Development of a dynamic resource allocation controller for partially reconfigurable FPGAs with service guarantee approach

Alexis Duhamel

▶ **To cite this version:**

Alexis Duhamel. Development of a dynamic resource allocation controller for partially reconfigurable FPGAs with service guarantee approach. Electronics. Nantes Université, 2022. English. NNT : 2022NANU4077 . tel-04088507

HAL Id: tel-04088507

https://theses.hal.science/tel-04088507v1

Submitted on 4 May 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Nantes Université

THESE DE DOCTORAT DE

NANTES UNIVERSITE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Électronique*

Par
Alexis DUHAMEL

Development of a dynamic resource allocation controller for partially reconfigurable FPGAs with service guarantee approach

Thèse présentée et soutenue à Nantes Université, le 7 décembre 2022
Unité de recherche : IETR UMR 6164

Rapporteurs avant soutenance :

M. DE LA TORRE Eduardo	Associate Professor, Universidad Politécnica de Madrid
M. TESSIER Russell	Professor, University of Massachussets

Composition du Jury :

Président :	M. VERDIER Francois	Professeur, Université Côte d'Azur
Examinateurs :	M. CHILLET Daniel	Professeur, Université de Rennes 1
	M. DE LA TORRE Eduardo	Associate Professor, Universidad Politécnica de Madrid
	M. TESSIER Russel	Professor, University of Massachussets
Directeur de thèse :	M. PILLEMENT Sébastien	Professeur, Nantes Université
Encadrante :	Mme. KOUKI Wiem	Ingénieure de recherche, Capgemini Engineering, Nantes

Contents

List of Figures

List of Tables

Résumé long

Les Field-Progammable Gate Arrays (FPGAs) reconfigurables dynamiquement présentent une solution prometteuse pour réduire l'empreinte des composants électroniques, leur coût, et leur consommation énergétique. Ce type de système embarqué dit auto-reconfigurable permet le chargement à chaud d'accélérateurs matériels dans des zones dédiées de la matrice FPGA, appelées Régions Reconfigurables (RRs), grâce à la Reconfiguration Dynamique Partielle (RDP). Des architectures FPGAs et des méthodes de gestion d'allocation de ressources dédiées à la RDP ont alors été conçues pour exploiter au mieux cette technique. En particulier, les architectures apportant une couche d'abstraction matérielle sont d'intérêt, car elles simplifient leur utilisation et leur intégration à des systèmes déjà existants.

Les latences introduites par la gestion de ces systèmes, et en particulier des opérations de reconfigurations dynamiques partielles, ne permettent pas d'égaler la performance de plus gros FPGA statiques. Néanmoins, la flexibilité offerte par le chargement à chaud de RRs permet d'exécuter plus de tâches applicatives, sur moins de ressources. Dès lors, la question de la garantie d'exécution de service se pose. La complexité des charges de travail (ou services) à exécuter sur le système se superposant à la complexité de la gestion des RRs justifient le besoin de méthodologies garantissant l'exécution de services.

Ce travail de thèse se concentre sur la gestion sur cible de l'exécution des applications sur systèmes à base de FPGAs reconfigurables dynamiquement avec une approche garantie de service.

Deux principaux verrous technologiques sont identifiés:

- **Moduler la complexité de la charge de travail du système à travers une modélisation de la notion de performance et de service.** En identifiant les paramètres d'une application impactant la qualité de service perçue, il est possible d'influer sur la prise de décision de la gestion d'allocation de ressources. Cette modulation doit permettre de résoudre le problème de l'allocation des ressources afin de satisfaire un niveau de contraintes de latences sur les différentes ressources du système. En exploitant les invariants face aux paramètres de l'application impactant la qualité, il serait aussi possible de réduire le nombre d'opérations de reconfigurations.

- **Sur la base des modèles de niveau de service précédents, aborder le problème de l'allocation dynamique des ressources au temps de l'exécution en garantissant l'exécution du service.** Si les modèles de qualité de service permettent l'identification d'actions sur les paramètres pour réduire la complexité du service à exécuter, l'application de ces décisions reste un problème conséquent. En particulier, il est nécessaire de bien anticiper les différents temps de reconfiguration et de prendre en compte la disponibilité des différentes ressources du systèmes (RRs et CPUs dans le cas de ressources hétérogènes).

  L'aspect garantie de service doit être couvert par un aspect temps réel, i.e. respecter une échéance pour l'exécution de l'application, et d'observer un niveau minimum de qualité du service défini par les concepteurs d'applications. De plus, le temps de décision des algorithmes de gestion d'allocation de ressources doit être pris en compte dans l'échéance de l'application afin que ce mécanisme apparaisse comme transparent pour l'utilisateur.

Pour résoudre la première problématique, nous nous intéressons à la définition de modes d'exécutions basé sur la décomposition en paramètres qualité d'une application. Ces paramètres influent sur la "Qualité d'Éxpérience" (QoE) perçue par l'utilisateur et peut être obtenue par définition mathématique de métriques objectifs (eg: rapport signal à bruit), ou par étude empirique (eg: appréciation de la qualité vidéo). Ainsi, il est possible d'attribuer des scores obtenus par l'étude de cette qualité perçue à plusieurs modes d'exécution d'une même application, définissant par là le mode de qualité dit "optimal", et des modes dégradés.

Puis, un modèle de qualité de l'implémentation de l'application est introduit. L'observation des métriques inhérents à l'exécution d'une application quelconque sur les systèmes auto-reconfigurables à base de FPGA dynamiquement reconfigurables (e.g. taux d'occupation des ressources, consommation énergétique de la puce,) permet de dresser un modèle de préférence dit "Qualité de Service" (QoS) dépendant de l'objectif du système embarqué (sa mission) et du contexte dans lequel il évolue (son environnement).

La multiplication attendue de ces modes d'exécution dû à l'augmentation du nombre de paramètres qualités est adressée à travers de moyens d'observer le rapport entre qualité d'expérience et de service des modes d'exécution. Ainsi, les concepteurs d'applications peuvent au temps de la conception, déterminer les modes d'exécution pertinents en fonction de la mission et de l'environnement du système embarqué. Cette réduction des modes d'exécution étant

particulièrement importante afin de réduire la complexité des décisions à prendre sur cible.

Après avoir défini ces deux modèles de qualité permettant de qualifier et quantifier le niveau de service perçu par l'utilisateur, et de l'implémentation, nous proposons deux méthodologies de gestion d'allocation des ressources:

- **Nous avons dans un premier temps proposé une méthodologie hybride découpée entre temps de la compilation et de l'exécution**. Cette méthodo-logie permet d'exécuter une partie des algorithmes de gestion d'allocation de ressources afin de générer un grand nombre de solutions. Ces solutions correspondent à différentes implémentations des différents modes d'exécution de l'application. Ces solutions ainsi pré-générées et pré-évaluées, la problème de la gestion au temps de l'exécution sur cible est réduit à une sélection d'une solution maximisant les métriques de qualités.

  Un compromis sur le nombre de solutions pré-évaluées à conserver sur cible doit néanmoins être observé. En effet, le temps passé à chercher une solution dans la base de donnée satisfaisant le niveau de contraintes imposé par l'environnement du système embarqué est lié à la taille (en nombre de solutions) de la base de données. Nous adressons ce compromis en introduisant une méthode permettant, lors de la génération, de ne pas évaluer les solutions d'implémentation d'un mode d'exécution ayant peu ou pas de chance de retourner un niveau de qualité de service suffisant. Puis, une fois les solutions générées, nous proposons une méthode permettant de garder le front de Pareto de l'espace de solution et un nombre de solutions voisines du front.

  Une fois sur cible, nous proposons l'utilisation d'une heuristique de recherche rapide qui permet de trouver une solution de la base de solutions satisfaisant les contraintes de l'environnement du système embarqué. Afin de garantir l'exécution du service, cette heuristique améliore incrémentalement la qualité d'expérience afin de réagir au plus vite aux variations de niveaux de contrainte.

  Enfin, l'approche proposée a été vérifiée fonctionnellement sur cible Zynq-7000, puis nous avons construit une simulation utilisant des métriques d'une implémentation d'une application d'encodage H.264 sur cette plateforme ainsi que de métriques issus de l'état de l'art. Les résultats obtenus montrent que cette méthode est capable de garantir l'exécution du service, caractérisé par le respect de l'échéance et par un niveau quantifiée

par le modèle de qualité de service non-nul. En générant des situations de contraintes d'environnement, la simulation montre que notre approche permet de réagir en moins d'une itération et sans couper le service dans 94% des cas. Cette approche est néanmoins moins efficace face aux situations où une ou plusieurs ressources tombent en panne, où elle ne peut garantir l'exécution du service que 62% des cas. L'approche est toutefois capable de maintenir un niveau de qualité d'expérience quantifié allant de 82% à 94% de l'optimal, tout en donnant une réponse sous la milliseconde en moyenne.

- **Dans un second temps, nous avons proposé une approche basée sur le calcul d'ordonnancement des tâches de l'application sur les ressources de la cible au temps de l'exécution**. Comparativement à la première approche, celle-ci ne nécessite pas le stockage d'une base de données de solution, et a l'avantage d'être déterministe. Un autre compromis se présente alors, car l'exécution d'algorithmes d'ordonnancement est plus long, et leurs temps de décisions évoluent avec le nombre de tâches composant l'application.

  Le calcul d'ordonnancement sur cible doit donc être rapide et performant afin d'obtenir une solution permettant l'exécution du service tout en respectant les contraintes. En ce sens, nous avons développé un heuristique d'ordonnancement permettant d'obtenir des solutions au problème d'allocation de ressources. En particulier, cet algorithme prend en compte l'unicité du gestionnaire de reconfiguration et du problème de partage de ressources que cela implique. Ainsi, l'heuristique d'ordonnancement proposé est capable de quérir les bitstreams des modules reconfigurables en avance de phase, pour que la RR assignée à une tâche soit reconfigurée au moment où cette tâche est prête à l'exécution. Cet heuristique est plus performant qu'une approche similaire de l'état de l'art en donnant des solutions dont la durée d'exécution sont plus courtes de 13% environ sur un ensemble de dix applications benchmarks issues de l'état de l'art. De plus, les résultats obtenus sur par génération de graphe de tâches d'applications montrent que bien que la complexité temporelle de évolue quadratiquement avec le nombre de tâches dans l'application, ses performances compensent le temps passé à calculer l'ordonnancement.

  Cet heuristique a été utilisé en temps qu'algorithme d'ordonnancement de la solution orientée qualité proposée. Des simulations ont été effectuées sur le même simulateur que la première approche en ciblant

l'application H.264 benchmark, ainsi qu'une application de contrôle de traction. Les résultats montrent que cette approche est capable de conserver la continuité de service dans 99% des cas en réponse à des contraintes de l'environnement, et jusqu'à 100% dans le cas de RRs ou CPU restreintes.

En résumé, ce travail de thèse répond au problème de garantie de service en prenant l'angle d'une gestion orientée qualité d'expérience et de service. À l'avenir, des approches statistiques (type *scoreboard*) ou d'apprentissage permettrai d'identifier des solutions particulières de la méthode hybrides. Elles correspondraient alors des solutions permettant de répondre à des niveaux de contraintes courants, et pourraient être sélectionnées plus fréquemment par la méthodologie afin de réduire le temps de décision. Enfin, l'optimisation énergétique étant une problématique importante de la littérature, l'impact sur la consommation du système dans sa globalité de l'utilisation de la RPD permettrait d'apporter des réponses sur l'intérêt de cette technique pour la réduction de la consommation. Deux axiomes se font alors écho: utiliser une matrice FPGA plus petite permet de réduire consommation en puissance, et la latence introduite par la gestion de l'allocation de ressource rend les petits FPGA reconfigurables dynamiquement plus lent que des plus gros FPGA statiques pour une même application. La réduction en consommation d'énergie reste alors à prouver.

Acknowledgments

First of all, I would like to express my gratitude toward my supervisor, Professor Sebastien Pillement. He allowed me to pursue this PhD and provided great support throughout this project. I acknowledge his continued assistance and the relevance of his numerous feedbacks have greatly contributed to this work. I would also like to thank my defense committee; professors Daniel Chillet, Eduardo de la Torre, Russel Tessier, and François Verdier, for their efforts and time invested in reviewing my dissertation and attending my defense. I would also like to thank Wiem Kouki, Chabha Hireche, Céline Mayousse and Ahmed Kammoun from Capgemini Engineering's Research & Innovation department for their assistance during those past years.

I express my gratitude toward Sandrine Charlier, Marc Brunet for their administrative and technical assistance, as well as all the members of the IETR laboratory for their engaging discussions. I would like to express heartfelt thanks to all the PhD students and interns with whom I shared a slice of academic life: Safouane Noubir, Quentin Dariol, Guillaume Martin, Antoine Laspeyres, Corentin Darbas, Loreine Makki, May Myat Thu, Tamar Mosiashvili, Mustafa Ibrahim, Reem Ashi, Juliette Pottier, Nolwenn Dreano. Throughout these years, you have all enriched my daily life and made this experience unforgettable.

Finally, I would like to thank Sélèna Marti for keeping faith in my abilities and standing by my side throughout this long journey. I also dedicate this thesis to my parents Valérie and Fabrice Duhamel, and my sisters Clara and Alizée, as an expression of gratitude.

Chapter 1

Introduction

While the number of transistors on chips has effectively doubled for fifty years [1], transistor miniaturization for single-core general-purpose CPUs has slowed down as it became more difficult for the industry. To mitigate these issues and to increase computing performances, a shift has been made toward parallel computing. Using multi-core CPUs or GPUs, applications can be executed faster, but with higher energy consumption and more expensive chips. These are major drawbacks for embedded systems as their batteries have limited energy supply, and the Internet of Things (IoT) tendency shifts toward more computational power on smart devices.

To answer the energy consumption issue, the embedded systems industry has considered Application-Specific Integrated Circuits (ASIC) instead. These chips can be fully customized to meet application and system requirements, achieving the best performance-energy trade-off [2]. However, the high design cost of ASICs can only be compensated by economies of scale. In addition, ASICs cannot execute other workloads than what they have been designed for, which fails to meet the flexibility in usage that embedded systems require.

As an alternative, Field-Programmable Gate Arrays (FPGAs) offer a middle ground between performance, energy efficiency, and flexibility. They have notably been used in the embedded system industry for signal processing [3] and hardware acceleration on satellites [4]. In addition, while they offer high application-specific performance, these chips can be reconfigured to support a broad range of applications, or to mitigate firmware issues with the systems at a distance.

This characteristic can also be exploited to reconfigure only some parts of an FPGA, called Reconfigurable Regions (RRs), through Dynamic Partial Reconfiguration (DPR). Dynamically reconfiguring the FPGA enables the system to change hardware functionality at runtime. This helps reclaim unused parts of an FPGA at a given time in such a way that only what needs to be executed is implemented in the design. Using this technique, designers are virtually capable of executing the same workload on a smaller FPGA chip, in terms of available logic elements. However, this comes at the cost of having to share and manage these regions, which introduces latency. While a fully

static FPGA most likely outperforms dynamically reconfigurable FPGAs due to this latency, DPR can help reduce chip footprint, costs, and power consumption [5].

To benefit from hardware acceleration offered by dynamically reconfigurable FPGAs, there is a need for abstraction layers and software libraries. As a result, new architectural and methodology challenges are introduced to make the most out of dynamically reconfigurable FPGAs. In particular, the savings offered by smaller dynamically reconfigurable FPGAs should not come at the cost of an embedded system becoming unreliable.

Providing accessibility for application designers helps reduce the time to market for embedded systems [6]. To do so, new methodologies aiming to manage dynamically reconfigurable FPGAs should not require deep knowledge of the underlying hardware for the end-users, but still, provide enough control over the architecture.

The environment in which such systems evolve can require reactivity and flexibility, therefore the system must be robust and guarantees the service execution. This means that on top of the targeted applications, additional services can be executed in real-time to react to these events (eg. a drone computing new trajectories to avoid obstacles), or to withstand a system failure (eg: gamma-rays causing errors in satellite applications). In the context of this work, the guarantee of service refers to the execution of a targeted application under a real-time deadline, while maximizing the performance of the application running on the hardware. This is a key factor for industry as it ensures the benefits of using dynamically reconfigurable FPGAs do not come at a loss in reliability, which is critical for embedded systems.

To do so, the dynamic resource allocation problem must be answered to exploit the RRs in a time-multiplexed manner. This allocation must take into account the latencies of DPR operations on the FPGA and must be done at runtime to react to the constraints on the system. In addition, the decision times of this management must be low enough that it doesn't impact negatively the capabilities of the system to withstand the constraints.

1.1 Contributions

To solve the challenges of guaranteeing service execution on dynamically reconfigurable FPGAs, we propose:

- **A quality model for the guarantee of service characterization**: a model of quality is introduced to characterize the level of service of

an application running on the hardware. This model is described at the functional level not to introduce extensive design efforts from the application designers. The introduced model of quality is based on the identification of parameters that impact the perceived quality. These parameters can then be tuned at runtime to increase or lower the compute-intensiveness of the targeted application.

- **Quality-oriented management methodologies**: using our model of quality, we introduce both a hybrid design-time/runtime methodology and a runtime scheduling-based methodology to autonomously manage the runtime of a targeted application in such a way that it ensures the respect of its real-time deadline. These methodologies are capable of upgrading and downgrading the modeled quality of the application to maximize its perceived quality. When doing so, they dynamically allocate resources to execute the applications. The runtime complexity of these methodologies is considered as their decision latency impacts the system's performance.
- **Fast and efficient runtime scheduling**: to increase the adaptability and scalability of the system, we introduce a novel scheduling heuristic to help meet application deadlines. This scheduling heuristic is capable of processing applications described at the functional task level to work with the introduced quality model. Special attention is brought to the management of FPGA dynamic partial reconfiguration management to shorten the obtained schedules.

1.2 Dissertation organization

The remainder of this document is organized as follows:

- **Chapter 2** introduces the architectural challenges introduced by dynamically reconfigurable FPGAs and describes different approaches of interest from the literature. We compare frameworks and operating system libraries for such systems on key features. In addition, we identify gaps in the literature on the management of dynamically reconfigurable FPGAs with a focus on the guarantee of service and application designer accessibility. Finally, we conclude with the adopted methods and hypotheses made in this thesis and introduce our targeted architecture.
- **Chapter 3** introduces our quality model and concept of execution modes of an application. This concept is then applied to a hybrid

design-time/runtime management methodology and illustrated with a H.264 application example. An experimental setup is introduced with emulation of the targeted architecture to introduce constraints on the FPGA and CPU. We then evaluate its performance in terms of perceived quality and guarantee of service, using application duration and time before deadlines as primary metrics.

- **Chapter 4** describes and discusses the scheduling heuristic for dynamic resource allocation, which is the second main contribution of this work. We first introduce its prediction capabilities to shorten schedule makespan (or duration), then how the dynamic reconfiguration operations of the FPGA are taken into account. State-of-the-art techniques for management latency minimization are introduced. Its performance is evaluated and compared with another recent approach from the literature, identified in chapter 2. Finally, the heuristic is integrated into the quality-oriented management methodology to increase the guarantee of service.

- **Chapter 5** summarizes the contributions of this thesis and discusses obtained results. We also identify future works required to improve the flexibility and performance of our approach.

Chapter 2

Background and model definition

Contents

To save logic elements resources on FPGA matrices and optimize energy consumption of the FPGA, Dynamic Partial Reconfiguration (DPR) has been used in the past to modify the behavior of hardware functionalities during runtime, such as for multi-standard software defined radio [3] and for high performance computing [7, 8]. Over the years, multiple works have been conducted to manage the DPR technique to dynamically reallocate the hardware over time, i.e. in a time-multiplexed way as with threads on a CPU. In this manuscript, we define self-reconfigurable systems as Systems on Chip (SoC) embedding a FPGA matrix capable of performing DPR in order to adapt the hardware accelerators.

This chapter first introduces the self-reconfigurable computing paradigm at large and its associated challenges for FPGA architecture design. Existing architectures are introduced, from which we propose our model. A focus is put on functional abstraction layers for hardware resources. Then, we review existing management methodologies and techniques for self-reconfigurable systems with guarantee of service.

2.1 Self-reconfigurable system design

Self-reconfigurable systems rely on the DPR capabilities of their FPGA matrix. To benefit from DPR, Reconfigurable Regions (RRs) must be defined offline when designing the FPGA architecture.

2.1.1 Reconfigurable region definition

The two mainstream competing vendors; AMD Xilinx and Intel Altera, propose FPGAs that are capable of performing DPR.

However, a fundamental difference lies in the way RRs are defined in their matrices. Figure 2.1 introduces two ways to define RRs in a FPGA matrix. The column-based definition takes whole columns in the matrix for partial reconfiguration. This was how RRs were defined when DPR was first introduced.

Since then, region-based definition has been introduced to perform DPR in rectangular regions, defined by the coordinates of the logic elements located at the corners of the defined RRs [9, 10]. Region-based definition is more flexible than column-based as it only defines which logic elements are going to be reconfigured, but requires more control.

The literature has mainly focused on Xilinx technologies as they natively support region-based RR definition, whereas Altera still supports column-

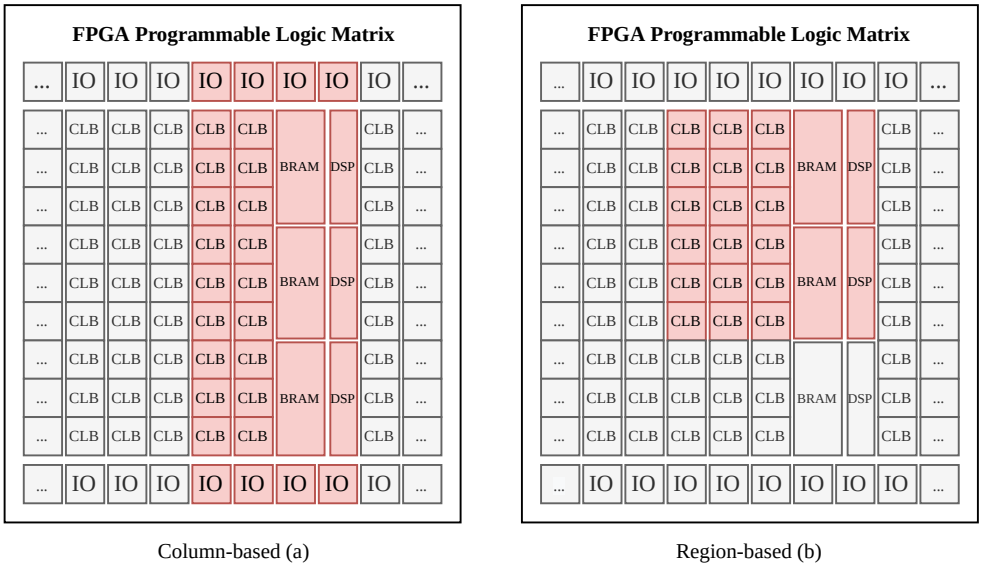


Figure 2.1: Column-based (a) and Region-based (b) Reconfigurable Regions (RRs) definition in FPGA matrices.

based definition. In addition, Xilinx currently offers an on-going support for their DPR design flow [10] and through dedicated software libraries to co-design software and DPR-capable hardware via the Vitis accelerated libraries [11] and the PYNQ project [12].

It has been proven that region-based definition can be used for Altera technologies by masking logic elements that should not be modified [9]. However this adds an additional step when designing with column-based definition.

Therefore in this thesis we will focus on Xilinx technologies and their terminology, yet self-reconfigurable systems using Altera technologies work similarly, this RR definition difference set apart.

To configure the logic elements of a FPGA, a configuration file must be loaded, called bitstream or bit file, in the FPGA's SRAM configuration memory. This read/write memory contains the states of Configurable Logic Blocks (CLBs), Block RAM (BRAMs) and DSP blocks, and the way they are connected to each other to perform a given hardware behavior.

The statically defined RRs can then be filled with hardware accelerators through a bitstream which will configure solely the defined part of the FPGA matrix. Hardware accelerators targeting a specific RR are denoted as Reconfigurable Modules (RMs). After implementation by the toolchain, the obtained configuration file is called partial bitstream, as opposed to the full static bitstream which configures the whole FPGA.

Typically, the configuration bitstreams are stored in an SD card or any read-only memory that's on the board, and are loaded at startup. The CPU of the system is generally used to load the configuration bitstream in the configuration memory through the Configuration Access Port (CAP) of the FPGA. The latter being the only channel with the read/write configuration memory. The Xilinx software development kit makes it possible to perform this operation with the First Stage Boot Loader (FSBL) before loading an Operating System (OS) [13].

Self-reconfigurable systems are still subject to this startup configuration. Whereas the full static bitstream is usually loaded in the configuration memory via the CPU (or JTAG for debugging) once at startup, partial bitstreams can be loaded at runtime via different methods. We'll be referring to any of such methods as DPR management methodology.

2.1.2 DPR management methodology

DPR controllers are responsible to manage the DPR process. Performance of this process, in terms of bandwidth, is crucial for self-reconfigurable systems. In practice, faster reconfiguration operations lower the latency overhead when changing the RM in a RR.

To load the configuration memory with a partial bitstream, two main methods or paths can be used: Processor Configuration Access Port (PCAP) and Internal Configuration Access Port (ICAP). Figure 2.2 illustrates both paths for commonly-used Zynq-7000 device [14].

The PCAP path is the most supported by Xilinx vendor toolchain and devices, notably with the support of dedicated software libraries [12, 15]. When loading a partial bitstream, the CPU sends a request to the Device Configuration (DevC) interface so that its embedded DMA engine fetches the bitstream in the DDR memory. The partial bitstream must have been transferred from local memories to the DDR memory beforehand. The bitstream is sent to the PCAP controller which verifies the integrity and validity of the bitstream. Finally, it is loaded in the configuration memory.

This path provides an easy way to handle DPR operations as it uses the dedicated device configuration interface. This path comes with no additional cost on the designer's side as this interface is already implemented in modern Xilinx SoPCs. However, the provided Python [12] and C [15] libraries suffer from relatively low bandwidth (2.04 and 13.6MBps) compared to other custom PCAP approaches which reach higher bandwidth at 145MBps as in [9]. Such differences can be explained by different DevC interfaces and further library

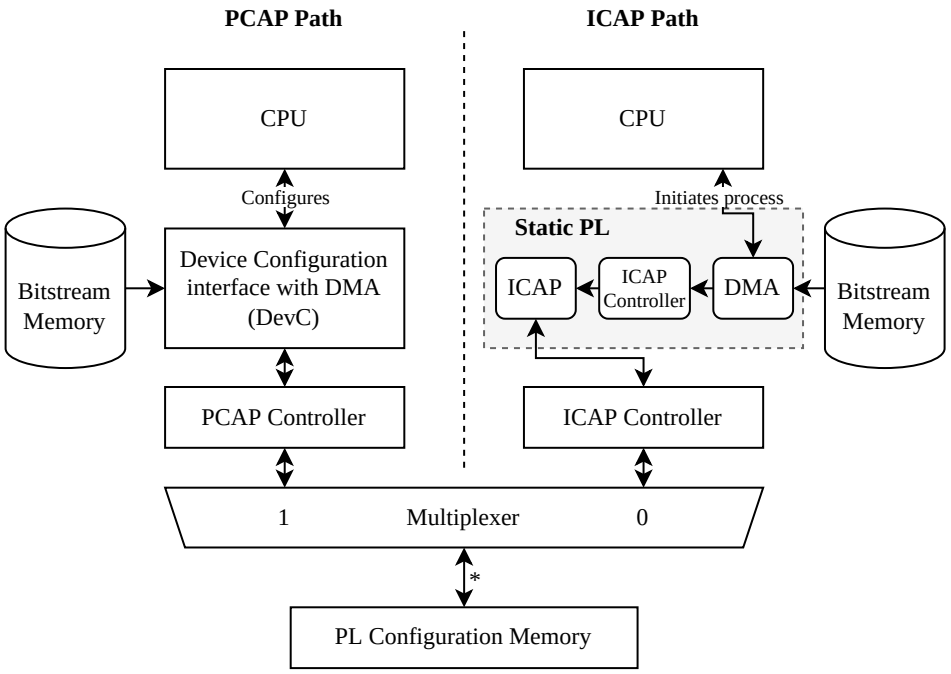


Figure 2.2: PCAP (left) and ICAP (right) paths for partial bitstream flow. The asterisk between the Multiplexer and PL Configuration Memory boxes indicates presence of another multiplexing step for JTAG PL configuration.

optimizations.

The PCAP path also actively uses the CPU as the latter configures the DevC interface DMA engine so the bitstreams are properly fetched from the memory (DDR or SD card).

The ICAP path is the other main method of performing DPR operations. The main advantages of this reconfiguration path is that they reach higher bandwidth, and free the CPU from the DPR operation. However, it comes at the cost of having dedicated logic within the FPGA matrix to perform the DPR operation.

Xilinx's SoCs allow the static programmable logic to access the ICAP resource. In theory, a static design could fetch a bitstream in DDR memory and reconfigure a RR without using the CPU at all. In practice, the CPU, or a soft processor like the Microblaze, is often used to initiate the DPR process through dedicated libraries.

Coupled with a DMA engine in the FPGA programmable logic, Xilinx's hardware ICAP IP (HWIP) can achieve up to 82MBps bandwidth [16, 17]. Further optimizations on the bitstream fetching part in [17, 18] brings the bandwidth up to 382MBps, which is near the maximum theoretic 400MBps

bandwidth for a 32b data bus at 100MHz.

Thanks to custom designs, works from the literature have achieved higher bandwidth than PCAP approaches. Using on the fly lossless bitstream decompression and ICAP overclocking, other approaches [19, 20, 21] were able to reach bandwidths as high as 1.5GBps. Those approaches store compressed bitstreams and decompress them in the FPGA matrix so there are less data to transfer from the memory.

To the best of our knowledge, the highest bandwidth of 2.2GBps has been achieved by [22] with a custom ICAP design overclocked at 550MHz. Although these approaches impose static design restrictions, authors of these works have made efforts to lower the amount of occupied logic elements to 1K LUTs and FFs on average (approximately 2% of LUTs and 1% of FFs in a ZYNQ-7000 XC7Z020 SoC FPGA) which makes it a relatively good trade-off between bandwidth and implementation cost.

Table 2.1 introduces a comparison of DPR controllers sorted by increasing bandwidth. Because self-reconfigurable systems require prior reconfiguration of RRs, higher bandwidth is crucial to minimize the reconfiguration latency overhead as higher bandwidth leads to lower reconfiguration time.

Spent resources for custom ICAP-based controllers takes between 1.8% to 7.52% of LUTs, and 0.38% to 1.42% of a Xilinx Zynq-7000 Artix-7 XC7Z020 FPGA matrix, which is a popular cost-optimized device. However, among all DPR controller approaches, the ICAP-based DPR controllers have a clear advantage over PCAP-based when it comes to bandwidth. ICAP-based approaches also free up the CPU to run OS services and application-related processes.

In our work, we distinguish two categories of ICAP-based DPR controllers: those who do not impose clock frequency constraints and the others. The first category implies a standard 100MHz clock as it is most often used in FPGA designs, which caps the theoretical bandwidth at 400MBps. In this category, the recent ZYCAP DPR controller [18] is the best trade-off between bandwidth and static implementation cost.

Table 2.1: Comparison of DPR controllers, sorted by bandwidth.

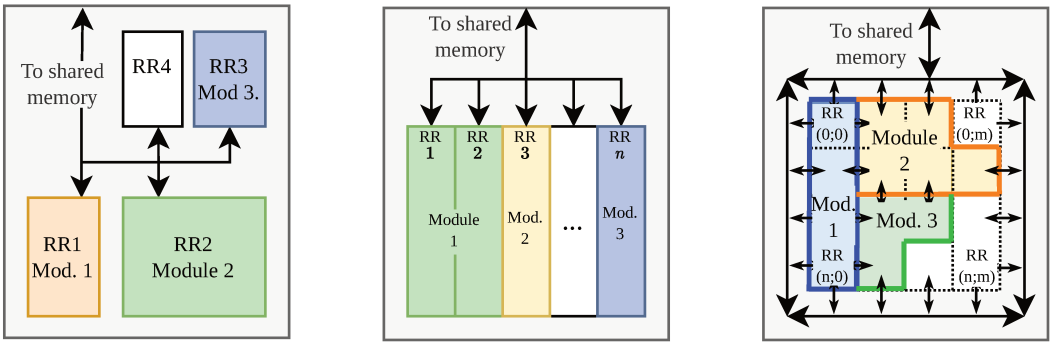
DPR controller	Bandwidth (MBps)	Resources usage (LUTs & FFs)	Frequency (MHz)
PCAP controllers			
PCAP Python Overlay [12]	2.04	-	-
XilFPGA PCAP C Library [15]	13.6	-	-
Custom PCAP bit. transfer [9][1]	145	-	-
ICAP controllers			
Xilinx HWICAP [16]	19	1K LUTs, 0.7K FFs	100
Xilinx HWICAP (DMA) [16]	67	4K LUTs, 6.2K FFs	100
DMA HWICAP [17]	82	4K LUTs, 0.9K FFs	121
BRAM HWICAP [17]	332	1K LUTs, 0.5K FFs	121
ZYCAP [18]	382	1.1K LUTs, 0.8K FFs	100
FaRM [19]	800	3.2K LUTs, 0.4K FFs	200
uPaRC [20]	1'433	4K LUTs, 1.1 FFs	362.5
Pham et al.[2] [21]	1'480	4.4K LUTs, 1.5 FFs	370
Hansen et al.[3] [22]	2'200	-	550

[1] Authors made use of a custom devC interface.
[2] Requires a Microblaze as introduced in [21], but can be modified to use the SoC's ARM CPU instead. Resources given without the Microblaze.
[3] Authors did not disclose the resources usage of the full reconfiguration controller.

The second category considers DPR controllers that overclocks the ICAP resource. Those approaches make implementation more difficult as timing requirements are harder to meet for the toolchain's place and route steps. Still, those approaches outperform the approaches from the first category with bandwidths up to 5 times higher. In this category, we will consider the profiled bandwidth of the uPaRC DPR controller [20].

2.1.3 Architecture design for reconfigurable systems

Self-reconfigurable systems can be classified based on their RR architecture style when designing around DPR. Each architecture style comes with their own pros and cons which can make them more suitable to specific paradigm of hardware accelerators (or RM). Three architectures styles cover the majority of DPR approaches: Island, Slots, and Grid styles, as illustrated in Figure 2.3.



Island-style (a) Slot-style (b) Grid-style (c)

Figure 2.3: The three main RR architecture styles in DPR-capable FPGAs. Black arrows denote communication channels between RRs and the external shared memory.

In this figure, the FPGA matrix is represented by gray squares which denote the static region, and RRs by colored rectangles inside. The footprint difference of the RRs can be noted, as their number and size respectively increases and shrinks from island style to grid style. This footprint difference is referred to as granularity in the literature [23, 24, 25], and ranges from coarse to fine grain. Coarse-grain defines monolithic accelerators which define RRs in the island styles, while medium-grain refers to functional units which corresponds to slot and grid styles [25].

The fine granularity refers to RRs which comprises just a few specific logic elements to fine-tune parts of the design [25]. However, this granularity fits specific designs and do not focus on supporting a wider range of generic hardware accelerators. For this reason, the fine granularity will not be discussed further in this work.

2.1.3.1 Island style

Island style consists in having few coarse-grain RRs fitting monolithic hardware accelerators (eg: vision application filtering, edge detectors...). A number of works have focused on this style of architecture [24, 26, 23], and it can be considered the most commonly used style.

The communication infrastructures used in these architectures can employ crossbar switches, which have the advantage of low congestion latency. However, crossbar switch resource usage scale exponentially with the number of nodes connected to it [27].

Monolithic accelerators imply that all the dedicated logic elements have to fit inside a single RR. This type of architecture is the easiest to design, as users can make use of High Level Synthesis (HLS) vendor toolchain to rapidly build hardware accelerators to populate the RRs.

Applications can then be described at the functional level to obtain a set of hardware accelerators to speed it up, without much knowledge of the underlying hardware except for what comprises the targeted RRs. This architecture style is often considered high level in comparison to the other styles for this reason.

Because the accelerators are functionally different, their synthesis can result very different number of spent logic elements. Hence the biggest RR in the design has to be at least big enough to fit the largest accelerator. When smaller accelerators are to be implemented, if they are placed in a bigger RR, it causes resource wastage (or resource fragmentation), as the rest of the free logic elements in the targeted RR cannot be used by other accelerators. An island style comprising only RRs of similar logic elements count is considered homogeneous.

Resource fragmentation can be reduced by introducing RRs of different size [24], but this introduces some limitations at runtime as to which accelerator can fit which RR. This hypothesis is called heterogeneous RR size. Using those, it is possible that certain RMs do not fit in all RRs and that execution times of RMs differ between RRs. Having RRs of different size can be beneficial to minimize the reconfiguration latency as the size of partial bitstream files scale with the region footprints, and not with the resource usage in the RR [10].

Finally, this style of architecture gets the most support from Xilinx and Intel as their dedicated toolchain official documentation explicitly supports such architectures [10, 28].

2.1.3.2 Slot style

Slot style architectures tackle the resource fragmentation issue from another point of view. They focus on accelerators that can span one or more RRs, while considering a larger number of fixed size RRs called slots [29]. The medium-grain RRs can fit functional units that may not always correlate to high level functional description as would monolithic accelerators.

This style of architecture usually supports RRs with similar size and logic elements content so that accelerators can be implemented in such a way that they occupy a minimum number of slots to minimize resource fragmentation.

Because the number of considered RRs in the FPGA architecture increases compared to island style, slot style greatly benefits from bitstream relocation [30, 31].

Bitstream relocation consists in taking a RM that targets a given RR i, and relocating it to another RR j. This can be achieved by modifying the configuration header of the bitstream files, which gives information on the localization of a RR in the matrix [10]. Runtime bitstream relocation imposes a latency overhead which in the range of tens of microseconds [31] to milliseconds [30].

However it is paramount that the content (in logic elements) of the original bitstream, and its rectangular shape in the matrix, are exactly identical to the RR that the bitstream is relocated to [31].

This imposes a heavy constraint on the FPGA architecture when defining RRs in the matrix. If slots are too big in proportion of the FPGA matrix, finding similar RRs with the exact logic elements content and shape is harder. This can be less constraining for high-end FPGA chips with Xilinx super logic regions [32, 33] and multi-FPGA reconfigurable systems [33, 34] as they have more resources to partition in slots.

2.1.3.3 Grid style

Finally, grid style further reduces the size of individual RRs. Here, the number of RRs increases significantly as RRs are reduced to a few logic elements organized in a systolic array [35, 36, 37]. Because of the very high number of RRs, grid style architectures rely on dedicated framework to generate flexible partial bitstreams and make extensive use of the bitstream relocation technique [38].

Compared to the other approaches, grid style architectures have the highest flexibility when it comes to implement RRs in the FPGA matrix. Thanks to bitstream relocation, RMs spanning multiple RRs can be moved within the RR grid, and arranged to minimize fragmentation [39, 40].

However, grid style architectures have a bigger static architecture overhead compared to other styles [24]. As shown in Figure 2.3 (c), multiple communication gates between the dynamic and the static region must be defined when designing the architecture. Each gate must be connected to a bigger communication infrastructure so that virtually any combination of RR implementation is valid.

While slot and grid style architectures can minimize dynamic resource fragmentation, the implementation costs brought by the different overlays and

frameworks can hinder their adoption by application designers. On the other hand, commercial-off-the-shelf (COTS) accelerators and HLS-built modules can easily be integrated into application designers design flow.

Slot and grid style approaches also require intensive use of runtime bitstream relocation. The latency introduced by bitstream relocation can become problematic to the service execution of real-time applications.

In this work, we believe the island style architectures with coarse-grained RRs of heterogeneous size are a good trade-off between resource fragmentation and ease of use [24].

2.1.4 Frameworks and libraries

To make use of the introduced DPR capable island style architectures, works have been focusing on designing dedicated operating system libraries and frameworks. Their goal is to provide software abstraction of dynamically reconfigured accelerators, so that application designers benefit from software flexibility and the computing performance of hardware accelerators. Researchers have then tackled the challenge of building hardware abstraction layers to manage the architecture's RRs efficiently.

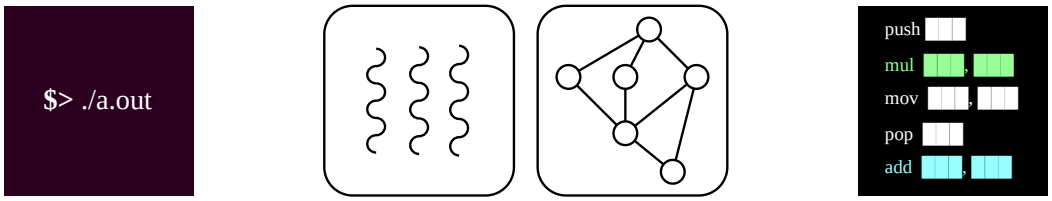
In the following sections, we first introduce the different abstraction levels used in the literature, and then introduce the hardware preemption mechanism for the real-time paradigm on self-reconfigurable systems. We then introduce the features of interest when reviewing the existing frameworks and operating system libraries.

2.1.4.1 Abstraction level

Hardware abstraction consists in providing the user with software routines to manage the self-reconfigurable system's FPGA architecture [41]. Choices made on the FPGA architecture and the targeted functional level of abstraction define different approaches.

As the goal is to provide users a generic methodology to implement their accelerated applications, a level of functional abstraction must be chosen. The level of abstraction must reflect the granularity of the FPGA architecture and its specificities.

We define 3 main levels of functional abstraction as illustrated Figure 2.4. The application level of abstraction makes the targeted FPGA architecture a black box for the user. By similarities with UNIX systems, the accelerated application is considered holistically as a single process that the user can interact with through a dedicated API.



(a) Application level

(b) Thread or task level

(c) Instruction level

Figure 2.4: Functional abstraction levels sorted from highest (left) to lowest (right).

This type of approach was first considered in the early stages of reconfigurable computing [42] with applications comprised of a single monolithic accelerators. It is still employed in multi-tenant deployment of FPGA-based accelerators [7], and can also describe recent frameworks that aim to compile software into functional units for grid-style DPR-capable architectures [43]. Application level abstraction is the easiest to comprehend as generating big monolithic accelerators is relatively easy with HLS languages and require no prior knowledge of the underlying architecture. Because this level of abstraction implies the usage of large RRs to fit big accelerators, it mostly relies on island-style coarse-grain architectures and multi-FPGA systems.

The thread or task level of abstraction, cf. Figure 2.4(b), consider a division of the targeted application into accelerated threads or tasks. The resulting partition of the application can then be represented as a task graph, and further parallel or pipeline execution optimization can be introduced at this step [41]. Following this partition of the application, tasks or threads can be identified as best candidates for acceleration and implemented as hardware accelerators. The frameworks and libraries' goals are then to manage those threads and tasks, with respect of the FPGA DPR-capable architecture [44]. Parts that may not benefit much from hardware accelerator, such as purely sequential code or monitoring tasks, can be executed on the self-reconfigurable system's CPU.

Smaller RRs in the architecture can fit with this abstraction level as the application execution is spread across multiple RRs. High performance of the communication infrastructure in terms of high bandwidth and low latency becomes important to maximize application speedup.

Finally, instruction set level of abstraction achieves very low level accelerators description. Its goal is to accelerate the execution of software by accelerating frequent instructions, for RISC-V CPU architecture [45] as a hot

topic example. This last level of abstraction requires application designers to know which instructions, or low-level software routines equivalent, can be accelerated when writing their software programs.

In our works, we focus on the task level of abstraction. Because it relies on a functional level definition, we believe such an abstraction level is suitable to define which parts of the application need to be accelerated.

Additionally, the application level might cause difficulties for guarantee of service execution as the former can lead to high logic element fragmentation because of the coarse-grain island style architecture, resulting in under performing systems. Instruction level on the opposite can help fine-tuning the targeted application's acceleration, but requires a complex framework which end-users can't really tune at run-time for their own goals.

2.1.4.2 Hardware preemption

In a software real-time operating system, task-level preemption momentarily interrupts the execution of a task to free up a CPU core to execute another task. Preemptive multitasking has been used in conjunction with priority-based scheduling to guarantee the execution of higher priority tasks over lower priority ones. Hardware preemption for a self-reconfigurable system aims to bring this feature to preempt hardware accelerators. This feature is interesting for a self-reconfigurable system with guarantee of service as it is notably used in the industry to increase systems reactivity.

To perform any kind of preemption, the system performs a context switch on the task. This context switch consists in three steps: interrupting and saving the task's progress, executing an other task, restoring the preempted task. In software, the task's progress consists of the states of CPU registers, stack pointer and program counter. In hardware however, progress of a task implemented in a RR consists of its FFs and local BRAMs states, plus any processed and unprocessed data that is in the shared DDR memory or still transiting in the communication infrastructure.

To tackle the challenge of hardware preemption, researchers have come up with two main methodologies: Configuration Access Port (CAP)-based and Task-Specific Access Structures (TSAS).

CAP-based methodology consists in using bitstream read back of the configuration memory. This feature allows to extract information on the logic elements states in the FPGA, and can be applied to RRs as well. Then, the partial bitstream which contains the FFs and local memories states can be saved in a local or external memory to be restored later. This idea dates back

as early as 2000 where authors of [46] designed a hardware context switch using this method.

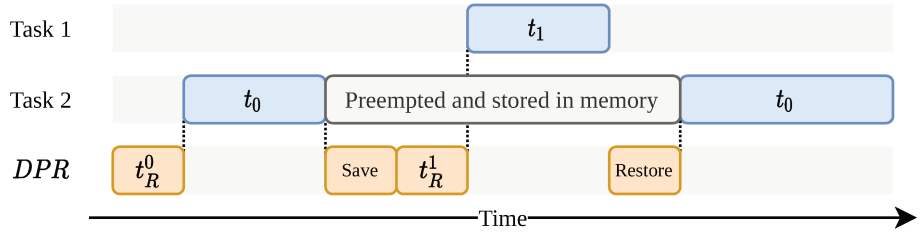


Figure 2.5: CAP-based hardware preemption of Task 1 over Task 0 on a single RR.

Figure 2.5 illustrates CAP-based hardware preemption of a task on a single RR. The *DPR* schedule shows how the context switch operations can impact the unique ICAP resource usage.

Bitstream size grows with the size of the corresponding RRs, and can reach multiple hundreds of kilobytes. Because local BRAM memories are limited within the FPGA, works have been trying to reduce the amount of extracted data by storing only state information of slices that are used in the RR [47, 48].

Finally, authors of [49] have used the CAP method on a recent framework for preemptive multitasking and disclosed the resulting time spent saving (T_{save}) and restoring ($T_{restore}$) during a hardware context switch. Obtained results on RR sizes S_1 and S_2 occupying respectively 6% and 2% of a Virtex-6 FPGA are presented in Table 2.2.

Table 2.2: Context switching results of CAP-based hardware preemption on a Xilinx Virtex-6 ML605 board from [49].

	FFs in RR	LUTs in RR	Bitstream size (kB)	T_{save} (ms)	$T_{restore}$ (ms)
S_1	17k	8.6k	741	16.0	9.7
S_2	5.8k	2.9k	361	10.3	5.5

The approach introduced by authors of [49] get a 30MBps bandwidth for the context switch operation as bitstreams are stored in the shared DDR memory. This bandwidth can be compared with the previously introduced DPR controllers which had up to 382MBps without overclocking the design, even if bandwidth optimizations cannot be applied to both the read and write bandwidth equally [22]. In addition, the saved bitstream can only be restored on the same RR, unless bitstream relocation is involved which also adds latency and require specific architectures.

This highlights the main issue with the CAP-based hardware preemption as it occupies the unique ICAP resource in the FPGA during the context switch operation. It can also be argued that CAP-based methodologies, and their storage size reduction optimization in particular, are technology-dependent. Recently, authors of [50] exposed the bitstream composition and required manipulations to save and restore context on 7-series and Ultrascale-series Xilinx FPGAs using ICAP read back capabilities, showing this approach can still be applied nowadays.

To mitigate the added occupancy on the ICAP resource, researchers have developed Task-Specific Access Structures (TSAS) methodologies [51, 52, 53, 54]. This type of hardware preemption consists in defining one or multiple task checkpoint(s) within the accelerator description. Different ways to create checkpoints have been studied such as using scan-chains copying the states of LUTs inside the RRs [51, 55], storing the states of specific FFs in a memory-mapped memory [51, 54], and using HLS tools to generate checkpoints [53]. Then, when a hardware task is to be preempted, only the content of those checkpoints is saved and restored, and the remaining uncheckpointed FFs and BRAMs states are dropped.

This type of hardware preemption can also support cross-RR preemption as the content of those checkpoints can be made in a way that they're not dependent on a specific RR. However, checkpoints add extra logic inside the RRs. This logic overhead depends on the used method, and recent approaches [53, 54] disclose up to 7% of LUTs and FFs overhead in RRs while achieving millisecond range context switches.

Finally, authors of [56] study the impact of preemption on priority-based real-time scheduling policies for applications comprised of software and hardware tasks. In this study, authors hypothesize an ideal hardware preemption mechanism with no logic nor time overhead. They compare this ideal preemption with non-preemptive scheduling (Block), and two other scheduling policies (Drop and Rollback). The Drop policy erase any progress made by the hardware task and restarts it from the beginning later on the same RR, while the Rollback policy restarts immediately the task in software after dropping the accelerator.

Comparison on AES encryption and Xvid video codec benchmarks show that whether the tasks were short and frequents, or intermittent and long, the ideal hardware preemption schedule is approximately 10% shorter than the non-preemptive scheduling policies. This paper highlights that latency cost of hardware preemption has to be seriously taken into account as they can quickly outweigh any gains in schedule makespan.

As a conclusion, if hardware preemption has been made possible, the inherent overhead in context switch duration, spent logic elements and design considerations hampers its use. Furthermore, works have shown that non-preemptive scheduling policies can reach similar performance without those limitations. For these reasons, we will not consider hardware preemption as a feature.

2.1.4.3 Features of interest

In this section, we define four features of interests to review frameworks and libraries for self-reconfigurable systems. We focus on approaches that target task-level accelerators in coarse-grained RRs of heterogeneous size organized in island style architectures. We define the following features:

- **POSIX-like API**: Presence of a dedicated API to manage hardware accelerators makes for an easier development on the software side [41]. This API should abstract hardware to hide specific architecture specifications from the application designer.
- **RR Point-to-point communication**: Data transfer between hardware accelerators using DMA access is a major bottleneck [24]. Point-to-point (P2P) communication between RRs within the FPGA helps reducing communication latency and decreases shared memory congestion.
- **Standardized RR interfaces**: Generic standard interfaces between the dynamic and the static regions helps designing the communication and control infrastructure. These interfaces need to be the same across all RRs so they are abstracted the same way on the software side through the API. In addition, a standard interface helps designing accelerators as designers only have one interface to study.
- **Runtime resource management**: Computational resources (RRs and CPU) should be managed by the framework, through real-time paradigm, task scheduling or autonomic management. In particular, the system must be resilient to RR unavailability as a result of a failure [4], or additional tasks or services to execute at runtime [57].

2.1.4.4 Related works

Operating System for Reconfigurable Systems (**OS4RS**) [58] was one of the first OS designed for self-reconfigurable systems. It was built with the Real-

Time Application Interface (RTAI) extension of a Linux kernel. The RTAI API was used to manage hardware accelerator as tasks in a standard real-time system, and RTAI's priority-based scheduler was employed as a mean to manage resources. The OS4RS targeted architecture made use of a standard RR interface to send data packets over a Network on Chip (NoC) which was capable to transfer data between RRs within the FPGA matrix. While its performance and used toolchain are now outdated, OS4RS still constitutes an interesting approach given our selected features of interest.

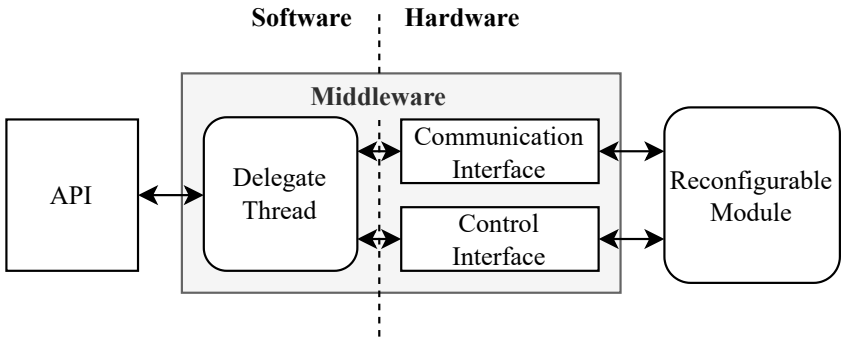


Figure 2.6: Software delegate thread concept enabling data communication and control of a reconfigurable module.

HThreads [59] is to the best of our knowledge the first approach that makes use of a POSIX-like hardware threads model. Hardware threads are abstracted by a software *delegate thread*, illustrated Figure 2.6, that the dedicated API can interact with as if it was a standard software thread. This concept is an interesting hardware abstraction solution as application designers keep a software stub to manage the hardware accelerator, and actions taken on the delegate thread are reflected to the accelerator. HThreads is based on a Real Time OS (RTOS) kernel that makes use of a priority-based scheduler that is able to manage hardware accelerators. However, while this approach provides an interesting take on POSIX-like API with delegate thread, this approach lacks most of the architectural considerations and offer no runtime management.

The **FOSFOR** project (Flexible Operating System FOr Reconfigurable platform) [60, 61] is an extension of RTEMS (Real-Time Executive for Multiprocessor Systems) for self-reconfigurable systems. Similarly to Hthreads, FOSFOR comprises middle ware to abstract hardware communication. This project's goal was to demonstrate the feasibility of abstracting the hardware accelerators for application designers by using RRs as threads.

FOSFOR's middle ware follows the concept of delegate thread introduced

in Figure 2.6. In FOSFOR, the RRs interface between static and dynamic region consists of a communication and control interface. The communication interface is plugged to a flexible NoC called DRAFT [62]. It is a reconfigurable fat-tree topology network for data flow communication, well adapted for RR to RR communication. The control interface interacts with a dedicated finite state machine located inside the RR that is customized to the accelerator.

The software part of FOSFOR's middle ware is comprised of a custom API to create and manage virtual communication channels. They consist of delegate threads that are software images of the DRAFT configuration.

This work has achieved a speedup up of 50× for selected applications, compared to full software RTEMS implementations. However, FOSFOR does not provide any runtime resource management.

FUSE (Front-end USEr framework) [63] is a framework extending the PetaLinux operating system (now one of Xilinx's tools) with a multi-threaded programming model approach. Its RRs, denoted as hardware threads, don't require a dedicated delegate threads as it was the case in FOSFOR. They instead rely on its software POSIX-like API for RR management. As illustrated in Figure 2.7, this management is done in the CPU from the user space via the Top-Level FUSE Component (TLFC), then handled in the kernel space by the Low-Level FUSE Component (LLFC).

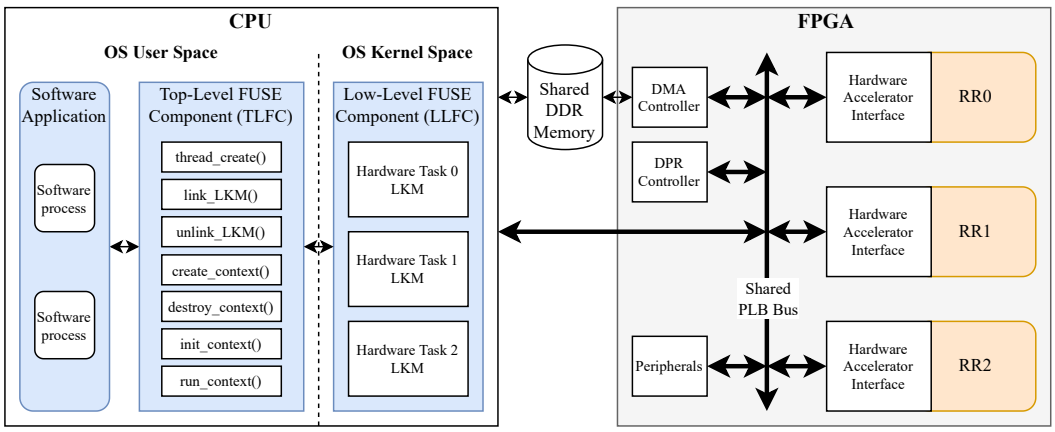


Figure 2.7: Overview of FUSE's targeted architecture [63].

When calling the API from the TLFC to instantiate hardware threads, management of the accelerator is abstracted by the LLFC. The latter is responsible for the management of the RRs and communicates with the FPGA. In the FPGA, each RR has the same hardware accelerator interface with the static region. Data and control communications are differentiated in the interface, and those can either be fully customized or made standard.

To manage API calls, a kernel space Loadable Kernel Module (LKM) is used instead of a user space delegate thread that the user can interact with. Terminating a user space delegate thread unexpectedly, or without care for the state of a RR, can leave an accelerator hanging with unprocessed data or non-empty local memories. Therefore handling such operations in the kernel space and only letting the user interact through the API ensures proper termination of the accelerators.

When managing accelerators, the LLFC autonomously assigns RRs for hardware accelerators via dedicated mapping policies [56]. This includes finding a valid real-time schedule and dynamically reconfiguring the accelerators to the assigned RRs.

However, because the API has been designed in such a way that a software application only is accelerated, each LKM manages only one accelerator. FUSE's API works in a way that data must be sent to an accelerator via the shared DDR memory, and the processed data from the RR must be sent back the LKM in the same way. So the support of RR P2P communication hasn't been studied using FUSE.

ReconOS (Reconfigurable OS for reconfigurable computing) [64, 65] is a long-term project aiming to develop a Linux-based OS and a multi-threaded programming model together for self-reconfigurable systems. The project was first introduced as open source in 2010 and was officially supported until 2017. ReconOS was developed as an open-source framework for self-reconfigurable systems to manage hardware accelerators. To do so, a POSIX-like API was provided with software delegate threads. The targeted architecture of ReconOS is illustrated in Figure 2.8.

In the FPGA, each RR gets connected to a dedicated control interface. It is in charge of passing commands to the hardware thread's operating system finite state machine control which can be customized for each accelerator, similarly to FOSFOR. The control FSM is inside the RR and is dynamically reconfigured with the RR. A second interface, namely the memory interface, allows the RR to access a shared DDR memory through the memory subsystem. It is comprised of a memory management unit (MMU) and an arbiter to queue memory access requests.

In 2018, an implementation of a multi-threaded real-time image-processing application in ReconOS has shown performance improvement compared to its software counterpart [66]. However, ReconOS does not natively make use of point-to-point communication between hardware tasks as SPREAD does. Such communication has to go through the memory subsystem, which increases the communication latency in the hardware part. The authors of [67]

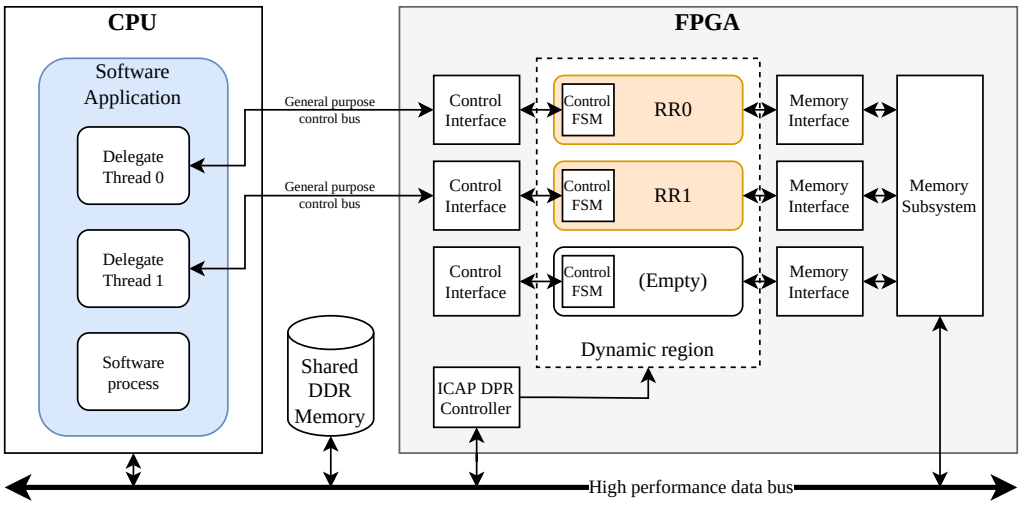


Figure 2.8: Overview of ReconOS's targeted architecture [65].

have compared their RR P2P approach with ReconOS, and have shown that it can increase communication bandwidths by up to 3,14×. This speedup comes at a cost of 1k slices and 360 FFs per communication interfaces, and dedicated OS libraries. The ReconOS project being open source, this optimization can be brought to the architecture. Although ReconOS's runtime management of applications is left to the designers, it offers a rich set of tools through the ReconOS Development Kit (RDK).

More recently, **FOS** (FPGA Operating System) [6] was introduced as an open source architecture-agnostic framework and OS libraries. It has been designed to be modular so that it does not depend on specific custom FPGA architecture considerations.

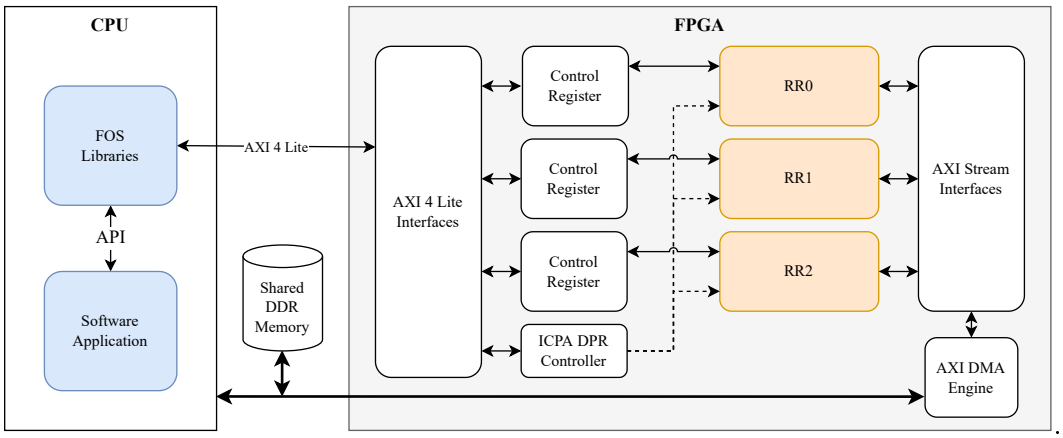


Figure 2.9: Overview of FOS' targeted architecture [6]

Although it has been thoroughly tested with a slot-style architecture, it is capable of running on island-style architectures as well. The only hardware design restriction is that each individual RR needs to be memory-mapped with an AXI4-Lite bus for control, and an AXI Stream bus for data communication. Figure 2.9 introduces an overview of a suitable island-style architecture for FOS.

In this framework, the static architecture is described using JSON file. Its purpose is to describe the number of RRs in the design, and what are their associated memory address and range. This helps the FOS libraries to connect with the accelerators in order to send control messages and read status words in the RR's control register. An API is provided in both `C++` and Python to communicate and dynamically reconfigure the RRs.

Each RR is described by its JSON accelerator descriptor which contains the list of partial bitstreams that can be implemented in the corresponding RR, or combination or RR in case of a slot style architecture. This is particularly helpful for application designers as in this programming model, a POSIX-like API is provided, and is able to call the provided scheduling algorithm to choose an accelerator implementation.

We mention now the cost of implementation of the FPGA architectures used in the previously introduced frameworks. These costs consist mainly of hardware control and communication infrastructure. The hardware control infrastructure is generally comprised of memory-mapped buses, some finite-state machines, and the ICAP DPR controller. The communication infrastructure includes DMA engine, interfaces (including local FIFOs), and interconnects to transfer data between RRs and the DMA engine.

The hardware control infrastructure has a relatively low cost of implementation (hundreds of LUTs and FFs) in comparison to the communication infrastructures (up to multiple thousands of LUTs and FFs) which in proportion is the biggest part of this implementation cost. The costs are greatly dependent on the scale of the system and should be considered in evaluation.

Table 2.3: Comparison of different frameworks on selected features of interest.

Features	OS4RS	HThreads	FOSFOR	FUSE	SPREAD	ReconOS	FOS
POSIX-like API	No	Yes	No	Yes	Yes	Yes	Yes
RR P2P communication	Yes	No	Yes	No	Yes	Yes	Yes
Standardized RR interfaces	Yes	No	Yes	Yes	No	Yes	Yes
Runtime resource management	Yes	Yes	No	Yes	Yes	No	Yes

We introduce Table 2.3 to compare the presented frameworks. As depicted by the focus of the literature, there is a tendency in provided frameworks

and libraries for hardware abstraction, and some work offers some level of runtime resource management. In addition, some approaches have considered increasing the performance of the self-reconfigurable systems via RR P2P communications.

We distinguish FOS as a notable work among the compared frameworks. FOS supports most features of interest and offers runtime resource management via a round-robin scheduler. The FOS scheduler was designed for a slot-style architecture to tackle the problem of implementing accelerators on multiple slots at once. In addition, as FOS is an open-source project, the runtime management can be modified to increase resilience through quality-oriented management.

In the FPGA, FOS requires dedicated bus adaptor interfaces to connect the RRs to the static design. This bus adaptator comprises the communication and control interfaces of each RR. In the version provided by the authors of [6], the bus adaptator also comprises a DMA controller for each RR, which brings a hardware overhead of 2k LUTs and 2.7k FFs per RR. Authors argue that this cost is relatively low in comparison to the available resources of their target (ZCU102 Evaluation Board). However, we can reduce this overhead as the architecture can be modified for smaller FPGAs.

2.1.5 Proposed architecture

To build our management methodology for the guarantee of service and the autonomic workload management problem, we introduce the proposed architecture in Figure 2.10. It has been implemented with four heterogeneous RRs for functional verification and metrics extraction.

This architecture is supported by the FOS libraries [6] and accelerator management and is inspired by features offered by other works. Notably, the streaming-based intra-FPGA RR-to-RR communications of [67] can be exploited to reduce communication latency and shared memory congestion. ReconOS' [65] control interface can also be used, and include command and status registers to communicate via the RR, as well as a simple internal FSM for the API to connect. To do so, we introduce a control and a communication interface for each RR in the static region of the FPGA.

The control interface contains a simple control FSM that is illustrated in Figure 2.11. It consists of three states: Idle, Running, and Reconfiguring. The Idle state denotes a reconfigurable module that's implemented on the RR and is awaiting its input data. Once the local input FIFO has been filled by the communication infrastructure, the task can enter the Running state. Then,

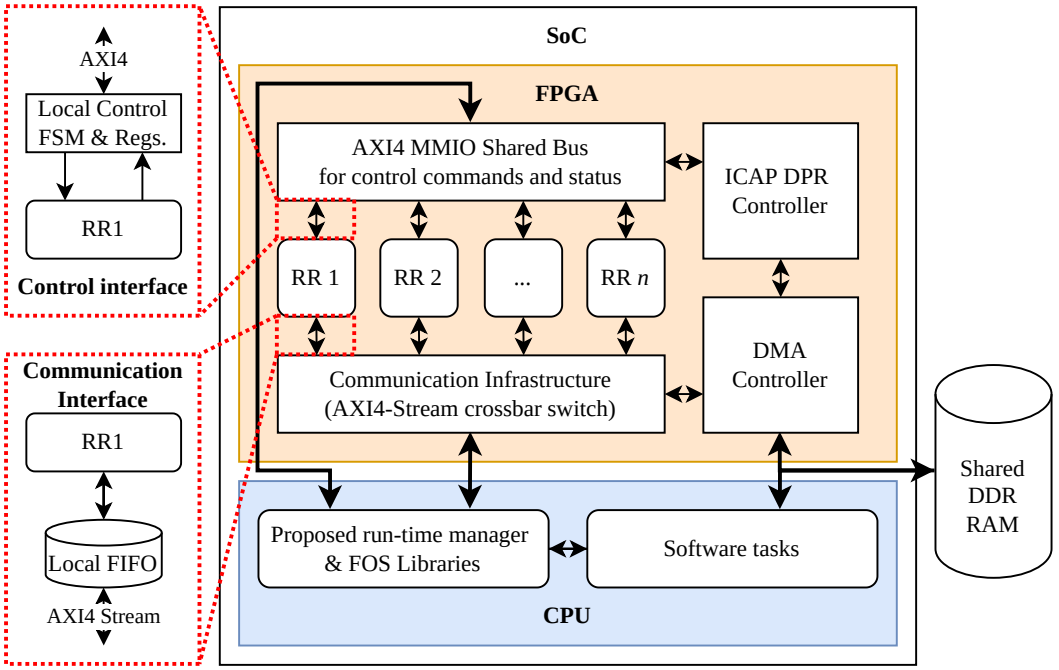


Figure 2.10: Overview of the proposed architecture

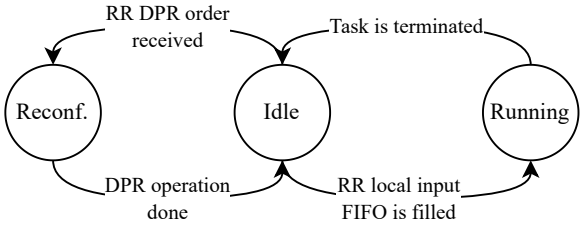


Figure 2.11: Local control interface FSM

once it has terminated its computation, the FSM returns to the Idle state, where a new DPR process can be performed (Reconfiguring), or the task be executed once again (Running). When the FSM goes to the Reconfiguring state, it awaits confirmation from the ICAP DPR controller to go back to the Idle state.

Each RR has a communication interface for input and output data transmission. These communication interfaces comprise input and output FIFOs capable of storing 1024×32-bit words, directly connected to the communication infrastructure, comprised of an AXI4 Stream crossbar switch. Those local FIFOs memories act like buffers to transfer data while the RRs is reconfigured to reduce data congestion in the AXI4 Stream switch. This enables faster data

transfers because intra-FPGA communication has a minimal time overhead.

However, crossbar switches use a lot of logic resources in the FPGA and this number grows exponentially with the number of regions: for k RRs, each of them needs to have its own bus interface with the other $k-1$ RRs. On the other side of the spectrum, a shared bus use fewer logic resources as only one instance of the bus exists. This can create a lot of communication delays as the bus becomes inaccessible to the other RRs as long as one RR occupies the bus. A middle ground approach would be the use of a Network on Chip (NoC) in the FPGA static architecture [68], which consists in implementing a network of buses to transfer packets between RRs in the design. Communication congestion prediction is out of this work's scope and is addressed in other works such as [69, 70].

Finally, both the RRs and CPU can access 512MB DDR3 RAM memory to store and transfer data. The RRs can do so via a DMA controller instantiated on the static part of the FPGA and connected to the communication infrastructure.

The RRs also have an AXI4 Lite bus connected to the Control Interface to receive orders from the run-time manager, and send back a status word for termination. The Control Interface also forwards orders from the CPU to the ICAP Controller.

The RRs are reconfigured by the ICAP controller which fetches bitstreams located in the DDR RAM. We used Xilinx's HWICAP IP as DPR controller for functional verification. Implementing other DPR controllers required extensive implementation work on the FPGA design and the FOS libraries, which was not the focus of this work. To profile the functionalities of such complex designs, we extrapolated bandwidth obtained by uPaRC [20] as metrics for our run-time methodology.

Finally, the system comprises a multi-core CPU: two cores for Zynq-7000 series, and four for Ultrascale-series SoCs. A core can be used to execute the run-time manager and OS services, while software tasks runs concurrently on the other(s).

A breakdown of resource usage per architecture element is made in Table 2.4. These implementation results were obtained on a ZedBoard Evaluation Board featuring a Zynq-7000 XC7Z020-CLG484-1 SoC. Overall, the static design used as architecture supporting the self-reconfigurable capabilities occupies approximately 15% of LUTs and 8% of FFs registers and BRAMs.

Table 2.4: Self-reconfigurable system resource usage by functional elements for the proposed platform. The summary section results are given as percentages of the available resources of a Zynq-7000 XC7Z020 SoC.

Available resources	**LUTs**	**FFs**	**BRAM36**	**DSPs**
RR1	6.4k	12.8k	20.0	40
RR2	10.0k	20.0k	30.0	50
RR3	12.8k	25.6k	30.0	40
RR4	9.6k	19.2k	30.0	60
Architecture element				
ICAP DPR Controller*	520	1.1k	1.0	0
DMA Controller	3.5k	4.5k	2.0	0
Communication Interface	1.5k	1.5k	8.0	0
Control Interface	2.9k	2.0k	0.0	0
Summary				
Static design	8.4k (15.8%)	9.1k (8.5%)	11.0 (7.9%)	0
Reconfigurable design	38.8k (72.9%)	77.6k (72.9%)	110.0 (92%)	190 (86.4%)
Total	47.2k (88.7%)	86.7k (81.4%)	110.0 (99.9%)	190 (86.4%)

*Xilinx's AXI HWICAP controller used in functional implementation

2.2 Self-reconfigurable systems management

In this section, we discuss the existing management methodologies, or runtime managers, of self-reconfigurable systems. As introduced in the previous section, we focus on the management of coarse-grain accelerators for island-style architectures containing RRs with heterogeneous sizes. These runtime managers are capable of allocating tasks to resources based on a given architecture and application information.

Generally speaking, most introduced frameworks possess some kind of runtime management that is either explicitly defined, as in FOS, with a dedicated mapping and scheduling algorithm. It can also be done with real-time OS libraries with priority-based schedulers [58, 59, 71]. In either case, these runtime manager focus on a task scheduling problem under time and space constraints.

However, more advanced runtime management methodology can increase the system resilience against faults [4, 72], or make use of hardware-software co-design approach to meet real-time deadlines [73, 74].

In particular, approaches that focus on quality-oriented management to render a guaranteed minimum service level are of interest to our work. Additionally, other approaches capable of efficiently managing application execution will be reviewed as it directly impacts the capability of a system to adapt itself to evolving execution context.

2.2.1 Management methodologies

To adequately manage self-reconfigurable systems, a dedicated runtime manager must find mappings of tasks on the system's resources to optimize an objective function. Mappings correspond to the assignment of tasks to the system's resources (RRs or CPUs).

When more than one task is assigned (or mapped) to a resource, the tasks are time-multiplexed. If the resource is a RR, multiple accelerators must be dynamically reconfigured in the RRs throughout an application's execution. In this case, scheduling algorithms are used to find a schedule of tasks on the resources. Scheduling algorithms typically minimize the execution time of targeted applications, but they can also minimize energy consumption [75] or maximize a quality metric [76].

Mappings can also be space-multiplexed, notably in the case of slot and grid-style architectures [39]. Then tasks can be mapped in such a way that they occupy multiple RRs at once to increase the speedup of a task or reduce resource fragmentation. Combinations of space and time-multiplexing mapping and scheduling approaches exist [6, 77] to minimize the execution time of an application.

Runtime management can also be used to increase system resilience. The authors of [4] introduce a runtime management method to increase the mean time to failure in an aerospace environment. In this environment, RRs become unstable due to gamma-ray exposition, and the proposed methodology must execute the application using fewer resources over time. Similarly, work in [78] introduces a runtime management method with fault mitigation capabilities to maximize the quality of service. The latter is characterized by the number of faulty application iterations (lower is better).

2.2.1.1 Quality-oriented management

Quality-oriented management requires the definition of quality models. In [79], authors introduce definitions of Quality of Experience (QoE) and Quality of Service (QoS). QoE is defined as the perceived quality by the end-user, tied to the functionalities of the executed application (eg. signal processing application SNR, video framerate...). QoS on the other hand is defined by the capability of the underlying hardware to execute the application under functional constraints (eg: the time before a deadline, resources occupancy...), regardless of QoE level. However, QoS can have an indirect effect on QoE (eg: schedule not respecting a deadline due to bad mapping decisions), and vice versa (eg. high QoE setting requiring a heavy workload to execute on the

resources). Using those two quality definitions, the goal of quality-oriented management is to guarantee a minimum level of QoE from the end user point of view, and optimization strategies are applied to maximize the QoE and QoS levels.

Quality-oriented management has been the focus of the literature for MPSoC and GPU management [80, 81, 82]. Through case-specific quality models, application designers can employ quality-oriented management methodologies to their own needs. This makes them capable of managing the runtime of the an application while maximizing its quantified quality.

The authors of [57] and [83] introduce a context and resource-aware runtime manager for autonomous UAV drones. In this approach, multiple applications constitute a workload to execute within a soft real-time deadline called the threshold. This workload evolves with additional or fewer tasks throughout the mission of the UAV, depending on its environment. To maintain a minimum level of QoE, the runtime manager checks if the workload can hold the threshold. If the resulting implementation doesn't, the runtime manager downgrades it to a lower QoE version, based on the hypothesis that a lower QoE level is associated with lower compute-intensiveness [76]. On the other hand, if the implementation's execution time holds the threshold by a significant margin, the workload is upgraded to a higher QoE version. This mechanism is called autonomic quality management and is illustrated in Figure 2.12.

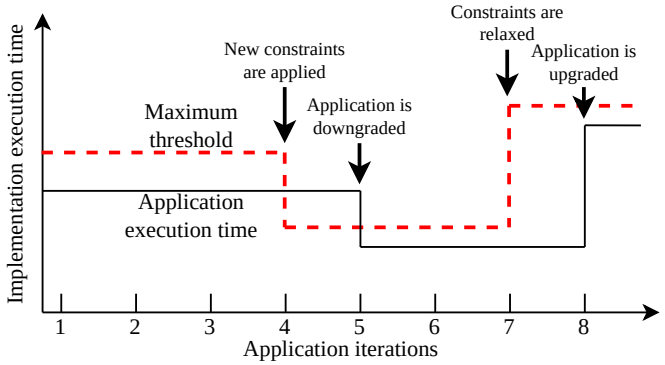


Figure 2.12: Autonomic quality management downgrade case.

At iteration 4, as the maximum execution time threshold is lowered, the application does not respect the new deadline. Its quality is downgraded to lower its computation intensiveness to meet the new constraint. Once the constraints are relaxed (step 7), the application is allowed to be upgraded again.

This autonomic quality management approach is close to our goal, but it was implemented as part of a mapping algorithm and did not tackle the challenge of runtime time-multiplexing scheduling of tasks on RRs.

The energy consumption and image quality trade-off have been studied by the authors of [84]. Their approach considers quality as a function of approximate computing error on the processed image. Although this approach does not target self-reconfigurable systems but static FPGA designs, it shows an insight as to how a QoE model can be defined from a functional point of view.

Recently, authors of [76] introduced an interesting quality-oriented management methodology for MPSoCs. Tasks from an application task graph can individually be tuned to different QoE levels. For each QoE level, a given task yields a different reward, which is a dimensionless score reflecting an end user's interest. Given this QoE model, this work's objective is to maximize the reward of a schedule while respecting a given deadline.

Then, a scheduling algorithm is executed to find a schedule that can run the application on the MPSoC within a defined deadline using the lowest QoE level for all tasks. This gives a base schedule, as in Figure 2.13 (a), where all tasks are assumed to be at their lowest QoE level, which gives an overall reward of 15. Then, the runtime manager introduced in [76] checks if each task can be upgraded to a higher QoE level without breaking the schedule's coherency. In Figure 2.13 (b), tasks T1 and T4 are upgraded which gives an additional 3 reward points as per the authors' example.

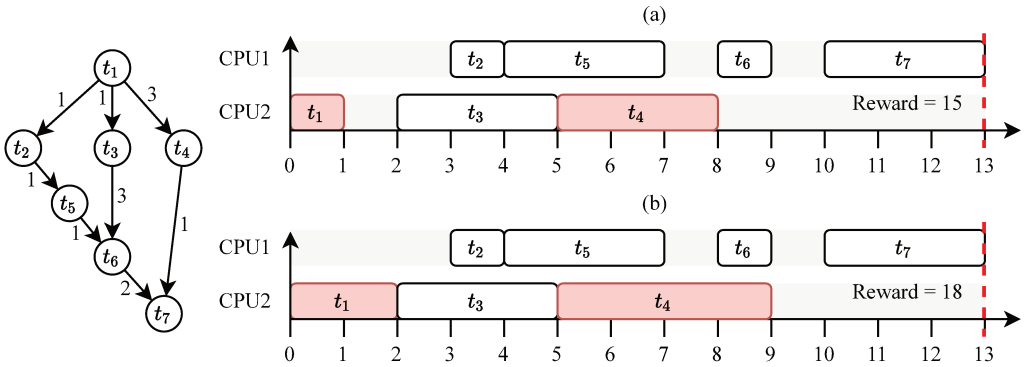


Figure 2.13: Quality-oriented scheduling principle from [76] for MPSoC architectures, applied to an example task graph. Schedule (a) contains only the lowest QoE level tasks for a reward score of 15. Schedule (b) upgrades the QoE of tasks t_1 and t_4 to higher quality levels to increase the schedule reward to 18.

While some introduced approaches can manage applications at runtime in MPSoC environments [76, 81], there is a lack of quality-oriented management for self-reconfigurable architectures. Moreover, the approaches that did focus on such architectures have been targeting specific use cases which cannot be applied generically to other applications.

The quality-oriented scheduling introduced in [76] is an interesting take for its genericity, but this approach does not target self-reconfigurable systems. In addition, individually managing the QoE level of tasks can be case-specific, and authors do not disclose a method to attribute rewards to application tasks.

The autonomic quality management from [83] offers another interesting take on the concept of upgrading and downgrading but did not tackle the scheduling problem for self-reconfigurable systems. To the best of our knowledge, no work exists regarding quality-oriented runtime management for self-reconfigurable architectures with runtime time-multiplexing scheduling.

2.2.2 Mapping and scheduling for self-reconfigurable systems

In the following paragraphs, we review works focusing on the mapping and scheduling problem for self-reconfigurable systems. Finding optimal mappings and schedules of workloads on such systems is an ongoing challenge in the literature. Given an appropriate set of metrics and constraints on the system and targeted application, computing mappings and schedules is an optimization problem with one or more objectives. As this problem is known to be NP-Hard, multiple heuristics have been developed over the years. Mapping and scheduling algorithms that are both fast and yield the shortest schedules are crucial for real-time embedded operating systems to hold application deadlines [24, 85].

Typically, these algorithms are executed alongside the software operating system. Some relatively rare approaches have considered implementing the scheduling algorithm in hardware [86, 87], although if decision times of those hardware schedulers are very low (sub-microsecond), these results are plagued by high LUTs and FFs implementation costs (multiple thousands of each).

Hardware scheduling could be more suitable for multi-tenant FPGA for cloud-computing. These platforms are based on much bigger FPGA matrices than those of cost-optimized FPGA-based embedded systems, and the proportion of used logic elements would be lower.

Compared with the multi/many-core domain which employs such algo-

rithms, scheduling for self-reconfigurable systems comes with the additional challenge of managing DPR operations. These operations can be seen as additional tasks to schedule on a DPR controller resource or be included within the tasks execution times. This management isn't trivial, and the schedule constraints brought by DPR introduce additional latency in the schedule. This is even harder when dealing with heterogeneous size of RRs in the platform.

In this section, we define formally the problem of mapping and scheduling for self-reconfigurable systems, and introduce methods from the literature to mitigate DPR-induced latencies. Finally, we review scheduling heuristics for runtime use.

2.2.2.1 Problem statement

As a general definition, the mapping and scheduling problems consist of time-multiplexing a set of tasks $\mathcal{T} = \{t_1, ..., t_n\}$ on a set of computational resources $\mathcal{P} = \{\mathcal{P}_1, ...\mathcal{P}_p\}$ for n tasks and p resources. In its non-preemptive general form, this set contains at least the following constraints: each task from the application must be executed once, and each resource can execute one task at a time. This implies that a task t_i can be executed at least one resource from $\mathcal{P}$. In this section, we formulate the mapping and scheduling problem applied to self-reconfigurable systems. Therefore, the set of resources $\mathcal{P}$ can be comprised of either RRs or CPU cores, and there exists an additional DPR resource onto which reconfiguration operations can be assigned and scheduled.

While mapping and scheduling are two separate problems, the literature comprises many examples of approaches that solve both together [88, 89, 90]. By doing so, the mapping can be computed in such a way that it minimizes the schedule duration.

We introduce Table 2.5 a list of parameters to describe systems and targeted applications.

Applications are represented by their directed acyclic graph (DAG), as illustrated by the canonical application task graph from [88], presented in Figure 2.14. In DAGs, vertices or nodes denote the tasks to execute, and edges illustrate data dependencies (i.e. communications) between tasks at the functional level.

DAGs can be formally modeled using an adjacency matrix A, which is a square matrix of size n. Elements $A(i; j)$ from this matrix denote the number of *tokens* (abstract unit of data) flowing through the DAG edge between tasks i and j. The corresponding adjacency matrix of the canonical example is presented in the right side of Figure 2.14.

Table 2.5: List of system and application parameters.

Symbol	Name	Size
n	Number of tasks	scalar
p	Number of resources	scalar
$\mathcal{T}$	Set of tasks	n
$\mathcal{P}$	Set of resources	p
A	Adjacency matrix	$n \times n$
C	Communication matrix	$p \times p$
W	Computation matrix	$n \times p$
R	Reconfiguration vector	$p \times 1$
D	Designation vector	$n \times 1$
E	Earliest start vector	$n \times 1$
F	Finish time vector	$n \times 1$

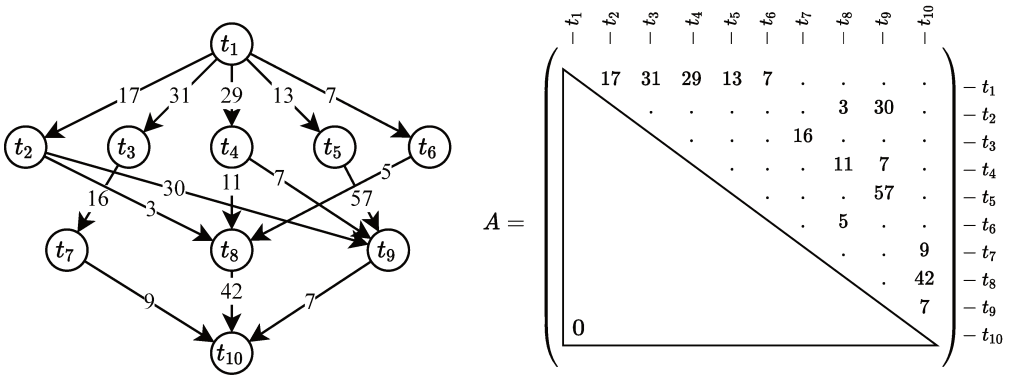


Figure 2.14: Canonical application from [88] with its adjacency matrix A. Dots denote zeroes for readability.

DAG adjacency matrices are given with their upper triangular matrix representation as in Figure 2.14. DAGs always produce such types of matrices for verification purposes: given an adjacency matrix A, if there exist no topological sorts such that for $i < j$, $A(i; j) > 0$, then A illustrates a task graph that doesn't comprise any loops and is therefore acyclic.

Tasks can be executed once all their parent (or predecessor) tasks have been executed. The tasks are atomic and executed until termination as preemption is not considered.

A $(n \times p)$ computation matrix W gives the execution times of the different tasks on the different resources (in milliseconds). As an example, Equation 2.1 is the computation matrix of the canonical example from Figure 2.14 on

three processing elements platform. Execution time $W(i;j)$ denotes the time during which a resource $\mathcal{P}_j$ is busy executing the task t_i. If $W(i;j) = 0$, then task t_i cannot be implemented on resource $\mathcal{P}_j$.

$$W = \begin{matrix} \begin{matrix} \mathcal{P}_0 & \mathcal{P}_1 & \mathcal{P}_2 \end{matrix} \\ \begin{pmatrix} 22 & 21 & 36 \\ 22 & 18 & 18 \\ 32 & 27 & 43 \\ 7 & 10 & 4 \\ 29 & 27 & 35 \\ 26 & 17 & 24 \\ 14 & 25 & 30 \\ 29 & 23 & 36 \\ 15 & 21 & 8 \\ 13 & 16 & 33 \end{pmatrix} \end{matrix} \begin{matrix} \\ -t_1 \\ -t_2 \\ -t_3 \\ -t_4 \\ -t_5 \\ -t_6 \\ -t_7 \\ -t_8 \\ -t_9 \\ -t_{10} \end{matrix} \tag{2.1}$$

As parameters for the platform, we finally introduce the $(p \times p)$ communication matrix C and $(p \times 1)$ reconfiguration vector R. Element $C(i;j)$ denote the communication time between two resources i and j. When $i = j$ (same resource), then the data don't need to be transferred over the communication infrastructure, and $C(i;j) = 0$. Element $R(i)$ denotes the reconfiguration time of a resource i, regardless of the task as this time depends solely on the size of the RR (in logic elements). If that resource is a CPU, then its reconfiguration time is null: $R(i) = 0$.

The mapping of the tasks is stored in a $(p \times 1)$ Designation vector. Its elements $D_i = \mathcal{P}_j$ returns the processing element that is assigned to the task t_i. In addition, the $(n \times 1)$ Earliest start time and Finish time vectors, respectively E and F, denote the start and termination time of a task t_i that is assigned to a resource.

Based on these system and application definitions, we define the following set of constraints that mapping and scheduling heuristics must answer to find a valid schedule.

The mapping and scheduling problem for self-reconfigurable systems is stated as follows:

Variables

- Set of tasks $\mathcal{T}$, of resources $\mathcal{P}$.
- Unique reconfiguration resource DPR to which reconfiguration tasks can be assigned

- Application parameters: adjacency and computation matrices A and W.
- System parameters: communication C and reconfiguration vector R.
- Designation vector D of size n, which elements $D_i \in \mathcal{P}$ denote the resource that has been assigned to task t_i.
- Earliest start vector E of size n, which elements E_i denote the start time of task t_i after being mapped on the resources.
- Finish time vector F of size n, which elements F_i denote the termination time of task t_i after being scheduled on the resources.

Constraint 1: Mapping completion

All tasks from the task graph must be assigned to a resource. Several tasks can be assigned to the same resource.

$$\forall t_i \in \mathcal{T}, \quad D_i \in \mathcal{P} \tag{2.2}$$

Constraint 2: No concurrency

Tasks are atomic and non-preemptible. Therefore a resource can execute only one task at a time.

$$\forall \mathcal{P}_j \in \mathcal{P},\ \{\forall t_i \in \mathcal{T} : D_i = \mathcal{P}_j\}, \quad \begin{cases} F_{i-1} \leq E_i \\ F_i \leq E_{i+1} \end{cases} \tag{2.3}$$

With

$$\forall t_i \in \mathcal{T},\ D_i = \mathcal{P}_j,\ F_i = E_i + W(i;j) \tag{2.4}$$

This ensures that no task can have a termination time while the resource $\mathcal{P}_j$ is busy executing the previous or next task assigned to this resource.

Constraint 3: Graph dependency

A task cannot start earlier than the termination time of its predecessor tasks. Assuming A is an adjacency matrix topologically sorted in its upper triangular matrix form:

$$\forall t_i \in \mathcal{T},\ D_i = \mathcal{P}_j,\ \forall t_k \in pred(t_i), \quad F_k \leq F_i - W(i;j) \tag{2.5}$$

With

$$pred(t_i) = \{\forall t_k \in \mathcal{T} : k < i,\ A(k;i) > 0\} \tag{2.6}$$

Constraint 4: Available data

A task cannot start unless it has received the data from its predecessors through the communication infrastructure.

$$\{\forall t_i \in \mathcal{T} : D_i = P_j\},\ \{\forall t_k \in pred(t_i) : D_k = \mathcal{P}_m\}, \quad F_k \leq F_i - [W(i;j) + C(m;j)] \tag{2.7}$$

It is to be noted that this constraint is a super set of constraint 3.

Constraint 5: Task reconfiguration

If the designated resource $\mathcal{P}_j$ for a task t_i is a RR, the latter must be reconfigured before its execution by a reconfiguration task t_{Rj} on the unique reconfiguration resource DPR.

$$\{\forall t_i : D_i = \mathcal{P}_j, R(j) > 0\}, \quad \{\exists t_{Ri} \in DPR :\ F_{Ri} < E_i\} \tag{2.8}$$

Once those constraints have been defined, we define the goal of the mapping and scheduling problem to minimize the schedule duration, or makespan.

2.2.2.2 DPR operations latency mitigation

Integration of DPR operations in the schedule has been the source of scheduling techniques to mitigate their inherent time overhead. In particular, bitstream reuse and pre-fetch techniques illustrated Figure 2.15, have been used to reduce the impact of DPR operations on the resulting schedules.

The pre-fetch technique, as employed in recent works [91, 90] consists in pipelining a reconfiguration job on the DPR controller in parallel with another task execution. This must be done in such a way that the RR is reconfigured before the corresponding task is ready for execution, and on an available RR (i.e. in Idle state).

Pre-fetch is illustrated in Figure 2.15 (a). The top schedule doesn't use any form of bitstream pre-fetching, while the bottom schedule makes use of it. In that bottom schedule, task t_2 can be executed right after t_1 has finished, effectively hiding the latency behind the execution of t_1. Without pre-fetching in the top schedule, the reconfiguration job of task t_2 starts at the end of task t_1 and introduces a delay equal to the reconfiguration job duration. Finally, t_4 cannot be pre-fetched as the targeted RR for t_4 is busy executing t_2, and the DPR schedule is busy executing the reconfiguration operation t_3^R. This technique is very useful in the case of relocatable bitstreams or for task with several potential RR targets. This is a substantial gain on schedule makespan

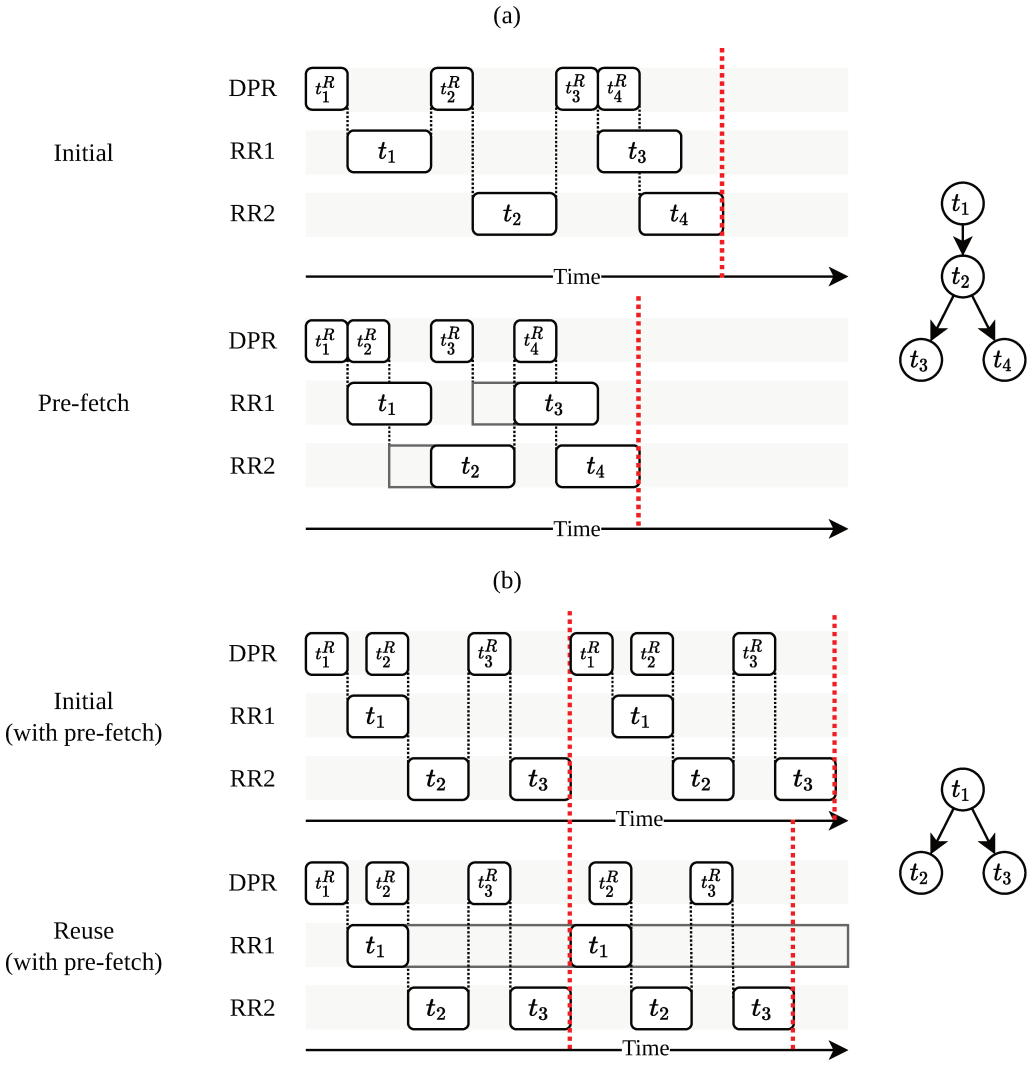


Figure 2.15: The pre-fetch (a) and reuse (b) scheduling techniques.

as DPR operations duration is non-negligible and the ICAP resource is unique in modern FPGAs [10, 28].

The reuse technique is illustrated in Figure 4.9 (b). Task t_1 is being reused between two iterations of the application, delimited by the dashed bold red line in the bottom schedule. In comparison with the top schedule that doesn't use this technique, the duration of the schedule has been reduced by one reconfiguration job, and this saving will be kept for any subsequent iteration of the application. RR reuse has notably been used in [92, 93] to limit the number of DPR operations during the execution of time-multiplexed mapping.

Between two iterations of an application or within a workload execution,

keeping a task idle for some time might be better than overwriting the RR with a new task. When keeping the task, there is no further need to reconfigure the used RR, so its DPR operation latency is introduced only once when it is first reconfigured. However, this implies that no other tasks can be implemented on this RR while the task is idle. This technique is particularly helpful when dealing with coarse-grain island-style architectures as DPR costs are higher than with smaller RRs. Also, when dealing with heterogeneous sizes of RRs, if one region can be used by only a few modules anyway, this might be of use.

Both techniques can be applied conjointly to reduce the impact of DPR operations on the resulting schedule [90].

2.2.2.3 Scheduling heuristics for self-reconfigurable systems

The reviewed multi-objective and quality-oriented approaches have in common that they all consider, in one way or another, the capability of mapping and scheduling the tasks on the resources to hold deadlines. In real-time systems, this time constraint is even more important as failure to meet a deadline can nullify any other accomplished scheduling objectives. In addition, when performing runtime scheduling, the decision time of the scheduling algorithm matters. As when new constraints are applied, systems need to react on time for reactivity [83]. Therefore, a trade-off exists between schedule duration (or makespan) reduction and the scheduler's time complexity.

The quest for obtaining an optimal schedule has always been a motivation for software engineers. NP-hard problems are notoriously difficult to solve optimally. MILP formulations run by complex solvers such as CPLEX or Gurobi have been used in the literature to find optimal schedules. Some approaches such as [94, 95] have been using MILP formulations on host computers targeting self-reconfigurable architectures, but as their results show, such algorithms cannot be ported to the target for runtime scheduling as their decision time is up to minutes on host computers.

Heuristics are employed to cut decision time short for runtime scheduling, accepting that the resulting schedules are sub-optimal. In this regard, the authors of [40, 96] have considered an range of scheduling heuristics based on the Earliest Deadline First (EDF) and the Lookahead heuristic [97] to schedule applications with respect of their respective mapping strategies. More recently, the authors of [98, 90] have implemented the pre-fetch feature on an ASAP scheduling policy to achieve a lower schedule makespan versus a non-pre-fetched scheduling policy.

In 2018, authors of [89] made a comparison of popular scheduling heuristics

for MPSoC with a focus on the trade-off between time complexity and resulting schedule makespan. The reviewed heuristics are the Heterogeneous Earliest Finish Time (HEFT) and Predict Earliest Finish Time (PEFT) [99, 88] algorithms, the Lookahead [97], and a Critical Path Aware (CPA) heuristic [99]. These heuristics are part of the list-based scheduling family. The main characteristic of this heuristic family is that throughout the scheduling of an application, they conjointly use a mapping strategy to minimize the schedule makespan.

In particular, the introduced heuristics are efficient in handling heterogeneous MPSoC systems, which self-reconfigurable architectures belong to, and for being capable of predicting the impact of scheduling decisions throughout the scheduling process. Predictive scheduling can be achieved by parsing the task graph once to detect potential execution bottlenecks, and then parsing it once again with this knowledge. The goal here is to find a topological order that maximizes the occupancy rates of resources to ensure maximum paralellization of task execution. The prediction feature lowers the resulting scheduling makespan, but has an impact on the time complexity of those approaches, as shown in Table 2.6.

Table 2.6: Time complexity of popular scheduling policies for heterogeneous computing for n tasks and p resources.

Name	Time complexity	Concept
ASAP [90]	$\mathcal{O}(n \cdot p)$	Greedy
PEFT, HEFT [88], [99]	$\mathcal{O}(n^2 \cdot p)$	Task prioritizing
CPA [99]	$\mathcal{O}(n^2 \cdot p)$	Critical Path
Lookahead [97]	$\mathcal{O}(n^4 \cdot p^3)$	Lookahead

Time complexity typically starts growing with the square of the number of tasks as the task graph needs to be parsed at least twice to properly predict decisions impacts.

The importance of predictive scheduling is highlighted in Figure 2.16 that showcases the CPA heuristic [99]. This figure illustrates three rounds of this algorithm. In this simple example, let tasks have the same execution time on any resource denoted by the number next to their respective node in the graph. In Figure 2.16 (a), task t_1 has already been selected for scheduling as it's the source task of the graph. Amongst the three possible paths, the critical path is $\{t_3; t_5; t_6\}$. Thus t_3 is added to the schedule, and the critical path search is repeated in (b) after having removed t_3 from the task graph. This step is repeated in (c), and up until schedule completion. Critical path search

ensures the topological ordering in which tasks are prioritized, according to the probability that they could cause a bottleneck in the schedule.

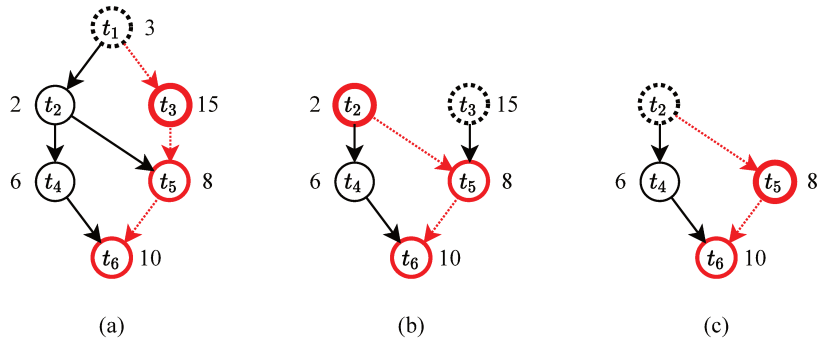


Figure 2.16: Critical Path Aware topological ordering principle.

The family of HEFT-based scheduling heuristics has been used recently in [100, 101] to schedule DAGs on Multi-FPGA architectures for cloud computing. However, authors do not address schedule optimizations (i.e. pre-fetch and reuse) nor runtime scheduling for embedded systems.

However, and to the best of our knowledge, no other works have considered employing the pre-fetching technique to benefit from the efficiency of PEFT on self-reconfigurable systems and mitigate DPR latency overhead. The closest approach would be As Soon As Possible (ASAP) with bitstream pre-fetching from [90], but that work does not consider architectures comprised of RRs of different sizes, which helps reduce resource fragmentation [102]. Among the compared scheduling heuristics for heterogeneous MPSoC in [89], the PEFT [88] has shown to be the most efficient in terms of schedule makespan reduction. We believe it can be adapted to with self-reconfigurable systems considerations.

2.2.3 Hybrid methodologies

As an alternative for runtime scheduling, we review here works on hybrid design-time/runtime methodologies. The main interest of such methodology is that it reduces the runtime management problem complexity by precomputing part of the problem offline on a host computer.

Hybrid management methodologies have found usage for self-reconfigurable systems [39, 98], and in particular to map tasks onto RRs for grid-style architectures. Their design time step consists in finding optimized implementations of tasks on RRs in the FPGA architecture (i.e. valid mappings). These implementations are evaluated in a vacuum at design time to find only specific

Pareto-optimum implementations on grid-style architectures. Then those optimum implementations are used at runtime to instantiate tasks onto the RRs.

However, while the approach from [98] did consider some form of ASAP scheduling, these approaches did not address the impact of heterogeneous RR sizes. Nevertheless, results in [98] show their hybrid design-time/runtime approach outperforms an OpenMP API in a MPSoC (CPU only) environment when taking decision times into account. This shows how efficient runtime management can deeply impact runtime performance.

As we target cost-optimized FPGA-based embedded systems, reducing the runtime management problem complexity is of interest. In particular, as it can decrease the decision time, we believe this could increase the reactivity of our system to evolving constraints.

2.3 Conclusion

Self-reconfigurable systems arouse great interest in the literature. Many styles of architectures and abstraction paradigms exist with their pros and cons, and works have been conducted to demonstrate the feasibility of such architectures and associated frameworks [66]. The island-style architectures, coupled to a coarse-grain tasks abstraction layer, is a good trade-off between performance and ease of use [67, 65, 6]. In particular, island-style accelerators are easy to manipulate by application designers as they can make use of custom-designed or vendor IPs. Xilinx vendor is pushing for such dedicated accelerator IPs with the Vitis accelerated libraries [11], and offers official DPR support through open source projects such as PYNQ [12]. In addition, some approaches have considered POSIX-like APIs [65, 6] to help software designers integrate such accelerators in their designs without requiring too much effort.

Self-reconfigurable systems management approaches have since been introduced to benefit from the full hardware acceleration potential that DPR offers. Efficient management isn't an easy problem to solve, and recent approaches have tackled this challenge through system resilience [78, 4], speedup performance [40, 90], and quality of experience maximization [83, 76]. To speedup the management process, hybrid design-time/runtime approaches have been introduced [39, 103]. Design Space Exploration (DSE) steps are conducted offline to either reduce the runtime management problem complexity or precompute a set of solutions that can be used at runtime.

To adapt to evolving system environment context, which can lead to increased resource constraints such as lowered RR availability [83], increased

workload [57] or restricted RRs [4], systems must provide management solution as efficiently as possible. In the case of real-time behaviors, this must be done on time, before reaching the application deadline. To answer this problem, scheduling heuristics have been introduced for MPSoC architectures [89], but few approaches have been considered in-schedule DPR operation management [90].

In this thesis, we build our architecture and framework on interesting features from the literature such as P2P RR communication [67] to lower communication times, and fast ICAP-based DPR controllers [18, 20]. The state-of-the-art FOS operating system framework [6] providing API and libraries to manage RRs from the software side.

In this chapter, we identified quality-oriented hybrid management methodology as an answer to the evolving system environment context problem to guarantee a minimum level of quality of experience. In addition, popular scheduling heuristics for MPSoC are efficient at time-multiplexing tasks on an architecture's resources, yet we identified a lack of such heuristics for self-reconfigurable systems.

Chapter 3

Quality-oriented application management

Contents

3.1 Overview

To guarantee service execution, self-reconfigurable systems must cope with run-time changes in RRs and CPU cores availability, and modifications in application requirements. These changes are introduced by additional tasks or services that must be executed on the resources (eg. evolving workload throughout a drone mission [57]), or restrictions on the resources themselves (eg. resources experiencing failures in highly constrained environment [4]). Reacting to those resource availability changes on time is crucial to maintain continuity of service. It is characterized by the capacity of the system to meet the application deadlines.

In this chapter, we introduce a quality-oriented hybrid manager to manage the introduced self-reconfigurable system. It can react to run-time evolutions of resource availability and additional workload while maintaining continuity of service. To do so, the proposed approach makes use of execution modes that are based on different combinations of functional parameters. Each mode is related to a quality score, and the runtime manager can change the current execution mode to react to the evolving situation while maximizing this quality score.

We first detail how execution modes can be defined using a H.264 application benchmark as an example. We then show how QoE and QoS models can be built for this application. Finally, we introduce our hybrid design-time/run-time methodology.

3.2 Quality model

3.2.1 Execution modes

Functional parameters (eg: image resolution for a video codec application, targeted SNR for signal processing ...) have an impact on perceived quality as they change how an application behaves. Different combinations of functional parameters of a given application denote different execution modes. In this thesis, applications are defined by their task graph, described at the functional level. Therefore changes in the functional parameters have an impact on the task graph, as in the scenario-aware synchronous data flow (SADF) model of computation [104]. This means that between execution modes, changes in the task graph can be introduced, by adding or removing tasks, modifying the tasks execution, or a combination of these changes.

When describing the application at the functional level, parameters that

have such an effect can be defined. Then, for each combination of these parameters, a new execution mode can be defined.

To illustrate this vision, we introduce the H.264 application benchmark. The H.264 video standard [105] is one of the most commonly used formats for video compression. Its high compression rate is achieved thanks to an integer discrete cosign transform (integer DCT) which can be accelerated by hardware. This application offers two functional parameters commonly manipulated to increase or reduce its compute-intensiveness: the video framerate and image resolution [106].

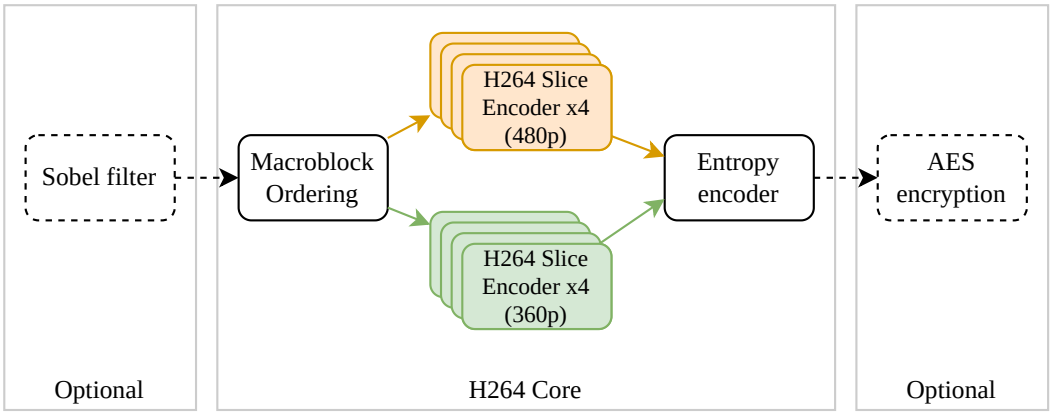


Figure 3.1: Task graph representation of the multi-resolution H.264 encoder benchmark with optional filtering and encryption tasks.

The H.264 encoder application task graph is illustrated in Figure 3.1. It is comprised of 'Core' H.264 tasks, and two optional tasks (Sobel filter and AES encryption). The Core H.264 is comprised of a Macroblock Ordering task which decomposes an image into four slices following the Flexible Macroblock Ordering Type 0 [107]. Each slice is encoded in parallel using the H.264 Slice Encoder tasks, which are comprised of an integer DCT transform and a quantization step. Then the encoded slices are merged into a compressed bitstream via entropy encoding. As only one image resolution can be selected at a time, only one set of H.264 Core tasks can be active at a time (orange for 480p and green for 360p).

The Sobel filter and AES encryption tasks are introduced to enhance the application and show their impact on execution modes.

We denote a unique execution mode of a targeted application as $\mathcal{M}_i$. The number of unique execution modes is n_{EM} $(0 < i < n_{EM} - 1)$. To define execution modes, we must identify functional parameters that have an impact on the perceived quality or quality parameters for short. Using the H.264

application as an example, we introduce Table 3.1 a list of the relevant quality parameters Q_i and their respective possible values.

Table 3.1: List of quality parameters for the H.264 application.

Parameter name	Values
Image resolution	$Q_0 = \{$ 360p; 480p $\}$
Video frame rate	$Q_1 = \{$ 30fps; 45fps; 60fps $\}$
Image filtering	$Q_2 = \{$ No; Sobel $\}$
Encryption	$Q_3 = \{$ No; AES $\}$

Each quality parameter must have at least two values. Changing values on these parameters must have an impact on the task graph properties (topology, execution times, or both). In this example, the image resolution impacts the execution times of the Slice Encoder tasks. Changing the Image filtering parameter adds or removes the Sobel task from the task graph. The video frame rate has an indirect impact on the application, as it changes the application deadline.

For a given application with n_p quality parameters Q_i, the number n_{EM} of execution modes is equal to the number of permutations with repetitions of quality parameters:

$$n_{EM} = \prod_{i=0}^{n_p-1} |Q_i| \tag{3.1}$$

The theoretical number of execution modes for the H.264 application is $n_{EM} = 2 * 3 * 2 * 2 = 24$. This number denotes a theoretical maximum limit of execution modes as some may not be implemented or desirable. For instance, AES and Sobel functionalities are considered optional for the H.264 application, they aren't made available for lower resolutions.

Execution modes should be exploited to minimize the number of changes when switching from one mode to another. Specifically, by putting a focus on the reuse technique which consists in reusing an accelerator that's already implemented on a RR between application iterations. Then if the task is the only one that's executed on this RR, there is no need to reconfigure it, which in turn frees up the ICAP DPR controller.

In this regard, we introduce the concept of quality-parameter-dependent tasks. These dependent tasks are impacted when changing a quality parameter when switching execution modes. Let two execution modes $\mathcal{M}_i$ and $\mathcal{M}_j$ ($i \neq j$) of an application. The task graph difference between $\mathcal{M}_i$ and $\mathcal{M}_j$ are the

dependent tasks that need modifications (inducing reconfiguration) due to a change of quality parameters.

For instance, with the H.264 application, the difference between execution modes $\mathcal{M}$(360p; 30fps) and $\mathcal{M}$(360p; 30fps; Sobel) is the Sobel task. The other tasks are quality parameter independent as they don't require reconfiguration. If one of those quality-parameter-independent tasks is already implemented on a RR and the only one in this schedule, then we can reuse it when transiting from one execution mode to another.

Table 3.2: Definition of the execution modes for the H.264 application.

Execution mode	$\mathcal{M}_0$	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$	$\mathcal{M}_4$	$\mathcal{M}_5$	$\mathcal{M}_6$	$\mathcal{M}_7$	$\mathcal{M}_8$	$\mathcal{M}_9$
Image resolution	360p	360p	480p	480p	360p	360p	480p	480p	480p	480p
Video framerate (fps)	30	60	30	60	30	30	30	60	30	45
Sobel filter	No	No	No	No	Yes	No	Yes	No	Yes	Yes
AES encryption	No	No	No	No	No	Yes	No	Yes	Yes	Yes

Out of the 24 theoretical execution modes, only 10 were retained as execution modes of interest for the H.264 application benchmark. These execution modes are summarized and defined in Table 3.2. This choice was based on the different feasible implementations on the targeted architecture, and coherency with the quality-assessment study results from [106] to introduce solely execution modes with significant changes in the task graph.

3.2.2 Quality of Experience

Quality of experience (QoE) indicates the user's perceived quality of a given execution mode. Depending on the nature of this perception, this metric can be very subjective (perceived quality from an end-user), or more objective (measured performance to reach a given task). For instance, the video frame rate may be significant for a car's infotainment system (eg: headrest video players) or a drone camera, but not as much as for an embedded system executing a high frame rate target acquisition program. As there is no unified QoE model, the effect of quality parameters on perceived quality can be modeled using empiric quality assessment [106], mathematical models [108], or a combination of both.

Empiric quality assessment is a study of the users' observations on perceived quality. It is best used for qualitative and subjective quality parameters, or when the number of different settings is relatively low and cannot be easily described mathematically. The typical usage of empiric quality assessment is to define video quality settings. There, a panel of end-users is asked

to give quality scores to various quality settings videos being played. The subjective nature of QoE in that use-case is highlighted when end-users speak in terms of image definition or details, which perception may vary with the content of the video and end-users eyesight. Finally, quality scores are used to define which settings the users prefer and illustrate how much better one setting is compared to another.

Comparatively, mathematical models are best suited for quantifiable quality parameters. Those are related to the application's performance, eg: Signal to Noise Ratio (SNR) for signal processing applications, or bandwidth of an encryption application. When such performance can be directly quantified, mathematical models can describe the preference. Eg: a 20dB SNR of a software defined radio has 10× less noise than one that has an SNR of 10dB.

Once a QoE model has been identified for the application with respect to the quality parameters, a QoE score Q_S^E function can be defined. This function takes a given execution mode $\mathcal{M}_i$ as parameter, which is comprised of its associated set of quality parameters (eg: $\mathcal{M}_0 = \{360p; 30fps\}$). To apply our methodology to other applications and QoE models, this function must return a score such that for any execution mode $\mathcal{M}_i$; $0 < Q_S^E(\mathcal{M}_i) \leq 1$.

Using the H.264 application quality parameters from Table 3.1, functional descriptions such as { 360p; 480p } and { No; Yes } are reduced to numerical values based on their indices (i.e. "360p" counts as 0 and "480p" counts as 1). The frame rate parameter retains its numerical value (i.e. "45fps" counts as 45). The quality of experience score Q_S^E function for the H.264 application has been extrapolated using a quality-assessment study from [106], and is defined in Equation 3.2:

$$Q_S^E(\mathcal{M}_i) = \frac{0.75Q_0 + 2.5Q_1.10^{-2} + Q_2 + Q_3}{4.25} \tag{3.2}$$

Finally, using this equation we can attribute a quality score to each execution mode, and the list of possible execution modes for a given application can be drawn in Table 3.3.

Table 3.3: Application execution modes for the H.264 application, sorted by increasing quality score Q_S^E.

Execution mode	$\mathcal{M}_0$	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$	$\mathcal{M}_4$	$\mathcal{M}_5$	$\mathcal{M}_6$	$\mathcal{M}_7$	$\mathcal{M}_8$	$\mathcal{M}_9$
Number of tasks	6	6	6	6	7	7	7	7	8	8
Quality score Q_S^E	0.18	0.35	0.35	0.53	0.59	0.59	0.76	0.76	0.82	0.87

3.2.3 Quality of Service

Quality of Service (QoS) answers the question "how well is the service executed by the system?". Whereas the QoE of an application can be evaluated offline, the QoS of a mapping and scheduling solution evolves at runtime. As it takes into account the system's current resource constraints, it reflects a runtime state of execution.

Two implementations of an application can only compare their QoS at an equal QoE setting. If those implementations run different execution modes, then their inherent functional difference might have an impact on the resources. Therefore the difference in QoS might be caused by the change in QoE. A trivial example is a 480p mode having to compute more pixels than a 360p mode, making the higher resolution implementation more demanding in terms of resource usage. However, two instances of the same execution mode can compare their QoS scores since they deliver the same service. Following the axiom that a higher QoS score implies a more efficient implementation, the system should use the one that maximizes QoS whenever possible.

Similarly to the QoE model, QoS is subjective to the system's specifications and the goal of the embedded system. We introduce here the example of real-time operating system QoS modelization, although QoS can also be modeled to consider mean time to failure [4] or energy consumption considerations [84].

In a real-time operating system, QoS would typically be related to occupancy rates of the CPU and soft or hard time constraints. Two paradigms exist concerning the QoS modelization of occupancy rates:

- *Load balancing* is performance-oriented and implies the system must be as busy as possible to make the most out of its available resources to execute an application. Here, the QoS level is higher the more the occupancy rates of the system's resources.

- *Slack* is power-oriented and rewards implementation solution that minimizes the occupancy rates and maximizes the time before the deadline. A preference is made toward systems that use as few resources as possible to execute the application.

A well-tuned QoS model based on the load balancing paradigm can value implementations that offload most of the work on hardware accelerators and free CPUs. However, it can over-value implementations that result in high occupancy rates for the sole reason of making accelerators and CPUs busy.

A QoS model based on slack favors implementations executing the targeted application as fast as possible. As they become more compute-intensive, high

QoE execution modes can return lower QoS scores with the slack paradigm. However, because QoS scores can only be compared at equal execution modes, this only means that high QoS might be harder to achieve as QoE increases.

For the H.264 application example, we chose to represent QoS using the slack paradigm which was also the choice made in [76]. Let a solution $\mathcal{S}$ be running on the system's resources. Its slack is defined as resource idle time duration between the end of the implementation execution $\tau_{\mathcal{S}}$ and its application deadline τ_D. We introduce the QoS score $Q_S^S(\tau_{\mathcal{S}}; \tau_D)$ in Equation 3.3.

$$Q_S^S(\tau_{\mathcal{S}}; \tau_D) = \begin{cases} \dfrac{\tau_D - \tau_S}{\tau_D} & \text{if} \quad \tau_S \leq \tau_D; \\ 0 & \text{else} \end{cases} \tag{3.3}$$

For the same reasons as the QoE model definition, this QoS function is defined in such a way that $0 \leq Q_S^S(\tau_{\mathcal{S}}; \tau_D) \leq 1$.

Finally, as we aim to guarantee the continuity of service execution while maximizing quality, we introduce the concept of continuity of service breaks, or service breaks for short. They are characterized by an unmet application deadline (failure to deliver the service), or when QoS is equal to zero.

3.3 Hybrid mapping and scheduling management

The proposed hybrid management approach is introduced in Figure 3.2. It is divided into two parts: design time and run time.

The design-time step's objective is to find a large number of mapping and scheduling solutions that take into account the application and system models introduced in chapter 2, and quality models. The mapping and scheduling solutions correspond to application implementations on the self-reconfigurable system. These solutions are then evaluated, sorted, and pruned. The remaining solutions are stored in a solution database, which is a JSON file containing mapping and scheduling solutions for the runtime manager to parse through. This database is then uploaded onto the self-reconfigurable system.

The runtime step of the proposed management approach is activated upon detection of new system constraints (e.g. additional workload changes, restrictions on resources), or in case of service break detection. It is responsible for the selection of a new solution from the pre-computed database to maximize the QoE first, then the QoS. The decision time to find this new solution is taken into account, as the system must react quickly to adapt to the new

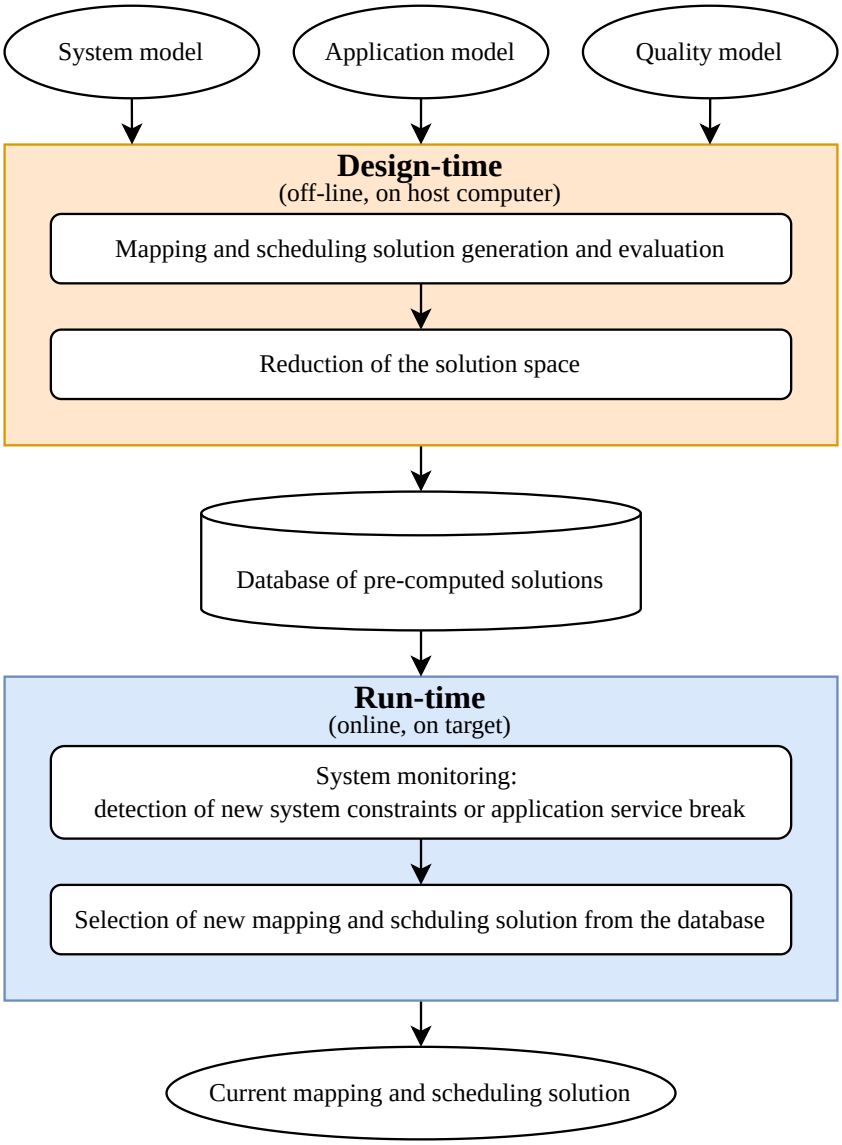


Figure 3.2: Overview of the proposed quality-oriented hybrid design-time/run-time management of self-reconfigurable system.

constraints or to fix the service breaks. Finally, the returned mapping and scheduling solution can be enforced via the FOS libraries [6].

This methodology can be used for any use-case described at the function task graph level targeting a self-reconfigurable system with similar architecture as introduced in Section 2.1.5.

3.3.1 Design-time computation

Defining a mapping and a schedule of the application task graph means assigning tasks to RRs or CPU cores, and defining their order of execution, or schedule, to respect the application schedule. As the system doesn't know beforehand what constraints will be imposed to the system at runtime, a variety of mapping and scheduling solutions must be generated to store on the target, and provide more freedom for the online stage. If the system doesn't have a solution that fits the system given the current constraints (i.e. non null QoS score), then it must downgrade the QoE and run a sub-optimal one. In the worst-case scenario, if the manager cannot downgrade further the QoE, then a service break occurs.

Here, the trade-off of hybrid design-time/runtime management between efficiency and runtime complexity is addressed by generating and evaluating a set of solutions for each execution mode. These sets must comprise a variety of high QoS solutions to answer the runtime events.

Given an application and its corresponding quality model, as well as the model of the targeted self-reconfigurable system, the design-time step consists of three consecutive operations:

1. Combinatorial fixed mapping generation;
2. Schedule evaluation of the mappings;
3. Reduction of the solution space and creation of the data set.

After which the data set is fully generated and compiled as a JSON file containing multiple valid solutions for each execution mode, ready to be exploited by the run-time step.

3.3.1.1 Combinatorial mapping generation

The mapping (i.e. assignation of tasks to resources) and scheduling (i.e. time-multiplexing of tasks on resources) are separated here as the scheduling heuristic aims to find mappings that minimize the schedule duration, or makespan. Because constraints can impact resources that will be used by the obtained schedules, we must provide solutions for the runtime manager that are sub-optimal when there are no constraints but may become useful when constraints are applied.

The list of mappings can be obtained by combinatorial mathematics formulas. A mapping corresponds to an assignment of tasks to computation

resources (RRs or CPU). Given a number n_{EM} of execution modes, a number n_i of tasks in the task graph of execution mode $\mathcal{M}_i$, and p resources to execute the tasks. Considering no restrictions on the mapping, i.e. all tasks can be executed on all resources, the maximum number of unrestricted mappings N_{map} is:

$$N_{map} \leq \sum_{i=0}^{n_{EM}-1} k^{n_i} \tag{3.4}$$

As an example, let's consider a system comprised of 4 RRs and a CPU core. Given the benchmark H.264 application execution modes from Table 3.3, we can draw Table 3.4 which computes the number of unrestricted mappings for each execution mode.

Table 3.4: Number of unrestricted mappings by execution modes for the H.264 application.

Execution mode	$\mathcal{M}_0$	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$	$\mathcal{M}_4$	$\mathcal{M}_5$	$\mathcal{M}_6$	$\mathcal{M}_7$	$\mathcal{M}_8$	$\mathcal{M}_9$
Number of tasks	6	6	6	6	7	7	7	7	8	8
Unrestricted mappings k^{n_i}	15k	15k	15k	15k	78k	78k	78k	78k	390k	390k

In total, this yields 1'156'250 fixed mappings that our methodology needs to find a schedule for. It becomes obvious that for larger self-reconfigurable systems and more complex workloads, this number grows considerably and severely hampers exhaustive algorithms. Constraints on the mapping generation and heuristics to evaluate and exploit the database are then needed.

This exhaustive mapping of Equation 3.4 doesn't limit the number of tasks assigned to a single resource (RR or CPU), i.e. for n tasks in the task graph, up to n task can be assigned to a single RR. In such a case, the sum of task execution times on this RR can become greater than the application timing deadline. Thus, the mapping can be considered invalid before even trying to find a valid schedule for it.

Figure 3.3 illustrates a timing deadline violation due to a bad mapping. In this thought experiment, many tasks are assigned through fixed mapping to RR1 and the scheduler has no choice but to execute them sequentially. However, such mapping had no chance to work in the first place, as the sum of task execution times τ_i assigned to resource RR1 is greater than the application deadline τ_D.

To prevent the design-time mapping generation from spending time computing schedules for invalid mappings, we introduce a constraint on the maximum number of assigned tasks to a single resource.

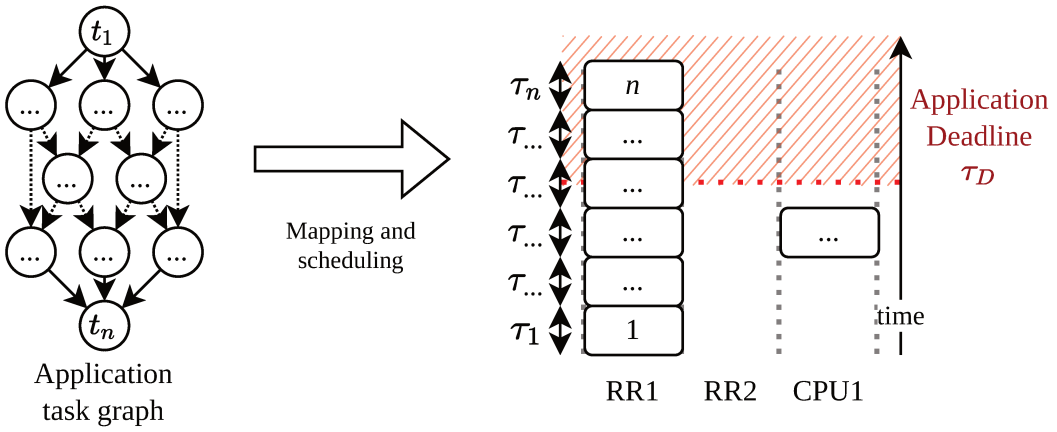


Figure 3.3: Deadline violation due to a task assignment overload on RR1.

Constraint 1: For a set of tasks $\mathcal{T}$ assigned to a resource $\mathcal{P}_j$ through fixed mapping. Given the computation time matrix W and reconfiguration vector R (from scheduling parameters in chapter 2), and the application deadline τ_D. Mappings are considered worthy to compute a schedule if:

$$\sum_{t_i \in \mathcal{T}} (W(i;j) + R(j)) \leq \tau_D \tag{3.5}$$

In addition, as we consider self-reconfigurable systems with different sizes of RRs, some tasks might not have a module for certain RRs. In such a case, no schedule can be computed for such mappings, and we introduce a second constraint on the fixed mapping generation.

Constraint 2: For a set of tasks $\mathcal{T}$ assigned to a resource p_j through the fixed mapping. Mappings are considered worthy to compute schedule if:

$$\forall t_i \in \mathcal{T}, \quad W(i;j) \neq \emptyset \tag{3.6}$$

Once the set of fixed mappings has been generated with the proposed constraints, the design-time step enters the schedule evaluation phase. These two constraints enable the phase to reduce the design space down to 35.8k solutions out of the 1'152k theoretical unrestricted mappings from Table 3.4. This 96.81% reduction ensures only mappings that have a chance of yielding a valid schedule with a high enough QoS score are being evaluated.

3.3.1.2 Schedule evaluation

The set of fixed mapping now defined, we run a fixed-mapping scheduling algorithm that considers the application's and system's specifications to define

a working schedule. Given the size of the solution space to evaluate, the choice of ASAP scheduling is motivated by its $\mathcal{O}(n \cdot p)$ time complexity for n tasks and p resources (RRs and CPU). Using fixed mappings, this time complexity even drops to $\mathcal{O}(n)$ because the scheduler doesn't need to find the best mapping according to its scheduling policy.

The fixed mapping scheduling heuristic used in this section is described in Algorithm 1 and illustrated in Figure 3.4 using application and system parameters from Table 2.5.

Algorithm 1 Fixed mapping scheduling heuristic

Input : task graph, application, and system information, and fixed mapping

Output : valid schedule

1: Put source tasks in the list of tasks ready for execution
2: **while** list of tasks ready for execution is not empty **do**
3: let task t_i be the next task in the list
4: let resource $\mathcal{P}_j$ be the selected resource for t_i from the fixed mapping
5: compute earliest start of t_i on resource $\mathcal{P}_j$ given R, C and W
6: **if** last task on resource $\mathcal{P}_j$'s schedule is identical to t_i **then**
7: *// Reuse the task that's already implemented*
8: **else if** $R(j) > 0$ **then** *// Resource $\mathcal{P}_j$ is a RR*
9: append reconfiguration job t_i^R for task t_i in the DPR resource
10: add t_i to the schedule of resource j
11: mark t_i as done
12: update list of tasks ready for execution
13: **return** schedule

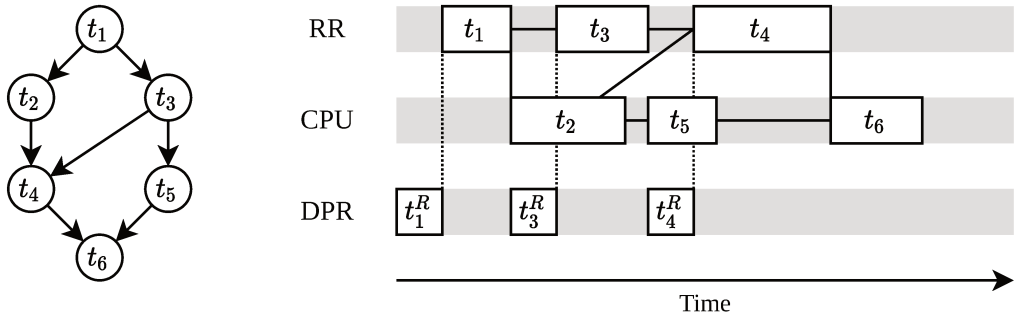


Figure 3.4: Schedule of a sample task graph on a single RR and CPU. Management of reconfiguration operations by the Configuration Access Port (CAP) is shown in the DPR timeline.

For each task t_i to schedule on a RR, the heuristic schedules the corresponding reconfiguration job t_i^R. As the Configuration Access Port (CAP) controller is unique, the reconfiguration jobs cannot be executed in parallel.

Taking the schedule on a single RR and CPU example from Figure 3.4 the reconfiguration operation t_1^R needs to be executed before the source task t_1. Once t_1 ends, t_2 and t_3 are added to the list of tasks ready for execution. The corresponding reconfiguration job of t_3, as it is executed on the RR, is added to the CAP schedule. t_3^R then delays the start of t_3 as it needs to wait for the end of t_3^R.

No tasks can be executed on a RR during the reconfiguration job of this RR because the targeted programmable logic is unavailable during the operation.

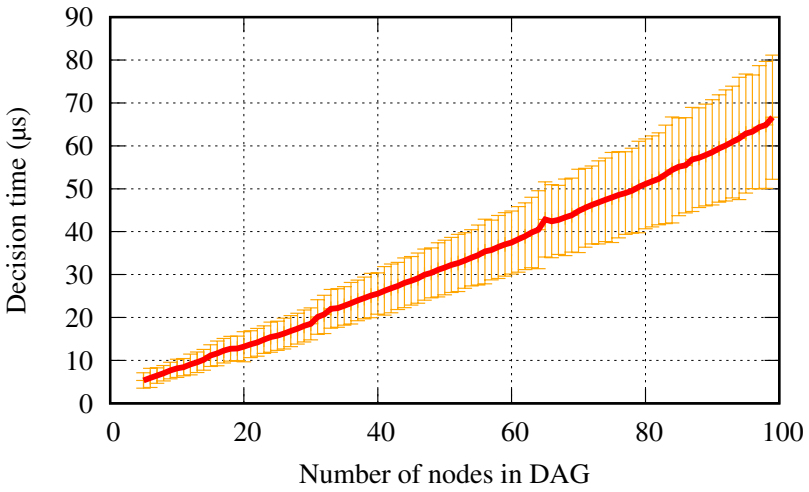


Figure 3.5: Scheduling decision time on a host computer of the fixed mapping ASAP heuristic by the number of nodes in the DAG task graph.

This fixed mapping scheduling respects the unicity of the ICAP controller for schedule coherency with the characteristics of self-reconfigurable systems. However, it does not make use of bitstream pre-fetching as this topic is discussed in chapter 4.

The scheduling decision time of the fixed mapping scheduler has been benchmarked using random synthetic application workloads. By generating a million random DAGs comprised of 5 to 100 nodes, we run the heuristic written in `C++` on a host computer's CPU (Intel i5-7300U 2.60GHz). The scheduling decision times are shown in Figure 3.5. Overall, obtained results show the heuristic yields schedule within 100μs, which is sufficiently low considering the high number of mappings to evaluate. This schedule evaluation is operated on the 35.8k mapping solutions from the combinatorial generation.

3.3.1.3 Solution space reduction

After having processed the schedule of all unique mappings and applied constraints, the design-time step enters the solution space reduction phase. As illustrated in Figure 3.6, the Pareto front can be drawn to find the best mapping and scheduling solutions for each execution mode. However, the final database that must be stored on target must contain more than only the Pareto-optimal solutions as the runtime manager must parse through it to find a solution that works given the resource constraints.

The trade-off between the number of solutions kept in the database and their quality of service must be addressed. Typically, solutions with a very low QoS score have little chance of being used at runtime as this implies they barely meet the requirements.

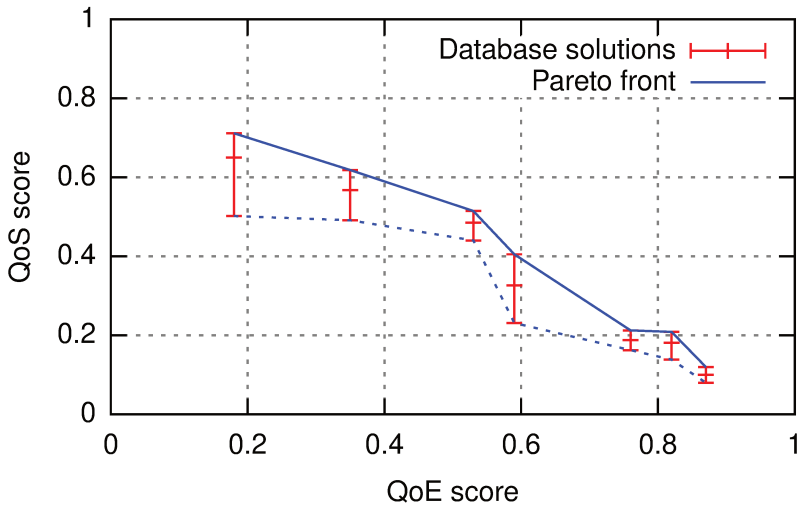


Figure 3.6: Database solution space in the QoE-QoS domain.

In the H.264 example, we propose here to keep the best QoS half of the solutions of an execution mode, if that set comprises more than 1k solutions. In addition, we prune any solution that has a QoS score below 0.10 as they are likely not to execute properly. The solution space profile is illustrated in the QoE-QoS domain in Figure 3.6. All database solutions are plotted in the red bars following with their evaluated QoS and QoE scores as coordinates. We see a correlation between the two scores, as when the QoE score grows, the QoS decreases. This validates the hypothesis that high QoE workloads are harder to implement for the self-reconfigurable system

The result of this step is shown in Table 3.5. In total, there are 15.8k mapping and scheduling solutions that are saved in the database for the H.264

application. It is to be noted that the execution mode $\mathcal{M}_9$ is heavily reduced, as there are few mappings that can make the application meet the deadlines. The constraints proposed at the combinatorial mapping step helped remove a large majority of the potentially invalid mappings.

Table 3.5: Application execution modes for the H.264 encoder benchmark application after the pruning.

Execution mode	$\mathcal{M}_0$	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$	$\mathcal{M}_4$	$\mathcal{M}_5$	$\mathcal{M}_6$	$\mathcal{M}_7$	$\mathcal{M}_8$	$\mathcal{M}_9$
Number of tasks	6	6	6	6	7	7	7	7	8	8
Unrestricted k^n	15k	15k	15k	15k	78k	78k	78k	78k	390k	390k
After constraints	7.8k	5.2k	6.9k	2.5k	6.4k	3.5k	1.2k	2.0k	2.2k	110
After pruning	3.3k	2.8k	2.4k	1.0k	2.9k	1.4k	0.4k	0.6k	1.0k	110
QoE score Q_S^E	0.18	0.35	0.35	0.53	0.59	0.59	0.76	0.76	0.82	0.87
Average QoS score Q_S^S	0.65	0.54	0.56	0.49	0.34	0.33	0.19	0.18	0.18	0.11

Figure 3.6 and Table 3.5 can help determine execution modes that may not be used frequently. Let two execution modes $\mathcal{M}_i$ and $\mathcal{M}_j$ such that $Q_S^E(\mathcal{M}_i) > Q_S^E(\mathcal{M}_j)$, if $Q_S^S(\mathcal{M}_i) > Q_S^S(\mathcal{M}_j)$, then execution mode $\mathcal{M}_j$ is likely to be used less than $\mathcal{M}_i$, if at all.

3.3.2 Run-time computations

Once the design-time step on the host computer is completed, a workload package is generated to be sent to the target. The components of this package are illustrated in Figure 3.7.

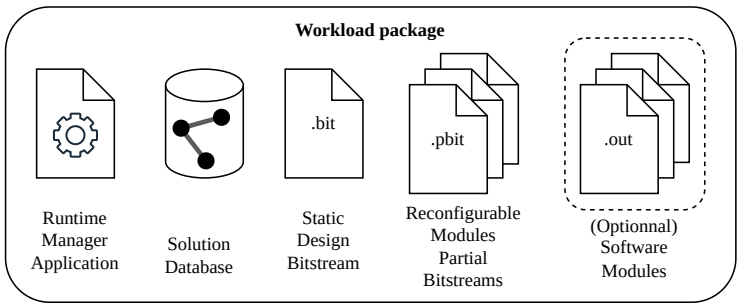


Figure 3.7: Content of the run-time workload package.

The Runtime Manager Application is the program responsible for fetching the best solution from the database to maximize the QoE and QoS scores at runtime. Once a mapping and scheduling solution from the solution database

has been identified, the corresponding reconfigurable modules (RMs) and software modules (SMs), if any, are used to execute the chosen mapping.

As mentioned in the quality model section, our runtime manager aims to avoid service breaks. Thus the runtime managers don't have to find the best solution in the database as long as respecting the application deadline and returning a non-null QoS solution is paramount. Finding solutions with a good QoE score is still important though, as once the service execution has been guaranteed, the runtime manager must maximize QoE.

In addition to the workload package, the self-reconfigurable system's FPGA must be properly set up using a static bitstream that contains the proposed architecture (see section 2.1.5). It is assumed that all reconfigurable modules' partial bitstreams that are considered in the application model must be provided by the application designers and have been correctly generated.

3.3.2.1 System monitoring

A new mapping and scheduling solution is needed for anytime the system cannot hold the application deadline. This can happen when the currently selected solution doesn't meet the deadline. We define here the causes of such delays, and the system monitoring implemented.

Hardware task execution times should not vary over time as their used logic elements have stable timings. Also, tasks are assumed to be atomic (no preemption on either hardware or software tasks as concluded in Section 2.1.4.2). Therefore once a task has its input data, we should not expect any delay other than what's been profiled. However, given the proposed architecture in section 2.1.5, shared resources can introduce execution delays due to congestion. In particular, the communication infrastructure management and shared memory accesses have been proven to be critical and has been the focus of many works to reduce congestion [27] or to predict communication delay [109].

An illustration of the source and targets of delays that cause discrepancies with the defined schedule is shown in Figure 3.8.

Any communication infrastructure approach that must be shared at some point can be confronted with the congestion problem, therefore discrepancies are expected. In addition, regardless of the communication infrastructure, access to the DDR memory by DMA is limited by the number and bandwidth of instantiated DMA controllers. Therefore some form of congestion is expected. While we do not aim to predict the level of congestion, we consider it as a form of constraint on tasks as they need to wait for their input to begin. By

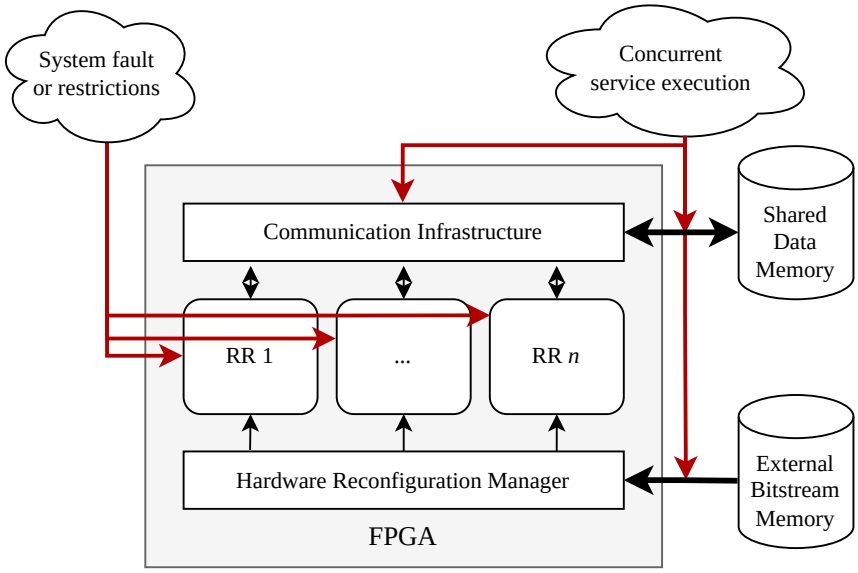


Figure 3.8: Illustration of the potential constraints on the FPGA resources (red arrows).

this logic, reconfiguration times can also vary slightly as bitstreams are stored on shared memory.

Finally, resources (RRs or CPUs) can become temporarily or permanently unavailable by the decision of the user or due to system faults. The user could typically manually implement a task on an RR that they wish to run permanently such as data flow services like encryption or signal filtering. Resources could either be fully faulty as a result of a gamma-ray exposition in aerospatial environments [4], or the user could restrict usage of one or more CPU or RR resources to reduce energy consumption or chip temperature, following their QoS model.

We define here two points of attention that needs to be monitored:

- Execution times and application deadlines;
- Resource restrictions (either from the user or system failure).

The targeted application deadline can be checked by a polling thread running in the CPU core dedicated to OS commands. It checks the state of execution after waiting for the expected application duration. To do so, it checks the content of a progression register of n bits for n tasks in the static design. When tasks executed in RRs are terminated, the bit that corresponds to their task index in the register is set to '1'. Similarly in software if tasks are executed on a CPU core, a size n register is stored in the software space.

At the end of the application deadline, those registers can be checked to see if all tasks have been executed. When OR-gating comparing those registers if not all bits have been set to ones at the end of the expected application duration, then we know the application wasn't fully executed by the end of the deadline.

Using this simple monitoring system, the system can know the tasks which are not set to '1' aren't finished yet. A resilience methodology could use this information to deduct the cause of this delay and introduce fault-mitigation capabilities. In the case of faulty resources, we believe that a statistical approach could take this decision. If mapping solutions repeatedly fail to execute tasks on the same resource, then the system could decide to stop using this resource until user intervention. However, as this type of resilience was not the main study of our work, it has not been implemented. We instead consider resource restriction coming from the user.

When the application terminates before the deadline, the difference between the pre-evaluated schedule duration time of the solution $\tau_{\mathcal{S}}$ and the real duration of the application $\tau'_{\mathcal{S}}$ gives the constraint level $\tau_{\mathcal{C}} = \tau'_{\mathcal{S}} - \tau_{\mathcal{S}}$. If $\tau_{\mathcal{C}}$ is positive, then the application took longer to execute than expected, and the level of constraints considered by the runtime manager needs to be updated. On the other hand, if $\tau_{\mathcal{C}}$ is negative, then there were less constraints than anticipated.

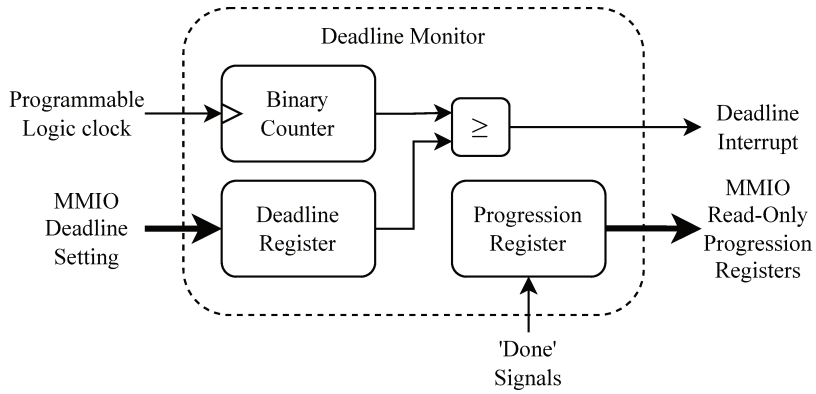


Figure 3.9: Hardware programmable deadline monitor.

This functionality can be implemented at a low logic elements cost on the programmable logic by implementing a binary counter (see Figure 3.9 for an illustration of the design). Memory-Mapped deadline and progression registers (MMIO) can be accessed by the CPU through AXI interfaces. When a new iteration of the application starts, the counter is reset and counts with each clock cycle of the programmable logic clock.

Whenever this counter reaches the set value in the deadline register, an interrupt is sent to the Run-time Manager Application. Upon termination, each task sends a 'Done' signal to the progression register, which sets the bit corresponding to the task ID (task t_1 sets bit 1). The progression register can be checked by the CPU to see the state of execution.

Other system or application specific metrics can be used as per the defined QoS model, such as temperature [110, 111] or energy consumption [75, 84] monitoring. In this chapter, without loss of genericity, we chose to use the application execution time deadline as it is simple to model and is coherent with the nature of our H.264 application benchmark.

The system monitoring pseudo-code is described in Algorithm 2 to illustrate the course of actions of our quality-oriented methodology.

Algorithm 2 System Monitoring pseudo-code

Input : workload package, deadline interrupt, user action
Output : call to the greedy-based run-time manager

```
1:  apply first scheduling solution
2:  set deadline register
3:  while true do
4:      wait for ( new set of constraints or deadline monitor interrupt )
5:      if deadline monitor interrupt then
6:          read Progression Register
7:          if all tasks aren't finished then // Service break → downgrade
8:              // Find a new solution with QoS score Q_S^S > 0
9:          else
10:             // Look for a solution maximizing QoE and QoS → upgrade
11:     if updated constraints then // Potential service break → downgrade?
12:         // Find a new solution with QoS score Q_S^S > 0
```

3.3.2.2 Greedy-based run-time manager

After receiving a call from the monitoring mechanism, a new schedule from the solution database needs to be found. Because the initial solutions were evaluated at design time without prior knowledge of the run-time system state, their fitness with the current constraints needs to be re-evaluated. Since the schedules and QoE scores have already been computed, this leaves the run-time manager to find which solutions fit best and maximize the QoE given the current state of the system.

The main interest of our approach is the capability of the run-time manager to automatically upgrade or downgrade the quality (and thus, compute-intensiveness) of the targeted application according to the runtime context. This behavior is the general case condition illustrated in line 10 of Algorithm 2.

Quality downgrades refer to the process of lowering the QoE to avoid service breaks. When the current implementation quality setting cannot hold the targeted application deadlines because of monitored constraints, a less resource-intensive setting is chosen. On the other hand, when the system's constraints are relaxed enough, an application upgrade aims to increase QoE.

The speed at which downgraded solutions are found by our run-time manager needs to be fast enough as once near-service breaks (i.e. when QoS is below a user-set threshold) are detected, the system must react accordingly. For instance, if the deadline trigger shows an incomplete application execution, the manager should aim to find a downgraded solution fast enough for it to be executed at the next application iteration.

Upgrades aim to increase the QoE first given the current level of constraints on the resources, then QoS. The upgrading process implies the current implementation is meeting the set deadlines and has a QoS score above the user-set threshold. So, as there is no urgency, the upgrading process can take more time to find a better solution.

The choice of whether the run-time manager downgrades or upgrades the application process is based on the monitored QoS score Q_S^S when the run-time manager is called. If the QoS score is non-null, then the run-time manager may upgrade to find a better QoE solution. If it is null, then the system is too constrained and a downgrade is required.

Because pre-computed scheduling solutions were computed on a system free of any constraints (i.e. for $\tau_{\mathcal{C}} = 0$), the heuristic needs to check if a selected solution has a better QoS than the one currently implemented. Once the decision to upgrade or downgrade has been made, the run-time manager uses a greedy-based search heuristic to find a new solution for the application that respects the constraint level, called a compatible solution.

A solution $\mathcal{S}_i$ is called compatible if the sum of the pre-computed schedule duration τ_{S_i} and the monitored constraint level $\tau_{\mathcal{C}}$ are inferior to the application deadline τ_D:

$$\tau_{S_i} + \tau_{\mathcal{C}} \leq \tau_D \tag{3.7}$$

and its QoS score is positive, i.e. $Q_S^S(\mathcal{S}_i; \tau_D) > 0$ using previously introduced QoS model in Equation 3.3. We remind here that the QoS score is not necessarily correlated with the execution times, hence the two constraints.

Figure 3.10 illustrates a downgrade and an upgrade operation. The top timeline illustrates the overall system resources occupancy caused by the monitored constraint level $\tau_{\mathcal{C}}$ and the execution of the solutions $\tau_{\mathcal{S}i}$ on the system resources. In this example, it is assumed that the constraint level $\tau_{\mathcal{C}}$ is constant. The bottom timeline illustrates the execution time spent by the run-time manager on the CPU looking for a solution in the database.

The first iteration runs a default solution $\mathcal{S}_0$ of the targeted application at an execution mode $\mathcal{M}_0$. This solution causes a service break, however, as once the deadline monitor sends an interrupt at the deadline mark τ_D, the application hasn't been terminated. This triggers the run-time manager which begins a downgrading process (denoted by $\mathcal{D}$). Let this process execute fast enough that a new solution $\mathcal{S}_1$ has been found such that $Q_S^S(\tau_{\tau_{\mathcal{C}}+\mathcal{S}1}) > 0$. Then the heuristic restored the continuity of service for the second iteration.

At the end of the second iteration execution, we illustrate a long upgrading process (denoted by $\mathcal{U}$) as the runtime manager is called to maximize QoE and QoS. This long term upgrade process starts right after $\mathcal{S}_1$ ends, and for example, spans up to nearly the end of the third application iteration. At the start of the third iteration, because the run-time manager hasn't returned its decision yet, the last valid solution is kept. Then the newly found solution $\mathcal{S}_2$ of the execution mode $\mathcal{M}_1$ is applied on the fourth iteration. In this example, as $Q_S^E(\mathcal{M}_1) > Q_S^E(\mathcal{M}_0)$, the runtime manager has upgraded the application.

The pseudo-code of the greedy-based heuristic is introduced Algorithm 3.

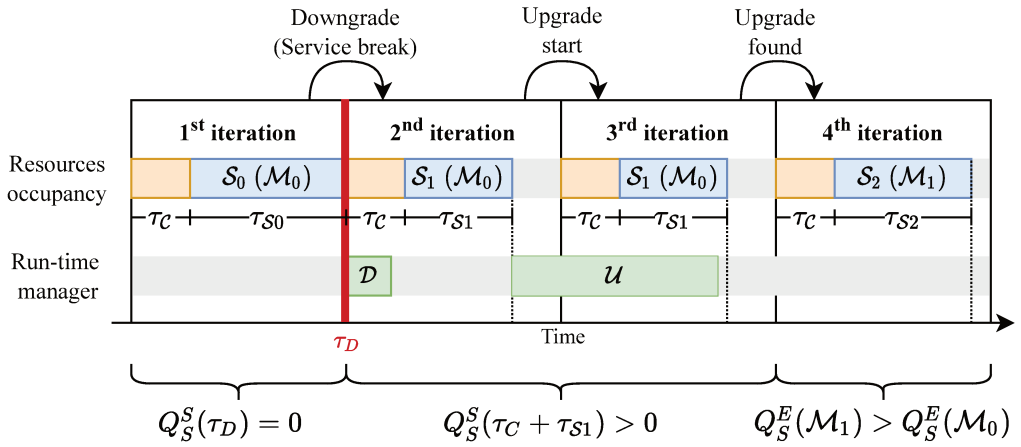


Figure 3.10: Illustration of downgrade $\mathcal{D}$ and upgrade $\mathcal{U}$ operations in the system timeline. $\mathcal{S}_0$, $\mathcal{S}_1$, and $\mathcal{S}_2$ denote different mapping and scheduling solutions of an application. $\mathcal{S}_0$ and $\mathcal{S}_1$ are solutions for the $\mathcal{M}_0$ execution mode, and $\mathcal{S}_2$ for $\mathcal{M}_1$.

In this heuristic, we must quickly compute the QoS given current constraints, as per lines 5 and 12. Those statements are the most time critical and need to be very fast for the heuristic to parse through the database as fast as possible.

Given the H.264 benchmark, as the QoS score has been defined as a function of a solution's schedule makespan, we add the constraints execution times τ_C to the pre-evaluated solution makespan to return a quick worst-case estimation of the solution's QoS.

Algorithm 3 Greedy-based search heuristic

Input : solution database, current context level, current QoS, and QoE score

Output : valid solution

```
1:  if downgrade then
2:      target QoE = current QoE score
3:      while no solution found do
4:          for new solution from database with QoE = Target QoE do
5:              compute QoS given the current constraints
6:              if new solution QoS > 0 then
7:                  return new solution
8:          decrease target QoE score to the next execution mode
9:  else if upgrade then
10:     while no compatible solution found do
11:         for new solution from database with QoE = Target QoE score do
12:             compute QoS given the current constraints
13:             if new solution QoS > 0 then
14:             if  (new QoS ≥ cur. QoS) or (target QoE > cur. QoE) then
15:                 return new solution
16:         increase target QoE score to the next best execution mode
```

This greedy-based heuristic stops once a compatible solution has been found. While greedy heuristics are known to be sub-optimal, it ensures a solution is yielded as soon as possible. We address this issue by incrementally maximizing the QoE and then QoS, therefore the greedy manager isn't stuck using a local optimum solution.

In case the manager doesn't find any compatible solution (cf. line 3 of Algorithm 3), then that means it has parsed all solutions from the database and the system is too constrained. Then the system is fully breaking the service in this worst-case scenario as it must compute a new schedule online. However, this situation is critical as the solutions in the database are among the best that were pre-evaluated. As a failsafe safety, the system can always

default to the solution with the highest QoS score in the database if the runtime manager doesn't return a new solution while experiencing a service break. This default solution is the less likely to fail the deadline, but at a cost of a low QoE score as per the trade off illustrated by the solution space in Figure 3.6.

3.4 Experiments

In this section, we evaluate our quality-oriented hybrid management methodology. The goal of those experiments is to:

- highlight how quality, through the proposed QoE and QoS models for our benchmark H.264 encoder application, is maximized over time;
- show our approach's capability to minimize service breaks;
- evaluate the overheads of the approach on the targeted platform.

3.4.1 Platform evaluation

The functional architecture is illustrated Figure 3.11 and is representative of the literature (see Section 2.1.5). The targeted board is a PYNQ-Z2 development board [12] which embeds a Xilinx Zynq-7000 SoC featuring 85K logic cells Artix-7 FPGA and a dual-core ARM Cortex A9 CPU. Such SoC is commonly found in the industry thanks to its cost-effective performance.

From this architecture, the following metrics have been acquired from the H.264 application benchmark: execution times of tasks on RRs, profiled reconfiguration times of RRs, and average communication time delay. These metrics are used to design a simulation environment to benchmark our methodology

Tasks execution times of the H.264 encoder benchmark application were measured on target using Xilinx's Integrated Logic Analyzer (ILA). The frequency of the clock used in this design is 100MHz. The measures from the ILA start upon reception of all input data and end after having finished processing and sending all output data. The execution times of the software implementation of the encoder's tasks were measured on the SoC CPU using the `time.h` library. Execution times of the profiled tasks are shown in Table 3.6.

It is to be noticed that RRs were large enough so that all tasks that can be executed in hardware can do so on all RRs in the design. This is not necessarily the case as we consider RRs of heterogeneous size in our design, as concluded in Section 2.1.3.1. In the case presented here, this is a consequence

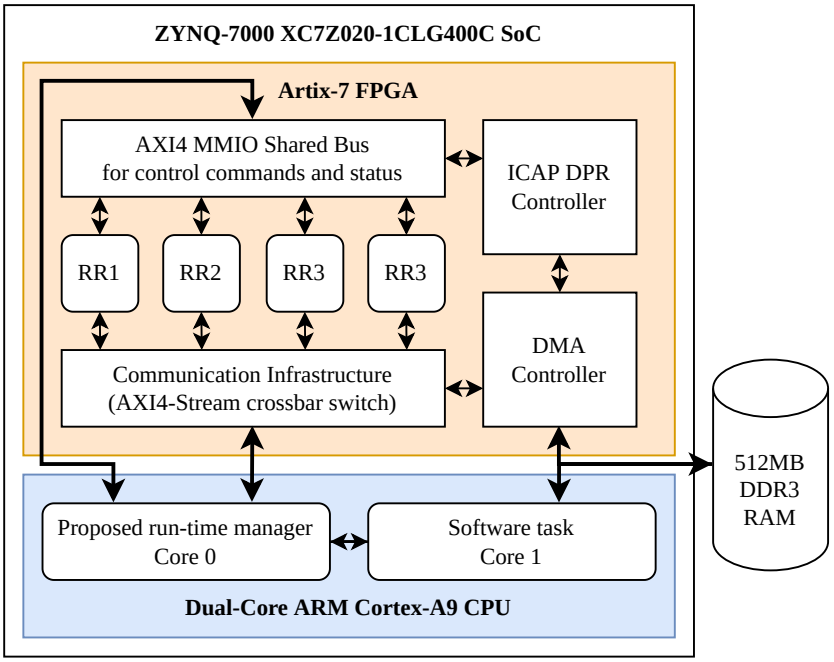


Figure 3.11: Considered architecture for the experiments (cf. Section 2.1.5).

of not optimizing the usage of logic gates in each RR, thus the number of used resources for any task is less than the number of resources in RR1, which is the smaller RR in this design (cf. Table 3.7). As partial bitstreams weigh the same regardless of the logic elements used in the RRs, we recommend making the most out of the RRs' available resources.

Table 3.7 lists the bitstream size of the considered RRs and the corresponding reconfiguration times τ_R profiled using the 1GBps ICAP controller uPaRC [20] data and guidelines. Because of the heterogeneous RR design choice made in Section 2.1.3.1, bitstream relocation cannot be practiced here. There is one bitstream for each different hardware implementable task from Table 3.6, and there are two sets of those corresponding to the two targeted resolution settings. Thus the total weight of the uncompressed bitstreams to store on target is 37.95MB. With compression, this size drops to 9.94MB.

Table 3.8 introduces the profiled worst-case delay on the communication interface. Those were measured using a task transmitting a full output FIFO content (1024×32b words) from a source in the design to a destination. The intra-FPGA RR to RR delay is caused by the AXI4 Stream switch. This Xilinx IP uses a round-robin arbiter. In the worst-case scenario, the source is last in the round-robin arbitration order, and all 4 RRs and the DMA need to send their 1024 words. The FPGA-CPU cross-domain communication (RR

Table 3.6: On-target profiled execution times of H.264 application tasks in hardware and software. Sobel Filter can only be executed in software, as Macroblock Ordering and AES Encryption can only be executed in hardware.

	Execution time (in ms)		
Task	Hardware	Software	Resolution
Sobel Filter	-	3.630	360p
Macroblock Ordering	0.332	-	
H.264 Slice Encoder	0.540	18.720	
Entropy Encoder	2.916	5.590	
AES Encryption	1.289	-	
Sobel Filter	-	5.842	480p
Macroblock Ordering	0.568	-	
H.264 Slice Encoder	0.960	30.445	
Entropy Encoder	5.184	11.803	
AES Encryption	2.579	-	

Table 3.7: Bitstream size and reconfiguration time τ_R by RRs profiled with 1GBps ICAP Controller from [20] with compression using X-MatchPRO algorithm. The full static design bitstream is provided for comparison with the size of the partial bitstream.

Region	Uncompressed size	Compressed bit size	τ_R
RR1	895 KB	231 KB	0.89 ms
RR2	1.18 MB	314 KB	1.21 ms
RR3	1.46 MB	386 KB	1.49 ms
RR4	1.23 MB	327 KB	1.26 ms
Full static design	3.85 MB	-	-

Table 3.8: Profiled worst-case delay on the communication interface using the AXI4 Stream interconnect.

Source	Destination	Worst-case delay (ms)
RR	RR	0.06
RR	CPU	0.30
CPU	RR	0.30
CPU	CPU	$<$0.01

to CPU and back) has a higher delay caused by the DMA controller IP. In the worst-case, all other RRs are sending their word to the CPU first, which causes DMA controller and switch congestion. The intra-CPU communication delay is negligible as software tasks can access DDR memory directly.

Finally, the design-time steps of our methodology have been executed on a host computer (Intel Quad-Core i5-7300U 2.60Ghz, 8GB DDR4 RAM). Given the targeted application, execution modes, and architecture, 1'156'242 solutions have been generated and pre-evaluated in 110s. After the solution-space reduction step, 15.8k solutions were kept in the solution database, achieving a 98.63% reduction. The composition of the solution database is shown in Table 3.5 introduced earlier. The resulting database has been stored in a `JSON` file, which size to load on target is 1.8MB. Considering all elements in the workload package, the latter weighs a total of 14.49MB to be stored on target, representing 2.93% of the DDR.

3.4.2 Simulation environment

Once the metrics have been profiled from the architecture, we built a simulation environment to benchmark and stress-test our run-time manager using those values. The motivation for simulating the environment was to introduce constraints level scenarios to resources. This is used to benchmark the methodology, as it otherwise requires development works on uPaRC's integration to FOS using the functional implementation introduced in Section 2.1.5.

An illustration of the simulation environment is given in Figure 3.12. This environment, including the run-time manager, has been written in `C++` and executed on the target's CPU.

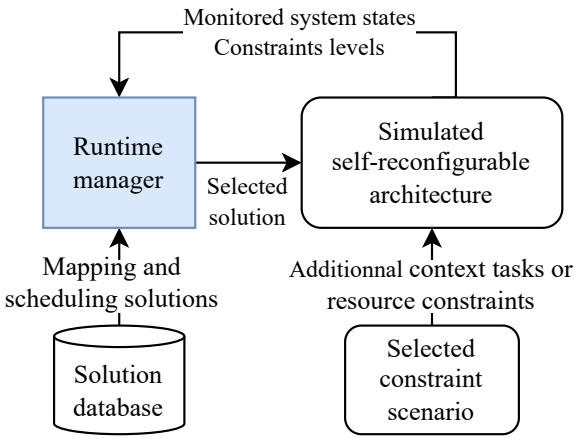


Figure 3.12: Simulation environment of the run-time manager.

In our experiments, constraints refer to the additional services or applications that share the resources with the targeted application or the monitored latency overhead. The constraint level for each resource denotes the amount of time in the schedule we consider to be completely unavailable for the application. To reflect the unpredictability of such constraints and test the resilience of our approach against changes in resource availability, we modeled random constraint levels. We consider two test scenarios:

1. **random constraint levels** where each resource gets independently assigned a constraint level;
2. **restriction of service** where a random number of resources are restricted from usage to execute the targeted application.

When generating random constraint levels, we used a uniform law taking values (in ms) between 0 and 75% of the deadline to guarantee a minimum of 25% of resource occupancy saved for the application. Similarly, the maximum number of resources that can be reserved is 4 out of 5 (4RR+1CPU), since some solutions can execute the H.264 at a low QoE setting with only one RR. This guarantees that a minimum service can be ensured and our approach tries not find a solution to an insolvable problem (as mentioned at the end of Section 3.3.2.2).

Each scenario is studied in an 'aggressive' and a 'periodic' manner with respectively new constraint levels values for each iteration, and every 10 iterations. The 'aggressive' scenarios are much more demanding for the system as the values vary a lot between two iterations. It makes for a good stress test as the system has to make lots of decisions to guarantee the service. The 'periodic' scenarios are more realistic in the sense that decisions from the user (or its mission management application) to increase or reduce the number of context services shouldn't happen this frequently, and typical system latencies don't vary as much in comparison.

In the following paragraphs, we compare the greedy-based search heuristic to an exhaustive search. The first goal is to determine how close our approach is to the optimal solution in terms of QoE score. The second goal is to show the capability of the approach to guarantee continuity of service execution in constrained scenarios.

3.4.3 Resulting quality scores

Figure 3.13 shows the evolution of the QoE score on 50 iterations of the greedy-based heuristic compared to the exhaustive search respectively for the

(a) 'aggressive' and (b) 'periodic' constraint scenarios.

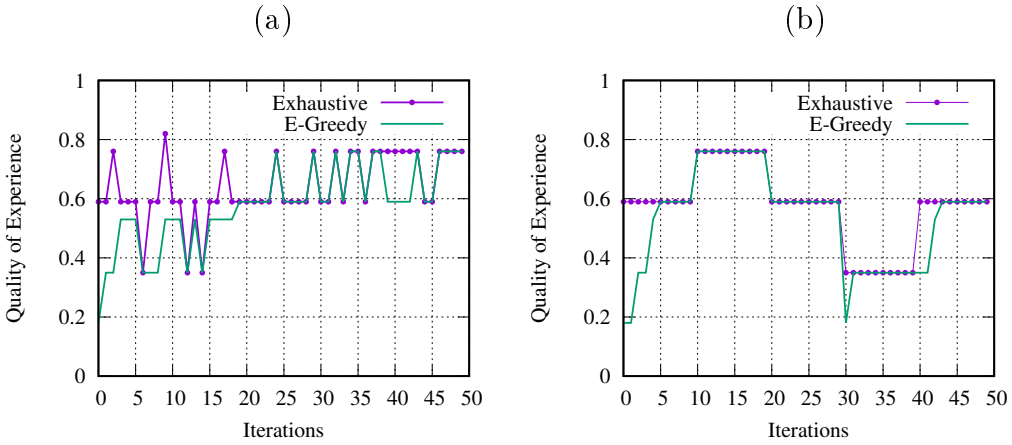


Figure 3.13: H.264 QoE score evolution of the greedy-based heuristic versus the Exhaustive search for the (a) 'aggressive' and the (b) 'periodic' random constraints scenarios.

The aggressive scenario introduces more changes and the greedy-based cannot precisely find every best solution. This is illustrated during iterations 38 to 43 where the QoE score is lower than optimal. As the greedy-based approach do not scan the whole solution database to find the appropriate solution, it returns a sub-optimal one.

In contrast, the periodic scenario makes the greedy-based heuristic find a solution with the optimal QoE score more often. When a context constraint is introduced, it can take a few iterations before finding an optimal QoE solution. Similarly, when downgrading the heuristic can downgrade the application too much as illustrated in iteration 30 of Figure 3.13 (b) as it pick the first feasible solution.

In Figure 3.14, we show the restriction of service scenario resulting QoE score evolution. In this scenario, entire resources can be restricted from usage for the targeted application. Thus a lot of solutions from the database are unusable and make the search harder for the heuristic. In the aggressive scenario Figure 3.14 (a), the greedy-based heuristic rarely reaches the optimal QoE score as it relies on whether the heuristic immediately finds one or not. In the periodic scenario (b), it has more time to process more solutions but still fails regularly to reach the optimal score.

Table 3.9 summarizes the obtained simulation results using the greedy-based approach. These are the average QoE score on the four studied test scenarios. The heuristic yields a better QoE score in the periodic scenar-

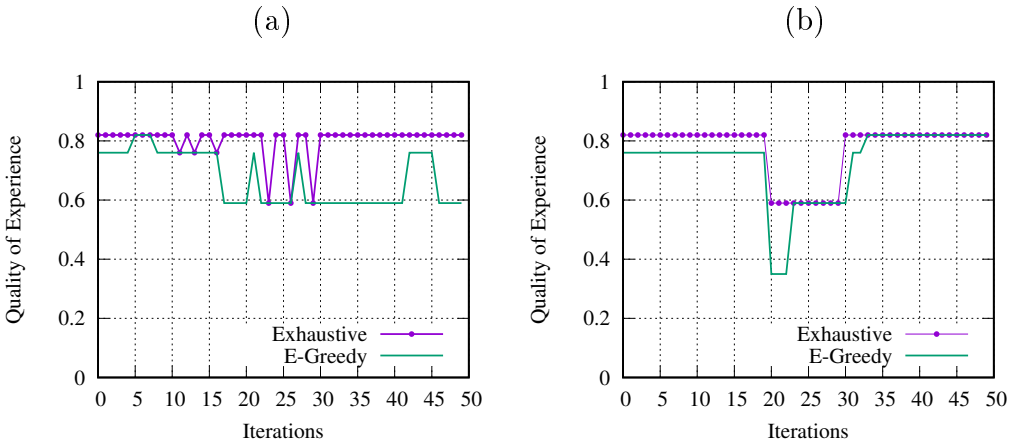


Figure 3.14: H.264 QoE score evolution of the greedy-based heuristic versus the Exhaustive search for the 'aggressive' (a) and the 'periodic' (b) restriction of service scenario.

ios than in the Aggressive ones. This is because the heuristic is in a more demanding environment where it can experience downgrade situations more frequently. As a consequence, it sometimes defaults to downgrade the QoE to find a suitable solution more easily, even if the exhaustive search proves there is a working solution while keeping the same execution mode. In the periodic scenario, however, the environment is a bit more relaxed and after a few iterations, the heuristic eventually finds a near-optimal solution by gradually upgrading the QoE.

Table 3.9: Average relative QoE score on 100k iterations in percentage of the average optimal score.

Scenario	Aggressive	Periodic
Random constraint	90.52%	93.76%
Restriction of service	81.37%	82.68%

The difference between random constraints and restriction of service is explained by the brutal changes brought by the latter. In restriction of service, more solutions from the database suddenly become infeasible due to the restriction of the resource(s) availability. This causes more frequent downgrade decisions from the heuristic, and a longer time before finding a better QoE solution when upgrading later on.

Finally, because the QoS varies a lot and we can only compare QoS at equal QoE score, we highlight the QoS evolution over a few iterations of the

random 'periodic' constraints scenario in Figure 3.15. At iterations 28 and 29, we see in inset (a) that the QoE score was stable at 0.59 as it had a few iterations to stabilize. This is also reflected in inset (b) with the QoS score sitting at 0.44.

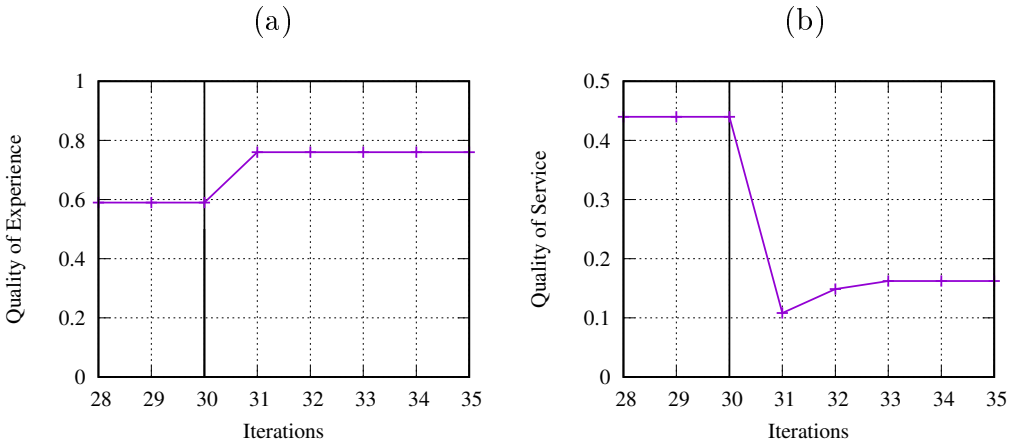


Figure 3.15: H.264 QoS score evolution (b) upon application of new constraint levels at iteration 30, with QoE evolution (a) as reference.

While constraints are relaxed at iteration 30, the run-time manager can upgrade the application at iteration 31 as the QoE score increases. In consequence, the QoS score changes as it is a new solution. Over the following iterations, the run-time manager changes solutions twice to increase QoS in iterations 32 and 33.

3.4.4 Resulting decision times

The average decision times measured on the targeted SoC's ARM A9 CPU for 100k iterations of the greedy-based heuristic and the exhaustive search are highlighted in Table 3.10.

Table 3.10: Average decision times of the greedy-based and exhaustive approaches on 100k iterations.

	Decision time (ms)
Greedy-based	0.28 ms
Exhaustive	5.29 ms

As expected, the greedy heuristic is much faster than the Exhaustive algorithm. To put those decision times in perspective, an ASAP scheduling

heuristic [90] yields schedules for the H.264 application within 0.24ms on average. While this is slightly faster than the average greedy-based search heuristic, computing a schedule online does not guarantee finding a solution that meets the application deadline at the given execution mode. Typically, in downgrade situations, this would mean a schedule is to be computed for each execution mode going downward from the starting execution mode until a valid schedule is computed.

In addition, online scheduling requires specific heuristics with low time complexity, which are generally outperformed by more compute-intensive algorithms that can be used at design-time [112].

To check how our approach is capable of respecting continuity of service execution, we measured how often the decision time overhead makes the application break the deadline. In other words, we check the inequality 3.8 for each iteration.

$$\tau_\epsilon + \tau_\mathcal{A} + \tau_C \overset{?}{<} \tau_D \tag{3.8}$$

With τ_ϵ as the decision time, $\tau_\mathcal{S}$ as the schedule duration of the selected application's solutions, τ_C as the overhead brought by constraints, and τ_D as the deadline:

We introduce Table 3.11 the percentages of downgrade iterations resulting in success or failure, 100% being the ideal case where continuity of service is always observed. 'In time' means the previous inequality is respected. 'Late' means a solution has been found, but because the found solution didn't have time to execute before the deadline, a single iteration has been dropped and the service has been restored at the next iteration. Considering our scenarios, lateness cannot be studied for Aggressive scenarios as constraints change every iteration, and the heuristic needs to adapt to those immediately. Finally, 'Failed' means that the run-time manager wasn't able to find a new valid solution before at least two iterations. Given the constraint scenarios, it is guaranteed there is at least one solution that can be found by the exhaustive search. Therefore the approach was always capable of finding a solution.

In the H.264 benchmark situation, a late solution implies a dropped frame which is not very noticeable by users [106], and a failed situation translates to multiple frames at once. This can be acceptable depending on the application constraints.

In the random constraints scenarios, our run-time manager successfully finds a valid solution in 94% of the cases. In the case of the periodic random scenario, 5.20% downgrade situations were solved late by one iteration,

Table 3.11: Resulting continuity of service execution checks on downgrades for the proposed approach.

	Aggressive Random	Periodic Random	Aggressive Restriction	Periodic Restriction
In time	94.03%	94.29%	61.69%	61.87%
Late	-	5.20%	-	21.15%
Failed	5.97%	0.51%	38.31%	16.98%

bringing the continuity of service ratio on downgrade situations to 99.49%.

The restriction of service scenario is a harder situation for the search heuristic as the resource availability changes are more brutal. In consequence, those scenarios show worse results than the random constraint ones with 61.87% of in-time solutions. Late results were registered in 21.15% of the situation, bringing the continuity of service ratio to a maximum of 83.02%.

To increase this continuity of service rate, we believe that organizing the database solutions in a tree structure could help quicken the search. The main cause of the lower continuity rate in the restriction of service scenario is that the runtime manager spends time evaluating solutions that aren't compatible with the given constraints. Organizing the solutions in a tree structure where each branch answers a specific type of constraint could help reach compatible solutions in fewer steps.

Another method could be to use a statistical approach or learning-based approaches to identify particular solutions of the hybrid method. These particular solutions would be those that answer common levels of constraints and could be selected more frequently by the methodology to reduce the decision time.

Compared to [39] that used space-multiplexing of modules on a RR grid, we made use of time-multiplexing of tasks on RRs of heterogeneous sizes and CPUs, while managing the reconfiguration processes through a dedicated fixed-mapping scheduler. Our approach maximizes the quality of experience through upgrades and downgrades of execution modes of self-reconfigurable systems, as in [83]. However, this work didn't make use of quality models and focused on discrete control of the application. Moreover, the homogeneous RRs size in their work does not correlate with our conclusions on the island-styles architectures for partially reconfigurable FPGAs in Section 2.1.3.1.

Finally, works from [76] in the MPSoC domain made use of a reward model to compute schedules on multi-core CPUs. This reward model could be compared to our QoE model as their goal was to maximize the reward

score of a schedule while holding a deadline. However, this work doesn't target self-reconfigurable systems.

3.5 Conclusion

In this chapter, we introduced our novel hybrid design-time/run-time quality-oriented manager. Its goals are to maximize the Quality of Experience and Quality of Service through models defined at design time and to minimize the number of service breaks when the system is constrained. To do so, our proposed methodology computes mappings on a host computer using combinatorial formulas. It then compute schedules for the targeted application, considering the specifications of our self-reconfigurable system. The size of the resulting solution space has been reduced using mapping constraints and pruning after schedule evaluation. A workload package is then generated, containing the pre-computed solutions and the partial bitstreams. Those solutions are sorted by execution modes, which are different functional implementations of the targeted application. They are ranked by QoE, under the hypothesis that a lower QoE is less compute-intensive than a higher one.

Once the workload package has been pre-computed, it is loaded on the target for the run-time steps of our methodology. The latter consists in monitoring the current state of the system to verify if it can hold a high level of QoS, which is an image of how well the service is currently being executed. If the QoS level is too low and the application risks a service break, then our run-time manager can find a new applicable solution.

Using metrics obtained from the implementation of a H.264 encoder application, we ran a simulation of different constraint scenarios on the target's resources. At the end of the design time phase, a total of 1.15M mappings have been identified. Using the proposed constraints, 35.8k mappings have been considered worthy to evaluate using the fixed-mapping scheduler. Finally, the final solution database contained 15.8k solutions for the runtime manager. This database accounted for 1.8MB to store on the target's RAM, in addition to the 12.69MB of compressed bitstreams. This adds up to 14.49MB, which represents 2.83% of the target's RAM.

The proposed hybrid methodology achieved continuity of service by finding valid solutions on time in 94% of the cases when resources are shared with context services and 62% when entire RR(s) are restricted from usage by the user. Those results go respectively as high as 98.20% and 83.02% on soft real-time constraints.

Chapter 4

Runtime scheduling for self-reconfigurable systems

Contents

4.1 Overview

In chapter 3, we used a hybrid design-time and runtime methodology to manage the targeted self-reconfigurable system. Experiments show we were able to maximize the quality of experience with a decision time small enough to guarantee the service execution thanks to a greedy search engine parsing a pre-computed set of solutions.

However, this comes at a cost of storing a mapping and scheduling solution database. The latter grows exponentially with the number of tasks in the application, RRs in the architecture, and quality-related execution modes. This can impact the performance of the proposed approach in two ways: increased solution database size in the system's memory, and as a consequence longer decision time to find a solution in the database. Although reduction of the solution space has been considered to minimize those impacts, it comes with its drawback of removing potential solutions that can be used at runtime.

In this chapter, we tackle this flexibility issue by adapting a state-of-the-art scheduling algorithm used in multi/many-core architectures to compute schedules for self-reconfigurable systems. The resulting scheduling algorithm keeps a good trade-off between time complexity and efficiency as it yields schedules that are shorter than current state-of-the-art approach.

4.2 List-based PEFT scheduling heuristic

In this section, we introduce the list-based Predict Earliest Finish Time (PEFT) scheduling heuristic [88]. This heuristic has historically been developed within the context of heterogeneous multi/many-core architectures such as ARM big.LITTLE, where execution times of tasks vary depending on the processor it has been assigned. As it has been shown in the literature, PEFT is one of the best-performing scheduling heuristics for heterogeneous processors [88, 89].

This is particularly of interest for our self-reconfigurable system as from a high level of abstraction, RRs can be abstracted as heterogeneous processors. As considered self-reconfigurable systems, introduced Section 2.1.5, make use of heterogeneous RRs, the execution times of implemented tasks in the FPGA exhibit similar heterogeneous processors behavior. Therefore PEFT can be applied to the scheduling problem for self-reconfigurable systems.

The main interest of this heuristic is its trade-off between its performance, in terms of application makespan reduction, and its $\mathcal{O}(n^2 \cdot p)$ (for n tasks and p resources) time complexity.

Its performance comes from its ability to foresee the impact of scheduling

decisions on the resulting application schedule duration, before beginning the scheduling steps. Such ability is also highlighted in the Lookahead heuristic [97], however the latter yields a $\mathcal{O}(n^4 \cdot p)$ time complexity.

An overview of the PEFT runtime scheduling heuristic is shown in Figure 4.1. It is based on an Optimistic Cost Table (OCT) which is a matrix containing all shortest paths to the exit task (or sink node) and acts as the prediction engine for this heuristic. From this table, we can compute ranks for each task, which relates to how impactful is each task on the shortest paths. A topological order can be obtained from these ranks. Finally, the tasks are then scheduled using the Earliest Finish Time (EFT) strategy, taking into account the prediction table, until there are no more tasks to schedule. As PEFT is a list-based scheduling heuristic, it finds both a mapping and its corresponding schedule.

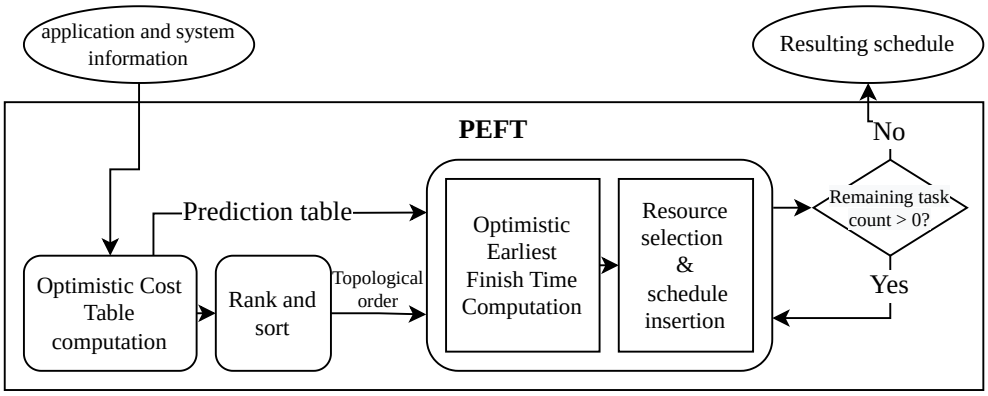


Figure 4.1: Overview of the PEFT scheduling heuristic.

4.2.1 Optimistic Cost Table

The Optimistic Cost Table (OCT) is a matrix that the PEFT heuristic uses to predict the impact of a single task scheduling decision on the resulting schedule. The OCT table is notably used to define a topological order that prioritizes tasks that have the biggest impact on the schedule and could potentially cause resource usage bottlenecks.

The OCT is computed in such a way that critical paths (in execution times) are evaluated for each task, and for each resource that the task is possibly being implemented on. This gives valuable information on what is left to schedule for the heuristic.

The OCT table is a matrix of size $n \times p$ for n tasks and p resources. Each value $OCT(t_i; \mathcal{P}_j)$ denotes the critical path from task t_i to the sink node,

considering the task t_i assigned to the resource $\mathcal{P}_j$. We remind here that the PEFT scheduling heuristic was originally developed for MPSoC architectures. Adaptations for self-reconfigurable systems are covered in Section 4.3.

The OCT computation considers and evaluate all task-resource implementations for schedule makespan prediction purpose. OCT values being indications of scheduling impact, they are not a perfect reflection of the final impacts.

Formally, for a task t_i and a resource $\mathcal{P}_j$, $OCT(t_i;\mathcal{P}_j)$ is defined recursively from the sink to source task in Equation 4.1, with parameters from Table 2.5.

$$OCT(t_i;\mathcal{P}_j) = \underset{t_j \in \text{ succ}(t_i)}{\text{Max}} \left[\underset{\mathcal{P}_w \in \mathcal{P}}{\text{Min}} \left\{ \begin{array}{l} OCT(t_j;\mathcal{P}_w) + W(t_j;\mathcal{P}_w) \\ +A(t_i;t_j) \cdot C(\mathcal{P}_j;\mathcal{P}_w) \end{array} \right\} \right] \tag{4.1}$$

For each successor task t_j of a given task t_i, the formula computes the shortest paths to the exit task (backward propagation) for each resource $\mathcal{P}_w \in P$ that t_j can be assigned to. The paths are evaluated in such a way that data dependencies and communication latencies are comprised in the evaluated paths. Element $OCT(t_i;\mathcal{P}_j)$ is then equal to the maximum of those shortest paths.

We illustrate Figure 4.2 an example of OCT table computation using the canonical workload using adjacency and computation matrices A and W (cf. Equation 4.2). In this canonical example, we consider a set $\mathcal{P}$ of three CPUs, and communication latencies between resources $C(\mathcal{P}_i;\mathcal{P}_j)$ being equal to 1 if $\mathcal{P}_i \neq \mathcal{P}_j$. The resulting OCT table of this example is given in Equation 4.2:

$$A = \begin{array}{c} \begin{array}{cccccccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & t_{10} \end{array} \\ \begin{pmatrix} \cdot & 17 & 31 & 29 & 13 & 7 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 3 & 30 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 16 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 11 & 7 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 57 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 5 & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 9 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 42 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 7 & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{array}{l} -t_1 \\ -t_2 \\ -t_3 \\ -t_4 \\ -t_5 \\ -t_6 \\ -t_7 \\ -t_8 \\ -t_9 \\ -t_{10} \end{array} \end{array} \quad W = \begin{array}{c} \begin{array}{ccc} \mathcal{P}_0 & \mathcal{P}_1 & \mathcal{P}_2 \end{array} \\ \begin{pmatrix} 22 & 21 & 36 \\ 22 & 18 & 18 \\ 32 & 27 & 43 \\ 7 & 10 & 4 \\ 29 & 27 & 35 \\ 26 & 17 & 24 \\ 14 & 25 & 30 \\ 29 & 23 & 36 \\ 15 & 21 & 8 \\ 13 & 16 & 33 \end{pmatrix} \begin{array}{l} -t_1 \\ -t_2 \\ -t_3 \\ -t_4 \\ -t_5 \\ -t_6 \\ -t_7 \\ -t_8 \\ -t_9 \\ -t_{10} \end{array} \end{array} \quad OCT = \begin{array}{c} \begin{array}{ccc} \mathcal{P}_0 & \mathcal{P}_1 & \mathcal{P}_2 \end{array} \\ \begin{pmatrix} 64 & 68 & 86 \\ 42 & 39 & 42 \\ 27 & 41 & 43 \\ 42 & 39 & 50 \\ 28 & 37 & 28 \\ 42 & 39 & 44 \\ 13 & 16 & 22 \\ 13 & 16 & 33 \\ 13 & 16 & 20 \\ 0 & 0 & 0 \end{pmatrix} \begin{array}{l} -t_1 \\ -t_2 \\ -t_3 \\ -t_4 \\ -t_5 \\ -t_6 \\ -t_7 \\ -t_8 \\ -t_9 \\ -t_{10} \end{array} \end{array} \tag{4.2}$$

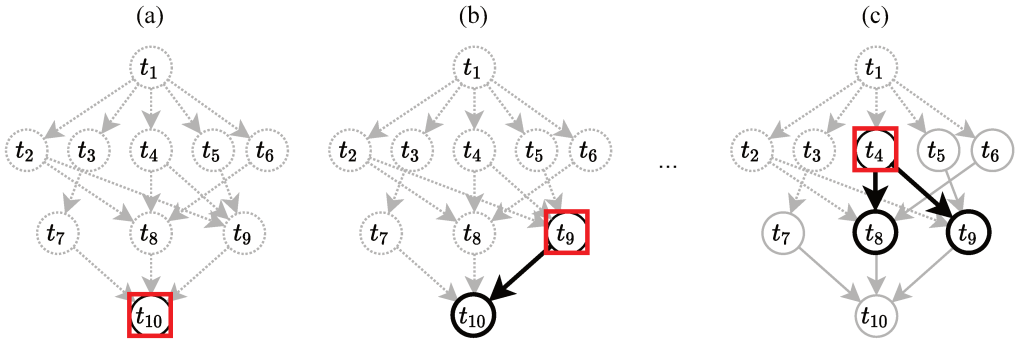


Figure 4.2: Example cases of OCT computation.

The OCT table is computed recursively by evaluating the paths of the first level of successors of each task. Sink tasks (i.e. t_{10} in the canonical example) don't have any successor per definition, cf. Figure 4.2 (a) where t_{10} is the sink task. Therefore OCT values of a sink task are always zero for any resource.

As an example in Figure 4.2 (b), the only successor of t_9 is t_{10}. There are three paths corresponding to t_{10} being assigned to $\mathcal{P}_0$, $\mathcal{P}_1$ or $\mathcal{P}_2$. The OCT value for t_9 being assigned to $\mathcal{P}_0$ is:

$$
\begin{aligned}
&OCT(t_9;\mathcal{P}_0) = \mathbf{Max} \\
&\left[\mathbf{Min} \left\{ \begin{array}{l}
OCT(t_{10};\mathcal{P}_0) + W(t_{10};\mathcal{P}_0) + A(t_9;t_{10}) \cdot C(\mathcal{P}_0;\mathcal{P}_0) = 0 + 13 + 7 \cdot 0 = 13 \\
OCT(t_{10};\mathcal{P}_1) + W(t_{10};\mathcal{P}_1) + A(t_9;t_{10}) \cdot C(\mathcal{P}_0;\mathcal{P}_1) = 0 + 16 + 7 \cdot 1 = 23 \\
OCT(t_{10};\mathcal{P}_2) + W(t_{10};\mathcal{P}_2) + A(t_9;t_{10}) \cdot C(\mathcal{P}_0;\mathcal{P}_2) = 0 + 33 + 7 \cdot 1 = 40
\end{array} \right. \right] \\
&= \mathbf{Max}[\mathbf{Min}\{13;23;40\}] = 13
\end{aligned}
\tag{4.3}
$$

Finally, in inset (c) we illustrate the case of $OCT(t_4;\mathcal{P}_0)$ (i.e. impact of t_4 assigned to $\mathcal{P}_0$) with multiple successors. In this case there is a total of six paths to evaluate in Equation 4.4: 3 task-resource assignment to evaluate for the 2 successor tasks of t_4: t_8 and t_9.

$$
\begin{aligned}
&OCT(t_4;\mathcal{P}_0) = \text{Max} \\
&\left[\begin{array}{l}
\text{Min} \left\{ \begin{array}{l}
OCT(t_8;\mathcal{P}_0) + W(t_8;\mathcal{P}_0) + A(t_4;t_8)\cdot C(\mathcal{P}_0;\mathcal{P}_0) = 13+29+11\cdot 0 = 42 \\
OCT(t_8;\mathcal{P}_1) + W(t_8;\mathcal{P}_1) + A(t_4;t_8)\cdot C(\mathcal{P}_0;\mathcal{P}_1) = 16+23+11\cdot 1 = 50 \\
OCT(t_8;\mathcal{P}_2) + W(t_8;\mathcal{P}_2) + A(t_4;t_8)\cdot C(\mathcal{P}_0;\mathcal{P}_2) = 33+36+11\cdot 1 = 80
\end{array} \right. , \\
\text{Min} \left\{ \begin{array}{l}
OCT(t_9;\mathcal{P}_0) + W(t_9;\mathcal{P}_0) + A(t_4;t_9)\cdot C(\mathcal{P}_0;\mathcal{P}_0) = 13+15+7\cdot 0 = 28 \\
OCT(t_9;\mathcal{P}_1) + W(t_9;\mathcal{P}_1) + A(t_4;t_9)\cdot C(\mathcal{P}_0;\mathcal{P}_1) = 16+21+7\cdot 1 = 44 \\
OCT(t_9;\mathcal{P}_2) + W(t_9;\mathcal{P}_2) + A(t_4;t_9)\cdot C(\mathcal{P}_0;\mathcal{P}_2) = 20+8+7\cdot 1 = 35
\end{array} \right.
\end{array} \right] \\
&= \text{Max}[\text{Min}\{42;50;80\};\text{Min}\{28;44;35\}] = \text{Max}[42;28] = 42
\end{aligned}
\tag{4.4}
$$

In practice, the number of paths to evaluate each value $OCT(t_i;\mathcal{P})$ table is equal to the number of successors tasks of t_i times the number of resources p. Hence the maximum number of OCT evaluations is $n^2 \cdot p$.

Once the OCT table is computed, we can obtain the OCT rank of each task using Equation 4.5, by averaging the OCT of each task.

$$
rank_{OCT}(t_i) = \frac{\sum_{k=0}^{p-1} OCT(t_i;\mathcal{P}_k)}{p} \tag{4.5}
$$

The rank vector can be used to get a topological sort of tasks in which they will be scheduled. In the canonical example, the resulting decreasing rank topological sort is $\{t_1;t_4;t_6;t_2;t_3;t_5;t_8;t_7;t_9;t_{10}\}$. Finally, the resulting OCT rank vector from the canonical example is introduced in Equation 4.6:

$$
rank_{OCT} = \begin{pmatrix} 72.7 \\ 41 \\ 37 \\ 43.7 \\ 31 \\ 41.7 \\ 17 \\ 20.7 \\ 16.3 \\ 0 \end{pmatrix} \begin{matrix} -t_1 \\ -t_2 \\ -t_3 \\ -t_4 \\ -t_5 \\ -t_6 \\ -t_7 \\ -t_8 \\ -t_9 \\ -t_{10} \end{matrix} \tag{4.6}
$$

4.2.2 Optimistic Earliest Finish Time

Once the OCT table and rank vector has been computed, the heuristic enters the task-scheduling phase. In this phase, tasks are processed in the decreasing

OCT rank topological order.

The mapping of a task t_i intends to minimize the Optimistic Earliest Finish Time O_{EFT}. Instead of choosing whichever resource minimizes the time at which task t_i finishes, the O_{EFT} formula represents how much it costs (in terms of schedule makespan) to implement it on a resource. O_{EFT} for a task t_i on a resource $\mathcal{P}_j$ is defined as:

$$O_{EFT}(t_i; \mathcal{P}_j) = EFT(t_i; \mathcal{P}_j) + OCT(t_i; \mathcal{P}_j) \tag{4.7}$$

With $EFT(t_i; \mathcal{P}_j)$ being the Earliest Finish Time (EFT) at which a task t_i can be terminated on a resource $\mathcal{P}_j$ given the task graph execution dependencies.

A task can be inserted in the current schedule (insertion policy) as illustrated in Figure 4.3 for the task t_4. It can be done so if there is enough idle time in a resource's schedule between tasks execution, also known as insertion windows.

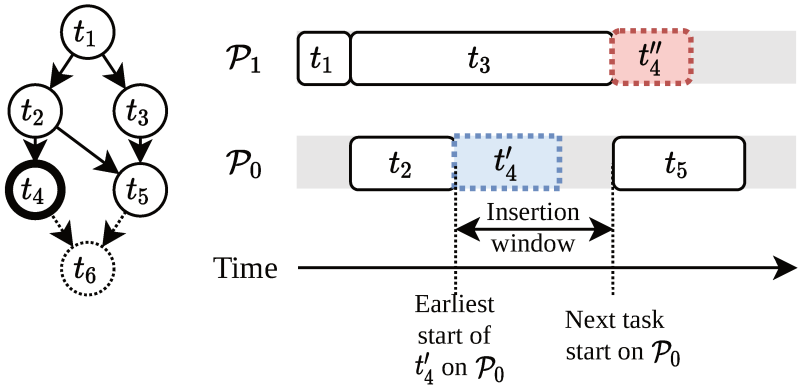


Figure 4.3: Illustration of insertion policy on task t_4 from an example task graph on two resources. t'_4 (in blue) is inserted in the schedule, while t''_4 (in red) is appended to the schedule.

The insertion policy's goal is to reduce the resulting schedule makespan by using the idle moments of the resources that are caused by task dependencies. In Figure 4.3, we give the example of a small task graph comprising six tasks, and the scheduling process has been paused at task t_4.

In this example, we suppose tasks $\{t_1; t_2; t_3; t_5\}$ have already been processed and thus are static in this schedule. Because t_5's execution requires prior execution of its predecessors t_2 and t_3, there is an idle time in resource $\mathcal{P}_0$. This lets a possible insertion of task t'_4, the latter requiring solely prior execution of task t_2. The alternative scheduling decision t''_4 illustrates how a

heuristic without insertion policy would insert this task on the earliest finish time basis. The earliest finish time scheduling policy with insertion is given in Algorithm 4.

This policy is executed for each resource $\mathcal{P}_j \in \mathcal{P}$ in the resource selection and schedule insertion step (cf. Figure 4.1). The resource that's assigned to the task is the one that minimizes the O_{EFT}.

Algorithm 4 Earliest Finish Time scheduling policy, with insertion

Input : Task t_i to schedule on resource $\mathcal{P}_j$, actual schedule state, application, and system information

Output : Earliest finish time (EFT) of t_i on $\mathcal{P}_j$

```
1: let LFT_P be the latest finish time of t_i's predecessors, or 0 if none
2: let n_T be the number of tasks scheduled on resource P_j
3: if n_T = 0 then // No tasks on P_j: run t_i as soon as possible
4:     t_i's start = LFT_P
5:     t_i's EFT = t_i's start + execution time of t_i on P_j
6:     return EFT of t_i on P_j
7: else
8:     for k ∈ [0 : n_T − 1] do // Parsing tasks in P_j's schedule
9:         if t_k = t_0 then
10:            // First task : can we insert t_i before t_0 starts?
11:            earliest start = LFT_P
12:        else
13:            // Can t_i be inserted between two tasks scheduled on P_j?
14:            earliest start = max( LFT_P ; t_{k−1}'s end )
15:        if earliest start + execution time of t_i on P_j < t_k's start then
16:            // If the insertion window is large enough
17:            t_i's start = LFT_P
18:            t_i's EFT = t_i's start + execution time of t_i on P_j
19:            return EFT of t_i on P_j
20:    // No insertion possible: append task at the end of P_j's schedule
21:    t_i's start = t_{n_T−1}'s end
22:    t_i's EFT = t_i's start + execution time of t_i on P_j
23:    return EFT of t_i on P_j
```

Once O_{EFT} has been computed for all tasks and the scheduling decisions have all been taken, the PEFT heuristic stops with the resulting schedule. The resulting PEFT schedule of the canonical graph is shown in Figure 4.4. Gaps in the schedule such as between tasks t_7 and t_{10} are explained by the communication latencies of the canonical application example.

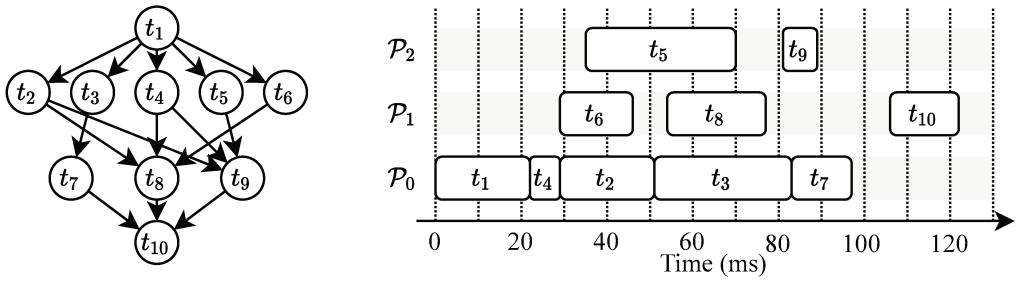


Figure 4.4: PEFT schedule of the canonical task graph on three CPU resources.

4.3 Self-reconfigurable system considerations

To enable the PEFT heuristic to schedule applications on self-reconfigurable systems, DPR operations must be considered in the system model. A simple method often used as an approximation, is to include reconfiguration and communication times in the execution time of tasks. However, this solution hides the complexity of sharing the unique DPR controller. Thus the resulting schedules by including reconfiguration in the execution times become wrong when two tasks start at the same time, implying two DPR operations happening at the same time. Moreover, this approach jeopardizes optimizations such as task pipelining and bitstream pre-fetching.

To successfully implement DPR operations in the schedule, modifications need to be made to the PEFT heuristic. Here, we introduce the following features:

- DPR operations management with the introduction of reconfiguration tasks when scheduling tasks on RRs;
- Bitstream pre-fetch management to reduce the impact of DPR operations on the schedule makespan;
- OCT table reuse and partial re-computation to accommodate for run-time changes in resource availability.

These features make our proposed heuristic capable of tackling the challenges of mapping and scheduling for self-reconfigurable systems with RRs of heterogeneous sizes. Providing a fast and efficient heuristic that can help to enhance the quality-oriented management methodology with flexibility.

4.3.1 Reconfiguration tasks

Reconfiguration tasks denote the process of dynamically reconfiguring a RR before executing a task in the FPGA. From the scheduling problem point of view, reconfiguration operations can be seen as additional tasks with conditional edges. Those tasks can be executed solely on a specific resource: the DPR controller.

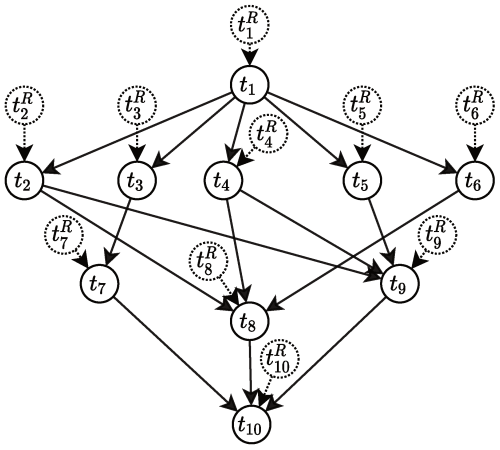


Figure 4.5: Canonical task graph with reconfiguration tasks denoted with dotted lines.

In Figure 4.5, we introduce a modified version of the canonical task graph to illustrate the concept of reconfiguration tasks. In this example, the canonical task graph is filled with reconfiguration tasks that are yet to be connected. Dependencies may appear depending on the mapping of tasks on the resources. If task t_1 and t_2 are executed on the same RR, then an edge connects the nodes "t_1" and "t_2^R". This must be done to represent the impossibility of reconfiguration task R2 to begin while task t_1 is still running: else this would erase the configuration in the respective RR. This must answer the 5th constraint (task reconfiguration) of the mapping and scheduling problem as introduced in Section 2.2.2.1.

Naively adding reconfiguration jobs increases the count of nodes in the graph to a maximum of $2n$ for a number n of original tasks, and increases significantly the number of edges. The PEFT heuristic increases quadratically with the number of nodes, and the edge growth increases the amount of predecessor termination checks in the EFT scheduling policy. Additionally, this rather simple method implies that a mapping decision has been taken before the scheduling process as the task graph needs to be fixed.

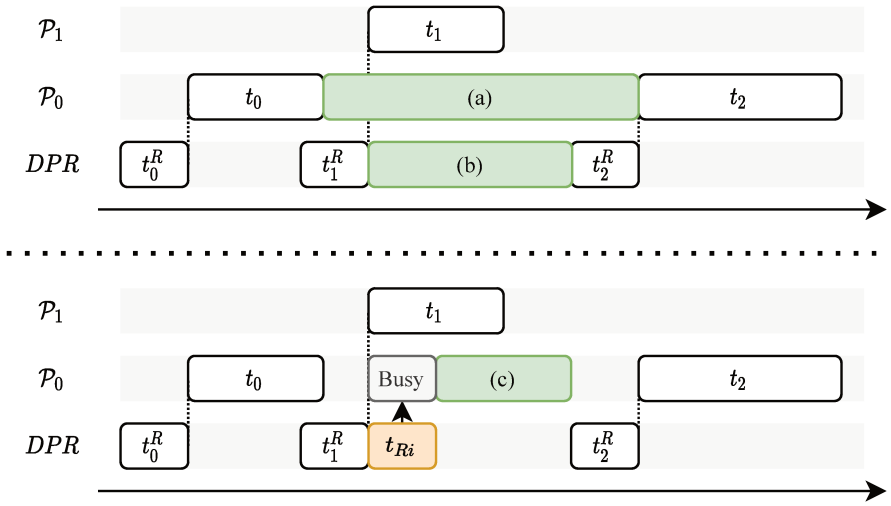


Figure 4.6: Illustration of EFT policy with insertion of a task t_i on two RRs $\mathcal{P}_0$ and $\mathcal{P}_1$, considering their reconfiguration tasks in three steps: (a) insertion window for t_i and t_{Ri}, (b) insertion windows for t_{Ri}, and (c) updated insertion window for t_i.

In the proposed approach, we instead let the scheduling heuristic add reconfiguration tasks on the fly. We illustrate the updated EFT with insertion policy in Figure 4.6. When computing the EFT for a task t_i on a resource $\mathcal{P}_j$, we first check for the insertion window (a) on $\mathcal{P}_j$'s schedule for both t_i and its corresponding reconfiguration task t_{Ri}. Since the DPR operation makes the targeted RR temporarily unavailable, it must be counted as part of the execution time in this insertion window.

When scheduling a software task, denoted by the absence of reconfiguration task, the insertion policy can stop here, otherwise, it must check if the corresponding reconfiguration task can be inserted in the DPR schedule. A new insertion window (b) then needs to be found to ensure the DPR controller is properly shared. We consider the previously scheduled reconfiguration tasks, as well as the other application tasks, to be fixed in the schedule. Once the reconfiguration task has been scheduled on the DPR insertion window (b), the original insertion window (a) is reduced to (c) to take into account the RR unavailability during the DPR process.

If the insertion window (c) becomes too small to fit the task t_i, then it is dropped and the algorithm looks for another insertion window in $\mathcal{P}_j$'s schedule. If there isn't, then t_i is appended at the end of $\mathcal{P}_j$'s schedule.

At the end of these three steps, we obtain the earliest finish task of t_i on $\mathcal{P}_j$, with respect to t_i's predecessors' terminations and the corresponding

reconfiguration process.

In addition to the EFT policy with insertion modification, the reconfiguration cost needs to be added to the OCT table computation. At the OCT computation step, the PEFT heuristic cannot know if the reconfiguration task can be pipelined with the communication time. This is reflected in the original OCT formulation in Equation 4.1 which now becomes:

$$OCT(t_i;\mathcal{P}_j) = \underset{t_j \in \text{ succ}(t_i)}{\text{Max}} \left[\underset{\mathcal{P}_w \in P}{\text{Min}} \left\{ \begin{array}{l} OCT(t_j;\mathcal{P}_w) + R(\mathcal{P}_w) + W(t_j;\mathcal{P}_w) \\ +A(t_i;t_j) \cdot C(\mathcal{P}_j;\mathcal{P}_w) \end{array} \right\} \right] \tag{4.8}$$

4.3.2 Bitstream pre-fetching for makespan reduction

The heuristic is now capable of managing hardware tasks with DPR operations. However, those reconfiguration tasks can benefit from further optimization. Namely, the bitstream pre-fetching technique which has been introduced in Section 2.2.2.2. The pre-fetching technique makes use of the precedence of the reconfiguration task t_i^R of a task t_i (end of $t_i^R \leq$ start of t_i) in a way that t_i^R's end can happen earlier than the start of t_i. Then the DPR controller is free earlier to reconfigure other tasks in the FPGA. The constraint is that the reconfigured task is mapped on a free (or idle) RR.

Bitstream pre-fetching can effectively parallelize the DPR operations with other tasks execution to minimize the latency introduced by reconfigurations. This is a desirable feature for our quality-oriented methodology as our quality of service model is based on slack (or time before deadline) optimization.

The pre-fetching technique creates an idle task constraint in the schedule. When a bitstream has been pre-fetched, there is a period in which the task is idle in the RR and waiting for its predecessor to terminate. The idle period effectively prevents other tasks from being reconfigured in this RR. This can be taken advantage of in the case of periodic tasks, which are not covered in this work. This constraint is the result of fixing previously scheduled tasks to limit the heuristic's time complexity. The idle task constraint is illustrated in Figure 4.7 (a).

To mitigate the idle task constraint, we introduce the Just-In-Time (JIT) bitstream pre-fetching. To do so, JIT pre-fetching minimizes idle times so there is no need to evaluate the idle task constraints if possible.

The idle task constraint can also be mitigated by delaying the reconfiguration of tasks that have an idle period, which could yield better schedules in terms of makespan reduction. This requires the scheduling heuristic to

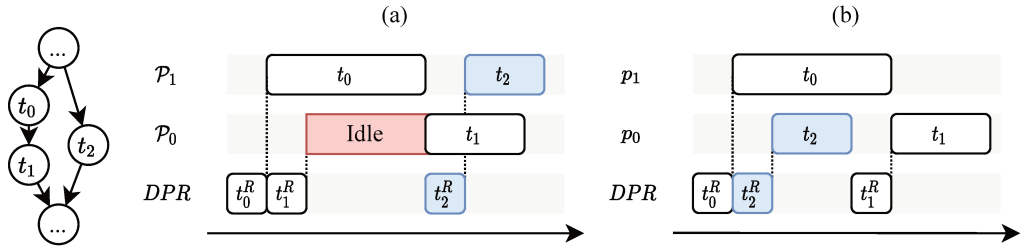


Figure 4.7: Illustration of idle task constraint on pre-fetching when scheduling task t_2 on a platform comprised of two RR resources. In (a), t_2 cannot be scheduled on $\mathcal{P}_0$ before t_1 as the RR has already been reconfigured and is waiting for t_0's termination. In (b), the pre-fetch has occurred just-in-time and lets t_2 run before t_1, assuming a large enough insertion window.

backtrack and re-schedule the reconfiguration task that causes the idle period. This would increase the heuristic's time complexity, however, which is a drawback as the goal of the proposed heuristic is to be efficient with relatively low decision time and a short schedule makespan.

To implement the JIT pre-fetching, three cases of reconfiguration task schedules are identified. The cases are illustrated in Figure 4.8. In those examples, we assume the heuristic tries to schedule task t_1 (colored in blue), and all other tasks are scheduled.

In the first case, the heuristic finds no insertion window in $\mathcal{P}_1$ as it computes $EFT(t_1; \mathcal{P}_1)$. It must append task t_1 at the end of the $\mathcal{P}_1$ schedule. In this example, t_1 requires t_0's termination, and so does t_1^R otherwise it would overwrite t_0's configuration. Therefore no pre-fetching can be applied here, and t_1's earliest start is delayed by the duration of reconfiguration task t_{R1}.

In the second case, the heuristic computes $EFT(t_1; \mathcal{P}_0)$, and the only other task in the schedule has been assigned to $\mathcal{P}_1$. The reconfiguration tasks t_1^R depend only on the other tasks assigned to $\mathcal{P}_0$ (none in this example), and the other events in the DPR schedule. Thus the earliest start of t_1 is later than t_1^R's as t_R^1 does not depend on t_0's termination. The insertion window between t_0^R's end and the earliest start of t_1 is assumed long enough to fit t_1^R in this example. Applying JIT pre-fetching, t_1^R is scheduled in such a way that t_1^R's end corresponds to the earliest start of t_1, avoiding idle time.

Finally, we illustrate JIT pre-fetching in the third case with a schedule insertion when computing $EFT(t_1; \mathcal{P}_0)$. In this case, there is an insertion window for t_1 and t_R^1, respectively in $\mathcal{P}_1$'s and the DPR schedule. This is possible thanks to the JIT pre-fetching of task t_3 that depends on t_2's termination. Here, pre-fetching is possible as it was in the second case, but the

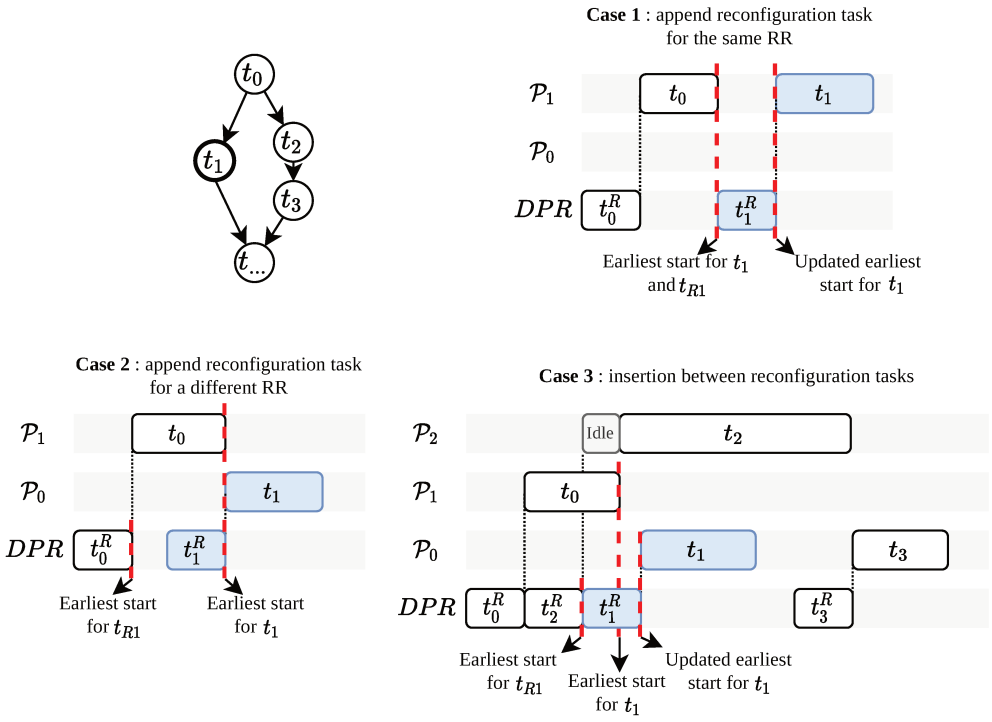


Figure 4.8: Just-In-Time (JIT) bitstream pre-fetching scheduling cases of an example task graph snippet.

end of the reconfiguration job t_1^R is later than the initial earliest start of t_1. Because t_1 cannot start before termination of t_1^R, its earliest start is updated. As the insertion window has been evaluated at large considering the execution time of t_{R1} and t_1 with no initial pre-fetch (cf. Section 4.3.1), it is guaranteed not to break the schedule's coherency.

Our heuristic considers those three cases to implement JIT pre-fetching on reconfiguration tasks. This feature does not require any modification on the original application task graph from the application designer.

4.3.3 OCT reuse and partial computation

In this section, a way to reduce the proposed heuristic's decision time is presented to tackle the efficiency challenge. To do so, we speed up our heuristic by pre-computing the OCT table offline as it takes a significant amount of time for path evaluations. The OCT table can then be reused fully or partially updated.

The first step of the PEFT heuristic is to compute the OCT table, which serves as prediction table, and to compute the optimistic earliest finish time

for computational resource selection. This can be used to our advantage as no modification is brought to this table in the later stages of the heuristic.

In the context of runtime quality-oriented application management, computing new schedules at runtime makes sense as constraints change over time (eg. deadlines, faulty or restricted resources...), or the workload to execute has changed in nature (eg. new QoE setting, additional services...).

However, reusing the OCT table isn't as trivial as it sounds. If the change brought by the context implies restricting one or more tasks on one or more resources, the OCT table must be at least partially recomputed. Indeed, the OCT's purpose is to reflect the impact of single-task scheduling decisions based on the computational resources pool and task graph dependencies. If one or more tasks cannot be implemented on one or more resources, then it is not guaranteed that the OCT table remains the same, as evaluated graph path costs are impacted.

To correlate OCT reuse with the proposed quality-oriented application management, different execution modes are represented by different adjacency and computation matrices. The corresponding OCT tables can be pre-computed and stored on-target to find schedules at runtime. Swapping OCT tables when implementing a different execution mode doesn't necessitate any modification and lets the heuristic reuse them in their entirety. This implies that for each execution mode, a different OCT table is stored for reuse purpose.

However, any hardware constraint or user restriction which directly impacts scheduling decisions requires partial OCT table computation. Such constraints and restrictions include forced implementation of specific tasks-resources assignments, denied resources for the scheduling heuristic, and can come from QoS user needs such as lowering chip temperature by reducing resource occupancy rates [113].

It is possible to partially compute the OCT table by computing only values that are impacted by a modification in their computation matrix W. Figure 4.9 illustrates the partial OCT computation when restricting task t_9 from using resource $\mathcal{P}_2$: $W(t_9; \mathcal{P}_2) = \emptyset$. For the canonical task graph example, three paths lead to the source node from t_9 through its first order predecessors: t_2, t_4, and t_5 and second order predecessor t_1. Because the OCT table is computed recursively from the sink to the source node, all predecessors on those paths need to have their OCT value updated.

The subset of tasks impacted by a modification on a task t_i can be found by graph analysis with Breadth First Search (BFS) or Depth First Search (DFS) from t_i to the source node (t_1 in the case of the canonical application).

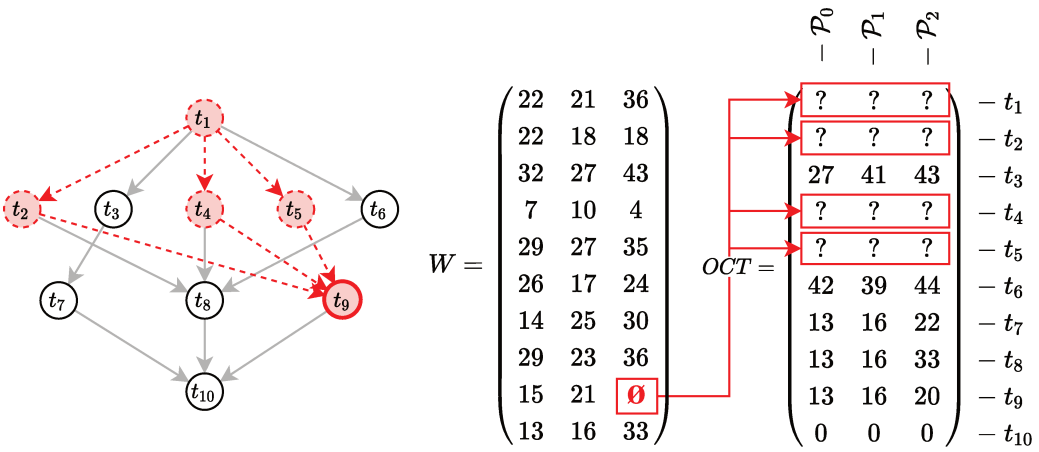


Figure 4.9: Partial OCT computation when restricting task t_9 from being implemented on resource $\mathcal{P}_2$. The paths highlighted with red dashes are impacted by the modification in the computation matrix W and cause the partial OCT computation of the corresponding tasks.

We introduce the inheritance matrix I in Equation 4.9 which is the result of this graph analysis on the canonical example. Element $I(i;j)$ indicates if task t_i is a predecessor of t_j, with $I(i;j) = 1$ if true, else 0. The diagonal elements of I are always equal to zero as a task cannot be its predecessor. This matrix can be computed offline at design time.

$$I = \begin{array}{c} \begin{matrix} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & t_{10} \end{matrix} \\ \begin{pmatrix} 0 & & & & & & & & & \\ 1 & 0 & & & & & & & & \\ 1 & . & 0 & & & & & & & \\ 1 & . & . & 0 & & & & & & \\ 1 & . & . & . & 0 & & & & & \\ 1 & . & 1 & . & . & 0 & & & & \\ 1 & 1 & . & 1 & . & 1 & 0 & & & \\ 1 & 1 & . & 1 & 1 & . & . & 0 & & \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & \end{pmatrix} \begin{matrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \\ t_9 \\ t_{10} \end{matrix} \end{array} \tag{4.9}$$

A last optimization can also be made here by benefiting from the DAG

property on the adjacency matrix. Similarly with the adjacency matrix A, because the studied graphs are DAGs there exists a topological order such that I is a lower triangular matrix (with diagonal elements equal to zeroes) and A is an upper triangular matrix. To avoid computing performing runtime graph analysis for partial OCT computation, we can merge the I and A matrices for runtime reference.

Finally, we introduce Pre-Fetch PEFT (PF-PEFT) in Algorithm 5, which takes into account the self-reconfigurable systems adaptations, JIT bitstream pre-fetching, and OCT reuse and partial computation optimizations presented in the previous Sections.

Algorithm 5 PF-PEFT scheduling heuristic

Input: Matrices A, C, W and vector R
Input (Optional): Pre-computed OCT table, Modified W matrix
Output: Application mapping and schedule for DPR capable architecture

if No OCT table or Modified W matrix **then** *//Complete PF-PEFT*
    Update OCT table recursively with Equation 4.8
    Compute OCT ranks vector using Equation 4.5
    Sort OCT ranks vector in decreasing order
Put source task in the list of tasks ready for execution (*ready-list*)
while *ready-list* not empty **do**
    $t_i \leftarrow$ highest OCT ranked task from *ready-list*
    for each $\mathcal{P}_j \in \mathcal{P}$ **do**
        Compute $EFT(t_i; \mathcal{P}_j)$ using insertion policy (cf. Algorithm 4)
        if $R(\mathcal{P}_j) > 0$ **then** *// If $\mathcal{P}_j$ is a RR*
            Set reconfiguration task t_i^R for RR $\mathcal{P}_j$ with JIT pre-fetch
            if delay > 0 **then** *// JIT pre-fetch case 1 or 3 from Figure 4.8*
                Update $EFT(t_i; \mathcal{P}_j)$
            else
                // JIT pre-fetch case 2 from Figure 4.8
                // $EFT(t_i; \mathcal{P}_j)$ doesn't need an update as there is no delay
        $O_{EFT}(t_i; \mathcal{P}_j) \leftarrow OCT(t_i; \mathcal{P}_j) + EFT(t_i; \mathcal{P}_j)$
    Assign task t_i to resource $\mathcal{P}_j$ that minimizes O_{EFT}
    Update *ready-list* with t_i's successors that are now ready for execution
return Application mapping and schedule for DPR capable architecture

4.4 Quality-oriented management with runtime scheduling

The proposed approach makes use of the quality of experience (QoE) and service (QoS) models introduced in chapter 3 as inputs.

By making use of the proposed PF-PEFT scheduling heuristic, the approach can benefit from being able to compute new schedules at runtime so there's no previously computed solution database to store on the target.

Execution modes are kept as the key element of the contribution, which enables the methodology to automatically downgrade and upgrade the targeted application to maximize the QoE score.

4.4.1 Proposed methodology

The proposed methodology is described in Algorithm 6. It starts from the highest QoE execution mode and downgrades it iteratively until it finds a valid solution with a positive QoS score.

The constraint update interrupt comes from either a hypervisor program (as in [83, 57]) or a manual demand from the user that increases the number of additional tasks or services to execute conjointly with the targeted application. This update can also come from a system failure detection on one or more RRs (as in [4]). Schedules can be kept between application iterations if constraints didn't change in this time frame.

QoS prediction in line 6 uses the QoS function defined by the application and system designers. In our case, as in chapter 3, the QoS value is equal to the proportion of slack in the schedule (time before the deadline).

As PF-PEFT is deterministic, there is always only one schedule to compute for each considered execution mode. Therefore, with j execution modes, n tasks and m additional context tasks, p resources and k restricted resources, the time complexity of the methodology can be estimated to:

$$\mathcal{O}(j \cdot (n+m)^2 \cdot (p-k^2)) \approx \mathcal{O}(j.n^2 p) \tag{4.10}$$

Algorithm 6 PF-PEFT based quality-oriented methodology

Input : Application and system information
Output : Valid schedule maximizing QoE

```
1:  while true do
2:      current execution mode ← mode of highest QoE
3:      wait for constraint update interrupt
4:      while no next schedule do
5:          compute new schedule with PF-PEFT heuristic (cf. Algorithm 5)
6:          if new schedule has predicted QoS > 0 then
7:              next schedule ← computed schedule
8:          else if Execution mode is at lowest QoE then
9:              // Can't downgrade anymore, will cause a service break
10:             next schedule ← computed schedule
11:         else
12:             downgrade QoE and pick the next execution mode
13:     apply new schedule
```

4.4.2 Using module reuse

The proposed methodology can be enhanced with module reuse capabilities. To benefit from module reuse, the system must be in a state where only one task has been assigned to a RR. In this case, the DPR operation on this RR can be skipped for each iteration as there is no functional need to reconfigure this task.

Because module reuse prevents other tasks from being executed on the same RR, it creates a restricted resource. Therefore there must be clear benefits for reuse on the resulting makespan as the proposed approach is capable of reducing the impact of DPR operations. It can be hypothesized that module reuse happens more frequently on applications that possess one parallel task with a longer execution time (in regards to the full schedule), and on systems where this task can be executed only on few RRs with high reconfiguration time.

We introduce Figure 4.10 an example task graph with two execution modes. In this example, the DAG topology does not change between execution modes and they only differ by their task computation times (W_1 and W_2), except for t_1 which is considered functionally identical between both execution modes.

In iteration 1, we let the methodology schedule the first execution mode, and the heuristics find a schedule in which t_1 is the only task assigned to $\mathcal{P}_0$.

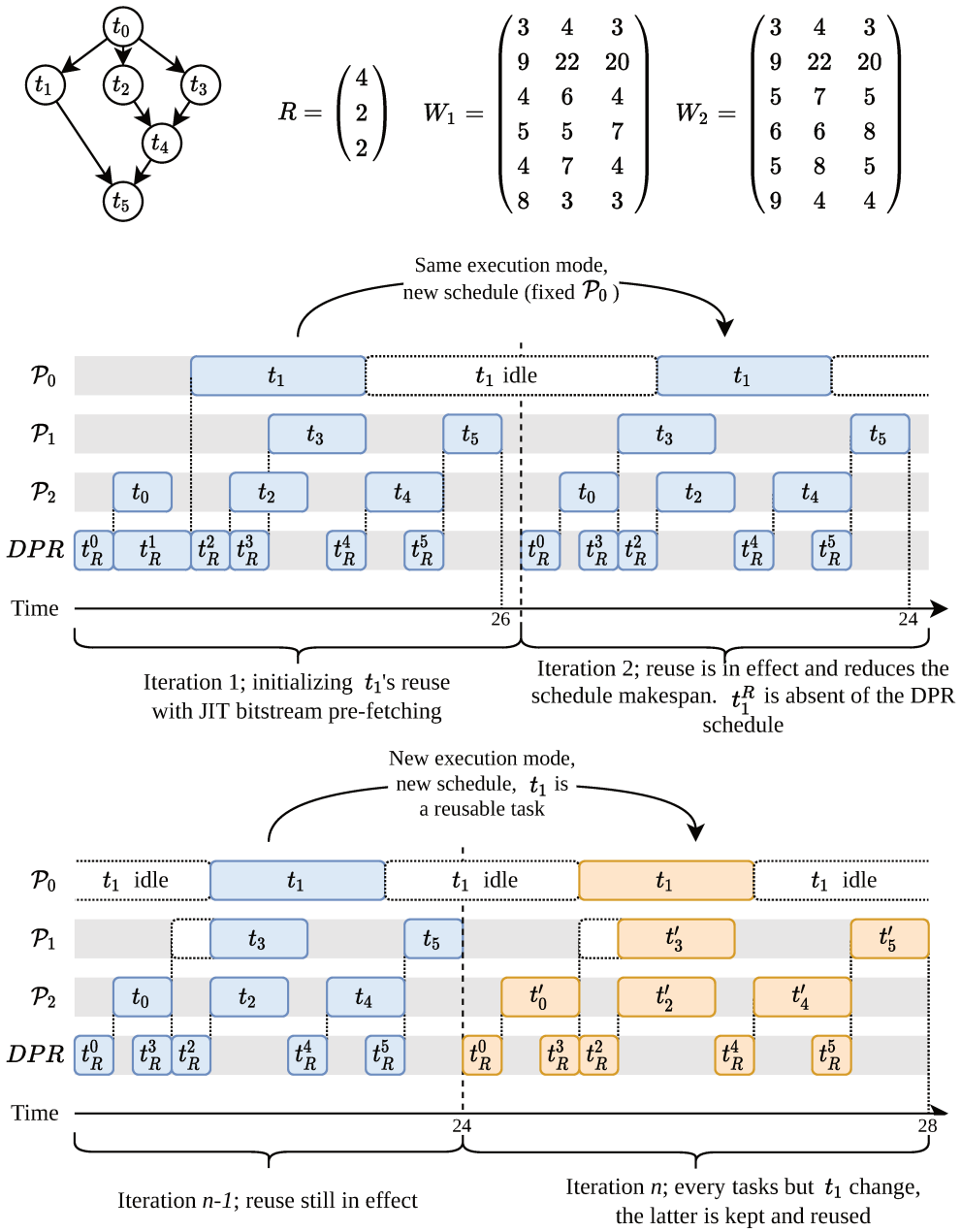


Figure 4.10: Example of reused task t_1 between iterations and execution modes with the schedule-base methodology. In this example we consider that t_1 can only be executed in $\mathcal{P}_0$ (illustrating a bigger RR), while $\mathcal{P}_1$ and $\mathcal{P}_2$ are smaller RRs. Communications times between resources are neglected in this example for readability.

With t_1 being alone on $\mathcal{P}_0$ and now reconfigured, the conditions for reuse are met, the methodology fixes t_1. For the scheduling heuristic, this means t_1 has to be scheduled on $\mathcal{P}_0$, and t_1^R is not required, as at this point t_1 is already implemented in $\mathcal{P}_0$. In addition, $\mathcal{P}_0$ is now restricted for the other tasks when mapping and scheduling.

A new schedule is then computed at the end of iteration 1 to make use of the reuse of t_1. It is to be noticed that the computed schedule with the reuse constraint will be applied only if its evaluated QoS value is higher than the one without. Reusing making the corresponding RR unavailable for other tasks.

The self-reconfigurable system can keep using this schedule until iteration n where in this example an upgrade is required (bottom part of Figure 4.10). Because t_1 is identical between execution mode 1 and 2, and is still in the reuse configuration, the methodology can reuse t_1.

Similarly as between iterations 1 and 2, if a solution with a positive QoS value is found when reusing t_1, then it is kept. A final schedule can be computed without considering the reuse of t_1 to verify if getting rid of the reused task increase the QoS.

It is important to note that tasks that can be reused between execution modes can do so only if it makes sense at the functional level.

4.5 PF-PEFT performance experiments

In this section, we measure the performance of our Pre-Fetch (PF) PEFT heuristic. We compare the obtained results to ASAP PF [90] that comprises reconfiguration job management and bitstream pre-fetching. Both heuristics can consider software resources in addition to RRs as they are special cases of null reconfiguration time. In addition, we differentiate Partial and Complete PF-PEFT, with the former making use of a pre-computed OCT table for decision time optimization, and the latter computing the OCT table from scratch. Conducted experiments aim to highlight the efficiency of our approach by measuring the obtained schedules makespan and the on-target decision time.

The heuristics are compared on a set of profiled task graphs from real applications from the literature and synthetic benchmarks.

4.5.1 Experimental setup

The PF-PEFT and the ASAP PF heuristics have been written in `C++` and cross-compiled for the ZCU102 Ultrascale+'s ARM A53 processor with `g++-aarch64-linux-gnu`. The `-O2` flag has been used to optimize execution speed, but no multi-core optimizations have been made in the code as embedded targets do not always comprise multiple CPU cores. To mitigate OS latencies and cache optimization, resulting decision times for given benchmark times are averaged on 125k schedules.

The heuristics take as inputs adjacency and computation matrices (A and W) which describe the DAG workload to schedule on the given system. It will also be given a communication matrix and a reconfiguration vector (C and R) to describe the time taken to transfer data using the communication infrastructure latencies and the time taken to reconfigure the underlying RR. Each different set of A, W, C, and R constitutes a different benchmark.

To analyze the efficiency of both algorithms, we ran the scheduling heuristics on task graph topologies that came from self-reconfigurable systems such as our H.264 application, using extrapolated results from heterogeneous platforms from the literature [76, 88, 114], or standard workload topologies [115].

We also made use of synthetic task graph benchmarks as in [4, 90, 88] to generate lots of benchmark task graphs and evaluate our heuristic on a large combination of graph topologies and latencies. The generated task graphs can then be scheduled on the platforms to compare the algorithms' performance.

Table 4.1: Considered architecture resource compositions.

Composition	RR sizes	Description
4RR+1SW	4 large	Coarse-grain island style[102]
8RR+2SW	3 large, 5 small	Balanced coarse-grain[6]
12RR+3SW	2 large, 10 small	Fragmented
16RR+4SW	1 large, 15 small	Highly parallel

In addition, we experiment with a range of resource compositions that comprise both CPUs and RRs resources. The studied resource composition can comprise up to 16 RRs to study the impact on the heuristics decision times and makespan reduction. The four studied platform compositions are illustrated in Table 4.1.

Architectures are represented by the communication matrix C and reconfiguration vector R. We give values from the communication matrix in

Table 4.2, obtained in the platform evaluation in Section 3.4.1. Reconfiguration times of the RRs scale with the number of logic blocks in the RRs, and the generated reconfiguration vector R illustrate this range of heterogeneous, coarse-grained RR size. We consider RRs holding 1k up to 10k slices of a ZCU9EG FPGA (ZCU102 board). The size of the resulting partial bitstreams ranges from 400KB to 2MB. Using high bandwidth ICAP controllers [18, 20], profiled reconfiguration times are comprised between 1.23ms and 5.24 ms.

Table 4.2: Profiled worst-case delay on the communication interface using the AXI4 Stream interconnect.

Source	Destination	Worst-case delay (ms)
RR	RR	0.06
RR	CPU	0.30
CPU	RR	0.30
CPU	CPU	<0.01

Finally, the results are evaluated using the Schedule Length Ratio (SLR). SLR denotes normalized schedule makespan to a lower bound and is the ratio of the sum of the critical path nodes running on an ideal system with no communication nor reconfiguration costs. It is defined as:

$$SLR = \frac{makespan}{\sum_{t_i \in CP_{MIN}} \left(\min_{\mathcal{P}_j \in \mathcal{P}} [W(t_i; \mathcal{P}_j)] \right)} \tag{4.11}$$

With $makespan$ as the schedule duration to normalize, $t_i \in CP_{MIN}$ denoting the tasks (or nodes) in the minimum critical paths, and $\min_{\mathcal{P}_j \in \mathcal{P}} [W(t_i; \mathcal{P}_j)]$ denoting the fastest computation time of t_i among the resources in the set of resources $\mathcal{P}$. A heuristic schedule always yields a value greater or equal to 1 as it cannot be less than the denominator of Equation 4.11, as it is not a realistic schedule makespan but the shortest path to sink node without platform constraints. When comparing heuristics with SLRs, the lower the better.

4.5.2 Real application benchmarks

We employ benchmark applications listed in Table 4.3. They are characterized by their tasks count in the workload, the number of resources they can be scheduled on, and their topology types. We distinguish two general types of graph topologies: Tall and Wide. They are illustrated in Figure 4.11 respec-

tively with the Epigenomics and CyberShake task graphs. The Tall topology denotes graphs that contain more tasks in their critical paths than the number of forked tasks and vice versa for the Wide topology. For instance, the Epigenomics task graph is a Tall graph with 8 tasks in its critical path and only 4 parallel tasks. The CyberShake on the other hand is a Wide graph with 6 tasks in its critical path and up to 8 parallel tasks.

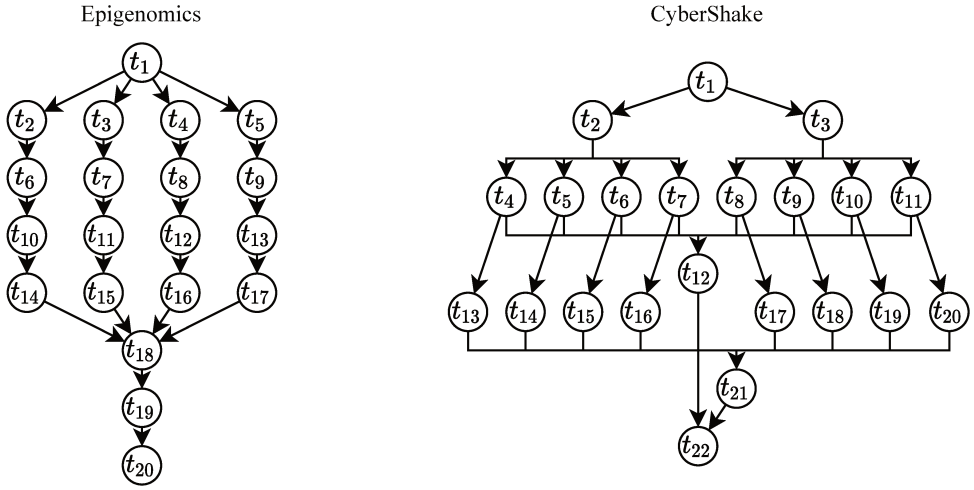


Figure 4.11: Illustration of Tall (Epigenomics) and Wide (CyberShake) task graph topologies.

Because there are too few data sets available online, such as our H.264 implementation, the StereoVision application from [91] and the Edge Detection mix from [98], we extrapolated datasets from other heterogeneous platforms approaches. Such platforms feature different CPUs and GPUs and provide realistic data for our scheduling heuristic, and FPGAs can have similar or better execution times as GPUs [24]. With this assumption, data sets from cryptography and lane detection applications from [114] were used, as well as the canonical task graph [88] and the traction control application from [76]. As these platforms were not implemented on FPGA, we extrapolated reconfiguration times by considering RRs that can fill 1k to 10k slices of a ZCU9EG FPGA (ZCU102 Board), assuming longer task computations on GPUs were loosely related with bigger RRs. Finally, Epigenomics, CyberShake, and Montage task graph topologies [115] were considered, although their data sets were incomplete and task computation times had to be extrapolated from the previously introduced applications.

As can be seen in Table 4.4, differentiating Partial and Complete PF-PEFT gives a lower and upper bound for the partial OCT computation optimization.

Table 4.3: List of benchmark application topologies.

Benchmark	Task count	Resource count	Topology type
CryptoGraphy[1] [114]	4	2	Tall
StereoVision [91]	7	3	Tall
Edge Detection Mix [98]	9	3	Tall
Canonical[1] [88]	10	3	Wide
H.264 + AES/Sobel	10	5	Wide
Traction control[1] [76]	10	5	Wide
Lane Detection[1] [114]	14	6	Tall
Epigenomics[2] [115]	20	4	Tall
CyberShake[2] [115]	22	5	Wide
Montage[2] [115]	26	5	Tall

[1] Extrapolated results from heterogeneous CPU-GPU platforms.
[2] Topologies only with synthetic computation times.

The ASAP PF from [90] yields a schedule in less than 100μs in all introduced application benchmarks, and grows linearly with the task count in the graphs. In comparison, our Complete PF-PEFT grows quadratically and reaches decision times up to 3 times longer than ASAP PF. However, those on-target decision times remain within half a millisecond, which is fairly low considering the duration of the resulting schedules is 10× to 100× longer.

Table 4.5 introduces the Schedule Length Ratios (SLR) of the resulting schedules. For each benchmark, we give the SLRs with (w/) decision times in the SLR computation, and without (w/o) to show how our higher decision times compared to ASAP PF impact the quality of the resulting schedule. This is because decision times need to be factored in for our runtime management methodology. Because PF-PEFT is deterministic, Complete PF-PEFT SLRs are denoted by '/' in Table 4.5 as it yields the same SLR as Partial PF-PEFT without factoring decision times.

On average, PF-PEFT approach outperforms ASAP PF by yielding schedules 11.54% shorter on average on the 10 benchmarks. Taking into account in the decision times in the SLR computation, this drops to 11.45%.

In the CryptoGraphy benchmark, both approaches yield the same SLR without factoring decision times and are worse for PF-PEFT by up to 0.18%.

Table 4.4: On-target scheduling heuristic decision times comparison.

	Decision times (μs)		
	ASAP PF	PF-PEFT Partial	PF-PEFT Complete
CryptoGraphy	12.55	12.30	23.42
StereoVision	25.14	31.47	58.21
Edge Detection Mix	32.00	38.26	72.57
Canonical	36.62	42.63	85.11
H.264 + AES/Sobel	37.79	43.51	72.69
Traction control	43.60	77.14	137.50
Lane Detection	45.95	78.13	131.82
Epigenomics	70.37	110.71	194.93
CyberShake	86.76	149.15	281.06
Montage	97.89	177.92	325.57

Table 4.5: Resulting Schedule Length Ratios comparison per heuristic, with decision time included in the SLR (w/) or without (w/o).

		SLR		
		ASAP PF	PF-PEFT Partial	PF-PEFT Complete
CryptoGraphy	w/o	18.24	18.24 (+0%)	/
	w/	18.27	18.27 (+0.17%)	18.27 (+0.18%)
StereoVision	w/o	1.329	1.304 (-1.94%)	/
	w/	1.329	1.304 (-1.94%)	1.304 (-1.94%)
Edge Detection Mix	w/o	1.770	1.654 (-6.54%)	/
	w/	1.770	1.655 (-6.51%)	1.655 (-6.50%)
Canonical	w/o	3.350	2.906 (-13.24%)	/
	w/	3.351	2.908 (-13.20%)	2.908 (-13.18%)
H.264 + AES/Sobel	w/o	1.809	1.460 (-19.26%)	/
	w/	1.810	1.461 (-19.20%)	1.462 (-19.17%)
Traction Control	w/o	2.253	1.971 (-12.53%)	/
	w/	2.254	1.972 (-12.44%)	1.974 (-12.38%)
Lane Detection	w/o	2.951	2.307 (-21.82%)	/
	w/	2.953	2.315 (-21.61%)	2.317 (-21.54%)
Epigenomics	w/o	1.775	1.742 (-1.84%)	/
	w/	1.775	1.742 (-1.84%)	1.742 (-1.84%)
CyberShake	w/o	2.167	1.588 (-26.75%)	/
	w/	2.167	1.588 (-26.75%)	1.588 (-26.74%)
Montage	w/o	1.946	1.869 (-3.97%)	/
	w/	1.946	1.869 (-3.97%)	1.869 (-3.97%)

This can be explained by the sequential nature of this benchmark. Because there aren't many bottlenecks to predict, the ASAP makes the same decisions as PF-PEFT by merely following the ASAP topological order.

In the Lane Detection benchmark illustrated in Figure 4.12, PF-PEFT outperforms ASAP PF by 21.54% with the complete version, including decision time. This can be explained by the topology of the Lane Detection task graph. The ASAP PF heuristic schedules the tasks in the numerical order, so it performs pre-fetch on tasks t_3 to t_6 in Figure 4.12 (b). But then tasks t_{12} is blocked as tasks t_7 to t_{11} need to be executed. Because ASAP PF couldn't predict the bottleneck on t_7, it cannot insert those tasks in the schedule and it has to append the sequence at the end of the schedule instead. By comparison in Figure 4.12 (c), PF-PEFT scheduled those tasks alternatively and allowed insertion windows thanks to JIT pre-fetching, regardless of the order in which the tasks were sorted originally.

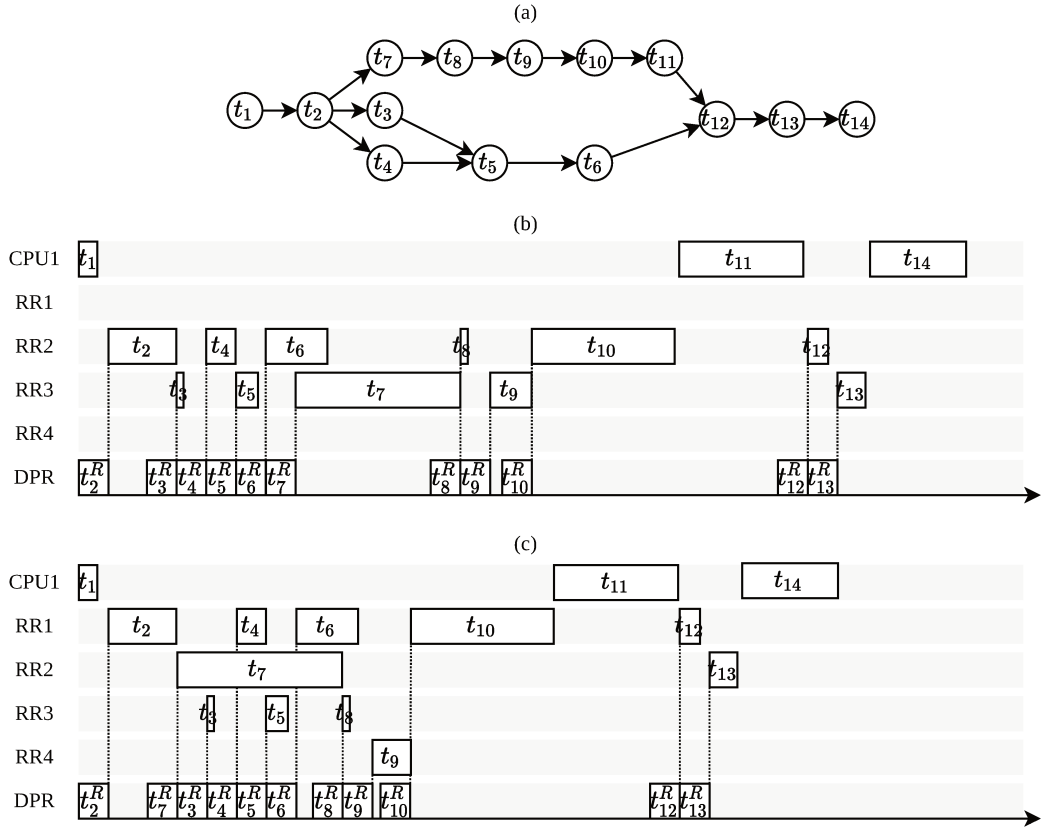


Figure 4.12: Lane Detection task graph (a) resulting schedules using ASAP PF (b) and PF-PEFT (c).

Finally, we draw attention to the minimal differences between Partial and

Complete PF-PEFT results when factoring in decision times in the SLR. This highlights the fact that even if our PF-PEFT approach is noticeably slower than ASAP PF, it has little impact down the line as the schedule duration takes up the majority of the resulting makespan.

4.5.3 Synthetic workloads

In this section, we evaluate the introduced heuristics with synthetic workloads. Workload generation can help generate a large amount of diverse data set, inspired by real application benchmarks, to characterize and evaluate the heuristics. Using our application and architecture models introduced in Chapter 2, we produce parameters previously introduced in Table 2.5 as inputs for the heuristics.

Synthetic workload topologies are made of layers comprising a random number of tasks to construct a random DAG topology. Each task in a layer is connected randomly with one or more tasks from previous layers. The characteristics of those DAGs are their height h (number of layers), and their width w (maximum number of tasks in a layer). Each node can be connected randomly with a number $m(X)$ of nodes from previous layers, with X being a random uniform value in $[0; 1]$ and $m(X) = \lceil \frac{n}{10} e^{-5X} \rceil$. This makes each DAG node capable of being connected with one to $\lceil \frac{n}{10} \rceil$ previous nodes.

We define Tall and Wide topologies as follows:

- Tall : $h \in [1; 50]$, $w \in [2; 4]$;
- Wide : $h \in [1; 4]$, $w \in [2; 50]$.

Both topologies are restricted to a number of nodes $n \in [20; 100]$ for a representative industry workload [57, 115].

Values in the computation matrix W, denoting execution times of tasks on resources, are generated uniformly in $[0.5; 500]$ (in ms). We defined this range as a set of execution times used in self-reconfigurable system benchmarks introduced previously. Such executions times are representative of task execution times on FPGA as they stay quite low compared to CPU implementations [24], and correlate with values from self-reconfigurable system implementations from [98, 91]. Each value of the computation matrix is generated independently to support the heterogeneous RR size hypothesis which typically causes reconfigurable modules to behave differently. We added a 2% probability that $W(i; j)$ is assigned a value of $+\infty$ to reflect the possibility of no implementation of a task t_i on a resource $\mathcal{P}_j$, while we guarantee that there

is a least one implementation of t_i on a resource. The generated topologies are kept for all platforms compositions.

4.5.3.1 Partial OCT computation savings

Figure 4.13 introduces the results for an 8RR+2CPU architecture composition on generated topologies, including 50% of Tall and Wide graphs respectively. Here, we show the part of the PF-PEFT decision time spent in OCT computation. PF-PEFT typically spends 48.2% of its decision time on average for the OCT table computation in this experiment. This can be used to the system's advantage when recomputing a new schedule, as decision times can be reduced by half (if no OCT update is required) while retaining the SLR reduction performance of our approach.

The SLR graph show values obtained while factoring in the decision times. The Partial and Complete PF-PEFT values are extremely close to each other. This indicates that the decision time is marginal compared to the resulting schedule makespan in the SLR computation.

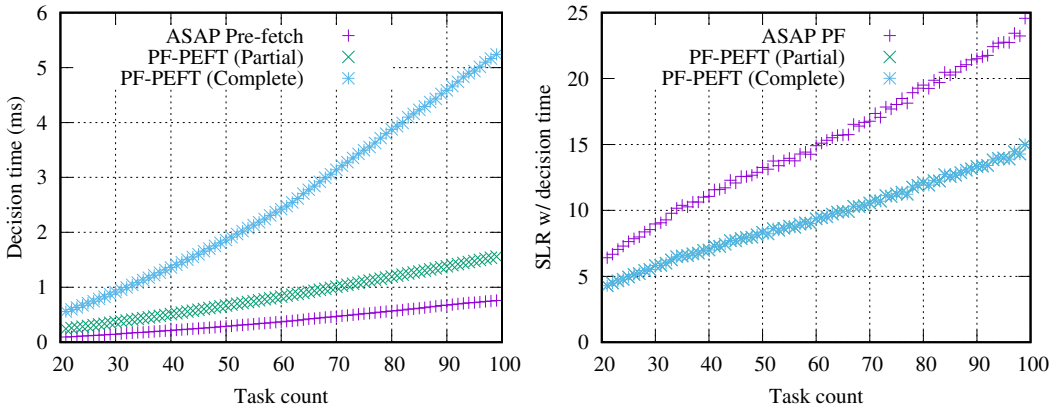


Figure 4.13: Random synthetic workload decision times (left) and resulting SLRs (right).

In addition, we introduce Figure 4.14, which summarizes the average decision time spent computing the OCT table on the four considered platform compositions from Table 4.1, for task graphs comprised of 20 to 100 nodes. Results show the more resources in the self-reconfigurable system, the more time it spends. This was expected from the PF-PEFT heuristic as the critical paths evaluations must be done for each resource (cf. Equation 4.8), and for each successor of each node in the graph. This highlights that while the OCT table can help predict the impact of individual task-resource assignments, its impact on the overall decision time can reach as high as 61%.

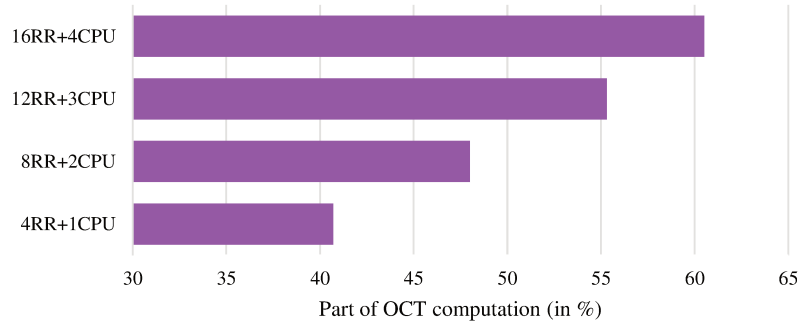


Figure 4.14: Part of the decision time spent computing the OCT table when OCT reuse is not possible, by system resources composition.

This also shows that attractiveness of reusing a pre-computed OCT table, the time taken by the heuristic to compute the OCT table must be considered as there are some cases where partial OCT computation or reuse cannot be applied. Typically, when resources are denied from being used by the methodology.

4.5.3.2 Effect of tasks computation times on SLR

In this experiment, we study how computation times from the synthetic workloads affect the resulting SLRs. The computation time values are reduced from $[0.5; 500]$ to $[0.5; 5]$ (in ms) to correlate with smaller values from self-reconfigurable system from [98]. Results are shown in Figure 4.15 for 100k DAGs on the same 8RR+2CPU platform.

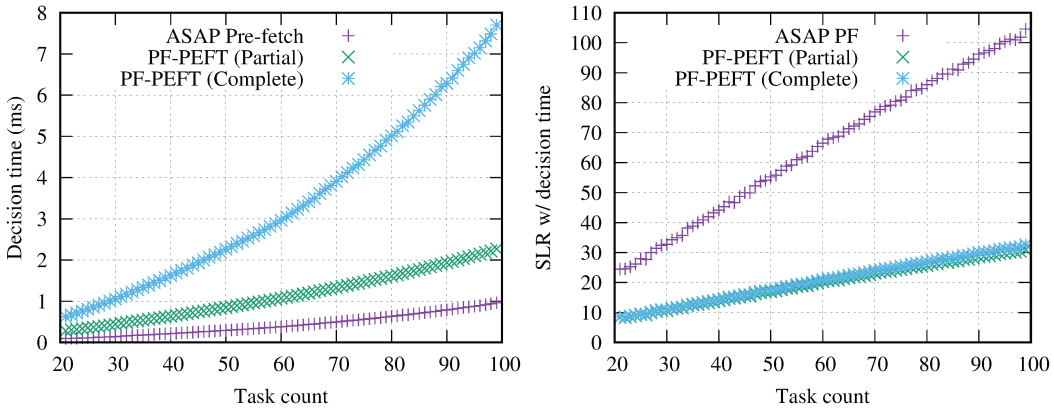


Figure 4.15: Random synthetic workload decision times (left) and resulting SLRs (right), with longer task computation times.

The decision times aren't impacted much by this change, as the main difference compared to the first experience was that there is more possibility

to insert tasks in the schedule thanks to JIT pre-fetching.

On the SLR graph though, the decision times play a bigger impact on the SLR computation. Thus the difference between Partial and Complete PF-PEFTs SLRs is bigger and can be distinguished for larger task graphs. However, this difference grows linearly in this experience from 2% at 20 tasks per graph, to 5% in the largest graphs.

The ASAP PF also does worse, as a consequence of reconfiguration tasks management impacting the schedule. Pre-fetch optimization saves on reconfiguration times and is a more effective makespan reduction solution when computation times are low. In this situation, computation times are lower and pre-fetching can help reduce makespan by at most one reconfiguration time (up to 5.23ms) per task. Thus pre-fetch management and schedule insertion play a larger role in makespan optimization.

This experiment features a very high SLR compared to the first one for both heuristics.

This is a reflection of the SLR metric computing the shortest path without considering reconfiguration or communication times. In this experiment, those times play a big part in the schedule and cannot compete with the ideal shortest path, leading to higher SLR.

4.5.3.3 Graph topologies impacts

In this experiment, we compare the impact of Wide and Tall topologies on the resulting decision times and SLR on the 8RR+2CPU platform with the original $[0.5 : 500]$ values (in ms) for the computation matrix W. Figure 4.16 introduces the resulting decision times in insets (a) and (b), and their corresponding SLRs in insets (c) and (d) for Wide and Tall topologies respectively.

Partial PF-PEFT measures the decision time when fully reusing an OCT table, whereas the Complete PF-PEFT takes into account the OCT computation in the decision time as it has to compute one from scratch. Therefore the difference between Partial and Complete PF-PEFT is the part of the decision time taken to compute the OCT table.

In insets (a) and (b), we see that the OCT computation takes a bigger part of the decision time in the Tall versus the Wide topologies, going as high as 75% for the Tall topology. This is because the OCT table is computed recursively, and many tasks in a Wide graph can have a common successor (eg: task t_{12} from the CyberShake application, introduced earlier in Figure 4.11). As the OCT value for a task needs to be computed only once for each resource, the OCT values of the common successors can be reused. This

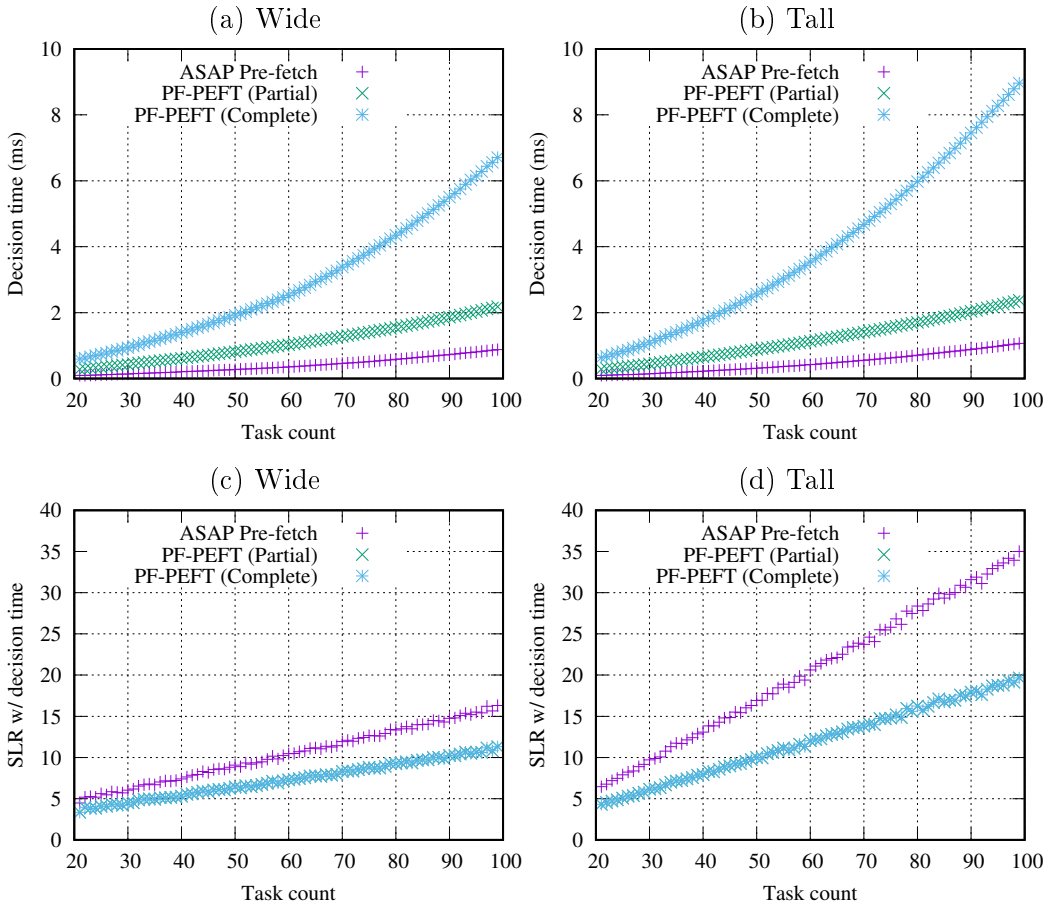


Figure 4.16: Comparison of the impact of Wide and Tall graph topologies on SLR and decision times.

optimization is topology-dependent as there are fewer common successors in Tall topologies.

In (c) and (d), we distinguish the differences in the resulting SLRs. Our approach has a lower SLR for both topologies, but the SLR reduction over ASAP PF is more important with the Tall topologies. Taller topologies give less space for parallel computations, and bottlenecks can happen as we've shown with the Lane Detection application in Section 4.5.2. ASAP PF's aggressive pre-fetching strategy can hinder the scheduling decisions to insert tasks in the schedule to cope with such bottlenecks.

By comparison, our resulting SLRs are 41% lower on the Taller topologies on average when scheduling 125k different synthetic task graphs between 20 and 100 tasks. The PF-PEFT approach is capable of predicting such bottlenecks and considers reconfiguration and communication times in its predic-

tion. Therefore it is capable of scheduling tasks that could cause bottlenecks in advance to maximize execution parallelization on RRs.

On the other hand, Wide topologies can be easier to schedule for the ASAP PF heuristic as at a given step in the scheduling process, there are more available tasks that are ready for execution. Thus, there is less opportunity for ASAP PF to wrongly choose a task that finishes earlier over a task that causes a bottleneck. Still, thanks to a more efficient insertion policy on the PF-PEFT heuristic, we were able to yield a shorter makespan by 29% on average on the Wide synthetic workloads.

4.5.3.4 Effect of platform composition

In this last experiment, we consider a set of 4 different architectures comprised of 4, 8, 12, and 16 RRs with respectively 1, 2, 3, and 4 additional CPUs, as introduced in Table 4.1. The goal here is to highlight the impact on both decision times and resulting SLRs and give insights into the design of coarse-grained self-reconfigurable systems.

A set of 125k random synthetic workload benchmarks is run for each of the 4 architecture compositions for both the Wide and Tall topologies, i.e. 250k benchmarks per platform, and one million synthetic task graphs in total for the four platforms.

Figure 4.17 introduces the result of decision times and SLR with the 4 self-reconfigurable system compositions. Each set of graphs compiles data from a combined 250k Wide and Tall synthetic workload benchmarks.

Results on decision times in the left insets show how the number of resources impacts our PF-PEFT heuristic. This result isn't surprising as the time complexity is $\mathcal{O}(n^2 \cdot p)$ on unconstrained systems. At most, on a hundred task on the more resourceful platforms containing 16 RRs and 4 CPU cores, the heuristic reaches a decision time of 21.34ms on average for the Complete OCT computation version.

Decision times are relatively high compared to ASAP PF which stays at most under the millisecond. However, the SLR results on the right side of Figure 4.17 show that the impact of decision time is marginal in the SLR, even for the highest decision times.

ASAP PF fails to yield better SLR results with more resources as it doesn't support task insertion. This coupled with a lack of prediction capabilities cannot guarantee a better usage of the resources. PF-PEFT on the other hand is capable of making locally sub-optimal scheduling decisions if that reduces the overall schedule duration without causing bottlenecks later.

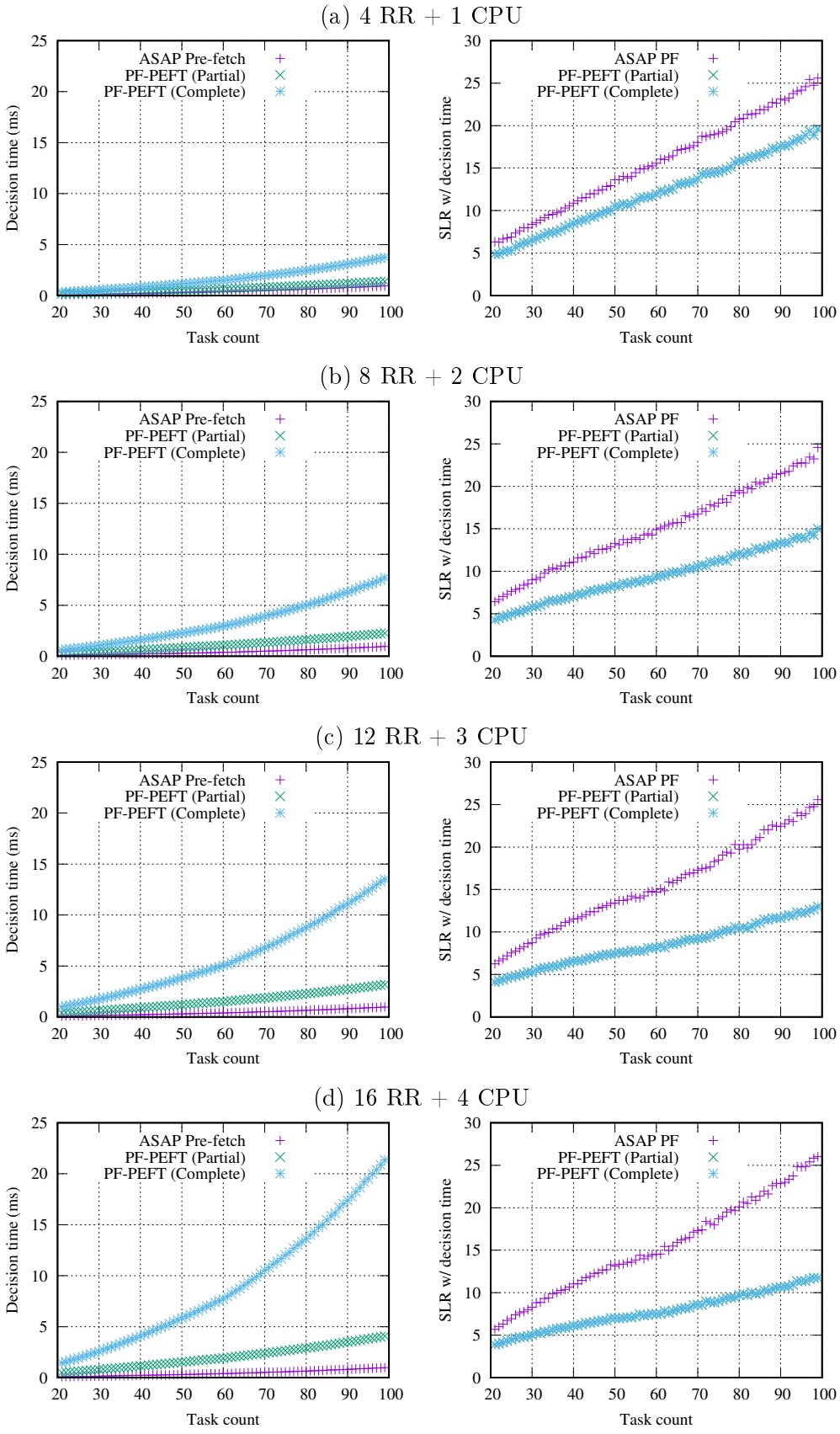


Figure 4.17: Platform composition impact on resulting decision times and SLRs.

4.6 Quality-oriented methodology experiments

4.6.1 Experimental setup

We make use of the same architecture as in Section 3.4.1 that is comprised of four heterogeneous RRs and a CPU core for task execution, and the same H.264 encoder implementation metrics and quality models.

In addition, we use the Traction Control application from [76] as illustrated in Figure 4.18. This application was used in a quality-oriented scheduling methodology for heterogeneous MPSoCs, and each task from the graph has a reward score attached that we can use to define a QoE score function for this application.

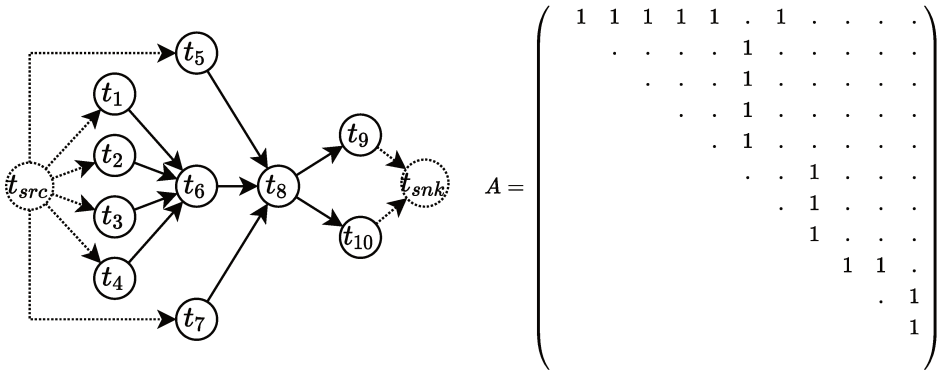


Figure 4.18: Traction control application task graph, with unique source and sink nodes t_{src} and t_{snk} added in the corresponding adjacency matrix.

This original task graph doesn't feature single source or sink nodes. This is an issue for the PF-PEFT heuristic as it requires a single entry and exit point to evaluate the longest paths. However, this can be mitigated for any similar DAG by creating dummy source and sink tasks. In Figure 4.18, placeholders t_{src} and t_{snk} are used to connect respectively tasks that do not initially have predecessors (t_1, t_2, t_3, t_4, t_5 and t_7), and those which do not have successors (t_9 and t_{10}). When running the scheduler, those tasks are considered to have a null computation time with no reconfiguration or communication delay.

The traction control management methodology in [76] made use of reward scores to measure the user's interest in defined execution modes, which can be assimilated into quality assessment. Using those rewards as metrics for a quality of experience model on the three execution modes (low, medium, and high precision), we obtain Table 4.6 from [76].

Table 4.6: Traction control execution modes and corresponding QoE values.

Execution mode	Reward scores	QoE value
Low precision	567	0.37
Medium precision	984	0.64
High precision	1'533	1

Conversely to the H.264 Encoder which features optional tasks that are executed only in some execution modes, the traction control application's graph topology doesn't change. Thus the adjacency matrix remains the same for all three execution modes and only the computation matrices differ as task execution times change between execution modes.

Finally, this application has a deadline of 134 ms which will be used to check service breaks.

Two scenarios are considered in the following experiments, repeated for both the H.264 and the traction control application:

1. Constrained workload: several random tasks are added to the set of tasks to schedule, on top of the targeted application;

2. Restricted resources: several resources are restricted from usage by the targeted applications.

Both constraints scenarios are similar to the two scenarios used in our experiments in chapter 3. In this set of experiments, however, there will be only aggressive scenarios (i.e. constraints change every iteration) as the scheduling approaches are deterministic.

4.6.1.1 Workload constraints

In this scenario, we generate a synthetic constraint task graph that needs to be scheduled along with the targeted application. These constrained task graphs contains between 2 to 12 nodes that can be executed on all the resources from the architecture. Their execution time are generated randomly between 0.5 and 5.0ms for the H.264 Encoder, and between 0.5 and 10.0ms for the Traction Control. These execution times ranges fit execution times from the targeted applications. The random synthetic workload generation is performed in such a way that the resulting constraint schedule makespan takes up to 75% of the targeted application deadline. Figure 4.19 illustrates the workload that needs to be scheduled at runtime and executed before the application deadline.

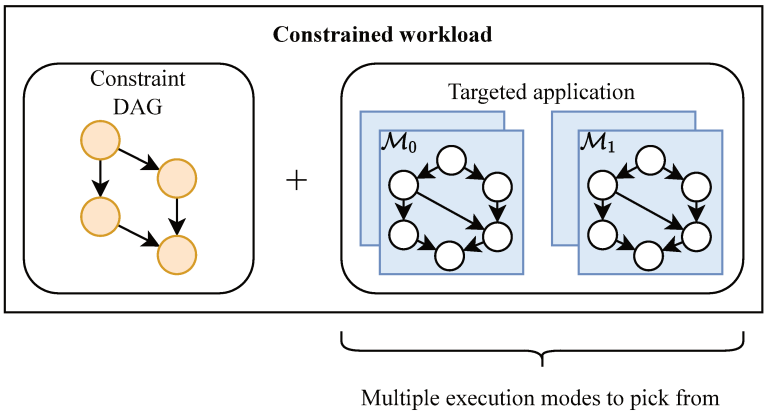


Figure 4.19: Constrained workload to implement on the targeted self-reconfigurable system.

4.6.1.2 Denied resources

In this scenario, we deny up to a number k out of p resources in the architecture ($k < p$) from being used by PF-PEFT to schedule the targeted application.

The H.264 Encoder benchmark features one CPU and 4 RRs, and up to 3 RRs can be denied. This application requires at least one CPU core and one RR to work, as some tasks can only be executed in software and some others in hardware. A scenario featuring all RRs being denied, or an unusable CPU cannot lead to a valid schedule by any scheduling heuristic.

Similarly, the Traction Control features 4 RRs solely, and up to 2 RRs can be denied at most. Should a third RR be denied, not even the optimal scheduling algorithm can find a valid schedule.

A simple schedulability analysis can be done. With only one RR available to schedule the application, in the lowest precision mode, the application makespan is equal to the sum of computation times of all tasks in this RR plus the sum of all DPR operations. Because every task and DPR operation would target the same RR, no optimization can be made. Therefore, the minimum schedule duration on a single RR would be equal to the minimum sum of execution times of tasks on a resource, and the reconfiguration latency times the number of tasks.

The computation matrix in the low precision mode and the platform reconfiguration vector are given in Equation 4.12.

$$W = \begin{pmatrix} 14 & 14 & 26 & 26 \\ 15 & 15 & 17 & 17 \\ 14 & 14 & 11 & 11 \\ 27 & 27 & 29 & 29 \\ 20 & 20 & 10 & 10 \\ 22 & 22 & 17 & 17 \\ 25 & 25 & 28 & 28 \\ 16 & 16 & 24 & 24 \\ 26 & 26 & 25 & 25 \\ 18 & 18 & 14 & 14 \end{pmatrix}, \quad R = \begin{pmatrix} 1.7 \\ 1.7 \\ 2.5 \\ 2.5 \end{pmatrix} \tag{4.12}$$

The minimum schedule makespan with only one RR is 214ms long using resource $\mathcal{P}_0$ or $\mathcal{P}_1$ as they are assumed equal. The makespan is greater than the application deadline (134ms), therefore no scheduling algorithm can hold this deadline.

4.6.2 Experimental results

4.6.2.1 Workload constraints

In Table 4.7, we introduce the decision times of the PF-PEFT-based methodology algorithm for the targeted applications in the constrained workload scenario.

Because the scheduling heuristic can be called multiple times, as a result of not finding a valid solution with a positive QoS score, those values can differ from that of the previous chapter where the heuristic was always called once. In this scenario, the partial OCT computation can be fully employed as there is no runtime restriction on which RR can be used for any task.

Table 4.7: Resulting average decision times of PF-PEFT based methodology heuristic in constrained workload scenario.

	H.264 Encoder	**Traction Control**
Decision times	0.171 ms	0.153 ms
Continuity of service	99.19%	97.71%

Using the scheduling-based methodology, obtained results show a high continuity of service with only 0.81% and 2.28% iterations resulting in a service break. Results show that our approach spends typically less than 0.2ms computing schedule at runtime. This is respectively 0.51% of the targeted

application deadline for the H.264 Encoder application, and 0.11% for the Traction Control application. Because the PF-PEFT-based methodology is efficient in terms of decision time and performance, the sub-one percent part of the deadline spent computing schedules puts rarely the system in a situation of a service break.

Table 4.8 summarizes the frequency of use of the targeted applications' execution modes.

Table 4.8: Part of iterations (in % of total) of H.264 Encoder and Traction Control execution modes and resulting average QoE score in the workload constraint scenario.

Execution Mode	**H.264 Encoder**	**Traction Control**
Break	0.81 %	2.29 %
Mode 0	0.27 %	22.56 %
Mode 1	0.001 %	60.74 %
Mode 2	4.38 %	14.41 %
Mode 3	0.008 %	
Mode 4	1.30 %	
Mode 5	25.95 %	
Mode 6	0.0004 %	
Mode 7	8.23 %	
Mode 8	58.74 %	
Mode 9	0.31 %	
Average QoE	0.724	0.634

The H.264 Encoder sees four dominant execution modes (2, 5, 7, and 8) that adds up to 97.3% of iterations. This doesn't mean that the PF-PEFT-based methodology finds modes such as Mode 6 (0.0004% usage) hard to schedule, but rather that Mode 7 provides a higher QoS value in this scenario, as Mode 6 and 7 shares the same QoE value.

The Traction Control on the other hand has a higher service break rate (2.28%). The presence of a further degraded mode with lower task computation times could help mitigate this higher rate, but there is a trade-off for application designers as there are only so many functionally working and desirable execution modes.

In comparison, authors of [76] did not tackle the challenge of scheduling for DPR nor runtime scheduling for constrained systems. Their approach was based on offline scheduling based on MILP formulations to obtain near-optimal schedules in terms of QoE. Running on a full software implementation in similar conditions as in [76], the resulting average QoE of PF-PEFT was

worse by 7.2%. However, we argue that in these conditions, those results do not highlight our contributions on the DPR operations management, although it shows the scheduling is relatively fast and efficient in comparison.

Finally, a direct comparison with our previous hybrid approach cannot be made fairly, as this scheduling-based approach is capable of runtime scheduling tasks from the targeted application in addition to tasks from the constraint DAG. This causes less resource fragmentation as using a dedicated runtime scheduler offers flexibility in resource usage.

On the other hand, using pre-computed solutions makes the system schedule one application and then the other. However, in an unconstrained scenario, pre-computed solutions can be better than runtime-computed ones. This is especially true if the offline scheduler is near-optimal as in [91, 76]. In addition, the runtime performance of the hybrid approach introduced in Chapter 3 does not scale with the number of tasks in the targeted application.

4.6.2.2 Denied resources

Table 4.9 shows the resulting decision times for the PF-PEFT based methodology when some resources are unavailable. In this scenario, because all tasks are being restricted from being scheduled onto one or more resources, partial OCT computation cannot be applied. This scenario is critical for Partial PF-PEFT and highlights why the self-reconfigurable system must be able to switch to Complete PF-PEFT.

Table 4.9: Resulting decision times of PF-PEFT based methodology heuristic in denied resources scenario.

	H.264 Encoder	**Traction Control**
Decision times	0.177 ms	0.287 ms
Continuity of service	100.0%	100.0%

Compared to the workload constraint scenario, one can see a reduction in decision time for the H.264 Encoder use case. This is due to the PF-PEFT-based methodology downgrading fewer times the execution modes until it finds a solution, as seen in Table 4.10 where the lowest QoE mode is Mode 5. Because there are fewer downgrades, there are fewer calls to the scheduler.

On the other hand for the Traction Control, decision times increase slightly as there are slightly more downgrades, impacting the average QoE value which drops to 0.582.

Continuity of service reaches 100.0% for both use cases, which indicates the

PF-PEFT-based methodology could find an execution mode of the targeted application that can be executed on the constrained platform.

These results can be nuanced as the challenge here is more to find a suitable scheduling solution and execute it before the deadline. The maximum number of restricted resources guarantees that there is at least one scheduling solution that can execute the targeted application.

Table 4.10: Part of iterations (in % of total) of H.264 Encoder and Traction Control execution modes and resulting average QoE score in the denied resources scenario. Dashes denote a usage rate of 0.0%.

Execution Mode	**H.264 Encoder**	**Traction Control**
Break	0.00 %	0.00 %
Mode 0	-	54.50 %
Mode 1	-	18.22 %
Mode 2	-	27.28 %
Mode 3	-	
Mode 4	-	
Mode 5	23.35 %	
Mode 6	-	
Mode 7	-	
Mode 8	47.92 %	
Mode 9	28.74 %	
Average QoE	0.780	0.582

Table 4.10 introduces the part of iterations spent in each execution mode. Here, a direct comparison with our previous hybrid approach is fairer as the scenario is the same and cannot use at all the denied resources. In this constraint scenario, he PF-PEFT-based methodology is more efficient as for each execution mode there is only one schedule to compute, with respect to the platform constraints, whereas the hybrid approach needs to parse multiple solutions before finding one that fits the constraints.

4.6.2.3 Module reuse impact

The impact of reconfigurable module reuse can be observed using the constrained workload experimentation on the H.264 Encoder. The Traction Control from [76] has three different quality modes (low, medium, and high precision) and the authors do not comment on the potential reuse of certain tasks. In addition, the DAG topology does not differ between execution modes, and the computation times of all tasks vary between those. Therefore, as authors

of [76] do not disclose if functional differences between execution modes are worth a reconfiguration, we cannot extrapolate reusable tasks for this application.

The H.264 Encoder execution modes can be functionally compared to see which task can be reused from going to a mode i to a mode j. From the 10 execution modes, we can identify the DAG impact of functional changes between execution modes, as represented in Table 4.11. As we can see, the H.264 encoder application offers many possibilities for task reuse.

Table 4.11: Reusable tasks between execution modes. Empty elements denote the absence of reusable tasks between a mode $\mathcal{M}_i$ and $\mathcal{M}_j$.

	0	1	2	3	4	5	6	7	8	9
0		(a)							(b)	
1	(a)				(b)	(b)			(b)	
2				(a)			(b)	(b)		(b)
3			(a)				(b)	(b)		(b)
4		(b)				(b)	(c)		(b),(c)	(c)
5		(b)			(b)		(d)		(b),(d)	(d)
6			(b)	(b)				(b)	(c)	(b),(c)
7			(b)	(b)					(d)	(b),(d)
8	(b)	(b)			(b),(c)	(b),(d)	(c)	(d)		(d),(c)
9			(b)	(b)	(c)	(d)	(b),(c)	(b),(d)	(d),(c)	

(a) all: only the framerate constraints change, all tasks can potentially be reused
(b) core: tasks from the core H.264 encoder can be reused (i.e. all but Sobel/AES)
(c) sobel and **(d) AES**: only Sobel and AES respectively can be reused

We then ran the constrained workload scenario again on the H.264 encoder application benchmark to check how many times the proposed methodology made use of the reuse technique. Obtained results show that 48.58% of the computed schedules featured a single task on a resource, that can be reused at the following application iteration as defined in Section 4.4.2. Then, upon a change in the constraints tasks to execute, only 14.52% of schedules that included a reused task led to another schedule with reuse.

This difference in reuse setups iterations and applied reuse schedules can be explained by the execution mode transitions. As seen in Table 4.11, if the H.264 Encoder is running on execution mode 5 (360p60fps+AES) with a reused task in the schedule, and the constrained workload makes our methodology capable of scheduling the application at execution mode 6 (480p60fps+Sobel),

no task can be reused as these two execution modes are too different on the functional level.

To apply the reuse technique, the methodology must detect when a schedule is computed with a task that's mapped alone on a RR (reuse technique setup). Then, a new schedule must be computed while fixing the task on the resource to properly apply the reuse technique. This comes at a cost of an additional PF-PEFT call in the methodology which increases the decision time by up to 28.52% (up to 0.364ms), with an average of 18.31% (0.336ms) using the complete PF-PEFT version.

However, as the decision times of the proposed methodology are still very short, this latency overhead caused only an overall 0.02% decrease in continuity of service. The benefits of the reuse technique outweigh this negligible decrease as it lowers the usage of the ICAP resource and energy consumption. Furthermore, because the reuse technique is more efficient with heterogeneous RR sizes [92], the DPR controller may avoid fetching larger partial bitstreams from the DDR, which could also help reduce the memory access congestion.

4.7 Conclusion

In this chapter, we introduced our task graph scheduling heuristic, PF-PEFT, for self-reconfigurable systems that can be used within our quality-oriented management methodology. The goal of this methodology is to efficiently find a mapping of tasks on the system resources to minimize the resulting schedule makespan while maximizing the quality of experience.

To do so, we determined a fast and efficient heuristic used in the field of heterogeneous multi/many-cores computing. This field shares similarities with our scheduling problem and predictive scheduling are effective to deal with task graph bottlenecks. This scheduling heuristic was then adapted for self-reconfigurable systems with the introduction of reconfiguration tasks, which takes into account the unicity of the DPR controller resource. Then, the Just-In-Time (JIT) bitstream pre-fetching is introduced to reduce resulting schedule makespan. JIT pre-fetching allows easier insertion of tasks in the schedule, and does not increase the time complexity of the algorithm.

The introduced heuristic has been thoroughly evaluated on a set of real-world data and extrapolated data sets from heterogeneous computing works. Synthetic workloads inspired by those data sets were also used as is customary in the literature. Those workloads were evaluated on different coarse-grained heterogeneous self-reconfigurable platforms to demonstrate the interest of our

approach.

Results show our approach outperforms a state-of-the-art scheduling heuristic for self-reconfigurable systems by up to 11% on average on a set of real application benchmarks. If the former is capable of finding valid schedules in less time than a millisecond on large task graphs of up to a hundred tasks, we argue that our longer decision times of up to 21ms on these large task graphs are mitigated by the resulting schedule makespan reduction.

In addition, we introduced a method for reducing significantly the decision times by pre and partially computing the Optimistic Cost Table (OCT) which is used for scheduling prediction. This reduction reaches up to 61% for runtime scheduling.

The PF-PEFT heuristic has then been employed as part of the quality-oriented management methodology. On-target experiments conducted with PF-PEFT have shown we were able to reach a high continuity of service and average QoE when conducting experiments with simulated constraint levels.

Chapter 5

Conclusion and discussion

This thesis work aimed to provide dynamic resource allocation methodologies to manage self-reconfigurable systems with a guarantee of service execution. The focus was put on FPGA architectures comprised of multiple RRs containing different numbers of logic elements (so called heterogeneous RRs). These types of architectures can be used to time-multiplex hardware accelerators (or hardware tasks) in the RRs, to execute an application workload while using a smaller FPGA matrix than doing a fully static implementation. This is particularly of interest as it can reduce costs, chip size, and potentially energy consumption. However, this makes the runtime management of the resources paramount to guarantee service execution. The latter is characterized by meeting a given application deadline (i.e. real-time constraint) and respecting a minimum level of quality of service.

To do so, we proposed models of quality and quality-oriented management methodologies to exploit the dynamic partial reconfiguration of FPGAs. Quality-oriented management has previously been used in CPU-GPU systems, as identified in chapter 2. However and to the best of our knowledge, no such approaches have been used for self-reconfigurable systems.

Using existing frameworks and operating system libraries, we introduced two quality-oriented management methodologies to dynamically allocate self-reconfigurable systems resources to targeted applications. Conducted experiments have shown that service execution can be guaranteed consistently while the system is constrained in execution times and communication latencies. These constraints are characterized by: unpredicted latencies within the communication infrastructure and or shared memories, additional tasks or services to execute concurrently with the targeted application or faulty resources that need to be deactivated at runtime.

5.1 Contribution summary

To achieve a high level of guarantee of service, we first introduced a quality model in chapter 3 to define quality parameters at the functional level, which impact end-users perceived quality, or Quality of Experience (QoE). In the

context of embedded systems, QoE can relate to the system's efficiency to reach a goal, such as reaching high precision with traction control, or a higher signal-to-noise ratio in signal processing applications.

These quality parameters can be exploited to define functional execution modes that application designers can associate with a QoE score to quantify either empirically or mathematically for their own needs. Then, a model of Quality of Service (QoS) that reflects the performance of the system when running the application can be defined. Using this quality model, application designers can qualify their definition of system performance for the quality-oriented management methodologies. Using the QoE and QoS models allows us to quantify how good a service is executed on the system, and how good this service is to reach a given goal.

In chapter 3, we introduced a hybrid design-time/runtime methodology that generates a pre-computed database of time-multiplexed implementations of a targeted application on the self-reconfigurable system's resources. These solutions are generated for each execution mode of the application, in such a way that the runtime part of the methodology possesses an array of statically optimized implementations. Then, to react to a given level of constraints on the system resources, the methodology can downgrade the quality of the application. Alternatively, the methodology can upgrade the quality of the application once the constraints are low enough that a higher-quality implementation can meet deadlines.

Using profiled metrics from an H.264 application implementation as a benchmark, the methodology achieved 94% continuity of service upon the addition of contextual services and latency, and up to 61% when some resources are being restricted from usage. Because this methodology does not require computing new mappings and schedules at runtime, it decreases the complexity of the runtime management as it only scales with the size of the database, and not the complexity (in terms of number of tasks) of the workload to execute. However, experiments have shown this has other impacts on the system as the database needs to be stored in the target's memory. To mitigate this issue, we pruned the solution database, which has shown an 18× speedup over the exhaustive search in the H.264 use case's database, for a decrease of at most 19% in quantified quality of experience.

To enhance the flexibility of the quality-oriented management approach, we introduced a runtime scheduling-based methodology in chapter 4. After identifying the Predict Earliest Finish Time (PEFT) as a scheduling heuristic offering the best trade-off between time complexity and application makespan reduction, we adapted it for self-reconfigurable systems with state-of-the-art

bitstream pre-fetch and reuse techniques. The introduced PF-PEFT heuristic outperforms a recent similar scheduling algorithm for self-reconfigurable systems by producing shorter schedules by 11% on average on a set of 10 applications from the literature.

The PF-PEFT heuristic has then been employed as part of a second quality-oriented management methodology. The latter makes use of runtime scheduling to find time-multiplexed implementation solutions for the targeted applications on self-reconfigurable systems resources. Obtained results show this helped reach a higher continuity of service rate than the first hybrid design-time/runtime approach. The continuity of service rate goes up to 99% on the additional service scenario and 100% on the resource restriction scenario, at the cost of implementing the scheduler in the platform.

5.2 Future works

In this work, the resource allocation methodologies focused solely on time-multiplexed resources for island-style architectures as we could not functionally verify functional behavior of our H.264 implementation on slot or grid-style architectures. As the end goal of our methodology is to execute more tasks using smaller FPGA matrices, time-multiplexing was performed on coarse-grained island-style dynamically reconfigurable architectures. However, works in the literature have shown an interest in dynamic resource allocation using relocatable tasks as it could reduce the fragmentation of resource usage. In particular, time-multiplexed grid-style architectures could make the most out of the available dynamically reconfigurable resources.

Space and time-multiplexing of FPGAs is a much harder challenge as other parameters need to be taken into consideration. As a consequence, the time complexity of the runtime management of the resources must be carefully studied so that it stays reactive. In addition, the partial bitstream management would be an issue that needs to be addressed as such architecture would require to either store additional bitstreams or compute runtime bitstream relocation, imposing respectively a memory and a latency overhead. However, space and time-multiplexing management could be used to further increase the system's efficiency in execution times, energy consumption, or chip temperature management.

Concerning the quality model, while execution modes introduce a method for application designers to easily define ways to upgrade and downgrade their targeted application at runtime, the quality of experience and service models

impact the decisions of the runtime management methodologies. An interesting future work would be to introduce multiple quality of service models for a given application and see how they impact the decisions. An interesting quality of service models for the industry would be energy [84] or temperature-oriented [113]. Then, based on a user's decision or a mission plan [57], the self-reconfigurable system could switch its quality of service goal at runtime. As a motivational example, a drone performing a target acquisition mission could use an energy-saving oriented quality of service model when reaching observation points. It could switch to a performance-oriented quality of service model during the mission, then return to the energy-saving model on its way back.

In the case of coarse-grain island-style architectures, we introduced different RRs compositions in our experiments with the PF-PEFT heuristic. These results show some preliminary works toward a definition of architecture guidelines when designing with DPR. It can be hypothesized that for a given application, there exists an optimal DPR-capable FPGA that best supports its execution. Considering slot or grid-style architecture, the FPGA footprint allowed for a hardware accelerator can differ. The topology of the architecture can then evolve to meet the requirements of targeted applications (eg: accelerator taking up to three slot-type RRs to reduce its execution time). Therefore, similarly to the quality of service metric, a quality of architecture could be introduced to monitor the efficiency of the current architecture to support the service execution.

In this work, we also made the hypothesis that application designers were capable of drawing the functional task graph of their applications. However, that is not always easy. Graph partitioning for complex workloads can result in very large task graph, notably with loop unrolling graphs into DAGs. In addition, graph partitioning for self-reconfigurable systems is similar to hardware/software co-design. This comes with added problem, such as defining task graphs that possess as little communication channels as possible to reduce congestion in the communication infrastructure. We believe graph analysis methods could help find the best resource compositions (in terms of logic elements) and application task graph decomposition. Addressing this trade-off could help reduce resource fragmentation, and decrease the reconfiguration latencies.

The proposed DPR-capable architecture could be enhanced with a NoC which possesses the advantage of better scaling with the number of RRs to interconnect compared to an AXIS crossbar, in terms of used logic elements. In addition, local cache memories could be used within the FPGA to store

frequently used variables. In the hypothesis that a lot of RRs are defined, this could reduce congestion on the DDR memory. Also, proper placement of these cache memories within the FPGA could help distribute the packets in the NoC.

Finally, multiple works from the literature consider energy savings as a motivation for using smaller dynamically reconfigurable FPGA versus bigger fully static ones [116, 75, 117]. If it can be verified that smaller FPGAs have lower average power consumption than bigger ones using vendor tools, we argue that for an equal workload to compute, we believe the smaller dynamically reconfigurable would take longer. Resources need to be shared on the latter and the DPR operations take a non-negligible time, whereas a fully static can fully pipeline and parallelize the workload. In addition, the fully static FPGA does not have a power overhead caused by the management of the self-reconfigurable system. Therefore, as the energy spent by the system is the quantity of power used over time to process the workload, it can be assumed that self-reconfigurable systems consume more energy than fully static FPGAs. As for future works, the study of the energy overhead could help characterize the best candidates for using self-reconfigurable systems in the embedded electronics industry.

Appendices

Bibliography

[1] Karl Rupp. 50 Years of Microprocessor Trend Data. *github.com/karlrupp/microprocessor-trend-data*, 2022.

[2] Qianru Zhang, Meng Zhang, Tinghuan Chen, Zhifei Sun, Yuzhe Ma, and Bei Yu. Recent advances in convolutional neural network acceleration. In *Neurocomputing*, 2018.

[3] Sherif Hosny, Eslam Elnader, Mostafa Gamal, Abdelrhman Hussien, Ahmed H. Khalil, and Hassan Mostafa. A Software Defined Radio Transceiver Based on Dynamic Partial Reconfiguration. In *2018 New Generation of CAS (NGCAS)*, pages 158–161, November 2018.

[4] S. S. Sahoo, T. D. A. Nguyen, B. Veeravalli, and A. Kumar. Multi-objective design space exploration for system partitioning of FPGA-based Dynamic Partially Reconfigurable Systems. *Integration*, 67:95–107, 2019.

[5] Ashutosh Dhar, Edward Richter, Mang Yu, Wei Zuo, Xiaohao Wang, Nam Sung Kim, and Deming Chen. DML: Dynamic Partial Reconfiguration with Scalable Task Scheduling for Multi-Applications on FPGAs. *IEEE Transactions on Computers*, pages 1–1, 2021. Conference Name: IEEE Transactions on Computers.

[6] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. FOS: A Modular FPGA Operating System for Dynamic Workloads. *ACM Transactions on Reconfigurable Technology and Systems*, 13(4):20:1–20:28, September 2020.

[7] Joel Mandebi Mbongue, Danielle Tchuinkou Kwadjo, Alex Shuping, and Christophe Bobda. Deploying multi-tenant FPGAs within Linux-based cloud infrastructure. *ACM Trans. Reconfigurable Technol. Syst.*, 15(2), December 2021.

[8] Chao Wang, Xi Li, and Xuehai Zhou. SODA: Software defined FPGA based accelerators for big data. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 884–887, March 2015. ISSN: 1558-1101.

[9] Kizheppatt Vipin and Suhaib A. Fahmy. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Comput. Surv.*, 51(4), jul 2018.

[10] Xilinx. Partial reconfiguration user guide. UG909 (2018).

[11] Xilinx. Vitis Accelerated Libraries Documentation, 2019. (v2022.1).

[12] Xilinx. PYNQ - Python productivity for Zynq, 2021.

[13] Xilinx. ZYNQ Ultrascale+ First Stage Boot Loader Documentation. 2021.

[14] Xilinx. Zynq-7000 SoC Technical Reference Manual, 2021. UG585 (v1.13).

[15] Xilinx. Baremetal Drivers and Libraries Documentation, 2020.

[16] Xilinx. AXI HWICAP LogiCORE IP Product Guide, 2016. PG134 (v3.0).

[17] Ming Liu, Wolfgang Kuehn, Zhonghai Lu, and Axel Jantsch. Run-time partial reconfiguration speed investigation and architectural design space exploration. In *2009 International Conference on Field Programmable Logic and Applications*, pages 498–502, 2009.

[18] Bushra Sultana, Anees Ullah, Arsalan Ali Malik, Ali Zahir, Pedro Reviriego, Fahad Bin Muslim, Nasim Ullah, and Waleed Ahmad. VR-ZYCAP: A Versatile Resourse-Level ICAP Controller for ZYNQ SOC. *Electronics*, 10(8):899, January 2021. Number: 8 Publisher: Multidisciplinary Digital Publishing Institute.

[19] François Duhem, Fabrice Muller, and Philippe Lorenzini. FaRM: Fast reconfiguration manager for reducing reconfiguration time overhead on FPGA. In Andreas Koch, Ram Krishnamurthy, John McAllister, Roger Woods, and Tarek El-Ghazawi, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, pages 253–260, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[20] Robin Bonamy, Hung-Manh Pham, Sébastien Pillement, and Daniel Chillet. UPaRC—Ultra-fast power-aware reconfiguration controller. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1373–1378, March 2012. ISSN: 1558-1101.

[21] Hung-Manh Pham, Van-Cuong Nguyen, and Trong-Tuan Nguyen. DDR2/DDR3-based ultra-rapid reconfiguration controller. In *2012 Fourth International Conference on Communications and Electronics (ICCE)*, pages 453–458, 2012.

[22] Simen Gimle Hansen, Dirk Koch, and Jim Torresen. High speed partial run-time reconfiguration using enhanced ICAP hard macro. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 174–180, 2011.

[23] Francesca Palumbo, Carlo Sau, and Luigi Raffo. Coarse-grained reconfiguration: dataflow-based power management. *IET Computers Digital Techniques*, 9(1):36–48, 2015.

[24] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 5–14, New York, NY, USA, February 2017. Association for Computing Machinery.

[25] Rafael Zamacola, Andrés Otero, and Eduardo de la Torre. Multi-grain reconfigurable and scalable overlays for hardware accelerator composition. *Journal of Systems Architecture*, 121:102302, 2021.

[26] Mohamad Najem, Théotime Bollengier, Jean-Christophe Le Lann, and Loïc Lagadec. Extended overlay architectures for heterogeneous FPGA cluster management. *Journal of Systems Architecture*, 78:1–14, August 2017.

[27] El Mehdi Abdali, Maxime Pelcat, François Berry, Jean-Philippe Diguet, and Francesca Palumbo. Exploring the Performance of Partially Reconfigurable Point-to-point Interconnects. In *12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2017)*, Madrid, Spain, July 2017.

[28] Intel. Partial reconfiguration user guide. UG-20136 (2019).

[29] Anuj Vaishnav, Khoa Pham, Dirk Koch, and James Garside. Resource elastic virtualization for FPGAs using OpenCL. 09 2018.

[30] Khoa Dang Pham, Edson Horta, and Dirk Koch. BITMAN: A tool and API for FPGA bitstream manipulations. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 894–897, March 2017. ISSN: 1558-1101.

[31] Godwin Enemali, Adewale Adetomi, Gopalakrishnan Seetharaman, and Tughrul Arslan. A functionality-based runtime relocation system for circuits on heterogeneous FPGAs. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(5):612–616, 2018.

[32] Xilinx. UG949: UltraFast Design Methodology Guide for Xilinx FPGAs and SoCs. (2021).

[33] Ghada Dessouky, Ahmad-Reza Sadeghi, and Shaza Zeitouni. Sok: Secure fpga multi-tenancy in the cloud: Challenges and opportunities. In *2021 IEEE European Symposium on Security and Privacy*, pages 487–506, 2021.

[34] Miho Yamakura, Ryousei Takano, Akram Ben Ahmed, Midori Sugaya, and Hideharu Amano. A multi-tenant resource management system for multi-FPGA systems. *IEICE Transactions on Information and Systems*, 104(12):2078–2088, 2021.

[35] A. Otero, E. De La Torre, T. Riesgo, T. Cervero, S. Lopez, G. Callico, and R. Sarmiento. Run-time scalable architecture for deblocking filtering in H.264/AVC-SVC video codecs. In *2011 21st International Conference on Field Programmable Logic and Applications*, pages 369–375, 2011.

[36] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 56–63, December 2015.

[37] Rafael Zamacola, Andrés Otero, Alberto García, and Eduardo De La Torre. An integrated approach and tool support for the design of FPGA-based multi-grain reconfigurable systems. *IEEE Access*, 8:202133–202152, 2020.

[38] Rafael Zamacola, Andrés Otero, Alfonso Rodríguez, and Eduardo de la Torre. Just-In-Time Composition of Reconfigurable Overlays. In Francesca Palumbo, João Bispo, and Stefano Cherubin, editors, *13th*

Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022), volume 100 of *Open Access Series in Informatics (OASIcs)*, pages 2:1–2:13, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[39] Andreas Weichslgartner, Stefan Wildermann, Johannes Götzfried, Felix Freiling, Michael Glaß, and Jürgen Teich. Design-Time/Run-Time Mapping of Security-Critical Applications in Heterogeneous MPSoCs. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '16, pages 153–162, New York, NY, USA, May 2016. Association for Computing Machinery.

[40] Zakarya Guettatfi, Paul Kaufmann, and Marco Platzner. Optimal and Greedy Heuristic Approaches for Scheduling and Mapping of Hardware Tasks to Reconfigurable Computing Devices. In *International Symposium on Applied Reconfigurable Computing*, pages 108–117. Springer, 2020.

[41] Dario Korolija, Timothy Roscoe, and Gustavo Alonso. Do OS abstractions make sense on FPGAs? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 991–1010. USENIX Association, November 2020.

[42] Robert Brodersen, Artem Tkachenko, and Hayden Kwok-Hay So. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pages 259–264, October 2006. ISSN: null.

[43] Rafael Zamacola, Alberto García Martínez, Javier Mora, Andrés Otero, and Eduardo de La Torre. IMPRESS: Automated tool for the implementation of highly flexible partial reconfigurable systems with xilinx vivado. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2018.

[44] Diana Göhringer, Michael Hübner, Etienne Nguepi Zeutebouo, and Jürgen Becker. CAP-OS: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor

architectures. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010. ISSN: null.

[45] Stefano Sordillo, Abdallah Cheikh, Antonio Mastrandrea, Francesco Menichelli, and Mauro Olivieri. Customizable vector acceleration in extreme-edge computing: A RISC-V software/hardware architecture study on vgg-16 implementation. *Electronics*, 10(4), 2021.

[46] L. Levinson, R. Manner, M. Sessler, and H. Simmler. Preemptive multitasking on FPGAs. In *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871)*, pages 301–302, April 2000. ISSN: null.

[47] H. Kalte and M. Porrmann. Context saving and restoring for multitasking in reconfigurable systems. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 223–228, August 2005. ISSN: 1946-1488.

[48] Krzysztof Jozwik, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. A Novel Mechanism for Effective Hardware Task Preemption in Dynamically Reconfigurable Systems. In *2010 International Conference on Field Programmable Logic and Applications*, pages 352–355, August 2010. ISSN: 1946-1488.

[49] Markus Happe, Andreas Traber, and Ariane Keller. Preemptive Hardware Multitasking in ReconOS. In Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, Lecture Notes in Computer Science, pages 79–90, Cham, 2015. Springer International Publishing.

[50] Marcel Eckert, Dominik Meyer, and Bernd Klauer. Context Save and Restore of Partial Reconfiguration Regions for Xilinx FPGAs. In *2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 5–12, July 2019. ISSN: 2642-7222.

[51] Dirk Koch, Christian Haubelt, and Jürgen Teich. Efficient hardware checkpointing: concepts, overhead analysis, and implementation. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, pages 188–196, 2007.

[52] Krzysztof Jozwik, Hiroyuki Tomiyama, Masato Edahiro, Shinya Honda, and Hiroaki Takada. Comparison of Preemption Schemes for Partially Reconfigurable FPGAs. *IEEE Embedded Systems Letters*, 4(2):45–48, June 2012. Conference Name: IEEE Embedded Systems Letters.

[53] Alban Bourge, Olivier Muller, and Frédéric Rousseau. Generating Efficient Context-Switch Capable Circuits Through Autonomous Design Flow. *ACM Trans. Reconfigurable Technol. Syst.*, 10(1):9:1–9:23, December 2016.

[54] Ye Tian, Jean-Christophe Prevotet, and Fabienne Nouvel. Efficient OS Hardware Accelerators Preemption Management in FPGA. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 367–370, December 2019.

[55] Slavisa Jovanovic, Camel Tanougast, and Serge Weber. A hardware preemptive multitasking mechanism based on scan-path register structure for FPGA-based reconfigurable systems. In *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007)*, pages 358–364. IEEE, 2007.

[56] Kyle Rupnow, Wenyin Fu, and Katherine Compton. Block, Drop or Roll(back): Alternative Preemption Methods for RH Multi-Tasking. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 63–70, April 2009.

[57] Chabha Hireche, Catherine Dezan, Stéphane Mocanu, Dominique Heller, and Jean-Philippe Diguet. Context/Resource-Aware Mission Planning Based on BNs and Concurrent MDPs for Autonomous UAVs. *Sensors*, 18(12):4266, December 2018. Number: 12 Publisher: Multidisciplinary Digital Publishing Institute.

[58] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Automation and Test in Europe Conference and Exhibition 2003 Design*, pages 986–991, March 2003. ISSN: 1530-1591.

[59] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass. hthreads: a hardware/software co-designed multithreaded RTOS kernel. In *2005 IEEE Conference on Emerging Technologies and*

Factory Automation, volume 2, pages 8 pp.–338, September 2005. ISSN: 1946-0759.

[60] FOSFOR - ANR Project - "Architectures du futur" Polytech Nice Sophia website.

[61] Fabrice Muller, Jimmy Le Rhun, Fabrice Lemonnier, Benoît Miramond, and Ludovic Devaux. A Flexible Operating System for Dynamic Applications. *XCell*, (73):30–34, November 2010.

[62] Devaux Ludovic, Sana Sassi, Sébastien Pillement, Daniel Chillet, and D. Demigny. Flexible interconnection network for dynamically and partially reconfigurable architectures. *International Journal of Reconfigurable Computing*, 2010, 01 2010.

[63] FUSE: Front-end user framework for OS abstraction of hardware accelerators.

[64] Enno Lübbers and Marco Platzner. Reconos: An operating system for dynamically reconfigurable hardware. In *Dynamically Reconfigurable Systems*, pages 269–290. Springer, 2010.

[65] Andreas Agne, Marco Platzner, Christian Plessl, Markus Happe, and Enno Lübbers. ReconOS. In Dirk Koch, Frank Hannig, and Daniel Ziener, editors, *FPGAs for Software Programmers*, pages 227–244. Springer International Publishing, Cham, 2016.

[66] Syam Sanal and J. Pinalkumar. Multithreaded Image Processing Using ReconOS on Reconfigurable Computing System. In *2018 International Conference on Emerging Trends and Innovations In Engineering And Technological Research (ICETIETR)*, pages 1–5, July 2018. ISSN: null.

[67] Ying Wang, Xuegong Zhou, Lingli Wang, Jian Yan, Wayne Luk, Chenglian Peng, and Jiarong Tong. Spread: A streaming-based partially reconfigurable architecture and programming model. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2179–2192, 2013.

[68] Andreas Ehliar and Dake Liu. An FPGA based open source network-on-chip architecture. In *2007 International Conference on Field Programmable Logic and Applications*, pages 800–803, 2007.

[69] Hai-Dang Vu, S. Le Nours, S. Pillement, Ralf Stemmer, and Kim Grüttner. A Fast Yet Accurate Message-level Communication Bus Model for Timing Prediction of SDFGs on MPSoC. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 17–22, January 2021. ISSN: 2153-697X.

[70] Ralf Stemmer, Hai-Dang Vu, Sébastien Le Nours, Kim Grüttner, Sébastien Pillement, and Wolfgang Nebel. A measurement-based message-level timing prediction approach for data-dependent SDFGs on tile-based heterogeneous mpsocs. *Applied Sciences*, 11(14), 2021.

[71] Cornelia Wulf, Michael Willig, and Diana Goehringer. RTOS-supported low power scheduling of periodic hardware tasks in flash-based FPGAs. *Microprocessors and Microsystems*, 92:104566, 2022.

[72] A. Pérez, A. Rodríguez, A. Otero, D. G. Arjona, Á Jiménez-Peralo, M. Á Verdugo, and E. De La Torre. Run-Time Reconfigurable MPSoC-Based On-Board Processor for Vision-Based Space Navigation. *IEEE Access*, 8:59891–59905, 2020. Conference Name: IEEE Access.

[73] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–12, November 2016.

[74] Alessandro Biondi and Giorgio Buttazzo. Timing-aware FPGA partitioning for real-time applications under dynamic partial reconfiguration. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 172–179, July 2017. ISSN: 2471-769X.

[75] Ta-Wei Liu, Yen-Fang Liu, and Ya-Shu Chen. Energy-aware run-time task partition and allocation in dynamic partial reconfigurable systems. *Journal of Systems Architecture*, 78:55–67, August 2017.

[76] Sanjit Kumar Roy, Rajesh Devaraj, Arnab Sarkar, and Debabrata Senapati. SLAQA: Quality-level Aware Scheduling of Task Graphs on Heterogeneous Distributed Systems. *ACM Transactions on Embedded Computing Systems*, 20(5):1–31, 2021.

[77] Tetsuro Nakamura, Shogo Saito, Kei Fujimoto, Masashi Kaneko, and Akinori Shiraga. Spatial- and time- division multiplexing in CNN accelerator. *Parallel Computing*, 111:102922, 2022.

[78] Siva Satyendra Sahoo, Tuan D. A. Nguyen, Bharadwaj Veeravalli, and Akash Kumar. QoS-Aware Cross-Layer Reliability-Integrated FPGA-Based Dynamic Partially Reconfigurable System Partitioning. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 230–233, Naha, Okinawa, Japan, December 2018. IEEE.

[79] Obinna Izima, Ruairí de Fréin, and Ali Malik. A Survey of Machine Learning Techniques for Video Quality Prediction from Quality of Delivery Metrics. *Electronics*, 10(22):2851, January 2021. Number: 22 Publisher: Multidisciplinary Digital Publishing Institute.

[80] Qingxiao Sun, Liu Yi, Hailong Yang, Mingzhen Li, Zhongzhi Luan, and Depei Qian. QoS-aware dynamic resource allocation with improved utilization and energy efficiency on GPU. *Parallel Computing*, 113:102958, 2022.

[81] Samuel Isuwa, Somdip Dey, Andre P. Ortega, Amit Kumar Singh, Bashir M. Al-Hashimi, and Geoff V. Merrett. QUAREM: Maximising QoE through adaptive resource management in mobile MPSoC platforms. *ACM Trans. Embed. Comput. Syst.*, 21(4), sep 2022.

[82] Boonyarith Saovapakhiran, Wibhada Naruephiphat, Chalermpol Charnsripinyo, Sebnem Baydere, and Suat Özdemir. QoE-driven IoT architecture: A comprehensive review on system and resource management. *IEEE Access*, 10:84579–84621, 2022.

[83] Soguy Mak-Karé Gueye, Eric Rutten, and Jean-Philippe Diguet. Autonomic Management of Missions and Reconfigurations in FPGA-based Embedded System. In *The 2017 NASA/ESA Conference on Adaptive Hardware and Systems*, page 8, Pasadena, California, United States, July 2017.

[84] Xiaokun Yang and Shi Sha. Exploiting Energy–Quality (E–Q) Tradeoffs: A Case Study on Color-to-Grayscale Converters with Approximate Design on FPGA. *Journal of Circuits, Systems and Computers*, page 2150062, June 2020. Publisher: World Scientific Publishing Co.

[85] Adil Iguider, K. Bousselam, Oussama Elissati, M. Chami, and Abdeslam En-Nouaary. Heuristic algorithms for multi-criteria hardware/software partitioning in embedded systems codesign. *Comput. Electr. Eng.*, 2020.

[86] Yi Tang and Neil W. Bergmann. A hardware scheduler based on task queues for FPGA-based embedded real-time systems. *IEEE Trans. Comput.*, 64(5):1254–1267, may 2015.

[87] Alexander Fusco, Sahil Hassan, Joshua Mack, and Ali Akoglu. Hardware-based scheduler implementation for dynamic workloads on heterogeneous socs, 2022.

[88] Hamid Arabnejad and Jorge Barbosa. List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table. *IEEE Transactions on Parallel and Distributed Systems*, 25:682–694, March 2014.

[89] Ashish Kumar Maurya and Anil Kumar Tripathi. On benchmarking task scheduling algorithms for heterogeneous computing systems. *The Journal of Supercomputing*, 74(7):3039–3070, July 2018.

[90] Reza Ramezani. A prefetch-aware scheduling for FPGA-based multitask graph systems. *The Journal of Supercomputing*, January 2020.

[91] Alexander Dorflinger, Mark Albers, Johannes Schlatow, Bjorn Fiethe, Harald Michalik, Phillip Keldenich, and S'andor P. Fekete. Hardware and Software Task Scheduling for ARM-FPGA Platforms. In *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 66–73, August 2018. ISSN: 2471-769X.

[92] E. A. Deiana, M. Rabozzi, R. Cattaneo, and M. D. Santambrogio. A multiobjective reconfiguration-aware scheduler for FPGA-based heterogeneous architectures. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, December 2015.

[93] Zhe Wang, Qi Tang, Biao Guo, Ji-Bo Wei, and Ling Wang. Resource Partitioning and Application Scheduling with Module Merging on Dynamically and Partially Reconfigurable FPGAs. *Electronics*, 9(9):1461, September 2020. Number: 9 Publisher: Multidisciplinary Digital Publishing Institute.

[94] Y. Ma, J. Liu, C. Zhang, and W. Luk. HW/SW partitioning for region-based dynamic partial reconfigurable FPGAs. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 470–476, October 2014. ISSN: 1063-6404.

[95] Qi Tang, Biao Guo, and Zhe Wang. Sw/Hw Partitioning and Scheduling on Region-Based Dynamic Partial Reconfigurable System-on-Chip.

Electronics, 9(9):1362, September 2020. Number: 9 Publisher: Multidisciplinary Digital Publishing Institute.

[96] Zakarya Guettatfi, Omar Kermia, and Abdelhakim Khouas. Over effective hard real-time hardware tasks scheduling and allocation. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–2, September 2015. ISSN: 1946-1488.

[97] Luiz F. Bittencourt, Rizos Sakellariou, and Edmundo R. M. Madeira. DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 27–34, February 2010. ISSN: 2377-5750.

[98] George Charitopoulos, Iosif Koidis, Kyprianos Papadimitriou, and Dionisios Pnevmatikatos. Run-time management of systems with partially reconfigurable FPGAs. *Integration*, 57:34–44, 2017. Publisher: Elsevier.

[99] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[100] Matteo Bertolino, Renaud Pacalet, Ludovic Apvrille, and Andrea Enrici. Multi-resource scheduling for FPGA systems. *Microprocessors and Microsystems*, 87:104373, 2021.

[101] Reza Ramezani. Dynamic scheduling of task graphs in multi-FPGA systems using critical path. *The Journal of Supercomputing*, 77(1):597–618, January 2021.

[102] Abhishek Kumar Jain, Douglas L. Maskell, and Suhaib A. Fahmy. Are Coarse-Grained Overlays Ready for General Purpose Application Acceleration on FPGAs? In *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 586–593, August 2016.

[103] Jan Spieck, Stefan Wildermann, and Jürgen Teich. Scenario-Based Soft Real-Time Hybrid Application Mapping for MPSoCs. In *2020 57th*

ACM/IEEE Design Automation Conference (DAC), pages 1–6, July 2020. ISSN: 0738-100X.

[104] B.D. Theelen, M.C.W. Geilen, T. Basten, J.P.M. Voeten, S.V. Gheorghita, and S. Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.*, pages 185–194, July 2006. ISSN: null.

[105] D. Marpe, T. Wiegand, and G.J. Sullivan. The H.264/MPEG4 advanced video coding standard and its applications. *IEEE Communications Magazine*, 44(8):134–143, 2006.

[106] Kurt Debattista, Keith Bugeja, Sandro Spina, Thomas Bashford-Rogers, and Vedad Hulusic. Frame Rate vs Resolution: A Subjective Evaluation of Spatiotemporal Perceived Quality Under Varying Computational Budgets. *Computer Graphics Forum*, September 2017.

[107] P. Lambert, W. De Neve, Y. Dhondt, and R. Van de Walle. Flexible macroblock ordering in h.264/avc. *Journal of Visual Communication and Image Representation*, 17(2):358–375, 2006. Introduction: Special Issue on emerging H.264/AVC video coding standard.

[108] Pamela C Cosman, Robert M Gray, and Richard A Olshen. Evaluating quality of compressed medical images: Snr, subjective rating, and diagnostic accuracy. *Proceedings of the IEEE*, 82(6):919–932, 1994.

[109] Quentin Dariol, Sebastien Le Nours, Sebastien Pillement, Ralf Stemmer, Domenik Helms, and Kim Grüttner. A hybrid performance prediction approach for fully-connected artificial neural networks on multi-core platforms. In Alex Orailoglu, Marc Reichenbach, and Matthias Jung, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 250–263, Cham, 2022. Springer International Publishing.

[110] Phillip H. Jones, Young H. Cho, and John W. Lockwood. Dynamically Optimizing FPGA Applications by Monitoring Temperature and Workloads. In *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, pages 391–400, January 2007. ISSN: 2380-6923.

[111] Behnam Khaleghi and Tajana Šimunić Rosing. Thermal-Aware Design and Flow for FPGA Performance Improvement. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 342–347, March 2019. ISSN: 1558-1101.

[112] Ashish Kumar Maurya and Anil Kumar Tripathi. Performance Comparison of HEFT, Lookahead, CEFT and PEFT Scheduling Algorithms for Heterogeneous Computing Systems. In *Proceedings of the 7th International Conference on Computer and Communication Technology - ICCCT-2017*, pages 128–132, Allahabad, India, 2017. ACM Press.

[113] Athena Abdi and Hamid R. Zarandi. HYSTERY: a hybrid scheduling and mapping approach to optimize temperature, energy consumption and lifetime reliability of heterogeneous multiprocessor systems. *The Journal of Supercomputing*, 74(5):2213–2238, May 2018.

[114] Dowhan Jeong, Jangryul Kim, Mari-Liis Oldja, and Soonhoi Ha. Parallel scheduling of multiple sdf graphs onto heterogeneous processors. *IEEE Access*, 9:20493–20507, 2021.

[115] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29(3):682–692, 2013. Special Section: Recent Developments in High Performance Computing and Security.

[116] Yohei Hori, Toshihiro Katashita, and Kazukuni Kobara. Energy and area saving effect of dynamic partial reconfiguration on a 28-nm process fpga. In *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*, pages 217–218. IEEE, 2013.

[117] Tian Gao, Zishen Wan, Yuyang Zhang, Bo Yu, Yanjun Zhang, Shaoshan Liu, and Arijit Raychowdhury. iELAS: An ELAS-Based Energy-Efficient Accelerator for Real-Time Stereo Matching on FPGA Platform. *arXiv:2104.05112 [cs]*, April 2021. arXiv: 2104.05112.

Appendix A

Scientific communications

Published

- **Duhamel, A.**, Pillement, S. "QoS Aware Design-Time/Run-Time Manager for FPGA-Based Embedded Systems" (2022). In: Desnos, K., Pertuz, S. (eds) Design and Architecture for Signal and Image Processing (DASIP). Lecture Notes in Computer Science (LNCS), vol 13425. Springer, Cham. DOI: 10.1007/978-3-031-12748-9_8

- **Duhamel, A.**, Pillement, S., Kouki, W. "Gestion orienté qualité d'expérience de systèmes embarqués reconfigurables par réutilisations de modules" [*Quality of experience oriented management of reconfigurable embedded systems with module reuse*] (2022). In: Groupe de Recherche et d'Etudes de Traitement du Signal et des Images (GRETSI). Available at: gretsi.fr/colloque2022/programme/

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Nantes Université

Titre : **Développement d'un contrôleur dynamique d'allocation de ressources pour FPGAs reconfigurables partiellement avec approche garantie de service**

Mots clés : FPGA, reconfiguration dynamique partielle (RDP), gestion d'allocation de ressources, garantie de service, accélération matérielle

Résumé : Les FPGAs dynamiquement reconfigurables permettent le changement d'accélérateurs matériels au temps de l'exécution. Cette technique permet notamment de réduire la taille des FPGAs dans les systèmes embarqués, réduisant les coûts de fabrication et la consommation d'énergie. Dès lors, de nouvelles problématiques de conception d'architectures et de leur gestion se posent, afin d'exploiter au mieux cette technique. La question de la garantie d'exécution des services se pose notamment en raison des besoins changeants des applications embarquées et de la complexité des algorithmes de gestion des ressources.
L'objectif de ce travail est de proposer une méthodologie de gestion d'allocation des ressources matérielles afin de garantir un niveau minimum de service d'une application. Pour cela, un modèle de qualité est présenté, permettant de qualifier le niveau de service d'une application exécutée sur une architecture dynamiquement reconfigurable. Ce modèle de qualité est utilisé afin de proposer deux méthodes permettant de gérer dynamiquement l'allocation des régions reconfigurables tout en maximisant la qualité du service rendu par le système. Enfin, un algorithme d'ordonnancement rapide et performant est introduit, permettant d'exploiter les caractéristiques des architectures dynamiquement reconfigurables.
Les résultats obtenus sur un ensemble de benchmarks démontrent l'efficacité de l'approche proposée.

Title : **Development of a dynamic resource allocation controller for partially reconfigurable FPGAs with service guarantee approach**

Keywords : FPGA, Dynamic partial reconfiguration (DPR), Resource allocation management, Service guarantee, Reconfigurable computing, Hardware acceleration

Abstract : Embedded systems based on dynamically reconfigurable FPGAs allow hardware accelerators to be swapped at runtime. This technique enables to reduce the size of FPGAs in embedded systems, reducing manufacturing costs and energy consumption. From then on, new challenges of architecture design and management arise, in order to make the most of this technique. Guarantee of service execution should be observed as embedded systems applications have changing computational needs, and the time complexity of resource allocation algorithms introduce latency overheads.
The objective of this thesis is to propose a methodology for managing the allocation of hardware resources to guarantee a minimum level of service of an application. A quality model is introduced, allowing to qualify the service level of an application executed on a dynamically reconfigurable architecture. This quality model is used to propose two methodologies to dynamically manage the allocation of reconfigurable regions while maximizing the quality of service provided by the system. Finally, a fast and efficient scheduling algorithm is introduced to exploit the characteristics of our dynamically reconfigurable architecture.
Results on a set of benchmarks demonstrate the effectiveness of the proposed approaches.