p.24–30.

[108] PLESSL, C. *et al*. Reconfigurable Hardware in Wearable Computing Nodes. In: INTERNATIONAL SYMPOSIUM ON WEARABLE COMPUTERS (ISWC), 6., 2002, Seattle, Washington. **Proceedings...** Washington: IEEE Computer Society, 2002. p.215–222.

[109] WIGLEY, G.; KEARNEY, D. Research Issues in Operating Systems for Reconfigurable Computing. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS (ERSA), 2., 2002, Las Vegas, USA. **Proceedings...** Nevada: CSREA Press, 2002. p.10–16.

[110] WIGLEY, G.; KEARNEY, D. The First Real Operating System for Reconfigurable Computers. In: AUSTRALASIAN CONFERENCE ON COMPUTER SYSTEMS ARCHITECTURE (ACSAC), 6., 2001, Queensland, Australia. **Proceedings...** Washington: IEEE Computer Society, 2001. p.130–137.

[111] WIGLEY, G. B.; KEARNEY, D. A.; WARREN, D. Introducing ReConfigME: an operating system for reconfigurable computing. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 12., 2002, Montpellier, France. **Proceedings...** Berlin: Springer, 2002. p.687–697.

[112] WALDER, H.; PLATZNER, M. **Reconfigurable Hardware OS Prototype**. Zurich: Computer Engineering and Networks Laboratory, ETH, 2003.

[113] WALDER, H.; PLATZNER, M. Reconfigurable Hardware Operating Systems: from design concepts to realizations. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS (ERSA), 3., 2003, Las Vegas, USA. **Proceedings...** USA: CSREA Press, 2003. p.284–287.

[114] WALDER, H.; PLATZNER, M. A Runtime Environment for Reconfigurable Hardware Operating Systems. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 14., 2004, Leuven, Belgium. **Proceedings...** Berlin: Springer, 2004. p.831–835.

[115] WALDER, H.; PLATZNER, M. Online Scheduling for Block-Partitioned Reconfigurable Devices. In: DESIGN, AUTOMATION AND TEST IN EUROPE (DATE), 2003, Munich, Germany. **Proceedings...** Washington: IEEE Computer Society, 2003. p.10290–10295.

[116] STEIGER, C. *et al*. Online Scheduling and Placement of Real-time Tasks to Paritally Reconfigurable Devices. In: INTERNATIONAL REAL-TIME SYSTEMS SYMPOSIUM (RTSS), 24., 2003, Cancun, Mexico. **Proceedings...** Washington: IEEE Computer Society, 2003. p.224–235.

[117] STEIGER, C.; WALDER, H.; PLATZNER, M. Operating Systems for Reconfigurable Embedded Platforms: online scheduling of real-time tasks. **IEEE Transactions on Computers**, Los Alamitos, CA, USA, v.53, n.11, p.1393–1407, November 2004.

[118] HANDA, M.; VEMURI, R. An Integrated Online Scheduling and Placement Methodology. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 14., 2004, Leuven, Belgium. **Proceedings...** Berlin: Springer, 2004. p.444–453.

[119] MERINO, P.; JACOME, M.; LÓPEZ, J. C. A Methodology for Task Based Partitioning and Scheduling of Dynamically Reconfigurable Systems. In: IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES (FCCM), 6., 1998, Napa Valley, USA. **Proceedings...** Washington: IEEE Computer Society, 1998. p.324–325.

[120] MERINO, P.; LÓPEZ, J. C.; JACOME, M. F. A Hardware Operating System for Dynamic Reconfiguration of FPGAs. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 8., 1998, Tallinn, Estonia. **Proceedings...** Berlin: Springer, 1998. p.431–435.

[121] ULLMANN, M. *et al*. On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 14., 2004, Leuven, Belgium. **Proceedings...** Berlin: Springer, 2004. p.454–463.

[122] DANNE, K.; MUEHLENBERND, R.; PLATZNER, M. Executing Hardware Tasks on Dynamically Reconfigurable Devices under Real-Time Conditions. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 16., 2006, Madrid, Spain. **Proceedings...** Washington: IEEE Computer Society, 2006. p.1–6.

[123] DANNE, K.; PLATZNER, M. An EDF Schedulability Test for Periodic Tasks on Reconfigurable Hardware Devices. In: ACM SIGPLAN/SIGBED CONFERENCE ON LANGUAGES, COMPILERS, AND TOOLS FOR EMBEDDED SYSTEMS (LCTES), 2006, Ottawa, Canada. **Proceedings...** New York: ACM Press, 2006. p.93–102.

[124] SIMMLER, H.; LEVINSON, L.; MÄNNER, R. Multitasking on FPGA Coprocessors. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 10., 2000, Villach, Austria. **Proceedings...** London: Springer, 2000. p.121–130.

[125] LEVINSON, L. *et al*. Preemptive Multitasking on FPGAs. In: IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES (FCCM), 8., 2000, Napa Valley, CA, USA. **Proceedings...** Washington: IEEE Computer Society, 2000. p.301–302.

[126] KALTE, H.; PORRMANN, M. Context Saving and Restoring for Multitasking in Reconfigurable Systems. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS (FPL), 15., 2005, Tampere, Finland. **Proceedings...** Los Alamitos: IEEE, 2005. p.223–228.

[127] BOBDA, C. *et al*. Task Scheduling for Heterogeneous Reconfigurable Computers. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI), 17., 2004, Pernambuco, Brazil. **Proceedings...** New York: ACM Press, 2004. p.22–27.

[128] DITTMANN, F.; GÖTZ, M. Applying Single Processor Algorithms to Schedule Tasks on Reconfigurable Devices Respecting Reconfiguration Times. In: RECONFIGURABLE ARCHITECTURES WORKSHOP (RAW), 13., 2006, Rhodes Island, Greece. **Proceedings...** Washington: IEEE Computer Society, 2006.

[129] DITTMANN, F. Reconfiguration Time Aware Processing on FPGAs. In: DYNAMICALLY RECONFIGURABLE ARCHITECTURES, 2006, Dagstuhl, Germany. **Proceedings...** Schloss Dagstuhl: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), 2006. n.06141. (Dagstuhl Seminar Proceedings).

[130] GHIASI, S.; NAHAPETIAN, A.; SARRAFZADEH, M. An Optimal Algorithm for Minimizing Run-Time Reconfiguration Delay. **ACM Transactions on Embedded Computing Systems**, New York, NY, USA, v.3, n.2, p.237–256, 2004.

[131] HANDA, M.; VEMURI, R. Area Fragmentation in Reconfigurable Operating Systems. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS (ERSA), 4., 2004, Las Vegas, USA. **Proceedings...** Nevada: CSREA Press, 2004. p.77–83.

[132] COMPTON, K. *et al*. Configuration Relocation and Defragmentation for Run-time Reconfigurable Computing. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, Piscataway, NJ, USA, v.10, n.3, p.209–220, 2002.

[133] BURNS, J. *et al*. A Dynamic Reconfiguration Run-time System. In: IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES (FCCM), 5., 1997, Napa Valley, USA. **Proceedings...** Washington: IEEE Computer Society, 1997. p.66–75.

[134] MIGNOLET, J.-Y. *et al*. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip. In: DESIGN, AUTOMATION AND TEST IN EUROPE (DATE), 2003, Munich, Germany. **Proceedings...** Washington: IEEE Computer Society, 2003. p.10986–10993.

[135] NOLLET, V. *et al*. Hierarchical Run-Time Reconfiguration Managed by an Operating System for Reconfigurable Systems. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS (ERSA), 3., 2003, Las Vegas, USA. **Proceedings...** Nevada: CSREA Press, 2003. p.81–87.

[136] NOLLET, V. *et al*. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING (IPDPS), 17., 2003, Nice, France. **Proceedings...** Wasington: IEEE Computer Society, 2003. p.174–180.

[137] PELLIZZONI, R.; CACCAMO, M. Adaptive Allocation of Software and Hardware Real-Time Tasks for FPGA-based Embedded Systems. In: IEEE REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM (RTAS), 12., 2006, San Jose, USA. **Proceedings...** Washington: IEEE Computer Society, 2006. p.208–220.

[138] STANKOVIC, J. A.; RAMAMRITHAM, K. The Spring Kernel: a new paradigm for real-time operating systems. **ACM SIGOPS Operating Systems Review**, New York, NY, USA, v.23, n.3, p.54–71, 1989.

[139] BURLESON, W. *et al*. The Spring Scheduling Co-Processor: a scheduling accelerator. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN (ICCD), 11., 1993, Cambridge, USA. **Proceedings...** Washington: IEEE Computer Society, 1993. p.140–144.

[140] LINDH, L.; STANISCHEWSKI, F. FASTCHART - A Fast Time Deterministic CPU and Hardware Based Real-Time-Kernel. In: EUROMICRO WORKSHOP ON REAL-TIME SYSTEMS, 3., 1991, Paris, France. **Proceedings...** Washington: IEEE Computer Society, 1991. p.36–40.

[141] LINDH, L. FASTHARD - A Fast Time Deterministic HARDware Based Real-Time Kernel. In: EUROMICRO WORKSHOP ON REAL-TIME SYSTEMS, 4., 1992, Athens, Greece. **Proceedings...** Washington: IEEE Computer Society, 1992. p.21–25.

[142] LEE, J. *et al*. A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE (ASP-DAC), 8., 2003, Kitakyushu, Japan. **Proceedings...** New York: ACM Press, 2003. p.683–688.

[143] KOHOUT, P.; GANESH, B.; JACOB, B. Hardware Support for Real-Time Operating Systems. In: IEEE/ACM/IFIP INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, 1., 2003, Newport Beach, USA. **Proceedings...** New York: ACM Press, 2003. p.45–51.

[144] KUACHAROEN, P.; SHALAN, M.; III, V. J. M. A Configurable Hardware Scheduler for Real-Time Systems. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS (ERSA), 3., 2003, Las Vegas, USA. **Proceedings...** USA: CSREA Press, 2003. p.95–101.

[145] LEE, J.; RYU, K.; III, V. J. M. A Framework for Automatic Generation of Configuration Files for a Custom Hardware/Software RTOS. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS (ERSA), 2., 2002, Las Vegas, USA. **Proceedings...** USA: CSREA Press, 2002. p.31–37.

[146] III, V. J. M.; BLOUGH, D. M. A Hardware-Software Real-Time Operating System Framework for SoCs. **IEEE Design and Test of Computers**, Los Alamitos, CA, USA, v.19, n.6, p.44–51, 2002.

[147] ANDREWS, D.; NIEHAUS, D. Architectural Frameworks for MPP Systems on a Chip. In: INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING (IPDPS), 17., 2003, Nice, France. **Proceedings...** Washington: IEEE Computer Society, 2003. p.265.2.

[148] ANDREWS, D. L. *et al*. Programming Models for Hybrid FPGA-CPU Computational Components: a missing link. **IEEE Micro**, Los Alamitos, CA, USA, v.24, n.4, p.42–53, 2004.

[149] JIDIN, R.; ANDREWS, D. L.; NIEHAUS, D. Implementing Multi Threaded System Support for Hybrid FPGA/CPU Computational Components. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS (ERSA), 4., 2004, Las Vegas, USA. **Proceedings...** USA: CSREA Press, 2004. p.116–122.

[150] ANDERSON, E. *et al*. Enabling a Uniform Programming Model Across the Software/Hardware Boundary. In: IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES (FCCM), 14., 2006, Napa Valley, CA, USA. **Proceedings...** Washington: IEEE Computer Society, 2006. p.89–98.

[151] ANDREWS, D. *et al*. hthreads: a hardware/software co-designed multithreaded rtos kernel. In: IEEE INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION, 10., 2005, Catania, Italy. **Proceedings...** Los Alamitos: IEEE, 2005. v.2, p.19–22.

[152] ANDREWS, D. *et al*. The Case for High Level Programming Models for Reconfigurable Computers. In: INTERNATIONAL CONFERENCE ON ENGINEERING OF RECONFIGURABLE SYSTEMS AND ALGORITHMS (ERSA), 6., 2006, Las Vegas, USA. **Proceedings...** USA: CSREA Press, 2006. p.21–32.

[153] KERSTAN, T. **Konzeption und Entwicklung einer konfigurierbaren Mikrokernarchitektur für ein komponentenbasiertes Echtzeitbetriebssystem**. 2006. Diplomarbeit (Diplom-Informatik) — Heinz Nixdorf Institut, Universität Paderborn, Paderborn, Germany.

[154] GRIESE, B.; OBERTHÜR, S.; PORRMANN, M. Component Case Study of a Self-Optimizing RCOS/RTOS System: a reconfigurable network service. In: INTERNATIONAL EMBEDDED SYSTEMS SYMPOSIUM (IESS), 1., 2005, Manaos, Brazil. **Proceedings...** [S.l.: s.n.], 2005. p.267–277.

[155] WANG, S. *et al*. Real-Time Component-Based Systems. In: IEEE REAL-TIME AND EMBEDDED TECHNOLOGY AND APPLICATIONS SYMPOSIUM (RTAS),

11., 2005, San Francisco, CA, USA. **Proceedings...** Washington: IEEE Computer Society, 2005. p.428–437.

[156] BUTTAZZO, G. *et al*. **Soft Real-Time Systems: predictability vs. efficiency**. 1.ed. New York, NY, USA: Springer US, 2005.

[157] BRINKSCHULTE, U.; SCHNEIDER, E.; PICIOROAGA, F. Dynamic Real-time Reconfiguration in Distributed Systems: timing issues and solutions. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING (ISORC), 8., 2005, Seattle, USA. **Proceedings...** Washington: IEEE Computer Society, 2005. p.174–181.

[158] ELES, P.; KUCHCINSKI, K.; PENG, Z. **System Synthesis with VHDL: a transformational approach**. 1.ed. Norwell, MA, USA: Kluwer Academic Publishers, 1998. p.114–119.

[159] BUTTAZZO, G. C. **Hard Real-Time Computing Systems: predictable scheduling algorithms and applications**. 1.ed. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[160] BUTTAZZO, G. C. Rate monotonic vs. EDF: judgment day. **Real-Time Systems**, Norwell, MA, USA, v.29, n.1, p.5–26, 2005.

[161] BRATLEY, P.; FLORIAN, M.; ROBILLARD, P. Scheduling with Earliest Start and Due Date Constraints. **Naval Research Logistics Quarterly**, New York, p.511–519, December 1971.

[162] User Core Templates Reference Guide (v1.4). San Jose, CA, USA: Xilinx Inc., 2003.

[163] KEROMYTIS, A. D. *et al*. Cryptography as an Operating System Service: a case study. **ACM Transactions on Computer Systems**, New York, NY, USA, v.24, n.1, p.1–38, 2006.

[164] SEDCOLE, N. P. **Reconfigurable Platform-Based Design in FPGAs for Video Image Processing**. 2006. PhD Thesis (Doctor of Philosophy) — Department of Electrical and Electronic Engineering, University of London, London, UK.