Towards Efficient Reuse of Software Programmable Streaming Coarse Grained Reconfigurable Architectures

Elias Barbudo Franco

Université Gustave Eiffel

École doctorale n° 532 :
Mathématiques et Sciences et Technologies
de l'Information et de la Communication (MSTIC)

THÈSE

pour obtenir le grade de docteur délivré par

UNIVERSITE GUSTAVE EIFFEL

présentée publiquement par

Elias Barbudo Franco

Towards Efficient Reuse of Software Programmable Streaming Coarse Grained Reconfigurable Architectures

Jury		
Prof.,	Alejandro Castillo Atoche	Rapporteur
Prof.,	Jean-François Nezan	Rapporteur
Prof.,	Michel Paindavoine	Examinateur
Prof.,	Carlos Valderrama	Examinateur
Prof. Associé,	Thierry Grandpierre	Co-directeur de thèse
Prof.,	Eva Dokladalova	Directrice de thèse

LIGM, Univ Gustave Eiffel
CNRS, ESIEE Paris, F-77454 Marne-la-Vallée

Abstract

Coarse-Grained Reconfigurable Architectures (CGRA) are designed to deliver high performance while drastically reducing the latency of the computing system. There are several types of CGRA according to the structure, application, type of resources, and memory infrastructure. We focus our work on a subset of CGRA designs that we call Software Programmable Streaming Coarse-Grained Reconfigurable Architectures (SPS-CGRA). An SPS-CGRA is a more or less complex array of coarse-grained heterogeneous hardware resources with a coarser granularity than the classical. An SPS-CGRA can perform spatial and temporal computations at low latency. Its stream-based processing provides high performance maintaining a level of flexibility. Although they are often highly domain-specifically optimized, they keep several levels of custom post-fabrication programmability, given by a set of parameters, so that they can be reused. However, their reuse is generally limited due to the complexity of identifying the best allocation of the processing tasks into the hardware resources. Another limiting point is the complexity of producing a reliable performance analysis for each new implementation since no mature tool exists.

To solve these problems, we propose a complete mapping and scheduling framework that targets SPS-CGRA. We introduce a generic hardware model allowing one to express these intrinsically custom levels of flexibility without neglecting data access and system configuration control. We also propose a performance estimation analysis based on resource latency description, allowing to obtain the upper bound of the computing cost. To complete, we present four different solutions for the mapping and scheduling problem: a List-based algorithm with backtracking, a Lookahead-based heuristic, a Bayesian-based heuristic and, a Q-Learning mapping algorithm. We evaluate and compare our solutions against an exhaustive approach in a real-life example and illustrate the benefits and efficiency of the proposed framework.

Résumé

Les architectures reconfigurables à gros grains (CGRA) sont conçues pour offrir des performances élevées tout en réduisant considérablement la latence du système informatique. Il existe plusieurs types de CGRA en fonction de la structure, de l'application, du type de ressources et de l'organisation de la mémoire. Nous concentrons notre travail sur un sous-ensemble de conceptions CGRA que nous appelons les architectures gros grain, flot de données, reconfigurables et programmables (SPS-CGRA). Un SPS-CGRA est une grille plus ou moins complexe de ressources matérielles hétérogènes à gros grains avec une granularité plus grande que les architectures CGRA classiques. Un SPS-CGRA peut effectuer de grandes quantités de calculs avec une faible latence. Son principe de traitement orienté flot de données offre des performances élevées tout en maintenant un niveau élevé de flexibilité. Bien qu'ils soient souvent hautement optimisés pour un domaine spécifique, ils conservent plusieurs niveaux de programmation après la phase de configuration Cette programmation s'effectue par le biais de paramètres, rendant ainsi possible leur réutilisation. Cependant, cette réutilisation est généralement limitée en raison de la complexité de l'identification de la meilleure allocation des tâches de traitement sur les ressources matérielles. Un autre facteur limitant la réutilisation est la complexité à produire une analyse de performance fiable pour chaque nouvelle implémentation car généralement il n'existe aucun outil spécifique pour explorer et exploiter pleinement le potentiel des architectures ainsi produites.

Pour résoudre ces problèmes, nous proposons un cadre complet de distribution et d'ordonnancement qui cible les SPS-CGRA. Nous introduisons un modèle théorique et générique de l'architecture matérielle permettant d'exprimer ces niveaux de flexibilité intrinsèquement personnalisés, ainsi que l'accès aux données et le contrôle de la configuration du système, souvent négligés dans les travaux existants. Nous proposons également une analyse d'estimation des performances, basée sur la latence des ressources. Pour compléter, nous présentons quatre solutions différentes pour résoudre le problème de distribution et d'ordonnancement : un algorithme avec retour en arrière basé sur des listes, une heuristique basée sur les algorithme de type "lookahead", une heuristique basée sur un algorithme Bayésien et un algorithme d'ordonnancement basé sur le Q-learning. Pour finir, nous évaluons et comparons nos solutions sur des ensembles d'architectures et d'applications dont les paramètres sont générés aléatoirement, ainsi que sur deux applications réelles.

Acknowledgements

First, I would like to express my gratitude to my director Eva Dokladalova, thank you for your guidance and support. I want to thank my co-director Thierry Grandpierre, it was a pleasure to benefit and learn from your knowledge.

I am very thankful to the reviewers of this thesis, Alejandro Castillo and Jean-François Nezan, your insights, and bits of advice are deeply appreciated. I also want to thank Michel Paindavoine and Carlos Valderrama for being part of my thesis jury. I am grateful for your advice and words towards this work.

I would like to thank CONACYT for financially support this work, and to the Université Gustave Eiffel, ESIEE Paris, and Laboratory Gaspard Monge, where this work was prepared.

To my parents, Ruddy and Elvira, I am deeply grateful for your support and love. To my sister Angelica, thank you very much for all your advice and help. To my brother Ruddy, thank you very much for your support. I thank all my family for being there for me. Special thanks to my lovely Noemi, who helped me through this entire journey.

I would like to thank Rosemberg Rodriguez for his friendship and support. My appreciation goes to all my friends from Mexico and France. I am deeply grateful that there was always someone to talk to and share my thoughts.

Contents

List of Figures

List of Tables

Acronyms

AAM Algorithm Architecture Matching methodology

ALU Arithmetic Logic Unit

ARM Advanced RISC Machines

ASIC Application-Specific Integrated Circuit

ASIP Application-Specific Intructions-Set Processor

CGRA Coarse-Grained Reconfigurable Architecture

CPU Control Processing Unit

DAG Directed Acyclic Graph

DSE Design Space Exploration

DSP Digital Signal Processor

FPGA Field Programmable Gate Array

GPP General-Purpose Processor

GPU Graphics Processing Unit

HPC High-Performance Computing

MCPU Morphological Co-Processing Unit

MPSoC Multiprocessor System on a Chip

NoC Networks-on-Chip

RL Reinforcement Learning

RRG Resource Routing Graph

SDF Synchronous Dataflow Graph

SFG Signal Flow Graph

SoPC System on Programmable Chip

SPS-CGRA Software Programmable Streaming Coarse Grained Reconfigurable Architecture

WCET Worst-Case Execution Time

VDHL VHSIC Hardware Description Language

XML Extensible Markup Language

Chapter 1

Introduction

Time-critical systems need to assure the accuracy of output while meeting hard timing constraints. This paradigm increases its importance with the advent of the industrial IoT, autonomous vehicles, drone-based applications, and smart grids. Often, these new applications must process a significant amount of data from a wide range of sensors. Furthermore, the data must be processed with maximal reactivity, respecting the capabilities of the hardware resources and ensuring a valid output. This creates a non-trivial problem for the latency optimization of processing systems.

To answer these constraints, an extensive number of hardware architectures have been proposed in the past, trying to find the best trade-off between efficiency and execution constraints. We can cite examples going from General-Purpose Processors (GPPs), Digital Signal Processors (DSPs), and Graphics Processing Units (GPUs) up to Application-Specific Intructions-Set Processors (ASIPs), Field Programmable Gate Array (FPGAs), Coarse-Grained Reconfigurable Architectures (CGRAs), and Application-Specific Integrated Circuit (ASIC).

1.1 Reusability of Time-Critical Systems

The increasing complexity of time-critical systems and their applications make designers lean towards the reuse of parts of hardware blocks and systems. Since time-to-market constraints are often an industrial priority, and to design a new hardware block from scratch takes a considerable amount of time, the best option is to exploit the already developed platforms' programmable or configurable blocks.

Reusability is the attribute of a system that allows to use it in different applications. It is the use of pre-designed and pre-verified platforms or hardware blocks that reduces time-to-market and ensures a previously known performance [16]. The level of reusability of a given platform includes the following [17]:

- Hardware features: programmability and parametrization of the hardware blocks, the complexity of the target applications, type of interfaces.
- Software tools: programming software and support, simulation environments.
- Standardization support (norms and protocols): documentation of parameters, requirements, and restrictions.

In Figure 1.1, we compare the efficiency with respect to the reusability exploitation of different platforms used for implementing time-critical systems. Efficiency is the relation of performance with regard to power consumption, as reported in the literature [18, 19]. There, we can observe that GPPs, DSPs, and GPUs have a broad support structure and can be used for many applications. However, their performance is lower than an FPGA and an ASIC. ASIPs are better than GPUs and GPPs in terms of efficiency, given the frequent addition of custom instructions

and functional units. However, they remain below FPGAs and ASICs, in terms of efficiency, due to their completely specialized execution engines [19, 20]. FPGAs offer high performance with low consumption and with a significant support environment. Nonetheless, this architecture's management usually requires low-level knowledge, and its performance is lower than the ASICs. ASICs offer the best efficiency, but they require a very rare low-level knowledge and a costly design process. Therefore their reusability is severely reduced. Finally, in the above-mentioned context, Coarse-Grained Reconfigurable Architectures (CGRAs) provide the best ratio between the increase of the overall performance while decreasing computing latency and minimizing the energy budget [21]. Their reuse capability depends on a degree of programmability provided by a set of parameters, often custom and positioned between general-purpose and fixed-function, defining the possibility of their "on-line or off-line re-programming" [18].

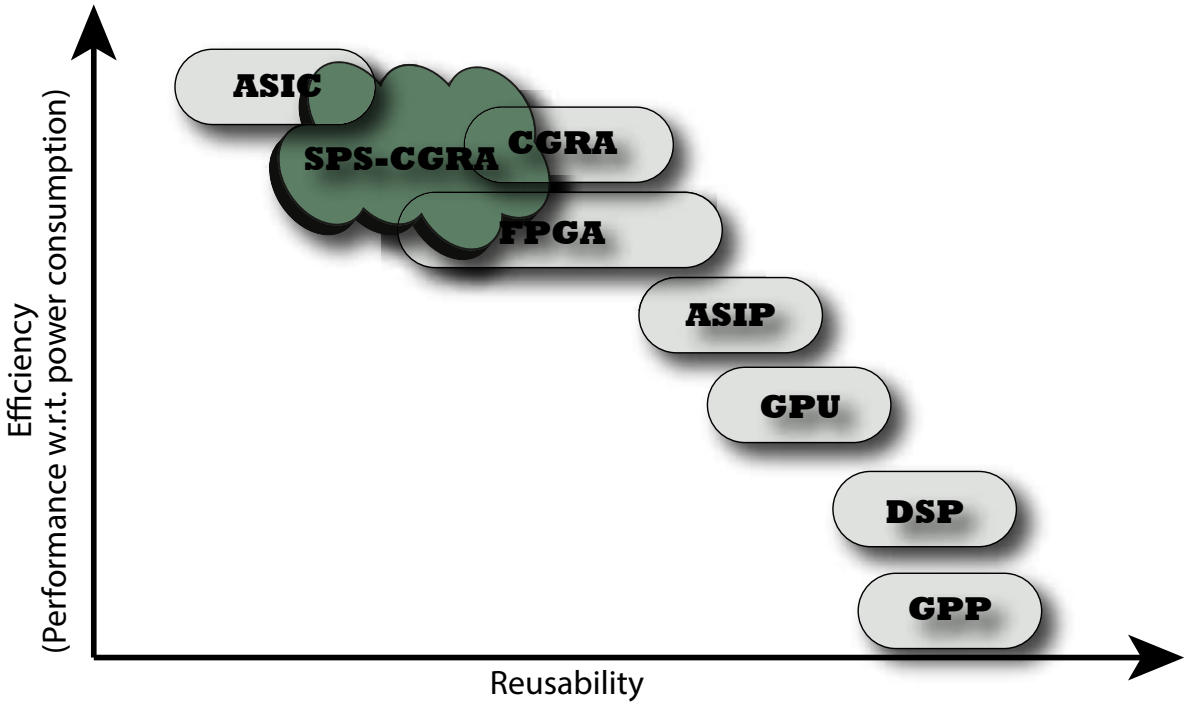


Figure 1.1 – Comparative study of hardware architectures from reusability point of view.

Effectively, CGRAs are optimized for a given application domain. There is a large diversity of CGRAs types. Their processing elements are not so generic as the ones of GPPs or DSPs. Usually, a CGRA consists of a set of ALU-like processing resources organized according to some well-defined topology for a dedicated application [22]. Hence, their interconnections and granularity have a reduced complexity than those of FPGAs. Also, their memory infrastructure allows decreasing the overall memory transactions compared to a GPU.

Based on these re-programming capabilities, we can distinguish an inner CGRA family that we call Software Programmable Streaming Coarse-Grained Reconfigurable Architecture (SPS-CGRA). An SPS-CGRA is a systolic, highly pipelined array of heterogeneous hardware resources. It is a spatially-configured overlay [23], based on an FPGA, can be realized as ASIC or be a part of a System on Programmable Chip (SoPC). Its hardware resources are defined by an initial "off-line" structural reconfiguration [24]. After this step, a degree of programmability is provided through reconfigurable or reprogrammable parameter sets, called configuration contexts [25]. Each hardware resource may be equipped with multiple contexts that may be individually switched through its internal registers' modification. An SPS-CGRA is a data-driven platform that may perform not only loops but entire applications at low latency. Its stream-based processing provides high performance while maintaining a level of flexibility.

The use of SPS-CGRAs is widespread across numerous fields. In the cryptocurrency domain, the implementation of hashing algorithms requires a massive amount of computational power. This may be solved using an FPGA-based SPS-CGRA, as depicted in Tetu et al. [8] where an implementation of the Lyra2REv2 algorithm, a standard in cryptocurrency, is presented. Mathematical algorithms can also take advantage of the systolic structure of an SPS-CGRA. For example, we can cite the FPGA-based implementation of an Algebraic Multigrid Solver presented by Haghi et al. [26]. Sound field rendering models the behavior of sound wave propagation in spatial and time domain using numerical methods, and its implementation and efficiency are crucial for several engineering application fields. Tan et al. [27, 28] proved that SPS-CGRA solutions offer better performance than a GPP implementation. In the computer vision domain, SPS-CGRAs are used to enhance the performance of image processing systems. Examples of these architectures are the FPGA-based SPS-CGRAs presented by Ngan et al. [29] and by Isavudeen et al. [10]; and also the ASIC-based SPS-CGRA introduced by Dokladalova et al. [30]. Deep Learning is another field where the use of SPS-CGRA is growing. Since a Neural Network (NN) is modeled as a Directed Acyclic Graph (DAG) and its dataflow is unidirectional, an SPS-CGRA is a perfect candidate to be used as an implementation platform. We can cite, for example, the ASIC-based Neuflow [31], and FlexFlow [32], the FPGA-based implementations of a Deconvolutional NN presented by Chang et al. [13], a Life Long learning Convolutional NN by Piyasena et al. [14] and a Randomly Wired NN develop by Kuramochi and Nakahara [33]. Some of these examples will be detailed in Section 1.4.

Unfortunately, despite the high performance that an SPS-CGRA can deliver while decreasing latency of processing, its reuse is not generalized due to the lack of generic tools or frameworks. This problem is inherited from the CGRA, based on manual or custom programming frameworks, that can not be easily transferred (or adapted) to other CGRA-based systems [11, 12].

Consequently, the programming and compiling frameworks remain immature. Hence, the mapping of any application onto an SPS-CGRA is often manual, requiring expert knowledge of both inner hardware mechanisms and application specificity. This reuse task could become overwhelming, leading to the rare reuse of these powerful SPS-CGRA platforms.

To face these issues, we propose a generic automated mapping and scheduling framework that targets most of the existing SPS-CGRA (Fig. 1.2).

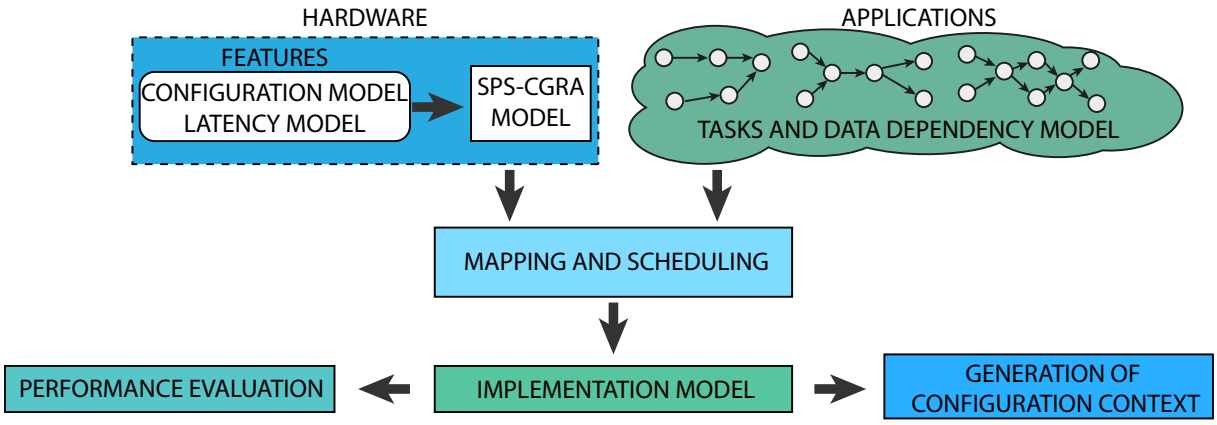


Figure 1.2 – Overall scheme of proposed modeling and mapping framework for SPS-CGRA.

Such a framework must be based on a model that allows describing the heterogeneity of any SPS-CGRA hardware element programming models. Simultaneously, this model has to be generic enough to match the maximum of SPS-CGRAs variants in granularity and application fields. Also, considering the need to optimize the system's reactivity, it has to provide an accurate, near-to-real performance latency estimation. Additionally, such a framework should offer mapping assistance to help the designer exploit this type of hardware architecture's inherent parallelism. Finally, the mapping assistance should aim to decrease the computing cost, a critical

metric in these systems.

In this context, the main contributions of this thesis are:

- A unified modeling framework that includes application, hardware and selected implementation.
- We propose to use processing latency as a metric of computing cost to be optimized. The latency can be described as a function of the hardware block programming parameters.
- A computation model to estimate the upper bound of the computing cost of the implementation selected by the mapping algorithm.
- We propose and evaluate four different mapping approaches.
    1. A List-based algorithm equipped with backtracking.
    2. A Topology-aware mapping heuristic.
    3. A Bayes-based mapping heuristic.
    4. A Q-learning-based mapping algorithm.

    To complete, we evaluate the algorithms in terms of exploration time and computing cost of the implementation.

Additionally, we introduce a complete benchmark framework with the previous items, including a configuration control generator.

The remainder of the chapter is divided as follows. In Section 1.2, we review state of the art on development tools and methodologies. Section 1.3 describes in detail the characteristics of a CGRA. Section 1.4 introduces the concept of SPS-CGRA. Section 1.5 presents the thesis contributions. Section 1.6 presents the outline of the thesis.

1.2 State of the Art: Development Tools and Methodologies

As introduced previously, reusability is the attribute of a system that allows one to use it in different applications. This aspect directly impacts time-to-market and provides the opportunity to use already developed and tested tools or frameworks. As stated in Section 1.1, reusability includes three aspects: hardware, software, and documentation. However, most of the designers of SPS-CGRA focus only on the hardware aspect, providing a set of programmable parameters to allow the use in different applications. Thus, we can observe a lack of tools and methodologies that could provide the software support. In this context, tools like Design Space Exploration (DSE) are often used to bridge this reuse technique gap.

A DSE is the systematic evaluation process of the different implementations (mappings) of an application onto a hardware system to determine optimal or best-suited hardware. This process can be manual, automatic, or hybrid. In a manual DSE, the engineer is in full charge to find the optimal implementation, while in an automatic DSE, the framework takes the implementation decision. In a hybrid DSE, the framework takes the decisions with a fine tunning from the engineer. This evaluation process may generate many possibilities due to the size of the design space. As the number of possible implementations increases, a complete, exhaustive exploration may be prohibitive. For this reason, several works propose different approaches for efficient DSE.

Giannopoulou et al. [34] presented a DSE framework for many/multicores, part of the Certification of Real-Time Applications designed for mixed-criticality (CERTAINTY) project, aiming at avionics applications. It uses a mixed-critically task application model with multiple critical levels and different task activation patterns and a platform model that allows to abstract of the memory and the communications of the system. A mixed-critically scheduling policy is

used, where only equal critical tasks can be executed in parallel. Finally, the implementation is evaluated using a response time analysis based on the Worst-Case Execution Time (WCET).

Grandpierre and Sorel [35] presented a prototyping methodology and a software tool (SynDEx) that aims to optimize implementations of real-time image and signal processing onto heterogeneous multiprocessors architectures. This methodology has been extended to integrated circuits [36, 37], VHDL code generation [38], and it has been the subject to improvements in terms of power consumption [39].

Jalier et al. [40] introduced a DSE framework for Multiprocessors System on Chip (MPSoCs). Their framework targets telecom applications. The application and hardware model are based on a SystemC description and specified using an XML standard. The modelization and the mapping are manual. However, the performance estimation is done automatically. Metrics such as application throughput, maximum latency, the utilization rate of resources, transient effects are used.

Castrillon et al. [41] presented the MPSoC Application Programming Studio (MAPS), a framework focusing on MPSoCs. The application model is described in C for Process Networks (CPN) and modeled as Kahn Process Networks (CPN). The architecture model is described with Extensible Markup Language (XML). The authors propose several heuristics for mapping and scheduling. The performance evaluation is a composability analysis, where the goal is to determine if a set of applications may be run in parallel.

Jovanovic et al. [42] presented a memory-aware mapping optimization tool for MPSoCs (MAMOT). The application model is a thread-based task graph. A memory-aware evolutionary algorithm is used as a mapping algorithm and is evaluated using performance and energy consumption as metrics.

Dauphin et al. [43] presented Ordonanceur DYNamique (Odyn), an approach for scheduling and memory management that targets Non-Uniform Memory Architecture (NUMA) platforms. The inputs are a Synchronous Data-Flow and a generic logical architecture. The scheduling aims to prevent deadlocks, using a static analysis of Memory Exclusion Graphs.

Suriano et al. [44] presented a framework that integrates the framework Parallel and Real-time Embedded Executives Scheduling Method (PREESM) [45] and the Xilinx SDSoC tool for Zynq devices. It aims to automatize the design and implementation of heterogeneous multicore multi-hardware accelerators. The framework uses PREESM to generate code, and then this code is manually adapted to be used within the Xilinx SDSoC software.

Bruneel et al. [46] introduced a tool for mapping applications to an FPGA. This tool allows a post-mapping reconfiguration by leaving a set of parameters that can be reconfigured on runtime. The applications are expressed as Boolean functions, and the main module of this tool is TMAP, a reconfigurability-aware technology mapper that produces a tunable LUT circuit.

Fricke et al. [47] presented an automatic tool-flow for an application-specific CGRA, called Virtual Coarse-Grained Reconfigurable Array (VCGRA). The architecture consists of layers of processing elements with virtual channels (switches) between them. The framework takes a data flow graph as an application model and produces a binary file used as a configuration context.

Peyret et al. [48] introduced a generic method for mapping application onto CGRA architectures. A Control Data Flow Graph is the application model, and the mapping is a combination between a heuristic and an exact method.

Chin et al. [49] presented CGRA Modelling and Exploration (CGRA-ME), a DSE framework for CGRAs. It includes modeling, mapping, and physical implementation. CGRA-ME modeling does not provide the descriptors for heterogeneous resources or latency because it only considers ALU-like processing resources. The framework offers two options for mapping: a simulated annealing approach and an integer linear programming. The mapping success of these approaches is directly proportional to the flexibility of the hardware fabric. The more flexible the platform, the higher is the mapping success rate. The resource's layout uses a mesh-type pattern, where the number of columns and rows requires to be specified and should be higher than one. Irregular structures are not allowed. Canesche et al. [50] introduced TRAVERSAL, a

graph-based greedy heuristic that provides a flexible, fast, and adaptive placement and routing for CGRAs. It's based on a novel traversal method called ZigZag. It exceeds the performance of CGRA-ME in terms of execution run-time, due to its execution on GPU. Li et al. [51] presented ChordMap, it focus on automated mapping of streaming applications, represented as Synchronous Dataflow graphs, onto a CGRA. Using modulo scheduling, it aims to exploit the parallelism of an application by partitioning it and using the notion of divide and conquer to map all the resulting partitions. Targeting High Processing Computing (HPC), Tan et al. [52] introduced ARENA, a DSE that generates specialized CGRA accelerators. The authors proposed to build CGRA clusters connected by a fast ring network, to increase the performance of the generated system. Shent et al. [53] opted for a different approach. The authors focus on the optimization of the FPGA routing. They proposed to divide the nets into three subsets: a subset of possible dependant nets and two subsets of possible independents. They showed that their approach speedup current approaches by 1.8x.

Even with the large number of works and frameworks dedicated to the reuse problem, to the extent of our knowledge, none can be directly applied to the SPS-CGRA. The inner mechanisms and the latency features can not be captured by the models used for many/ multicores or MPSoCs frameworks. Furthermore, the mapping and scheduling algorithms are not aware of the complexity of both the application and the hardware. Thus, the resulting mapping or implementation will not be near-optimal. Similar works, such as CGRA-ME, use code generation techniques to build hardware blocks according to the application. They may not be capable of reusing an already deployed system's parameters.

1.3 Coarse-Grained Reconfigurable Architectures

Initially, the introduction of the CGRA was motivated to solve development bottlenecks of the FPGA-based systems. Despite its high performance, the FPGA granularity (bitwise logic) imposed a significant challenge for the designers. Additionally, the synthesis-time could take hours or days to finish.

The coarser granularity allows designing more specialized processing resources. This, in turn, increases the performance. Additionally, the new granularity decreased the synthesis time, and the possibility of multiple contexts was introduced. The new type of architecture was named Coarse-Grained Reconfigurable Architecture.

There are many variations of CGRAs. Depending upon the type of interconnections or processing resources, the structure and application of a CGRA may change. Despite these variations, the usual basic processing element of a CGRA consists of interconnections, a register file, and an ALU block [22, 1]. Figure 1.3 illustrates a generic processing resource. The variations of the processing resources may cover the addition of several register files, different types of processing modules. Also, the internal structure of a processing element can be completely different from the one shown in Figure 1.3, with only one processing module or without a register file.

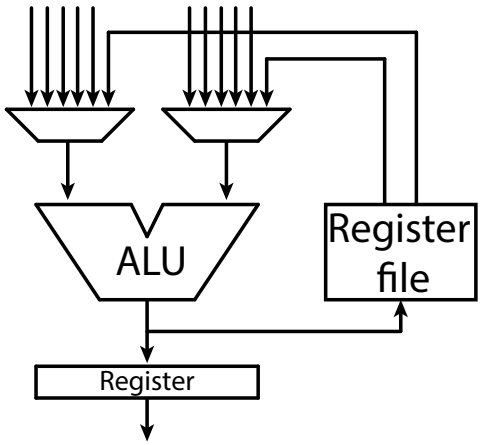


Figure 1.3 – Generic processing resource of a CGRA [1].

In terms of interconnection structure, the typical configuration is a mesh of processing resources. In Figure 1.4 a generic structure is showed, where we can see the mesh of processing resources with the addition of the data and instruction memory. Other types of interconnections may be torus, linear array, or ring.

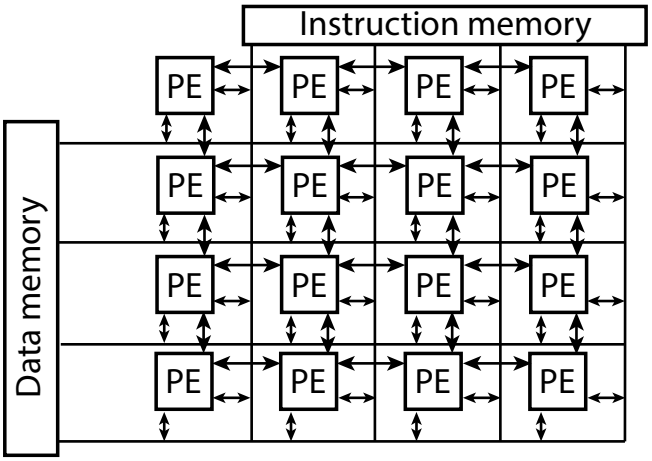


Figure 1.4 – Generic structure of a CGRA [1].

The main role of a CGRA is to perform the computational-demanding inner loops of an application. A CGRA may process the input data in stream-based processing at low latency. Due to this, a CGRA acts as a hardware accelerator inside a system controlled by the main processor. The coupling of the main processor and the CGRA may be tight or loose. In a loose coupling, the CGRA is connected directly to the main memory and may execute code concurrently with the main processor. On the other hand, a tight coupling means that the CGRA shares resources with the main processor and can not execute code in parallel [22].

The last but not least advantage of a CGRA is the context switching time. Specifically the possibility to switch contexts in a very short time. The implementation of an application onto a CGRA may result in several configuration contexts. Each configuration context represents a part of the application and contains the resources' configuration and one or more addresses to the next configuration context. The time needed to request and gather a configuration context and deploy it to the resources to start the new processing process is the context switching time [54]. This feature opens the door to a temporal execution, as the configuration time is short enough to allow the use of multiple contexts.

Examples

A large number of CGRAs have been developed during the last 30 years. Most of them show similarities regarding the type of processing resources. According to the type of topology, we can distinguish two categories: mesh-based and linear-based CGRAs.

Within the category of mesh-based topology, Marshall et al.[2] introduced CHESS. CHESS is a 4-bit ALU-based system, one of the first CGRA that allowed the use of its processing elements as a scratchpad. Figure 1.5 shows a CHESS processing element and Figure 1.6 the interconnection. An evolution of this CGRA is the Elixent D-Fabrix Reconfigurable Algorithm Processor (RAP) [55], where the functionality of the CHESS switch-boxes are improved.

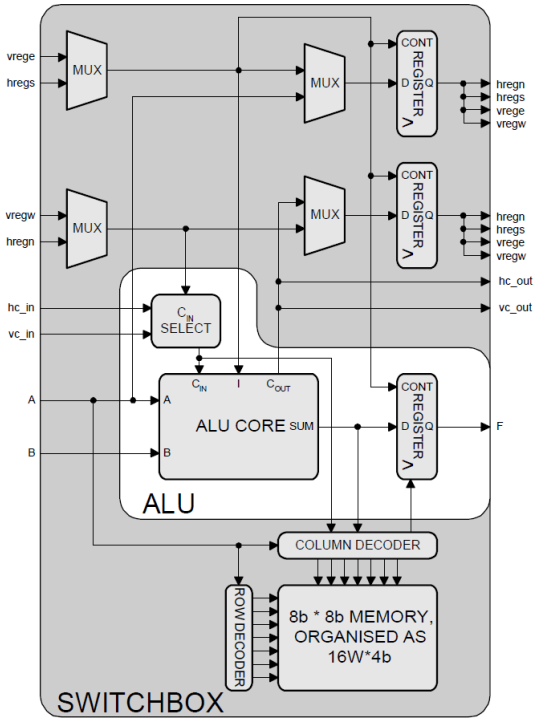


Figure 1.5 – Logical structure of a CHESS processing element [2].

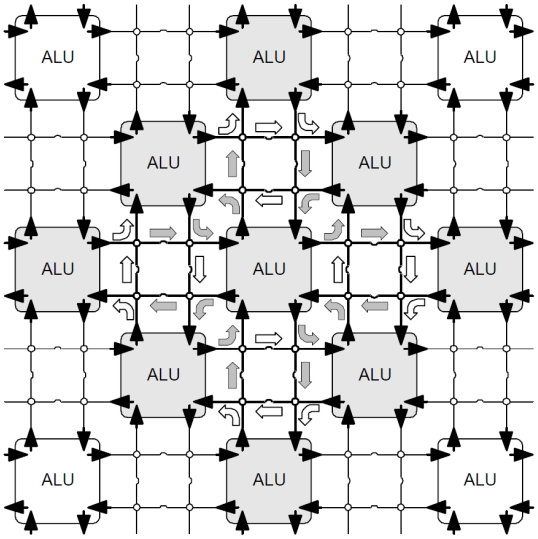


Figure 1.6 – CHESS layout [2].

MorphoSys [3, 56] is a reconfigurable platform that includes a CGRA and a RISC processor loosely coupled. As illustrated in Figure 1.7 the basic processing element tasks also include a multiplier. Additionally, it allows multiple configuration contexts and dynamic reconfiguration. The interconnection network consists of quadrants that allow three hierarchical levels of interconnectivity (Figure 1.8). The first level is the nearest neighbor, which is the interconnectivity between a hardware resource and its direct neighbors. The second level is the intraquadrant connectivity, which enables the interconnection between hardware resources that belongs to a quadrant. The last level is the interquadrant connectivity that allows the interconnectivity between quadrants. These levels of interconnectivity increase the parallelism of the system and its performance. This platform has been improved to allow floating-point operation and implemented in silicon [57].

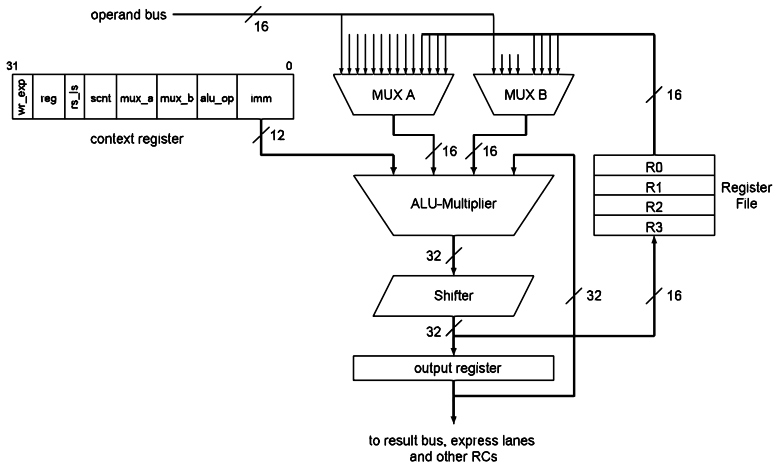


Figure 1.7 – Logical structure of a MorphoSys reconfigurable cell [3].

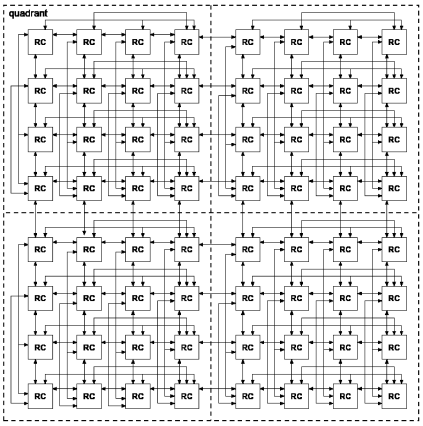


Figure 1.8 – MorphoSys reconfigurable array [3].

The Architecture for Dynamically Reconfigurable Embedded Systems (ADRES) [4, 5] features a mesh of heterogeneous processing elements (Figure 1.10) and is widely used for research. These heterogeneous processing elements feature a functional unit that supports multiple operations and a distributed register file (Figure 1.9). The main advantage of this platform is its flexibility and easy-to-use framework.

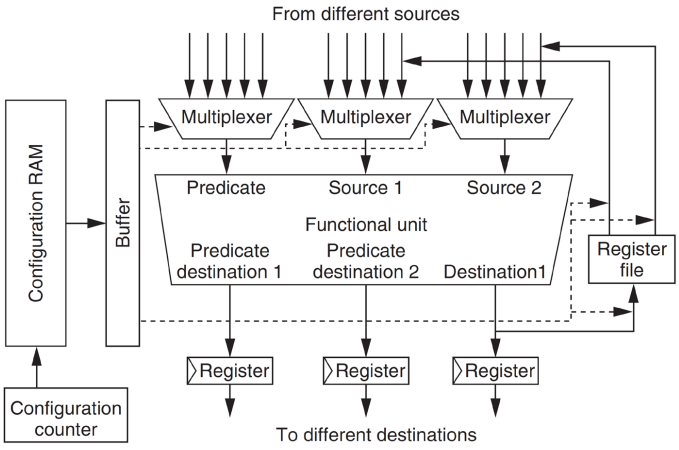


Figure 1.9 – Logical structure of a ADRES processing element [4, 5].

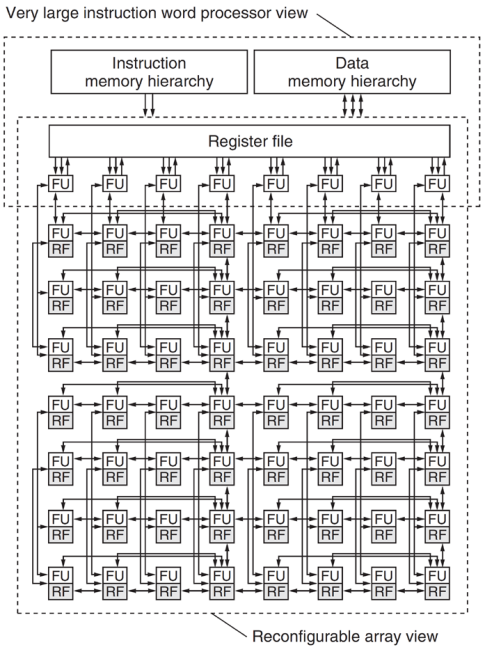


Figure 1.10 – ADRES interconnection [4, 5].

PipeRench [6, 7] is a CGRA with a linear interconnection. The basic structure of its processing element is shown in Figure 1.11, as we can notice it is an ALU-based element with an interconnection layer. The interconnection layer is the main feature of this CGRA, as its structure is based on independent layers interconnected by the interconnection layer (Figure 1.12). This structure allows for implementing independent applications in each layer. This increases the parallelism of this platform.

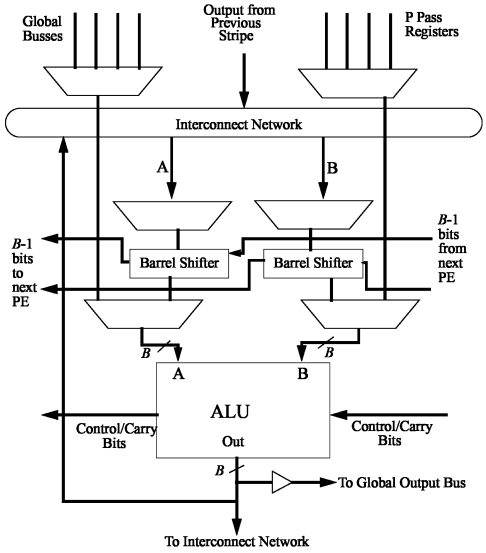


Figure 1.11 – Logical structure of a PipeRench processing element [6, 7].

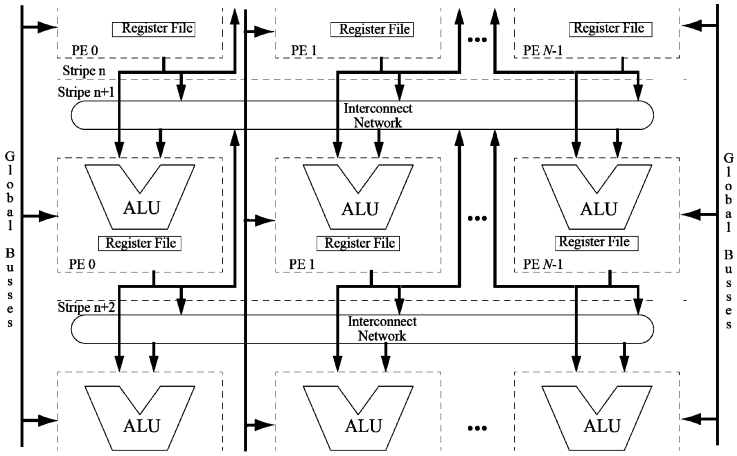


Figure 1.12 – PipeRench interconnection [6, 7].

Most of the current work on CGRA development applies the same principles as the previous CGRA examples. They use the same type of connectivity, and the processing elements perform ALU-like operations. Nonetheless, there have been some improvements in the use of the register file, different internal structures, the addition of scratchpads, processing elements heterogeneity. A significant development is in the part of the granularity of the processing elements. We can observe and corroborate the heterogeneity of resources that will impact the processing latency and final computing cost. As previously stated, the CGRA was a solution for the FPGA development process bottlenecks. However, its performance may still be improved. For this, the designers chose to increase even more the granularity of the processing elements and use a stream-based interconnection. These new systems can perform time-critical tasks and shown an increase in the performance of several applications. In the following section, we will introduce such architectures.

1.4 Software Programmable Streaming Coarse-Grained Reconfigurable Architectures

An SPS-CGRA is a specific subset of CGRAs that consists of a systolic array of heterogeneous processing resources. To better understand the features of this type of architecture, we can use several classification methods. Liu et al. [18] propose a CGRA multidimensional taxonomy based on the models of programming, computation, execution, and micro-architecture. According to the programming model, an SPS-CGRA belongs to the category of the parallel programming model. Usually, an application for an SPS-CGRA consists of a sequence of expressions that describe its behavior with partially implicit parallelism. The aim of the mapping tool is to completely identify and abstract the parallelism to exploit it and provide an efficient implementation. The second classification is based on the computation model, where the SPS-CGRA falls in the category of multiple configurations, multiple data (MCMD). However, an SPS-CGRA can switch between configurations and allows a dynamic parameter-based reconfiguration on runtime. The third classification is based on the execution model. An SPS-CGRA belongs to the category of static-scheduling sequential execution where the mapping tool is in charge of the initiation and mechanism of the reconfiguration.

In the work of Wijtvliet et al. [22] a CGRA classification is proposed using four main elements, structure, control, integration, and tooling. According to this paper, structurally, an SPS-CGRA is a multi granular direct connection network with a cache memory hierarchy. The

granularity may be coarser than ALU-based operations. In terms of the control, SPSP-CGRA scheduling is done at compile-time, where the mapping tool creates the spatial and temporal scheduling. Still, both the resources and network's reconfiguration can be done at run-time using a pre-defined configuration. It also allows custom operations since the granularity of the resources is higher than the usual CGRA operations. Regarding the integration, SPS-CGRAs are usually used as co-processors (hardware accelerators) with loose coupling and no resource sharing. The last element of the classification, tooling, claims that the compiler is usually based on a mix of dataflow/custom language, as each SPS-CGRA has different types of operations. The framework of an SPS-CGRA does not consider the place and route since the architecture is usually already deployed, and only a design space exploration is done to find the best mapping possible.

Hartenstein [58] proposes different classification. The author uses the type of architecture, granularity, fabrics, mapping, and the intended target application to distinguish between CGRAs. According to this classification, an SPS-CGRA would be a multi granular 1-D array. Both the fabrics and the target application may vary. The mapping in these types of architectures is described as scheduling. Tehre and Kshirsagar [59] propose a similar classification to [58]. However, the authors added the category of reconfiguration model, which for an SPS-CGRA is dynamic.

Chattopadhyay [60] proposes a taxonomy based on three axes, the design choices, the class, and the modeling and design space exploration tools. The design choices refer to three aspects, the microarchitectural view, the programming model, and the platform type. In Figure 1.13, we summarize the different microarchitectural view categories applied to our SPS-CGRA target. An SPS-CGRA is usually used as a special-purpose co-processor. Its pipelining interconnection architecture allows a data-driven flow. It typically consists of a linear array of coarse-grained resources that may be configurable statically/dynamically.

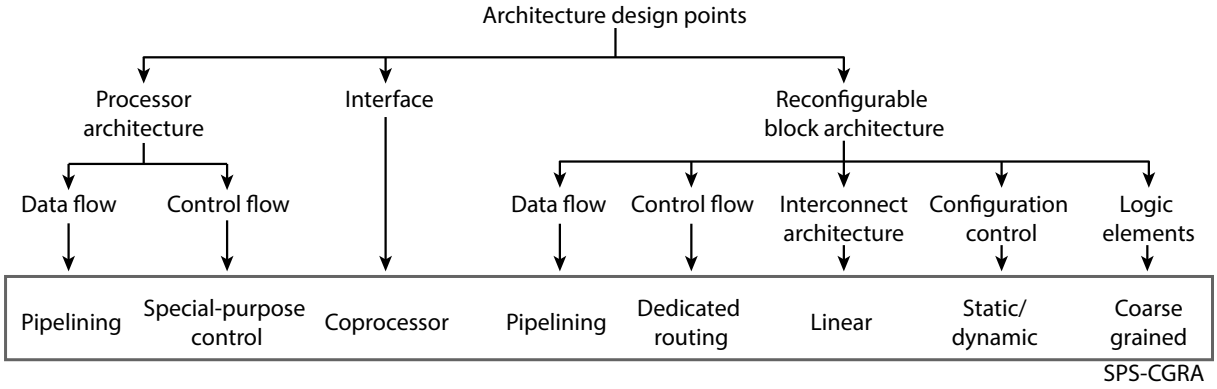


Figure 1.13 – Proposed microarchitectural classification of an SPS-CGRA.

According to Chattopadhyay [60], an SPS-CGRA belongs to Class III: custom processing and reconfigurability. The broad range of possible structures and architectures make this family of accelerators a good candidate for various real-time applications, although it increases the complexity for a general framework that could cover all types. The last axis of Chattopadhyay is the modeling and design space exploration tools. Usually, the tools that target SPS-CGRA allow only the modification/optimization of the (software parameters) implementation, and they are highly specialized for a given platform.

As stated previously, there are several types of SPS-CGRA. They share similarities in architecture and properties, but the tools and their frameworks are custom. This aspect overburdens its use and also the possibility to apply tools from one SPS-CGRA to another. A seamless design with accurate latency estimation and exploration capability is required to increase the use of this type of architecture and exploit all its benefits.

Examples

As quickly presented in the introduction, an extensive number of SPS-CGRAs have been developed for research and commercial purposes. They belong to different fields, and they are application-specific. Due to the higher level of granularity of these architectures, reusability within the application field may be narrow, but it is compensated by the high performance and low latency that can provide.

Hashing is an algorithm that calculates a fixed-size bit string value from a file. Within the cryptocurrency domain, hashing algorithms ensure the integrity of the transactions. The implementation of hashing algorithms requires a huge amount of computational power. In this regard, Tetu et al. [8] propose an SPS-CGRA-based implementation of the Lyra2REv2 hashing algorithm (Figure 1.14). It features a pipelined stream-based architecture, and it is loosely coupled to an ARM processor.

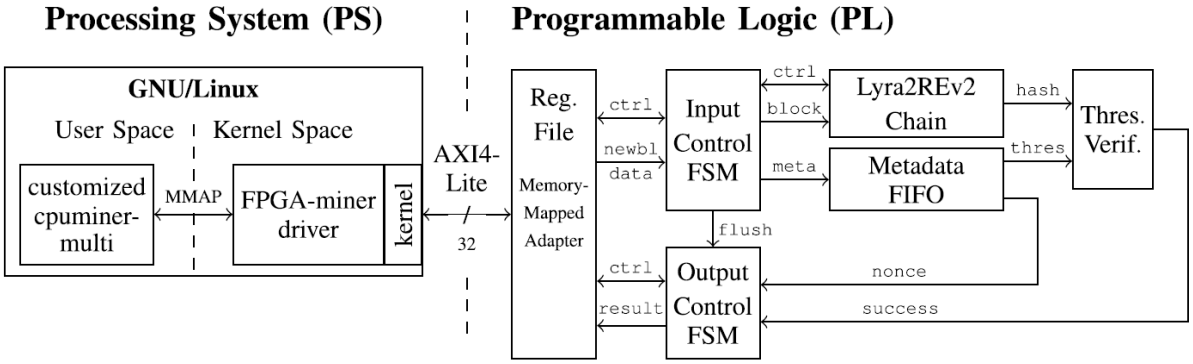


Figure 1.14 – Architecture of the SPS-CGRA implementation of the standalone Lyra2REv2 miner [8].

Sound field rendering uses numerical methods to model the behavior of sound wave propagation in the spatial and time domain. The finite difference time domain method has been widely adopted and becomes an essential algorithm in room acoustics owing to its high accuracy, easy implementation, and parallelization. Tan et al. [27, 28] present an SPS-CGRA implementation of this method. Its highly pipelined architecture and stream-based processing allow the computation at continuous time steps and reduced access to external memory.

In the computer vision domain, SPS-CGRAs are used to enhance the performance of image processing systems. Examples of these are the FPGA-based SPS-CGRAs presented by Ngan et al. [29, 9] (Figure 1.15) and by Isavudeen et al. [10] (Figure 1.16). These architectures allow handling multiple parallel pixel streams given their adaptable, highly pipelined datapaths. Another example is the ASIC-based SPS-CGRA introduced by Dokladalova et al. [30]. This platform features reconfigurable datapaths, and its processing resources are able to implement different contexts.

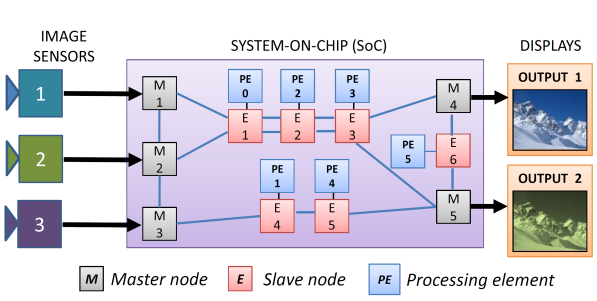


Figure 1.15 – Architecture of the SPS-CGRA proposed by Ngan et al. [9]

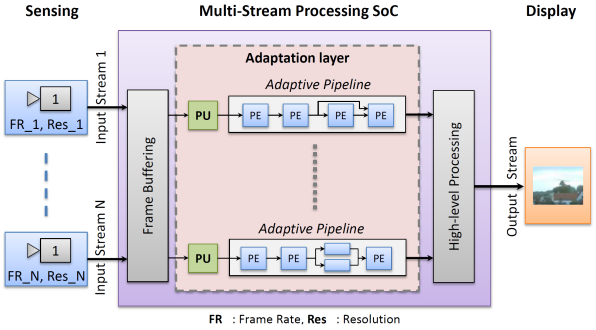


Figure 1.16 – Architecture of the SPS-CGRA proposed by Ali et al. [10]

Additionally, within the computer vision domain, another example of SPS-CGRA is the Programmable Pipeline Image Processor (P2IP) introduced by Possa et al. [11]. The P2IP is a scalable systolic-based architecture that provides low latency processing and allows run-time reconfigurable datapaths. Figure 1.17 illustrates the functional diagram of this architecture. Notice the main blocks (named PE1, PE2, ..., PEn) of the system, which correspond to the typical structure and data flow of an SPS-CGRA. The architecture consists of the main controller (P2IP controller) responsible for delivering the configuration context to all the hardware resources. The PEn blocks represent the processing resources of the hardware, and the connection of them is through the Reconfigurable Interconnection (RI) blocks. Observe that the interconnection of the blocks is in a pipeline form. This feature emphasizes the stream-based processing of an SPS-CGRA. Another example is the Morphological Co-Processing Unit (MCPU) introduced by Bartovsky et al. [12]. The MCPU assembles several efficient dilation/erosion units with geodesic units and ALUs to support a large collection of morphological operations. It is integrated as a coprocessor in an FPGA-based platform. Figure 1.18 layout the architecture of the MCPU. As we can notice, the MCPU follows the principles of an SPS-CGRA, with its pipeline-based array of processing resources (Large SE pipeline and Geodesic Pipeline). It has a loose coupling since there is no resource shared with its host processor. A set of multiplexers, configuration registers, image buffers, and a memory controller ensures its correct operation.

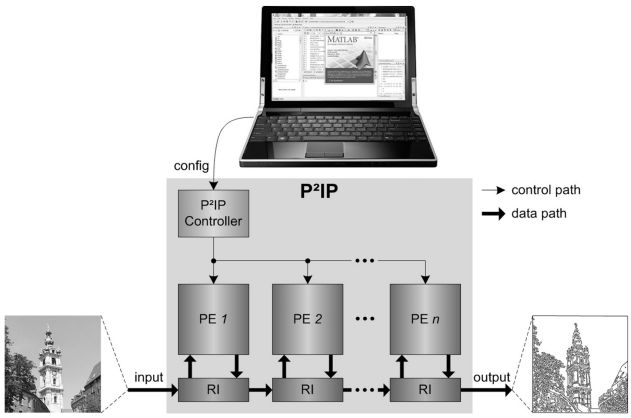


Figure 1.17 – Simplified functional diagram of the P2IP [11].

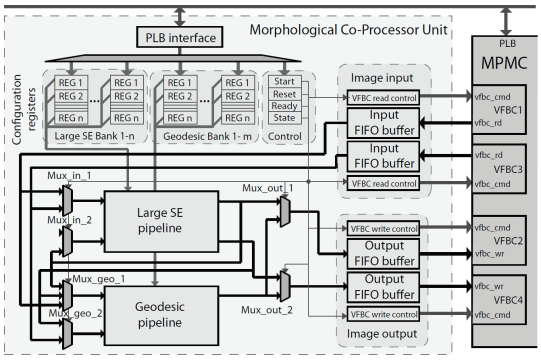


Figure 1.18 – Architecture of the MCPU [12].

Another field that benefits from the SPS-CGRA high performance is Deep Learning. The basic computation model of Deep Learning is a NN, and this can be modeled as a Directed Acyclic Graph (DAG). Given that the dataflow of such computation model is unidirectional, the stream-based low latency processing of an SPS-CGRA may exceed the performance of other platforms. Examples of them are the ASIC-based Neuflow [31] (Figure 1.19) and FlexFlow [32] (Figure 1.20), both similar in processing resources and interconnection.

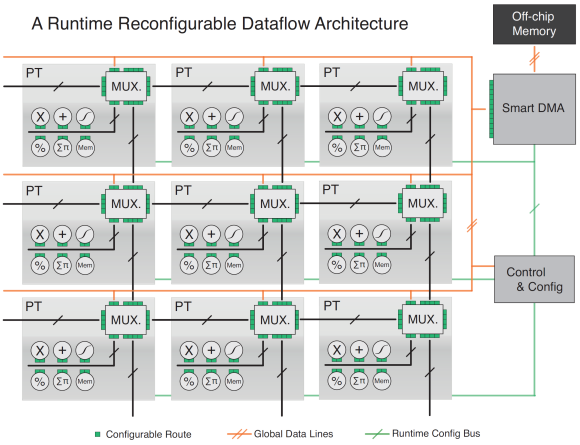


Figure 1.19 – Architecture of the NeuFlow.

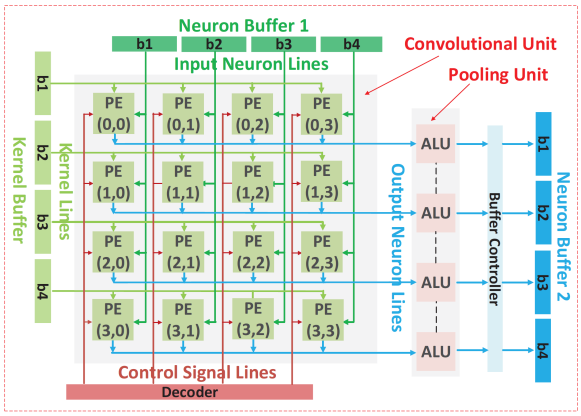


Figure 1.20 – Architecture of the FlexFlow.

Other examples are the FPGA-based implementations of a Deconvolutional NN presented by Chang et al. [13] (Figure 1.21), a Life Long learning Convolutional NN by Piyasena et al.[14] (Figure 1.22) and a Randomly Wired NN develop by Kuramochi and Nakahara [33].

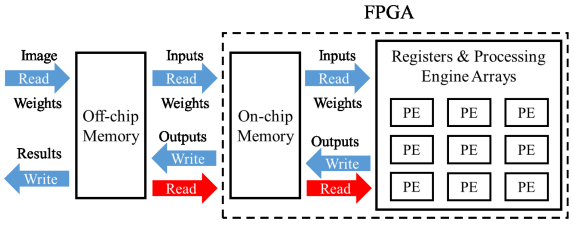


Figure 1.21 – Architecture of the Deconvolutional NN [13].

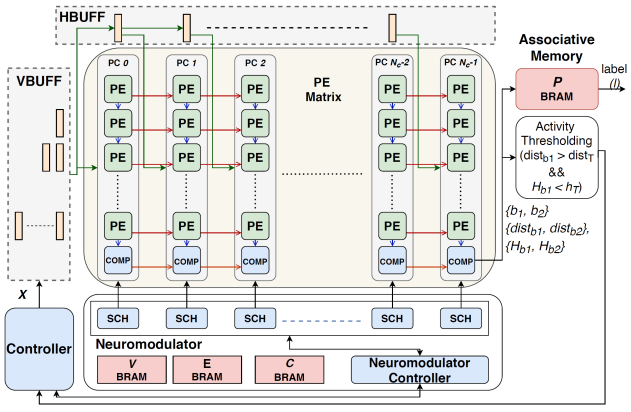


Figure 1.22 – Architecture of the a Life Long learning Convolutional NN [14].

1.5 Thesis Contributions

We can see from the examples presented in Section 1.4 that there are many application fields benefiting from SPS-CGRA-based systems. The examples presented above exceed the performance of other types of platforms considerably by at least one order of magnitude. Nevertheless, their reuse opportunity is severely limited. Not only the cited examples but most of the SPS-CGRAs are not equipped with tools or software support that allows one to reuse them. Usually, they depend on custom mapping tools, or worst, manual mapping.

Even if there is a mapping assistance tool, such a tool is not flexible enough or easy to adapt. Generally, it requires a broad knowledge of the platform. Moreover, it can not be used for another SPS-CGRA, mainly due to such tools are designed specifically for the target hardware. This situation is understandable if we consider the large diversity of structures and applications of the SPS-CGRA. However, in general, they share similar characteristics, and those can be exploited as a foundation of a generic framework that can capture those similarities and provide enough flexibility for all the SPS-CGRAs. This framework could open the doors for the easy implementation of more applications onto already deployed SPS-CGRAs. This will decrease or eliminate the re-design time and reduce the time-to-market.

In this thesis, we present a **complete framework** for easy reuse of **SPS-CGRAs**. It is an

end-to-end framework that begins with the **specification files**, which describes at a high level the application and the main features of the hardware. We introduce a **unified graph-based model** covering **application**, **hardware** and **implementation**. We present four **mapping and scheduling algorithms**. We propose a **performance evaluation** based on the latency-based estimation of the upper bound of the computing cost. Finally, we introduce a software solution, showed in Figure 1.23, that accepts the two specification files (application and SPS-CGRA) and produces, in parallel, the configuration context for a particular application and a latency report that shows the computing cost.

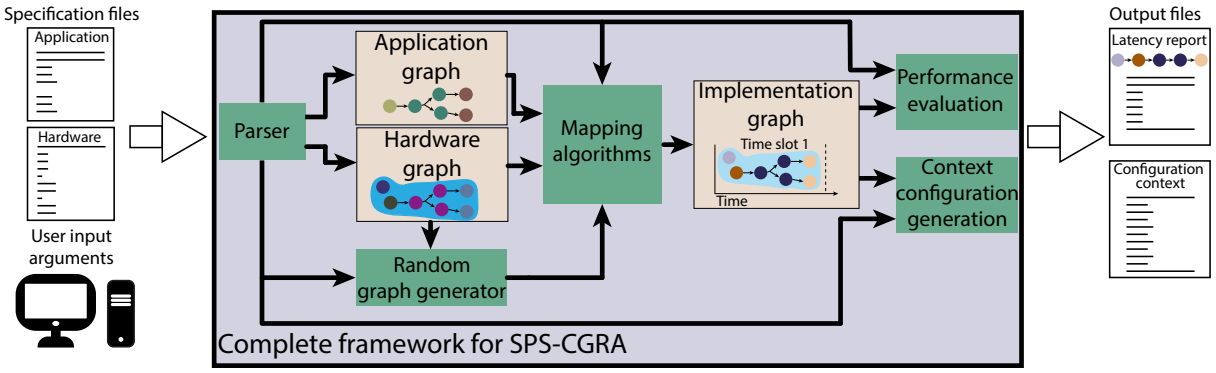


Figure 1.23 – Overview of the software solution for the easy reuse of SPS-CGRA.

1.6 Thesis Outline

A general overview of the thesis is showed in Figure 1.24.

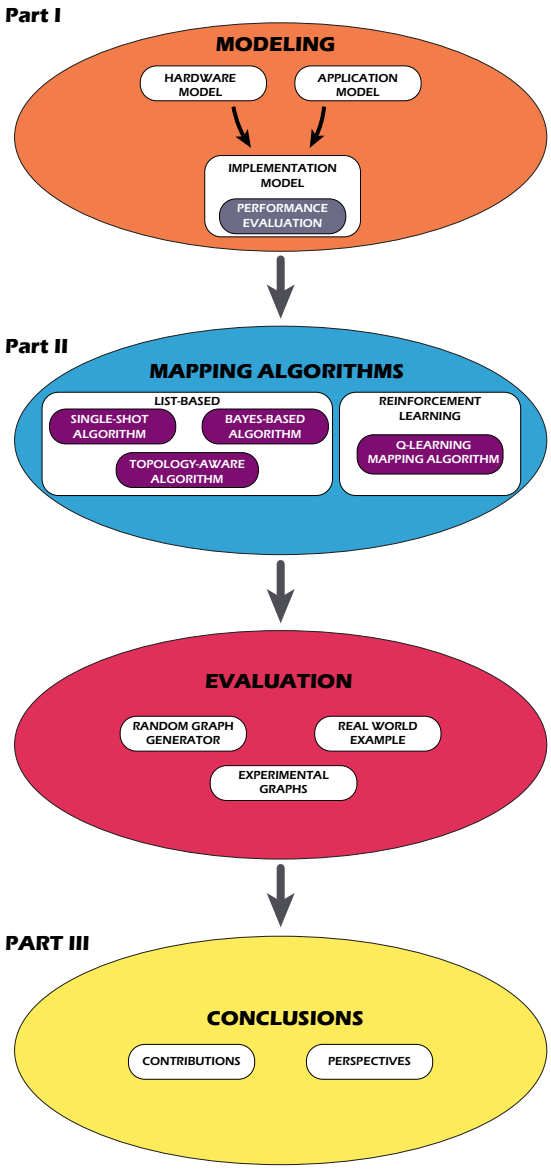


Figure 1.24 – General overview of the thesis

We divide the thesis into two parts: Modeling and Mapping algorithms. The outline of the thesis is:

I Modeling

Chapter 2 introduces our application model. We start with the current state of the art of application models. Considering that models that apply to our problem are not that many, we include models for other platforms such as multicore processors and MPSoCs. We describe their main features and benefits. We argue that a graph-based model, and specifically an extension of the DAG model, could fulfill our requirements. Furthermore, we propose a specialization of the Parameterized and Interfaced Synchronous Dataflow (PiSDF) and the AAM methodology. Next, we present the formalization of our model. We introduce the information that the nodes include and the different tasks that we consider in our model. Finally, we present two real-life applications that help us illustrate the modeling process and the benefits and capabilities of our application model.

Chapter 3 introduces our hardware model. A state of the art of current hardware models is presented. We divided it by type of platform, and we describe the features that the models

highlight. We claim that current approaches neglect essential elements such as configuration control and latency properties. We then present the main hardware characteristics of the SPS-CGRA. Considering these characteristics, we introduce the formalization of our hardware model. We describe all the types of hardware resources that the model is capable of abstract. Additionally, we present our approach for modeling latency properties of the processing resources. It is based on the input and computing latency of the processing resource, and it is a function or the programmable parameter and the physical realization of the resource. Next, we use several experimental examples to illustrate the modeling process, including some transformations required to fit our hardware model. Finally, we use a real-life SPS-CGRA to demonstrate the benefits and capabilities of our hardware model.

Chapter 4 introduces our implementation model. We present the formal definition of the model. We describe how the implementation model is composed. We introduce a special subset of resources called data dependency resources. These resources allow modeling data dependence between time slots and between datapaths. Next, we present several experimental examples that illustrate the capabilities of the implementation model and its main features. We introduce our **performance evaluation** methodology. It is based on the estimation of the upper bound of the computing cost. From the implementation graph, we produce a subgraph that we call evaluation graph. This graph is used to estimate the upper bound of the computing cost through an equation that we also propose in this work. Within this equation, we introduce the variable ω, which abstracts the latency imbalance among processing resources. We show with some experimental examples that this equation can accurately estimate the upper bound of the computing cost.

II Mapping algorithms

Chapter 5 presents the current state of the art of mapping and scheduling algorithms. We focus on list-based scheduling algorithms, linear programming, and reinforcement learning. We argue that list-based and reinforcement learning algorithms show a high probability of adapting and solving the SPS-CGRA mapping problem. Usually, list-based algorithms target limited processing elements, and their principle is based on a list of priorities, in which resources and tasks are ordered according to a fixed metric. We can use these characteristics to adapt a list-based algorithm to solve the SPS-CGRA mapping problem. Additionally, list-based algorithms can include heuristics to improve their performance. Reinforcement learning is a subset of Machine Learning, where an agent interacts within an environment. From these interactions, the agent aims to improve its actions towards the maximization of a reward. We argue that this approach is suitable for our problem. Given the trial-and-error method of this approach, we may extend the exploration design space without reaching a case explosion usually produced by an exhaustive approach.

Chapter 6 introduces three list-based mapping and scheduling algorithms. We first propose a Single-Shot List-based mapping algorithm. This algorithm aims to produce a feasible mapping. It is equipped with backtracking. However, it does not include an optimization heuristic. Secondly, we introduce a Topology-Aware mapping algorithm. The principle is based on a look-ahead-based heuristic. The locality of the mapping decision of the single-shot mapping algorithm is increased by considering the mapping of the successors of the given task. Additionally, it includes the notion of computing latency within the heuristics. The third and last algorithm is the Bayes-based mapping algorithm. We use the Bayes theorem to solve the SPS-CGRA mapping problem. We include two hyperparameters that the user can tune to obtain better final implementations.

Chapter 7 presents our Q-learning mapping algorithm. We adapt the reinforcement learn-

ing framework to our SPS-CGRA mapping problem. We propose to define the agent as a mapping function. It will interact with the hardware graph, which is defined as the environment. The agent will receive information about a defined subgraph that contains the given task and its successors and predecessors from the environment. Also, information about the upward and downward rank will be given. With this information, the decision no longer remains local. We introduce a hierarchical reward policy. The agent will receive a reward for each allocation of a task onto a processing resource. Additionally, the agent will receive a reward for the allocation of the entire application onto the hardware. We propose two types of training, one offline using randomly generated graphs and another online using the target application. This approach will allow the framework to be used for other applications rather than only for the given one.

Chapter 8 presents an evaluation of our mapping and scheduling algorithms. The evaluation consists of several sets of experimental graphs, one pair of randomly generated graphs, and a real-life example. The sets of experimental graphs will help us to characterize the algorithms. The randomly generated graphs represent graphs with more complex structures than the normal SPS-CGRA and its applications. The final evaluation uses a real-life example. It is the proof of concept of our complete framework and its mapping algorithms.

Chapter 9 summarizes our contributions and outlines the perspectives and future lines of research generated by this work.

Part I

Modeling

Chapter 2

Application Model

2.1 Introduction

An application model is an abstraction of an application, its role is to capture its main features. It should be able to represent the tasks of the application and, it should provide an accurate description of them and of their important parameters. As one of the inputs of a mapping algorithm, an application model must share similar features with the hardware model. This is crucial, as the level of abstraction of the application model should comply with the hardware model. The description of the application's tasks should allow a matching with the description of the resources of the hardware. Since the mapping algorithm aims to find the best allocation of a task onto a resource, this goal will not be achieved without its correct representation. As a result of this, an adequate abstraction of the application is a priority. This importance has attracted the attention of researchers, and several works aim to create new and more accurate application models.

Important advances have been made in the development of application models. In parallel and distributed computing, a well-known application model framework is the Directed Acyclic Graph (DAG) model [61]. The DAG model allows to capture the parallelism of the applications and exploit the advantages of a distributed system. Several extensions to this model have been proposed, introducing new features like information about different configurations (scenarios) [62], parameters [63], interfaces [64], or parameter dependency trees [65]. We propose an specialization of the Parameterized and Interfaced Synchronous Dataflow (PiSDF) [65] for SPS-CGRA. It allows exploiting the parallelism of the SPS-CGRA's applications and its heterogeneity, and thanks to it, increase the possibility of achieving the optimal mapping. Furthermore, we exploit the parameter descriptors in order to define the behaviour of the SPS-CGRA and estimate its performance in terms of computing cost.

This chapter introduces the proposed application model. We describe the current state of the art, discuss the different models and define the requirements of our problem. Finally, we present our application model and its features.

We organize the remaining part of the chapter as follows. Section 2.2 reviews state of the art. Section 2.3 presents the proposed application model. In Section 2.4, we introduce the formalization of the model. In Section 2.5, we present two application examples for a better understanding of our model. Finally, Section 2.6 summarizes this chapter.

2.2 State of the Art

A particularity of the application models is that their structure and characteristics should match the ones of the target architecture. This aspect allows decreasing the complexity and the number of steps of the mapping algorithm. In this section, we review and describe the main features of several application models that target different platforms.

2.2.1 Dependence Graph

One of the first application models is the Dependence Graph for Very-large-scale integration (VLSI) array processors. Lo et al. [66] presented the Dependence Graph, which is a directed graph where nodes represent computations of variables associated with the node and edges are data dependence between nodes.

2.2.2 Task Model

A interesting application model is the task model. Liu and Layland [67] introduced the task model, which characterizes task behavior with the execution time and the period of task activations. Following this first task model, other works integrate different descriptors to enhance this model. Namely, in the field of multi-core processors, several works use as the main task descriptor the Worst-Case Execution Time (WCET)[68, 69] of the task.

For the same target, multi-core systems, Marinho et al. [70] introduced a task model for sporadic tasks. They consider a task set of n sporadic tasks. A relative deadline and its minimum time span or period between two consecutive activations of the same task are used as descriptors. Lakshmanan et al. [71] used the task model and represented each task with the number of computations segments, the number of parallel threads, the WCET of the sequential segments, the WCET of each thread, and the period of the task. Kim et al. [72] described a task model with separate setup times. They divide the setup time into processing time and transmission time. Also, their task model allows them to deal with different physical realizations of hardware resources.

2.2.3 Parallel Synchronous Task Model

The parallel synchronous task model [73] uses a directed graph that consists of parallel jobs integrated by many computation segments. Each segment may contain many synchronous parallel threads. The descriptors of each segment are the number of threads and the worst-case execution requirement. The main target for this model is multi-core system applications.

2.2.4 Digraph Real-Time Task Model

Stigge et al. [74] presented the Digraph Real-Time (DRT) task model for processor-based systems. It consists of n independent tasks, each one characterized by a directed graph. The authors describe each node with its WCET and its relative deadline.

2.2.5 Non Cyclic Recurrent Real-Time Task Model

Baruah et al. [75] presented the non-cyclic Recurrent Real-Time (RRT) task model targeting processor-based systems. It is, in fact, a DAG with a unique source and one or more sinks. The authors describe each node with the execution requirement and the deadline of the task.

2.2.6 Generalized Multiframe Task Model

Peng et al. [76] presented a generalized multi-frame task model (GMF) with parameter adaptation for processor-based systems. Usually, a generalized multi-frame task model (GMF) represents each task with a three tuple. The first element represents the execution times, the second the relative deadlines, and the minimum inter-arrival time between consecutive frames is the last one. The arrival time, the deadline, and the WCET characterize each frame. The authors modified this model where the deadlines and the period can be defined as an interval of values.

2.2.7 Recurring Real-Time Task Model

The recurring real-time model (RRT) is a generalization of the GMF model where a relaxation of the types of tasks is included, allowing to model different job types. Chen introduced the concept of varying task implementations for sporadic tasks, which allows modeling of different worsts case execution times, increasing the accuracy of the model [77].

2.2.8 3-Phase Model

The 3-phase model is a generalization of the Predictable Execution model (PREM) [78]. The PREM model consists of two phases, a memory phase and an execution phase. It targets single-core systems but was extended to multi-cores [79]. The 3-phase model [80] divides each task into three successive phases. During the Acquisition phase (A-phase), the task loads its instruction and data into the local memory. Later, in the Execution phase (E-phase), the task executes non-preemptively. Lastly, in the Restitution phase (R-phase), the main memory stores the processed data. The model allows bus accesses only during the A- and R-phase. With this feature, the time required to perform each phase can be accurately computed. In the work of Maia et al. [80], A_i, E_i and R_i represent the maximum execution time of the A-, E- and R-phase of task t_i, respectively. The WCET in isolation of t_i (without suffering any interference) is the sum of each phase's execution times. The authors describe each task with its period and the constrained deadline.

2.2.9 Synchronous Dataflow Graph

The synchronous dataflow graph was originally created for Digital Signal Processors' applications on parallel hardware. It is a graph where the nodes represent actors and the edges FIFOs. It is widely used, and several works exploit and enhance it. In the field of processor-based systems, Sih et al. [81] use an Acyclic Precedence Expansion Graph (APEG), which is a modification of the Synchronous Dataflow Graph, they describe each with a weight representing the amount of data. Berler et al. [82] propose a synchronous dataflow model with timing specifications for cyber-physical systems. The model includes timing specifications for I/O nodes with side effects (in other words, nodes that interact with the physical world). Pelcat et al. [83] use the synchronous dataflow model for applications that target MPSoCs. They use a specific model called Parameterized and Interfaced Synchronous DataFlow (PiSDF). The PiSDF is an extension of the Parameterized Synchronous Dataflow (PSDF) [63] and the Interface-Based SDF (IBSDF) [64]. PiSDF model allows to describe a broad range of applications by parameterization, hierarchy and configuration [84]. This is done through the addition of a set of parameters associated to the configuration of each task. Additionally, a set of dependencies of the parameters defines the hierarchy of the parameters and the influence of a given parameter to its child's and parent's parameters. Inherit from the IBSDF and also taken from the Algorithm Architecture Matching Methodology [35, 85], the PiSDF includes the concept of source and sink special nodes, that interact with the outside world. These special nodes allow defining the inputs and output of the system, increasing the feasibility to be instantiated in any design [64].

2.2.10 Algorithm Architecture Matching

The Algorithm Architecture Matching methodology (AAM) [35, 85] covers all the steps of rapid prototyping in a seamless flow of graph transformations from algorithm and architecture specification to distributed executives automatic generation [86]. A Data Dependence Graph represents an algorithm (application). This graph is a hypergraph where the nodes are operations, and edges represent data dependency. The methodology defines four types of nodes, memory (delay), computation (action), conditioning (action), and sensors/actuators. In order to support FPGA, Kaouane et al. [36] presented an extension of the DAG model of the AAM methodology called the factorized data dependence graph model. It includes the fork, join, diffusion, and iterate

nodes, so the temporal or spatial repetition of a pattern can be modeled. They target real-time image processing applications and general signal processing applications.

2.2.11 Directed Acyclic Graph

The Directed Acyclic Graph (DAG) model allows describing complex applications with a high degree of parallelism and the data dependence between tasks. It has been used for several targets.

In the field of processor-based systems, several works consider the DAG model. Qin et al.[87] uses a DAG model and describes the nodes with their execution communication cost. Wang et al. [88] used a DAG to represent an application and complement it with a matrix of the estimated time to execute a certain task in a particular processor. Melani et al. [89] presented a modified DAG model. They focus on sporadic conditional parallel tasks. The WCET describes each node of the DAG (which represents each task). Also, they make a difference between nodes. They introduce a specification for regular nodes and conditional nodes. Qamhieh et al. [90] introduced an algorithm to convert from parallel DAGs to sequential in order to reduce the complexity of the mapping and scheduling algorithm. Topcuoglu et al. [91] used a DAG model, where each node is described with the average execution cost. Yuan et al.[92] used a DAG model that specifies the execution cost of the task and the cost of the communication. On the other hand, as multi-core systems technology evolves, the possibility to use them for parallel applications increases.

Concurrently, different application models can improve the exploitation of parallelism. Saifullah et al. [73] introduced a decomposition method for DAG models to a sequential model that decreases the complexity of the scheduling. Pathan et al. [93] used a DAG model and describe each task with the minimum inter-arrival time of consecutive jobs, the relative deadline, and a real-time constraint that limits the execution time of the task.

The field of Multi-Processor Systems-on-Chip (MPSoC) uses similar approaches. Zadrija et al. [94], and Frid et al. [95, 96] used a weighted DAG. The authors describe each node with the number of operations (computational intensity), and the weight of the edges represents the amount of data transacted. Sinaei and Fatemi [97] used the Kahn process network model, a network of concurrent processes interconnected via FIFO channels. Youness et al. [98] used the DAG model, where the node's labels are the computation cost and the weight of the edges is the communication cost. Jalier et al. [40] modeled telecom applications (OFDM, demodulation, MIMO decoding) as a SystemC[1] function, describing the behavior of each task: communications with external tasks or buffers, the complexity of the data processing and control flow. They allow not only tasks but buffers to be in the application model.

Regarding CGRAs, the works in this area also use variations of the DAG model. Bingfeng et al. [100] introduced a Data Dependence Graph with two types of edges: data edges and precedence edges. Data edges indicate that data need to be routed between the connected operation terminals. Precedence edges indicate that the operation needs to be ordered, but no data is routed between the operations. Ma et al. [101] used a Control Data Flow Graph (CDFG) with the information of the operation nodes and the data and control dependence among them. Das et al.[102] used a graph-based model called Control Data Flow Graph (CDFG), where each node represents a basic block. Each basic block represents, in turn, a Data Flow Graph, where each node has a set of operations as a descriptor. Other works [103, 104, 105] use Data Flow Graph to represent an application for a CGRA.

Within the field of reconfigurable architectures, Kota et al. [106] presented a DAG model for applications that target reconfigurable system-on-chip (rSoC). Each node of the DAG may represent multiple versions of implementations for that task. Each version has its corresponding hardware requirements in terms of area, functional units, and time to execute the task. The

[1]SystemC is an ANSI standard C++ class library for system and hardware design for use by designers and architects who need to address complex systems that are a hybrid between hardware and software [99]

authors consider functionality implementation for each node. This particularity takes full advantage of the nature of the DAGs to model heterogeneous applications. Lai et al.[107] used the DAG model where the nodes are operators, and the edges represent data flow. Cong and Xu [108] used a boolean network, a DAG, where every node represents a logic gate.

Namazi et al. [109] used the DAG model to represent applications for Networks-on-Chip (NoC). The authors describe each node with the priority assigned to the task and the number of clock cycles of the task worst-case execution time. In the same matter, the authors defined the weight of the edges with the maximum volume of data and the required communication bandwidth between two nodes. Lu et al. [110] used a weighted directed graph. The weight represents the packet amount and the edges, the packet transmission.

Regarding other platforms, such as High-Performance Computing systems, Ma et al. [111] used a DAG model, where each node has a parameter representing the number of instructions and each edge the volume of transferred data. Jiang et al. [112] used a DAG to represent workflows for clouds. The model uses, as labels, the computation cost, the communication cost, the earliest start time, and the estimated finish time.

2.2.12 Directed Graph

Gamatié et al. [113] used a directed graph of runnables. Runnables are functions that represent the smallest unit of code schedulable by an operating system and are associated with non-functional attributes. It is dedicated to multi-core systems. The nodes represent the runnable and the *release*, which is an attribute specifying whether the release mode of a task t_i is periodic, sporadic, or aperiodic. Also, the authors provide the means to describe the data communication, but it is not compulsory to define.

2.2.13 Boolean Dataflow Graph

In the field of code generation, Li et al. [114] presented a model of computation based on a boolean dataflow graph, particularly for FPGAs. Their model targets irregular applications, which are usually built around loop constructs. They classify the loops constructs into for-all (parallel execution) and for-each (sequential execution).

2.2.14 Others

Rajeev et al. [115] presented a transformation from the formal model of a distributed system. They represent a distributed system with a tuple. The first element is the set of schedulable objects, which can be tasks or messages. The second element is the set of resources, which can be elements or hardware. The third element describes the allocation of a task, either an Electronic Control Unit (ECU) or a bus. The last element is the set of data dependency between tasks and messages (edges). The descriptors of each task or message are the initial offset, the period and the best and worst-case execution time, and lastly, the deadline.

2.2.15 Discussion

The extensive literature provides a large set of models with different characteristics. Each one is optimized to match its hardware model pair. In terms of structure, the most used is the DAG model. The DAG model provides the means to abstract parallelism of applications and the data dependence of the tasks. It is ideal for parallel processing architectures [89, 90, 100, 111]. Furthermore, an important extension of the DAG model, introduced by the AAM methodology [35, 85], which uses hypergraphs, allows the broadcast of the output data. This feature is important for our problem. Normally, an SPS-CGRA application can have tasks connected to one or more tasks, and the model needs to represent the explicit broadcast of output data. Moreover, the DAG model allows abstract complex applications that target similar platforms like CGRAs [101, 102, 103, 104, 105].

Regarding the descriptions and the information associated with the tasks, most works emphasize the WCET [68, 69, 72, 71] in the multi-core field. Other approaches include a division of the setup time and the transmission time [72], or a division between phases (acquisition, execution, and restitution) [78, 80]. Given the characteristics of our target hardware, the application model does not need information about execution time. Das et al. [102] take another approach by providing the means to specify a set of operations for a given task. In the works of Kota et al. [106], the descriptors also allow to describe heterogeneous implementations of a task. However, these descriptors do not allow the description of explicit parameters. Pelcat et al. [65] efficiently uses the concept of parameterization of the tasks introduced by Bhattacharya and Bhattacharya [63] to include functions of the input parameters. Aditionally, the authors include, from the works of Piat et al. [64] and the AAM methodology, the notion of source (sensors) and sink (actuators) special nodes that allows to model the external input and output data. To efficiently map applications onto an SPS-CGRA, the previously stated examples are not enough. Our target architecture requires both information on the type of task and its parameters. Usually, the user knows beforehand the application's computing requirements. The task type and its parameters become a priority. This information will ultimately affect the behavior of the hardware accelerator and impact the computing cost of the implementation.

In this work, we propose an specialization of the PiSDF model and the AAM methodology that targets SPS-CGRA applications. This application model provides the precise level of abstraction, by introducing new descriptors, required to exploit the features of an SPS-CGRA.

2.3 Proposed Application Model

The applications that target an SPS-CGRA usually consist of tasks with mixed granularity, from bit-wise operations to complete transformations. Also, the application tasks exhibit heterogeneity in their parameters (kernel size, kernel shape, computing constants, length of input), which needs to be considered during the mapping. Additionally, the application computing requirements are commonly known in advance, and they are usually compatible with the target SPS-CGRA.

Recall that SPS-CGRAs are configuration and dataflow driven platforms [18]. These architectures employ configuration control to define the behavior of their resources. As a result, the latencies of the resources, and consequently the computing cost of the final implementation of an application onto an SPS-CGRA, depend directly on the type of task and its parameters, and the hardware physical realization of the resource that executes the task. This defines the necessary information that the application model should have. Since we consider coarse granularity of computing, the descriptors of the application model do not require to represent any latency information but the type of task and its parameters. The latency function that models the material realization of the resource is a part of the hardware model (see Section 3.3.2).

Since there is an extensive number of application fields where an SPS-CGRA is used, we propose a generic application model adapted to most of these application fields. We consequently need to define an application:

Definition 2.3.1 (Application)**.** An application consists of a set of ordered heterogeneous tasks with data dependency between them. An application consists of a set of indivisible tasks with mixed granularity.

Definition 2.3.2 (Data dependency)**.** Data dependency is the relation between a task and the preceding tasks. A task might need the completion of all its preceding tasks to start its execution. The data dependency defines the order of execution of the tasks.

Definition 2.3.3 (Task)**.** A task is an atomic transformation applied to the input data to obtain output data. In our context, it can represent an image processing operator, an arithmetic operation, a bitwise operation, among others. Each task is defined by a set of parameters (numerical constant, a boolean value, a string of characters).

A task may represent a transformation that requires one or multiple input data.

Definition 2.3.4 (Input(s) of a task)**.** A task might have one or more inputs of data.

The result of the transformation is represented by the output data. A task might forward or transfer its output data to one or more successor tasks. Hence, creating data dependency between the task and its successors.

Definition 2.3.5 (Output(s) of task)**.** The task output results from the transformation applied to the data and can be transferred to the following task or broadcast to several tasks (successors).

Each application requires at least one source of data that usually comes from a sensor, camera, or electronic device, producing data to be processed.

Definition 2.3.6 (Source of data)**.** The source of data is the origin or producer of data to be processed. We define a source of data of an application as a **sensor**. An application can have one or more sensors, each one with different parameters.

A sink of data will consume the transformed data. This data sink might be an actuator, a display, a monitor, between other possible devices.

Definition 2.3.7 (Sink of data)**.** The sink of data receives or consumes the processed data. We define a sink of data of an application as an **actuator**. An application can have one or more actuators, each one with different parameters.

2.4 Formal Application Model

Let $G_{APP}(T, D)$ be a directed hypergraph that models an application. A hypergraph allows modeling a task that may broadcast its output data to several successor tasks. T is a set of nodes that represent the tasks of the application. D is a set of oriented hyperedges that represent data dependency between tasks.

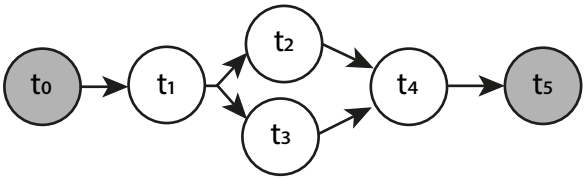


Figure 2.1 – Example of an application model hypergraph.

We call a task $t_i \in T$, so that $t_i = (type_i, p_i)$, where $type_i$ is the type of atomic transformation applied to the data and p_i is the set of the transformation parameters. We call also $type_i$ and p_i as a descriptors of t_i. We can see in Figure 2.1 an example of an application model hypergraph.

Observation 2.4.1 (Sensor and actuator descriptors)**.** A node t_i that represents a **sensor** or an **actuator** will have $type_i = interface$ and p_i are the parameters of the input/output data. A complete example is given in Section 2.5.

We represent an atomic task with a single node. Notice in Figure 2.2 that the input edge represents the input of data to the task (node) and the output edge the output of processed data from the task. In the case of the task of Figure 2.2 the input degree of the node is one (one input data), and the output degree of the node is also one (one output data).

Definition 2.4.1 (Input degree of a task)**.** We define the input degree of a task t_i as the number of different input of data (number of predecessors). The input degree of a task should be a natural number greater than zero.

$$deg^-(t_i) > 0,\ deg^-(t_i) \in \mathbb{N} \tag{2.1}$$

Definition 2.4.2 (Output degree of a task)**.** We define the output degree of a task t_i as the number of successors connected to t_i. Notice that the output data, sent to all the successors, is the same (broadcast). The output degree of a task should be a natural number greater than zero.

$$deg^+(t_i) > 0,\ deg^+(t_i) \in \mathbb{N} \tag{2.2}$$

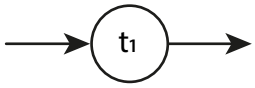


Figure 2.2 – Example of a single task.

Graphically, we represent a task with a node. This node will have the name of the task as a label. In Figure 2.2 the name of the task is t_1.

Observation 2.4.2 (Sensor and actuator representations)**.** Additionally to the name of the task as a label, a node t_i that represents a **sensor** or an **actuator** will have a different coloration than the other type of tasks.

We represent a sensor node (Figure 2.3) or an actuator node (Figure 2.4) with a different color than the other nodes. Notice in Figure 2.1 that the sensor and the actuator nodes are in gray and the other nodes are in white.

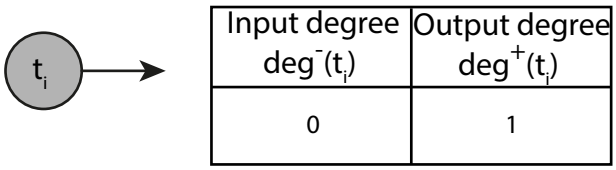


Figure 2.3 – Example of a single sensor.

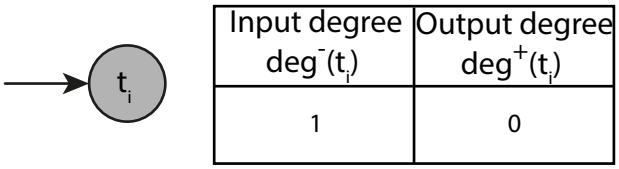


Figure 2.4 – Example of a single actuator.

Also, we can see differences in the degrees of the nodes from Figure 2.4 and 2.3.

Definition 2.4.3 (Input degree of a sensor node)**.** The input degree of a sensor node t_i is always zero.

$$deg^-(t_i) = 0 \tag{2.3}$$

Definition 2.4.4 (Output degree of an actuator node)**.** The output degree of an actuator node t_i is always zero.

$$deg^+(t_i) = 0 \tag{2.4}$$

The output degree of a task or a sensor node can be from one to $n \in \mathbb{N}$. Figure 2.5 shows the graphical representation of a task that broadcasts its output data.

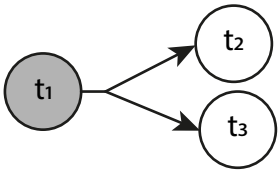


Figure 2.5 – Example of a task that broadcast is output data.

Equally, the input degree of a task or an actuator node can be from one to $n \in \mathbb{N}$. In Figure 2.6, we can see an example of the graphical representation of a task with input degree two.

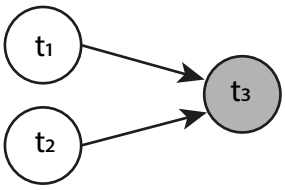


Figure 2.6 – Example of a task with input degree two.

2.5 Examples of Real-Life Applications

In this section, we introduce two application examples. These examples, taken from image processing, will help us describe the modeling process. The objective is to illustrate how the proposed application graph applies to real-life problems.

Image processing uses digital computers to perform some operations on an image that allows to extract meaningful information or enhance the image [116]. Image processing algorithms are used in numerous time-critical applications such as autonomous navigation systems, unmanned aerial vehicles, or industrial control. From the set of image processing algorithms, we can mention an important subset grouped under the name of mathematical morphology [117]. Mathematical morphology groups algorithms capable of extract image components that are useful in the representation and description of region shape, such as boundaries, skeletons, and the convex hull [116]. Additionally, mathematical morphology includes algorithms used for pre- or post-processing operations, such as filtering, thinning, and pruning. The importance of morphological mathematics has attracted the attention of many hardware designers, who seek to implement their operators efficiently. One of these hardware implementations is the Morphological Co-processor Unit (MCPU) [12]. The MCPU assembles several efficient dilation/erosion units, geodesic units and ALUs to support an extensive collection of morphological operations (See Section 1.4 and 3.5). In addition, an interesting feature is that the shape and size of structuring elements are programmable. Also, MCPU belongs to one of the hardware realizations dealing with latency minimization. Due to internal knowledge of this SPS-CGRA, throughout the thesis, we use it as an example of a real-life SPS-CGRA. We select two applications that target the MCPU to describe the proposed modeling process.

2.5.1 Alternated Sequential Filter

The first example is an Alternated Sequential Filter (ASF) [118]. The ASF is extensively used for a nonlinear filtering of images, preserving the topology characteristics. It is known for its computing cost. In our context, it represents a long linear pipeline of tasks with the possibility

to overpass the length of the hardware resources (Figure 2.7). The Equation 2.5 provides its formal definition.

$$ASF^{\lambda}(f) = \varphi^{\lambda}\gamma^{\lambda} \dots \varphi^{1}\gamma^{1}(f) \tag{2.5}$$

where f denotes the input image, λ the SE size, γ, and φ are the operators of opening and closing defined in [117]. In this example, we consider an ASF^{λ} with $\lambda = 4$. Here we consider that the MCPU can only implement erosion and dilation operations. Due to these hardware constraints, we need to decompose both closing and opening, operations with even numbers for size of structuring element.. The closing and opening can be decomposed into two basic morphological operators, erosion (ϵ) and dilation (δ). By doing this decomposition, we can transform the ASF^4 equation using erosion and dilation operators, and in the same way, reduce it as shown in Equation 2.6.

$$ASF^4 = \varphi^9\gamma^9\varphi^7\gamma^7\varphi^5\gamma^5\varphi^3\gamma^3 = \delta^9\epsilon^{17}\delta^{15}\epsilon^{13}\delta^{11}\epsilon^9\delta^7\epsilon^5\delta^3 \tag{2.6}$$

After the transformation and reduction, the application will consist of nine tasks. The input parameters for both erosion and dilation are the size and shape of the structuring element[2]. We can see the resulting application model in Figure 2.7. Notice the use of one sensor node and one actuator node, both in gray. Also, the respecting parameters of each task are hidden for visualization but contained in a joint specification file.

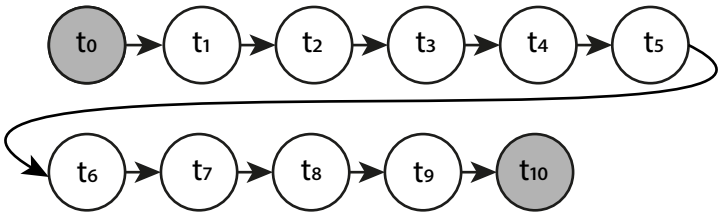


Figure 2.7 – Application model of the example ASF^4.

We list the entire set of parameters in Table 2.1. As we can see, for the sensor and actuator nodes, the *type* is $interface$. The remaining nodes take their *type* according to the specification of the application.

Table 2.1 – Descriptors of the tasks for the ASF^4 example

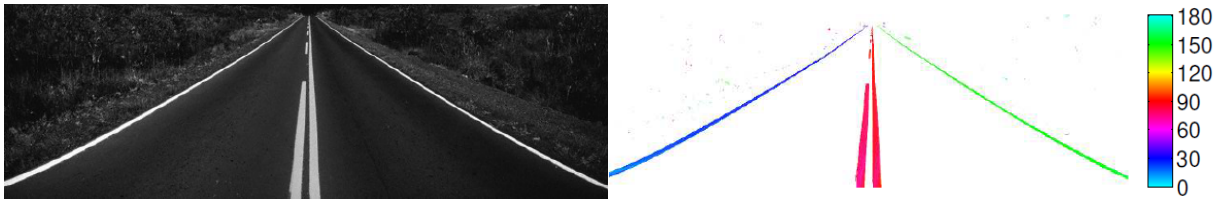
Task	$type_i$	p_i		
		Size	Shape	Resolution
t_0	interface	N/A	N/A	height, width
t_1	dilation δ	3	square	N/A
t_2	erosion ϵ	5	square	N/A
t_3	dilation δ	7	square	N/A
t_4	erosion ϵ	9	square	N/A
t_5	dilation δ	11	square	N/A
t_6	erosion ϵ	13	square	N/A
t_7	dilation δ	15	square	N/A
t_8	erosion ϵ	17	square	N/A
t_9	dilation δ	9	square	N/A
t_{10}	interface	N/A	N/A	height, width

[2]The structuring element is a computing window with different possible shapes: square, line, circle, custom.

In Table 2.1, *N/A* means *Not Applicable*, which are descriptors of a task that do not correspond to its type. For example, t_0 is a task that models a sensor, only the descriptor of resolution applies to it. Contrarily, t_2 is a task that models an erosion operation, in which resolution is not one of its descriptors. The parameter Resolution (height and width) represents the number of input data samples to process.

2.5.2 Road Line Orientation Detection

Our second example is a road line orientation detection [119, 15]. This second application represents a highly parallel task organization. The principle is the computing of oriented linear openings of the input (Figure 2.8).



(a) Original Image. (b) Composite image indicating the line orientation.

Figure 2.8 – Road line orientation detection [15].

Figure 2.9 shows the complete road line orientation detection. We only focus on the computation of the oriented openings. The rest of the application is processed in a CPU.

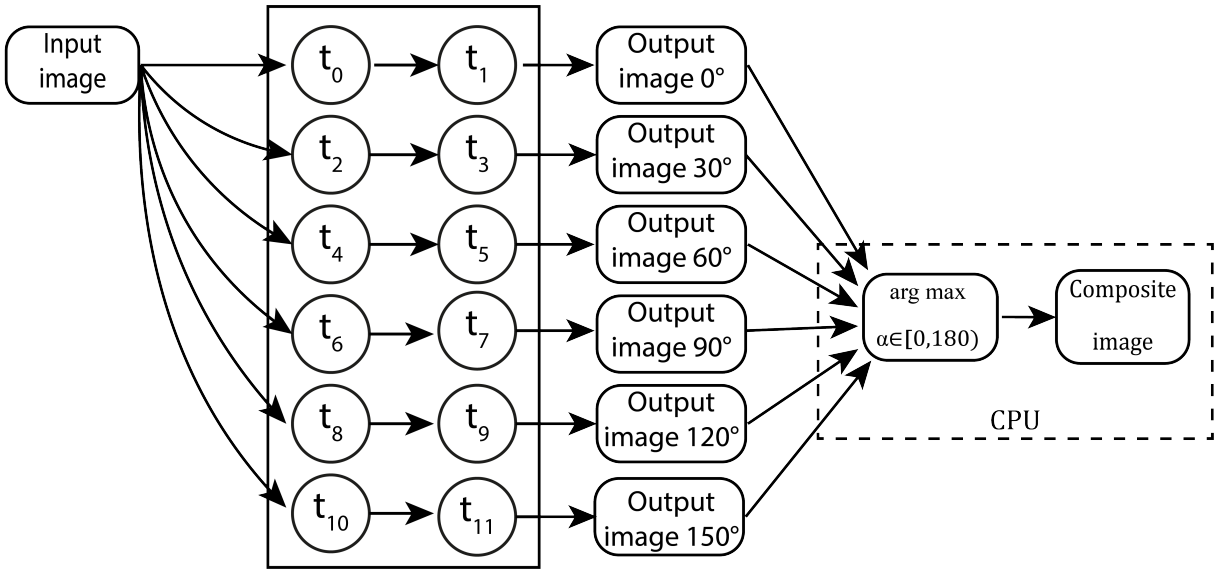


Figure 2.9 – Complete road line orientation detection.

Again, we decompose the opening in basic morphological operators (erosion and dilation). We can see in Figure 2.10 the corresponding application graph. The actuators $t_{3,6,9,12,15,18}$[3] are the resulting processed images that will be transferred to the CPU. In Table 2.2 we describe the parameter for each task.

[3]Here and in the remaining part of the thesis, we will use $t_{i,j,k,l}$ as an abbreviation of t_i,t_j,t_k,t_l.

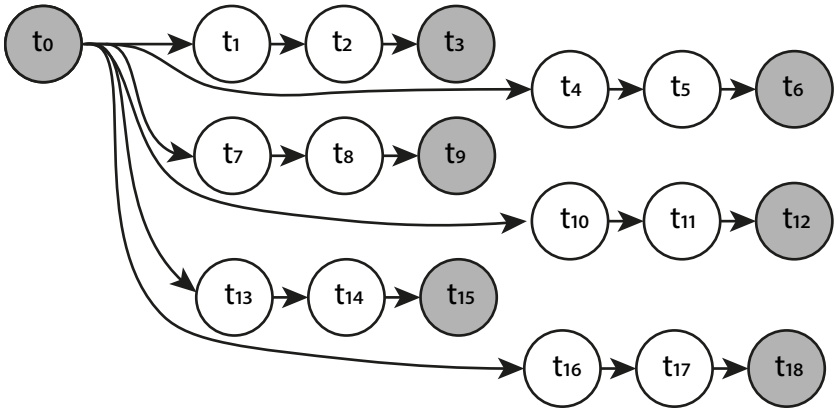


Figure 2.10 – Application model of the example of the road line orientation detection.

Table 2.2 – Descriptors of the tasks for the Road Line Orientation detection (we consider default position of the center of structuring element).

Task	$type_i$	p_i			
		Size	Shape	Angle	Resolution
t_0	interface	N/A	N/A	N/A	height, width
t_1	erosion ϵ	31	line	0	N/A
t_2	dilation δ	31	line	0	N/A
t_3	interface	N/A	N/A	N/A	height, width
t_4	erosion ϵ	31	line	30	N/A
t_5	dilation δ	31	line	30	N/A
t_6	interface	N/A	N/A	N/A	height, width
t_7	erosion ϵ	31	line	60	N/A
t_8	dilation δ	31	line	60	N/A
t_9	interface	N/A	N/A	N/A	height, width
t_{10}	erosion ϵ	31	line	90	N/A
t_{11}	dilation δ	31	line	90	N/A
t_{12}	interface	N/A	N/A	N/A	height, width
t_{13}	erosion ϵ	31	line	120	N/A
t_{14}	dilation δ	31	line	120	N/A
t_{15}	interface	N/A	N/A	N/A	height, width
t_{16}	erosion ϵ	31	line	150	N/A
t_{17}	dilation δ	31	line	150	N/A
t_{18}	interface	N/A	N/A	N/A	height, width

As described in Section 2.5.1, N/A refers to descriptors that do not apply to a particular task. We can observe the additional parameters of *angle* needed to describe the tasks. We can validate that our model can describe several types of parameters of the applications. And more importantly, we can describe either sequential and parallel applications and the combination of both.

2.6 Conclusions

In this chapter, we have introduced our application model. It is a specialization of the PiSDF model and the AAM methodology applied to applications that target an SPS-CGRA. While several application models are being used for different platforms, direct use of one of them is not possible. In this regard, we propose to use a hypergraph and a different set of descriptors

for the tasks. The hypergraph allows one to model a possible broadcast of the task's output to several successors' tasks or tasks with several inputs of data. Additionally, it can be used to model the basic application structures, sequential and parallel, and their combinations. Thus, fully exploit the parallelism that an SPS-CGRA can offer. The descriptors of the task only focus on the type of task and its parameters. This decreases the complexity of the application model. And also matches the features of the hardware model and reduces the complexity of the mapping algorithm.

We describe the modeling process with two application examples. We select two applications from the set of applications that the MCPU can implement. Although the examples come from image processing, our application model can be used for other application fields. The task's descriptors allow one to model any parameter, such as integers, strings, or boolean values. Furthermore, the graph-based representation can capture most of the current application structures. As a whole, our application model is generic, and the modeling process is low complexity.

Chapter 3

Hardware Model

3.1 Introduction

Recall that an SPS-CGRA consists of an irregular systolic array of heterogeneous processing, communication, and memory resources.

The processing resources are functional units able to perform a specific set of tasks, such as morphological operators, ALU-like tasks, hashing algorithms, image filtering operators, or others. Not all the processing resources have the same latency or use the same range of parameters. Even if the operation is the same, the physical implementation may vary. Thus the latency characteristics or the required parameters may be significantly different, even for the same function. This is due to the inner implementation of the given functional unit.

The communication resources are dedicated to transfer, copy and perform read/write memory operations. They realize datapaths. Like the processing resources, each communication resource can have a different physical realization. Each physical implementation may have different latencies. We may have a multiplexer with a latency of one clock cycle or an crossbar that requires two clock cycles or more to assign the input to the correct output.

The memory resources are memory blocks (RAMs, FIFOs, Flash or erasable memories). An SPS-CGRA can have one or more memory resources from different types. This will impact the communication resources that deal with the read/write operations.

In addition, an SPS-CGRA has a configuration control layer that assigns the configuration context to all the resources. The number of configurable parameters defines the size of the configuration context. The largest size of the configuration context will occur when the user programs the entire SPS-CGRA. But, the user may not need complete programming of the SPS-CGRA. In these cases, the configuration context will be reduced. The configuration context size has a direct impact on the configuration control cost. Thus, the configuration control cost in an SPS-CGRA is a function of the number of parameters to program.

To take full advantage of these architectures, we need an abstraction of the hardware that can capture all these details. This abstraction comes in the form of a hardware model. To help the labor of the mapping algorithm and also increase the probability of getting the optimal implementation, the hardware model should be able to provide the correct information of the architecture. Information about the resource features and latency functions is the priority. Also, the hardware model should provide the means to abstract the greatest number of types of SPS-CGRAs possible.

Several hardware models have been proposed targeting platforms from processor-based up to hardware accelerators. Most of the works related to processor-based and multicore-based architectures use the task model to define the latency requirements. To allocate the task to the optimal processor/core, the mapping algorithm considers the WCET, a descriptor widely used by the task model. Other works provide the means to deeply detail the latency of the tasks [78, 72], either dividing the latency or consider different physical realizations. As we can see, for these platforms, the details of latency are on the side of the application model, but

as the SPS-CGRA is a data-driven architecture, the hardware side (model) is the one that should define the latencies. On the other hand, some works include a specific hardware model [110, 100]. Commonly they are custom to a particular architecture or do not provide the means to accurately describe the latency of the resources. This lack of solutions creates a significant gap for a hardware model of an SPS-CGRA.

In this chapter, we introduce a new hardware model for SPS-CGRA. It provides the means to accurately define the latencies of hardware resources and generate a mapping. It is general enough to model several types of SPS-CGRAs. We define the structure and elements of the hardware model. We provide information about the descriptors and use an example to explain the process of the modelization of an architecture.

We organize the remaining part of the chapter as follows. Section 3.2 reviews state of the art. Section 3.3 presents the proposed hardware model. In Section 3.4, we introduce the formalization of the model. In Section 3.5, we present two hardware examples for a better understanding of our model. Finally, Section 3.6 summarizes this chapter.

3.2 State of the Art

A hardware model aims to abstract efficiently a system architecture [83]. Its purpose is to capture the inner mechanism and latency characteristics of the architecture and be able to provide the best amount of information to the mapping algorithm. The amount of information that the model delivers is in direct relation to the complexity of the mapping algorithm. A complex model might increase the exploration time and the complexity of the mapping algorithm without benefits for the implementation. Therefore, we make a trade-off between the description of the architecture and the complexity of the model to relax the complexity of the mapping algorithm. Thus, the hardware models abstract only the crucial points, such as type of resources, working parameters, and latency characteristics.

Since the hardware models focus on some features of the architecture, usually, they are custom. From this perspective, we divide this section into types of hardware targets, and at the end, we present a general discussion of the presented works.

3.2.1 Processor-Based Systems

In a processor-based system, a bus connects one or more processors. The interconnections may be point-to-point or fully connected. The memory of the systems may be shared or distributed. Few works detail a specific hardware model for this type of system. Most of the works use the task model or its variations to describe the latency characteristics [80, 68, 69, 71].

Lo and Jean [66] introduced a Signal Flow Graph (SFG) for processor-based systems. It is a graph-based model that consists of processing nodes, communication edges, and delays. The nodes of the graph represent zero delay arithmetic or logic functions. The graph integrates edges and weighted edges. The edges represent data channels. The weighted edges represent a time delay operator. Sinaei and Fatemi [120] presented another hardware graph-based model for processors. The nodes represent processing elements and memory elements. Each element has a descriptor for its capacity, energy consumption, and cost. Wang et al. [121] represented a high computing environment [122], consisting of a set of machines [123], as a directed acyclic graph. They described each node with a value of the estimated expected execution time of subtask s_i on machine m_j.

Tafesee et al. [124] divided its hardware model into two aspects. An architecture model, which is a set of processors and memories. Each element, either processor or memory, is associated with its architecture, type, power, clock rate, interrupt, and memory. Particularly for the processors, the model includes the set of tasks that the processor can implement, the WCET, and the power cost. The second element of the hardware model is a topology model, which describes the physical layout of the components and defines the communication cost. Ma et

al. [111] modeled a high-performance computing system with a set of homogeneous processing elements with a parameter of state: active, idle, and shutdown.

Castrillon et al. [41] used a list to describe the processing elements and the communication primitives. This approach also targets processor-based systems. They defined each processing element with the operation cost, a descriptor of the multitasking support, including the context switch time and the scheduling policies. The communication primitives include three descriptors. The offset represents the overhead as a constant. The start is a variable that increases as a function of the transferred bytes. And the stair, which is a function of the communication cost.

3.2.2 Algorithm-Architecture Matching

The Algorithm-Architecture Matching (Adequation) (AAM) methodology [35, 36, 37] and its extension (SynDEx-Mix) [86] model an heterogeneous distributed architecture as an oriented hypergraph. The architecture may be based on programmable (processor-based), reconfigurable (FPGA), or ASIC components. The methodology defines four subsets of nodes, operator, communicators, memory, and bus/mux/demux. The memory nodes may be Random Access Memories (RAM) or Sequential Access Memories (SAM), which can be shared for data communications. The operator node is a sequencer node that represents the execution of a finite subset of operations (a WCET is associated with each operation). A communicator node is a sequencer node that executes communication operations to/from its connected memories (RAM, SAM). The communication operation latency is a function of the size of data to transmit and the available bandwidth.

3.2.3 Multiprocessor System-on-Chip

A Multiprocessor System-on-Chip (MPSoC) uses multiple processors along with other hardware subsystems to implement a system [125]. Frid and Sruk [96] introduced a graph-based model for MPSoCs. It uses a weighted directed graph where they divide the nodes into processing elements and memory nodes. The execution time of each task of the application is the primary descriptor of a processing element. If the processing element can not execute the task, the execution time is infinite. The memory nodes use the number of read, write, or read-write ports as a descriptor. The weight of the edge is associate with the speed of the write/read operation. The same authors present in [95] a simpler model. It only considers the processing elements and computation speed of each one.

Pelcat et al. [83] presented the Linear System-Level Architecture Model, which consists of an undirected graph that includes the set of processing elements, the set of communication nodes, the set of unidirectional edges, a function that represents the computing cost, and the lagrangian coefficient setting the computation to communication cost ratio. The computation cost can be energy, area, price, amount of memory depending on the model created. This model is used within the framework PREESM [45, 44].

Jalier et al. [40] described an MPSoC with a SystemC description. They defined only three types of entities, processor, memory, and communication media.

Zahaf et al. [126] modeled a heterogeneous architecture as a set of execution engines. Each execution engine is characterized by its execution capabilities, the engine's tag, and its scheduling policy. The engine's tag allows identifying different physical implementations. Using the scheduling policy property allows supporting different scheduling policies, which can be preemptive or non-preemptive.

3.2.4 Network-on-Chip

A Network-on-Chip (NoC) is a structured and scalable interconnection architecture that consists of multiple segments of wires and routers, structured as a grid [127]. Lu et al. [110] used a tuple (N, λ) to describe a NoC. N is the set of nodes representing the routers, and $\lambda_i \in \lambda$ represents

the number of available neighbors nodes at the X or Y coordinate of node i. Yang et al. [128] used an array of tiles to model an NoC, where each tile represents an intellectual property IP, a router, and a set of network interfaces. Mehran et al. [129] used a directed graph called Architecture Characterization Graph (ACG), where each vertex is a tile and the edges the routing path. Each node has a descriptor that represents the average energy consumption of sending one bit of data. In previous work, Mehran et al. [130] introduced an order list called Platform Priority List, according to the connection degree of each tile.

3.2.5 Coarse-Grained Reconfigurable Architectures

Mei et al. [100] presented Modulo Routing Resource Graph (MRRG), a variation of the Resource Routing Graph (RRG). The RRG is a time-space graph in which all resources (space dimension) are modeled with vertices. There is one such vertex per resource per cycle (time dimension) in the schedule being generated. Directed edges model the connections over which data values can flow from resource to resource [131]. The MRRG is a directed graph where the set of nodes corresponds to the ports, wires, or artificially created nodes. Each node represents an execution time t. The edge set corresponds to switches that connect the nodes. Furthermore, the initiation interval (II) of the loop is one of the descriptors of an MRRG. Several works use this model [131, 104] or a variation of it [103]. Yoon et al. [105] used a directed graph to model a CGRA, where the edges represent a data dependence between processing elements.

3.2.6 Field Programmable Gate Array

Li et al. [114] presented a hardware model based on a set of composable templates for FPGA accelerators. The templates match the possible elements of a boolean dataflow graph used as an application model. Each template is a building block for the architecture and its datapaths.

3.2.7 Discussion

A summary of the previously introduced works is presented in Table 3.1.

Table 3.1 – Hardware models state of the art summary

Platform	Model	Features
Processor-based	Task Model [80, 68, 69, 71]	Graph-based model Homogeneous/ Heterogeneous resources Parameters: WCET, deadlines, overall latency characteristics
	Signal Flow Graph [66]	Graph-based model Able to characterized communication delays
	Sinaei and Fatemi [120]	Graph-based model Processing and memory resources Parameters: capacity, energy consumption, and cost
	Wang et al. [121]	Graph-based model Able to characterized the execution time
	Tafesee et al. [124]	Divided into an architecture model (description and parameters of the resources) and topology model (interconnections)
	Ma et al. [111]	Able to characterized the status of the resource (active, idle, and shutdown)
	Castrillon et al. [41]	A descriptor of the multitasking support, including the context switch time and the scheduling policies
MPSoC	Frid and Sruk [96, 95]	Graph-based model Processing and memory resources Able to characterized the execution time
	Linear System-Level Architecture Model [83]	Graph-based model Able to characterized the computing cost as a function
	Jalier et al. [40]	Processor, memory, and communication media resources
	Zahaf et al. [126]	Parameters: execution capabilities, the engine's tag, and its scheduling policy
NoC	Lu et al. [110]	Considers the number of available neighbors nodes as parameter
	Mehran et al. [129]	The average energy consumption of sending one bit of data is consider as parameter
CGRA	Modulo Routing Resource Graph [100]	Graph-based model Set of nodes corresponds to the ports, wires, or artificially created nodes
FPGA	Li et al. [114]	Set of composable templates (building block) for FPGA accelerators

As one of the inputs of a mapping algorithm, the level of abstraction of a hardware model is critical. An accurate hardware model without high complexity is not trivial to define.

A hardware model for SPS-CGRA requires three main features. In the first place, the means to define heterogeneous resources. In the field of multiprocessors, usually, the processors are homogeneous, and the models do not provide any means to define heterogeneous resources [80, 68]. As the requirements of the applications started to increase, the systems begin to incorporate heterogeneous resources [69, 71], like in the AAM methodology [85]. MPSoCs, NoCs, and CGRAs models normally support heterogeneous resources. Usually, these models are graph-based and make use of the set of properties of the nodes to describe heterogeneous resources. Moreover, the graph's edges are used to describe interconnections between resources. [131, 45, 86].

The second feature is a fine grain latency modelization. Most of the works model the latency of allocating a task onto a resource using the WCET [68, 85]. This approach might lead to a pessimist performance evaluation (See Section 4.4.1). Few works allow to model different latencies for the same resource, like in [80], which is crucial in the context of programmable architectures. Moreover, the possibility to model different physical implementations that are able to execute the same task is not possible, with the exception of [126], however, without a fine grain latency modelization.

The last important feature is the configuration control layer. Usually, the hardware models allow to abstract only the hardware resources and their interconnections [131, 114, 100]. The configuration control layer of the system is often neglected and not considered within the hardware model. However, this layer directly impacts the computing cost of the implementation. The time elapsed during the reconfiguration of the system should be included in the performance evaluation for a cycle-accurate result.

Clearly, there exists a gap in the field of hardware models for SPS-CGRA. In the following section, we describe the structure of an SPS-CGRA and the latency of its elements.

3.3 Software Programmable Streaming Coarse Grained Reconfigurable Architectures

In Section 1.4, we introduced our target architecture, which we name in this work Software Programmable Streaming Coarse-Grained Reconfigurable Architectures (SPS-CGRA). Examples of this architecture can be found in several application fields (see Section 1.4). Moreover, it can be an overlay based on an FPGA, ASIC, or SoPC technology. These different application fields and technologies used to build an SPS-CGRA allow a great diversity of types of hardware resources and interconnections. However, preserving many similar characteristics, which allows one to differentiate the SPS-CGRA from the rest of the hardware platforms. In this section, we will describe the internal characteristics of an SPS-CGRA.

Definition 3.3.1 (SPS-CGRA)**.** An SPS-CGRA is an irregular systolic array of heterogeneous hardware resources with fixed interconnections. The hardware resources are heterogeneous in both functionality and physical realization. It may contain different random access memory blocks. An integrated interface allows it to receive/transmit data from outside sensors/actuators. The degree of programmability is given through software reconfiguration. It may allow partial reconfiguration.

As we can see in Figure 3.1, an SPS-CGRA is a complex architecture that includes resources with heterogeneous characteristics. From memory blocks up to different types of processing resources (PR_n), and usually including an interconnection block. It can be used to process data from one or several sensors and produce processed data for one or several actuators. Each processing resource can be itself composed of some hardware accelerators or processor cores. Additionally, a configuration control layer is able to program the processing resources and modify the interconnections through a set of parameters. In the following section, we will describe the details of each resource.

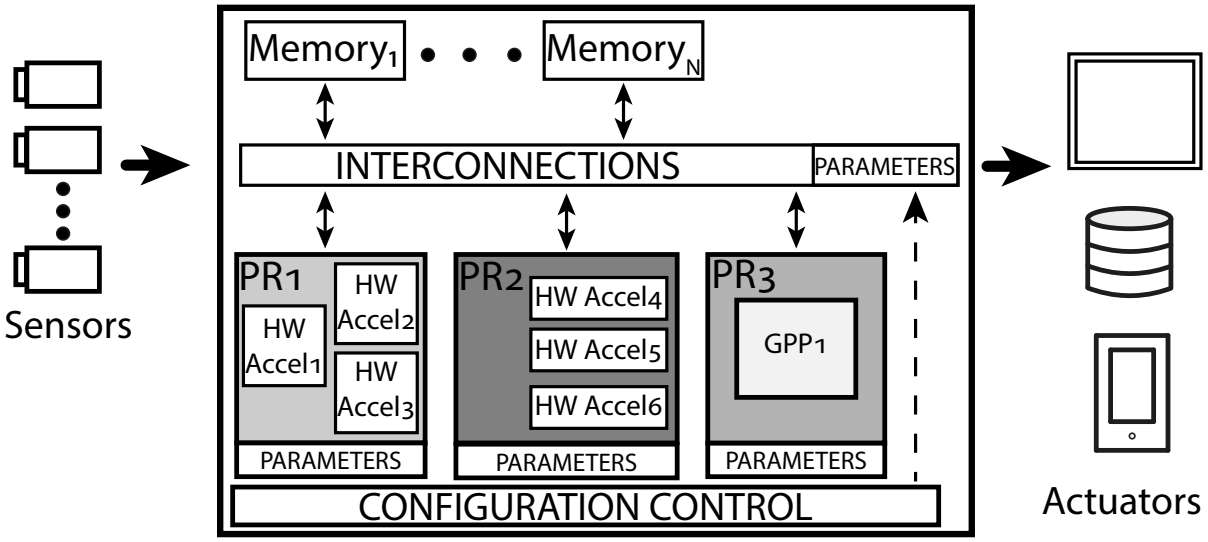


Figure 3.1 – Global architecture of an SPS-CGRA.

3.3.1 Architecture Resources

As it was already introduced, an SPS-CGRA consists of a set of **processing**, **communication**, **memory** and **configuration control** resources. The **configuration control** resources are in charge of delivering the configuration context to the hardware resources. In the following paragraphs, we bring the definitions of each hardware resource with the objective of clearly delimit the terminology.

Definition 3.3.2 (Configuration control resource)**.** A configuration control resource defines the behavior of the hardware resources. It is in charge of distributing the configuration context.

Depending on the capabilities of the SPS-CGRA, a configuration control resource might support partial reprogramming of the hardware resources, through the selection of a subset of hardware resources and distributing only to them the new configuration context.

A **processing** resource can perform a transformation of its input data to produce output data. This transformation can be from a simple arithmetic operation to a complex application-specific operation. A processing resource can have several inputs and also several outputs. The input type can be the same or different depending on the transformation. The processing resource can broadcast its output to one or several other resources through an unique output.

Definition 3.3.3 (Processing Resource)**.** A processing resource applies a given transformation to its input data. It may be able to implement one or several transformations. Each transformation may require a different set of input parameters. Additionally, the processing resource is characterized by latency, which is a function of the implemented transformation and its input parameters. The processing resource may have one or more inputs. The transformation's output may be broadcast to one or several resources.

To manage the unused processing resource correctly, each hardware resource can systematically perform two special operations: *copy* and *disable* (see Section 3.4.4).

The **communication** resources can transfer, copy data from a memory resource to a processing resource, copy data from a processing resource to a memory resource, copy data from a memory resource to another memory resource, and route the data (as multiplexer).

The communication resources may represent multiplexers, buses, or crossbars. These resources may have several inputs and several outputs.

Definition 3.3.4 (Multiplexer / Bus / Crossbar)**.** A communication resource represents a multiplexer/bus/crossbar. It is a resource that performs a copy operation from one or more inputs

to one or more outputs. In addition, they are described by the latency function related to the data transfer.

A subset of the communication resources is dedicated to perform read/write operations. These resources are able to transfer data from and to a memory resource. They only have one input and may have one or more outputs. If the resource has several outputs, this represents the broadcast of the same output data to several resources.

Definition 3.3.5 (Memory access)**.** Memory access resources are a subset of the communication resources that perform read/write operations from or to its associated memory resources. Each memory access resource is associated with a finite address space of a memory block.

Another subset of communication resources is the interface resources. They represent the external producer and consumer of data. The external producer is the resource that provides the data for the system. It may be a camera, a personal computer, transducers, or a flash memory.

Definition 3.3.6 (External Producer of Samples)**.** An external producer of data is the main source of data for the system. There may be one or more external producers.

The external consumer of data is the sink of the output processed data. There may be one or more external consumers. The external consumer may be a display, a personal computer, or a monitor.

Definition 3.3.7 (External Consumer of Samples)**.** An external consumer of data is the sink of processed data of the system. There may be one or more external consumers.

The **memory** resources are memory blocks (RAM, FIFO, Flash or erasable memory). There may be one or more memory resources in the system. Each memory resource is unique. Each memory block may have one or more read/write dedicated channels, which are exposed to the memory access or interface resources.

Definition 3.3.8 (Memory resources)**.** A RAM / FIFO / Flash or erasable memories are memory resources used to store a finite amount of data given by its address space.

3.3.2 Hardware Described Through Latency

We can divide the architecture characteristics latency into two main categories, **configuration cost** and **hardware resources latency**. The former relates to the configuration control resource (see Definition 3.3.2). The delivery of the configuration context to all the hardware resources requires a specific amount of time, which is the time consumed during the programming of the hardware resources. This time may be unique for each hardware resource, as it is a function of the number of parameters and information to program.

Definition 3.3.9 (Configuration cost)**.** The configuration cost is the time consumed during the programming or configuration of all the hardware resources. This time is a function of the parameters of each hardware resource, and therefore unique for each resource.

The **hardware resources latency** relates to the latency generate by each hardware resource in the system. The heterogeneity of the hardware resources makes the latency of each hardware resource unique and a function of its parameters. Consider the example in Figure 3.2, where the same task (t_i) has been allocated twice onto the same resource (r_j) but with a different set of parameters. In the first case, a set of parameters has been applied to the resource. The outcome of this set of parameters will be a pair of input and computing latencies (defined formally in the next paragraph). On the other hand, if we change the set of parameters, these latencies will change.

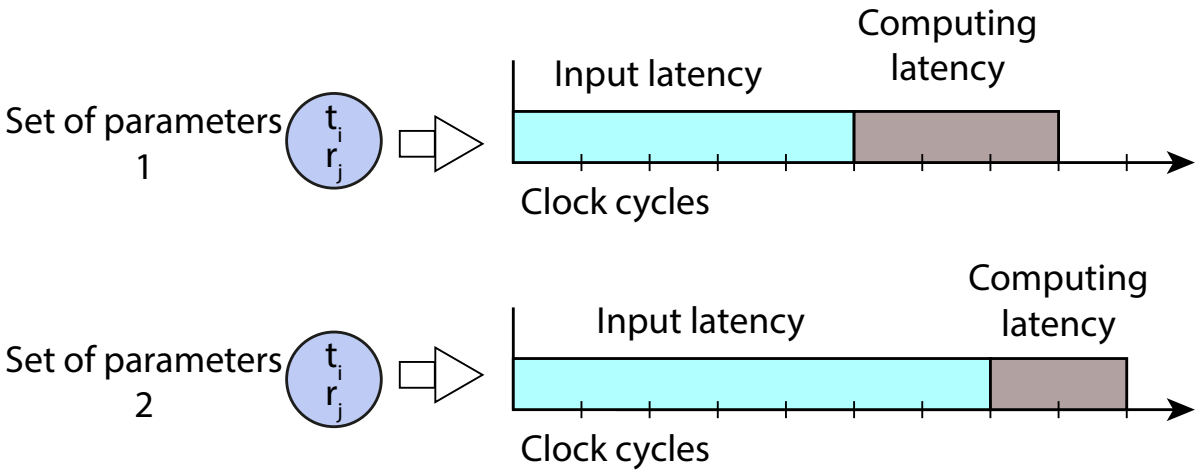


Figure 3.2 – Impact of the change of parameters on the latency of a resource

Recall that a processing resource applies a given transformation to the input data to obtain output data (see Definition 3.3.3). This transformation generates a latency value which is a function of the type of the transformation and its parameters. Since an SPS-CGRA is a stream-based processing platform, this latency function is, in fact, the union of two latencies: **input latency** and **computing latency**.

Definition 3.3.10 (Input latency)**.** The input latency is the amount of time (measured in clock cycles) from the arrival of the first sample until the arrival of the last sample required to start the processing of the samples. At this moment, the processing resource is ready to output the first processed sample.

Definition 3.3.11 (Computing latency)**.** Computing latency is the amount of time (measured in clock cycles) elapsed between the output of two consecutive processed samples after the computing (processing) pipeline is full and the feed of new samples is constant.

The **input latency** and **computing latency** are functions of the type of transformation and its parameters. They are defined by the physical hardware realization for a given task.

A communication resource that applies a copy operation (i.e., from memory to memory) also generates a latency value. This value is a function of the physical realization, and it is constant. Equally, the communication resource that applies a read/write operation (i.e., between memory and processing resource) generates a latency value, which is a function of its physical realization. Lastly, the external producer and consumer of samples generate a latency value representing the latency of producing one input sample or consuming one output sample. In our model, the memory resources do not generate any latency since, we propose to integrate it into the communication resources.

3.4 Formal Model

In this work, a directed hypergraph $G_{HW}(S, K)$ models an SPS-CGRA architecture. The set of nodes (S) represents the entire SPS-CGRA resources. The set of oriented hyperedges (K) models the hardware resources interconnections. The S is a union of several subsets representing different hardware resource classes. Their relations are depicted in Figure 3.3. Each of them is defined in the following sections.

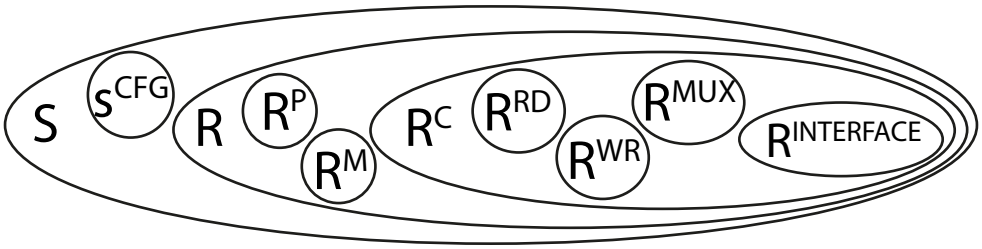


Figure 3.3 – Composition of the set S

3.4.1 Set S of SPS-CGRA Resources

We define the set of nodes S as the union of the node s^{CFG} and the set R.

$$S = s^{CFG} \cup R \tag{3.1}$$

The unique node s^{CFG} is the sequencer node that represents the configuration control of the system (See Definition 3.3.2). The set R represents the set of hardware resources of the system. This set is a union of the following subsets.

$$R = R^P \cup R^C \cup R^M \tag{3.2}$$

R^P represents the processing resources, R^C the communication resources and R^M the memory resources. These three subsets will be respectively detailed in Sections 3.4.4, 3.4.5 and 3.4.6.

3.4.2 Sequencer Node s^{CFG}

The sequencer node s^{CFG} controls the system configurations (See Definition 3.3.2). It is in charge of the modification of the resources configuration between different applications or between partial configurations required to realize one application. We define s^{CFG} as

$$s^{CFG} = (Cfg^{fun}, Cfg^{param}) \tag{3.3}$$

where Cfg^{fun} is a set of designer-defined functions that express the configuration cost of each hardware resource according to its configuration mechanism. Cfg^{param} is the set of configuration parameters of the hardware resources. These parameters will be fixed by our mapping algorithms.

Observation 3.4.1 (Unique sequencer node)**.** In a hardware model, we assume that there will be only one sequencer node, and its descriptor will cover all the possible configuration functions and parameters of all the hardware resources. There is at least one tuple $(Cfg_i^{fun}, Cfg_i^{param})$ for all $r_i \in R$:

The sequencer node is connected to all the hardware resources through a F-hyperedge [1]. We can see in Figure 3.4 an example of the connection of one s^{cfg} to two hardware resources. Notice the F-hyperedge, which has dotted lines.

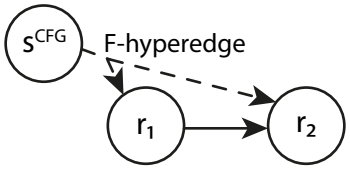


Figure 3.4 – Graphical representation of the connection of a s^{CFG}

[1] A Forward hyperedge or F-hyperedge is an hyperedge $E = (T(E), H(E))$ with $\mid T(E) \mid = 1$ [132].

As the general complexity of an SPS-CGRA is beyond two hardware resources, we introduce a different graphical representation of the F-hyperedge, which allows better readability. Figure 3.5 shows this graphical representation. We use a simple closed shape in a two-dimensional plane to represent the interconnections. All the nodes inside the closed shape in blue are connected to the node s^{CFG} through the F-hyperedge. Notice the arrow direction of the node s^{CFG}, which represents that the connection goes from the s^{CFG} to all the nodes inside the closed shape in blue. This graphical representation can be used also to represent an B-hyperedge[2], an example will be given in Section 3.5.2.

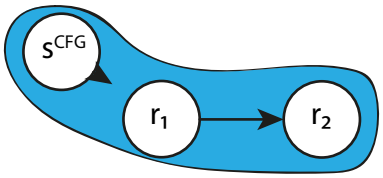


Figure 3.5 – Graphical representation of the connection of a s^{CFG} through a solid plane

As we can see in Figure 3.4 and Figure 3.5, the node s^{CFG} has an input degree of zero. This means that there is no input connection for this node. On the other hand, the output degree is equal to the rest of the hardware resources.

Observation 3.4.2 (Degree of sequencer node)**.** The sequencer node degree is given by:

$$deg^{-}(s^{CFG}) = 0 \wedge deg^{+}(s^{CFG}) =\mid S \mid -\ 1 \tag{3.4}$$

3.4.3 Hardware Resources R

The set R represents the hardware resources (see Section 3.4.1): subset R^P (P stands for process) dedicated to transform, subset R^M (M stands for memory) to store and subset R^C (C stands for communication) to communicate data. We represent each hardware resource as a node $r \in R$, where the input degree represents the input data connections, and the output degree represents the output data connections. Figure 3.6 shows an example of a single resource.

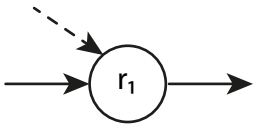


Figure 3.6 – Example of a single resource

In Figure 3.6, we can notice two input edges, one represented with a solid line and another with a dotted line. The solid line edge represents an input data connection, and the dotted line represents the connection with the configuration control node made through the F-hyperedge presented earlier. In order to alleviate the hypergraph, as explained earlier, this hyperedge is removed and replaced by a closed shape.

Definition 3.4.1 (Input degree of a resource)**.** The input degree of a hardware resource only considers the input data connections. The connection with the configuration control node is implicit.

[2] A Backward hyperedge or B-hyperedge is an hyperedge $E = (T(E), H(E))$ with $\mid H(E) \mid = 1$ [132].

Definition 3.4.2 (Output degree of a resource)**.** The output degree of a resource represents the output data connections of the resource. If the output degree is higher than one, it means that the output data will be broadcasted to all the successors.

For example, the hardware resource r_1 of Figure 3.6 has an output degree of one.

Now consider the resource r_1 of Figure 3.7. This hardware resource has a connection to the hardware resources r_2 and r_3. Consequently, the output degree of this resource is equal to two. The output data of r_1 will be broadcasted to the other resources r_2 and r_3. Figure 3.8 depicts the same structure as Figure 3.7 but is drawn without the connection to the configuration control node in order to lighten the graph.

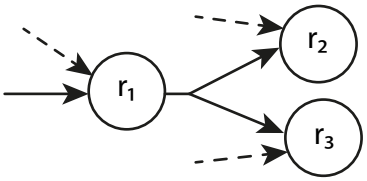


Figure 3.7 – Example of a single resource broadcasting its output

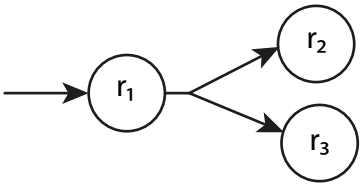


Figure 3.8 – Example of single resource broadcasting its output without the connection to the configuration control node

3.4.4 Processing Resources Subset R^P

The subset R^P represents resources that apply a given transformation of the input data. We define node $r_i^P \in R^P$ as

$$r_i^P = (\mathcal{T}_i, \Pi_i, \mathcal{L}_i, Cfg_i) \tag{3.5}$$

where $\mathcal{T}_i$ is the set of transformations that r_i^P can perform, and Π_i is the set of allowed parameters of each transformation.

Recall that an SPS-CGRA allows two different levels of heterogeneity. On one level, each processing element may be able to perform different types of tasks (addition, multiply, fast Fourier transform, hash algorithm). On the second level, the heterogeneity resides on the different physical implementations that a set of processing resources that performs the same type of task may have. This second level of heterogeneity directly impacts the computing cost of the system. A naive implementation will be more computationally costly than an optimized implementation. A set of parameters for a processing resource can result in a reduced computational cost compared to another set of parameters. Our model is able to deal with this feature using the latency descriptor $\mathcal{L}_i$.

The descriptor $\mathcal{L}_i$ describes the latency functions of a hardware resource. We define latency of a hardware resource as the time interval elapsed between the arrival of data at the input and the update of the output with a value corresponding to the transformation of that particular data. In our context, this time includes input latency and computing latency. Stream-based processing resources require a certain time to fill the input pipeline and start output data. We

define the internal pipeline filling time as the input latency. Computing latency is the time consumed to output data after the input pipeline is full. This, in fact, involves two functions, one for the input latency and another for the computing latency. Our approach solves this with the possibility that $\mathcal{L}_i$ includes two different functions.

$$\mathcal{L}_i = (\mathcal{L}_i^{IN}, \mathcal{L}_i^{CL}) \tag{3.6}$$

The descriptor $\mathcal{L}_i^{IN}$ represents the input latency and $\mathcal{L}_i^{CL}$ the computing latency of the resource i. Both are function of $\mathcal{T}_i$ and Π_i. Finally,

$$\mathcal{L}_i = (\mathcal{L}_i^{IN}(\mathcal{T}_i, \Pi_i), \mathcal{L}_i^{CL}(\mathcal{T}_i, \Pi_i)) \tag{3.7}$$

An example of the modeling process of the latency features of a real-life SPS-CGRA is given in Section 3.5.5.

As a result of the differences between the type of processing resources and physical implementations, the configuration cost could not be the same for all the resources. The fourth and last descriptor $Cfg_i \in Cfg^{fun}$ (see Section 3.4.2), defining r_i^P, is used to store the configuration cost function of r_i^P. Cfg_i is a designer-defined function that assigns the configuration cost value of r_i^P as:

$$Cfg_i(\mathcal{T}_i, \Pi_i) \tag{3.8}$$

Observation 3.4.3 (Complementary operations)**.** We assume that each r_i^P is equipped with the operations *copy* and *disable* allowing to manage correctly unused processing resources.

For some applications, after the configuration step of the whole SPS-CGRA, some resources may remain unused. The mapping algorithm will automatically assign one of the complementary operations to the unused resource. In the case of *copy* operation, the resource will be used as a bypass (their inputs are directly connected to the outputs without any transformation). In the case of *disable*, the resource will be completely disabled.

Consider the processing resource of Fig. 3.9. This resource has two inputs and one output. To implement the *copy* operation, the user needs to specify which input should be copy to the output. This information should be detail in the configuration parameters of the resource.

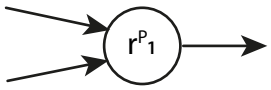


Figure 3.9 – An r^P with two inputs and one output

3.4.5 Communication Resources Subset R^C

The subset R^C represents the resources dedicated to the data transfer, read/write operations to memories and interfaces, copy, and data-path control. Recall that (Section 3.4.1), we distinguish four types of communication resources: multiplexers R^{MUX}, input/output interfaces $R^{interface}$, data reading and writing R^{RD}, R^{WR}.

$$R^C = R^{MUX} \cup R^{INTERFACE} \cup R^{RD} \cup R^{WR} \tag{3.9}$$

We describe each subset of R^C in the following sections.

R^{MUX}

A node $r_i^{MUX} \in R^{MUX}$ provides a set of inputs and outputs that performs a copy operation from a selected input to the selected output. A r_i^{MUX} can describe a multiplexer/bus/crossbar and is model as a four-element tuple

$$r_i^{MUX} = (I_i^{port}, O_i^{port}, \mathcal{L}_i, Cfg_i) \tag{3.10}$$

The descriptors I_i^{port} and O_i^{port}, represent the set of input and output ports of r_i^{MUX}. $\mathcal{L}_i$ is the latency of r_i^{MUX} and $Cfg_i \in Cfg^{fun}$ represents the configuration cost.

R^{WR} **and** R^{RD}

An $r_i^{WR} \in R^{WR}$ and an $r_i^{RD} \in R^{RD}$ are resources able to perform a *read/write* operation from or to a memory resource. We define r_i^{WR} and r_i^{RD} with a three-element tuple

$$r^{WR}, r^{RD} = (\mathcal{A}_i, \mathcal{L}_i, Cfg_i) \tag{3.11}$$

The descriptor $\mathcal{A}_i$ defines the address space to access. $\mathcal{L}_i$ models the latency of the *write/read* operation and $Cfg_i \in Cfg^{fun}$ represents the possible configuration cost.

Each processing resource can be connected directly to one or more communication resources from the subset R^{RD} and R^{WR}.

One particularly of the descriptor $\mathcal{A}_i$, is that it allows one to define the exact address space available for a hardware resource. As there may be some restrictions on the memory address space, we can use this descriptor to associate each hardware resource with the correct address space. This feature helps us to model systems where the re-injection (recomputation) of data between datapaths is available. Re-injection of data occurs when the processed output of one datapath is the input data of a second datapath. Consider a communication resource r_i^{WR} as the last element of a datapath, and it writes data to a memory module. On the other hand, assume that a communication resource r_j^{RD} is the first element of a datapath, and it reads data from the same memory module as r_i^{WR}. If the descriptors $\mathcal{A}_i$ and $\mathcal{A}_j$ are the same, or either $\mathcal{A}_i$ range covers the range of $\mathcal{A}_j$ or vice versa, the system allows re-injection of data.

Observation 3.4.4 ($\mathcal{A}_i$)**.** $\mathcal{A}_i$ allow us to define if re-injection of data is valid, if and only if an r^{RD} is able to access the address space of a r^{WR}.

An example of the use of the subsets R^{RD} and R^{WR} is given in Section 3.5.2

$R^{INTERFACE}$

The external sources and consumers of the data are nodes $r_i^{sensor} \in R^{INTERFACE}$ and $r_i^{actuator} \in R^{INTERFACE}$. We describe an r_i^{sensor} with the tuple

$$r_i^{sensor} = (\mathcal{T}_i, \Pi_i, \mathcal{L}_i) \tag{3.12}$$

where Π_i are the allowed parameters and $\mathcal{L}_i$ is the latency of producing one data sample.

Observation 3.4.5 ($R^{INTERFACE}$ descriptor $\mathcal{T}_i$)**.** To complete, each r_i^{sensor} and $r_i^{actuator}$ has an implicit descriptor operation $\mathcal{T}_i = interface$

We consider that an r_i^{sensor} has an internal r^{WR}, which allows it to transfer data directly to an r^P or an r^{MUX}, or write data to an r^M. In Figure 3.10, we show this concept graphically. In this work we use the labels *sensor* (r_i^{sensor}) and $SNSR$ (r_i^{SNSR}) interchangeably.

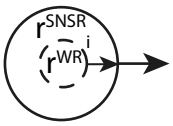


Figure 3.10 – Example of a r_i^{sensor}

Likewise, we describe a $r_i^{actuator}$ as

$$r_i^{actuator} = (\mathcal{T}_i, \Pi_i, \mathcal{L}_i) \tag{3.13}$$

where Π_i are the allowed parameters and $\mathcal{L}_i$ is the latency of consuming one data sample. We consider that an $r_i^{actuator}$ has an internal r^{RD}, which allows it to transfer data directly from an r^P or an r^{MUX}, or read data from an r^M. In Figure 3.11, we show this concept graphically. In this work we use the labels *actuator* ($r_i^{actuator}$) and $ACTR$ (r_i^{ACTR}) interchangeably.

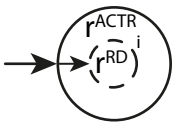


Figure 3.11 – Example of a $r_i^{actuator}$

When an $r^{actuator}$ or an r^{sensor} is directly connected to a r^M, the latency of the resource should include the latency of reading or writing one data sample.

To describe the behaviour of the subset $R^{interface}$, two simple examples are given in Section 3.5.1.

3.4.6 Memory Resources Subset R^M

The subset R^M represents the hardware memory resources (RAM modules, sequential memory modules). We describe each $r_i^M \in R^M$ with a tuple

$$r_i^M = (\mathcal{A}_i, C_i^{RD}, C_i^{WR}) \tag{3.14}$$

where $\mathcal{A}_i$ represents the addressing space of r_i^M. C_i^{RD} is the number of read channels available, and C_i^{WR} the number of write channels. A channel is a specific range of addressable memory space which is accessible to some hardware resource. We assume the following conditions:

1. The sum of C^{RD} of all $r^M \in R^M$ of the hardware should be equal or more than the number of all r^{RD} plus the number of all $r^{actuator}$.

$$\sum_{i=1}^{n} C_i^{RD} \geq \mid R^{RD} \mid + \mid \{r^{actuator}\} \mid \tag{3.15}$$

   where $n = \mid R^M \mid$.

2. The sum of C^{WR} of all $r^M \in R^M$ of the hardware should be equal or more than the number of all r^{WR} plus the number of all r^{sensor}.

$$\sum_{i=1}^{n} C_i^{WR} \geq \mid R^{WR} \mid + \mid \{r^{sensor}\} \mid \tag{3.16}$$

where $n =\mid R^M \mid$.

3. An r_i^M is always paired with at least one resource node from $R^{RD} \cup R^{WR} \cup R^{interface}$.

Notice that the memory resources do not have the expression of the latency. This one is always integrated into the associated r^{RD} and r^{WR} nodes.

We consider the following interconnection patterns for an r_i^M.

- A linear sub-graph with a sensor r_k^{sensor} as a predecessor of a r_i^M resource, and a memory read r_j^{RD} as a successor of this r_i^M resource (Figure 3.12).

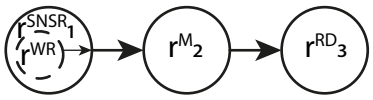


Figure 3.12 – Example of a node r_2^M with r_1^{SNSR} as predecessor and r_3^{RD} as successor.

- A linear sub-graph with a r_k^{WR} as a predecessor of a r_i^M resource, and an actuator $r_j^{actuator}$ as a successor of this r_i^M resource (Figure 3.13).

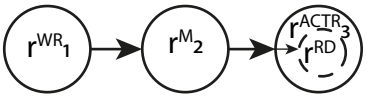


Figure 3.13 – Example of a node r_2^M with r_1^{WR} as predecessor and r_3^{ACTR} as successor.

- A linear sub-graph with a r_k^{WR} as a predecessor of a r_i^M resource, and a r_j^{RD} as successor of this r_i^M resource (Figure 3.14).

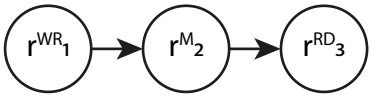


Figure 3.14 – Example of a node r_2^M with r_1^{WR} as predecessor and r_3^{RD} as successor.

Notice the need for an r^{RD} and an r^{WR} in each pattern. Either in an implicitly form, as in the r^{sensor} and the $r^{actuator}$, or in an explicit way with the r^{RD} and r^{WR} itself.

Our model is a directed hypergraph, and one property that we comply with is acyclicity. An issue with the acyclicity may appear when we have to model a system where the datapath uses the same memory resource as a source and sink of the process. During the modelization, to remove this cycle of the hardware, we split the memory resource according to the number of channels or the number of datapaths. Consider the hardware model showed in Figure 3.15. For didactic reasons, we only show the memory resource, and we represent a set of hardware

resources with the cloud shape. Notice that the data go from and to the memory resource r_i^M. This creates a forbidden cycle in the model.

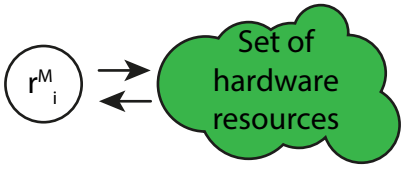


Figure 3.15 – Representation of a cyclic hardware model

To solve this, we split the memory resource into two memory resource nodes but sharing the same name. We can see in Figure 3.16 the resulting graph. Notice that now the memory resource r_i^M is split into two, a $r_i^{M_{SRC}}$ (source node of the datapath) at one end and $r_i^{M_{SINK}}$ (sink node of the datapath) at the other. This transformation eliminates the cyclicity. In order to show that these two nodes model the same memory resource, we add the dotted lines to connect them. The hardware resources that the dotted lines enclosures are considered one datapath. We include these dotted lines only if both ends of the datapath are the same.

Figure 3.16 – Memory resource modelled by two connect memory nodes

Now, let's see another example. In Figure 3.17 we show a different hardware model. In this case, the hardware graph has two cycles.

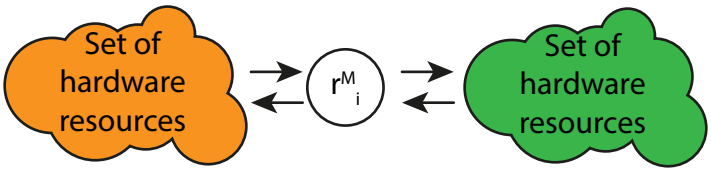


Figure 3.17 – Representation of a cyclic hardware model

Since we can not have a cycle in our hardware model, we again split the memory resource, but this time we end up with two independent datapaths. As we can see in 3.18, the two independent datapaths are enclosure with dotted lines.

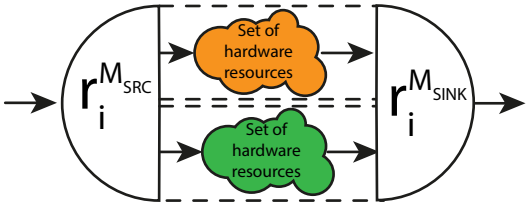


Figure 3.18 – Memory resource modelled by two connect memory nodes with two independent datapaths

Several examples of the memory resource modeling process are given in Section 3.5.3.

3.4.7 Fork-Join Special Nodes

In some cases, the sensor (i.e., a camera) produces a burst of n samples of data, where n can be from 2 to the total amount of expected input samples. Assume that the actuator consumes a burst of m samples of data, where m can be from 2 to the total amount of expected output processed samples. Given that the hardware platform is stream-based, the input burst will be stored entirely and then processed one by one. Next, the processed samples will be again stored, and after the completion of the processing of all input samples, the processed data will be sent to the actuator. This condition creates three different regions. It is no longer a pure stream-based system where as soon as one sample of data is ready, it moves forward to the next resource. In this case, we need to wait until a specific number of samples are ready to move to the following resource.

To model this different type of processing, we introduce two special nodes that allow to represent this behavior.

A **join** node allows to model the case where we need to wait n number of samples to move these samples to the following resource. We represent a join node graphically with the difference of an edge crossing line with a 1 for the input edge and the n samples of the burst in the output edge. We can see in Figure 3.19 the graphical representation of a join node.

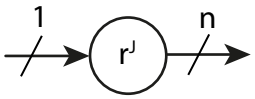


Figure 3.19 – Special join node

A **fork** node allows to represents the case where we read a burst of n data samples, and these samples are processed one by one. We represent a fork node graphically with the difference of an edge crossing line with the n samples of the burst and a 1 in the output edge. We can see in Figure 3.20 the graphical representation of a fork node.

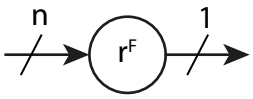


Figure 3.20 – Special fork node

An example of the use of the fork-join nodes is detailed in Section 3.5.4.

3.5 Examples

In this section, we provide several examples of the modeling process of hardware architectures. The examples make use of virtual hardware architectures. Additionally, we provide the modeling of real-world hardware. We focus on the Morphological Co-processor Unit (MCPU) [12]. This co-processor will help us illustrate and validate the benefits of our hardware model.

3.5.1 Communication Resources $R^{INTERFACE}$

To describe the behavior of the subset $R^{INTERFACE}$, consider the stream-based hardware in Figure 3.21. The hardware takes data from a camera as input and displays the processed data on an actuator (e.g., display).

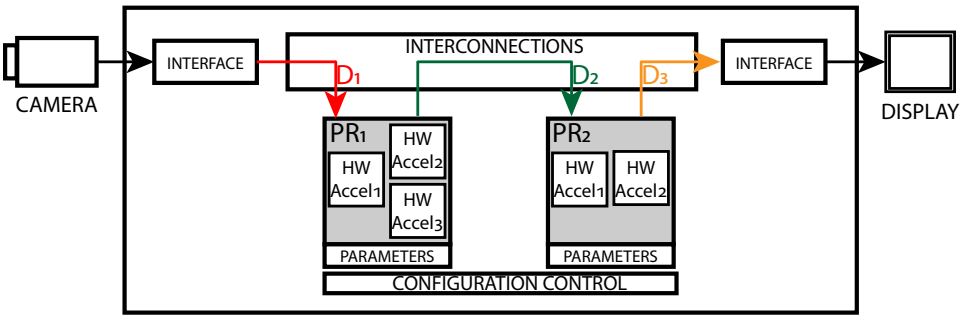
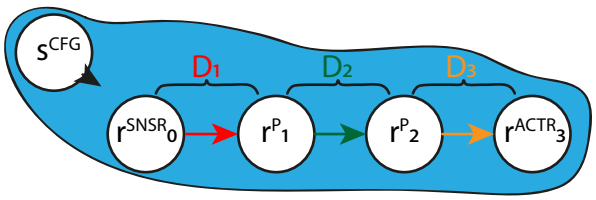


Figure 3.21 – SPS-CGRA example 1

Figure 3.22 shows the corresponding hardware model of this basic example. The input data goes directly to the processing resource r_1^P. We represent the camera with r_0^{SNSR}. The reading of the camera memory is implicitly represented by r_0^{SNSR} itself. The output data produced by r_2^P goes directly to the actuator r_3^{ACTR}. This last node implicitly writes in the actuator (display) memory. Note the presence of s^{CFG} connected by an F-hyperedge (closed blue shape) to all the nodes to configure them.



Hardware resources	Hardware model nodes
Configuration control	s^{CFG}
Interface/ camera	r^{SNSR}_0
PR_1	r^{P}_1
PR_2	r^{P}_2
Interface/ display	r^{ACTR}_3

Figure 3.22 – Hardware model of SPS-CGRA example 1

In the second example, we study the modeling of a hardware architecture able to process images using pipelined processing of the pixels. It is depicted in Figure 3.23. The camera produces an image (pixel by pixel) which is read and stored in the memory module of the system. After the first input data (first pixel) is available in the memory module, the processing can begin by reading the data, processing it, and storing back the result in the same memory. After the first processed pixel is available in the memory module, it can be displayed by the actuator.

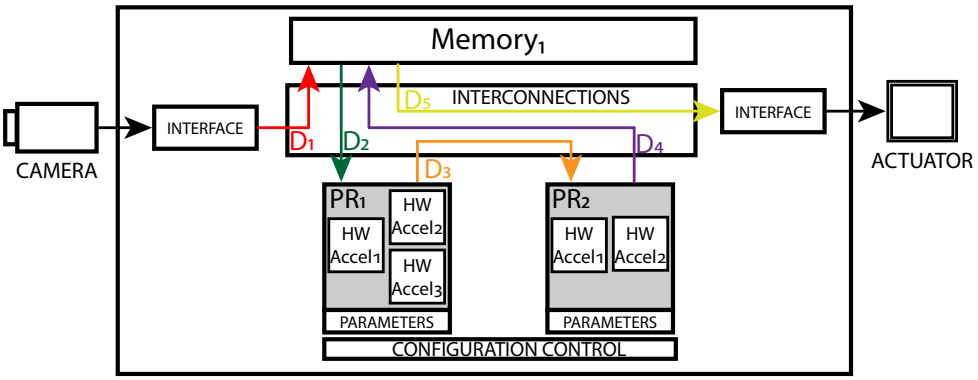
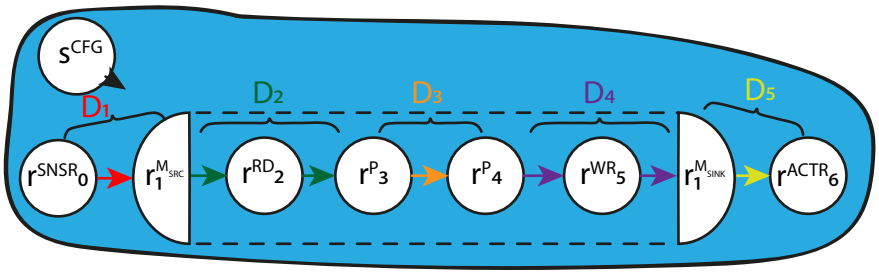


Figure 3.23 – SPS-CGRA example 2

Figure 3.24 shows the corresponding hardware model. The camera is represented using r_0^{SNSR}. This resource represents the input data (pixel) generation and the writing to the memory module r_1^M. The datapath $D1$, showed in red, is modeled by the edge (r_0^{SNSR} , $r_1^{M_{SRC}}$). The datapath $D2$, showed in green, is modeled by the edges ($r_1^{M_{SRC}}$, r_2^{RD}) and (r_2^{RD}, r_3^P). The datapath $D3$, showed in orange, is modeled by the edge (r_3^P, r_4^P). The datapath $D4$, showed in purple, is modeled by the edges (r_4^P, r_5^{WR}) and (r_5^{WR}, $r_1^{M_{SINK}}$). The last datapath $D5$ is modeled by the edge ($r_1^{M_{SINK}}$, r_6^{ACTR}). Symmetrically, the actuator is represented using r_6^{ACTR}. This resource represents the data reading (processed pixel) from the memory module and the actuator's consumption data. Notice the split of the memory module and the dotted lines that enclosure the remaining hardware resources, as explained in Section 3.4.6.



Hardware resources	Hardware model nodes
Configuration control	s^{CFG}
Interface/ camera	r^{SNSR}_0
Memory1	$r_1^{M_{SRC}}$/ $r_1^{M_{SINK}}$
PR1	r^{P}_3
PR2	r^{P}_4
Interface/ display	r^{ACTR}_6

Figure 3.24 – Hardware model of the SPS-CGRA example 2

3.5.2 Communication Resources R^{WR} and R^{RD}

Consider the hardware depicted in Figure 3.25 and its associate hardware model in Figure 3.26.

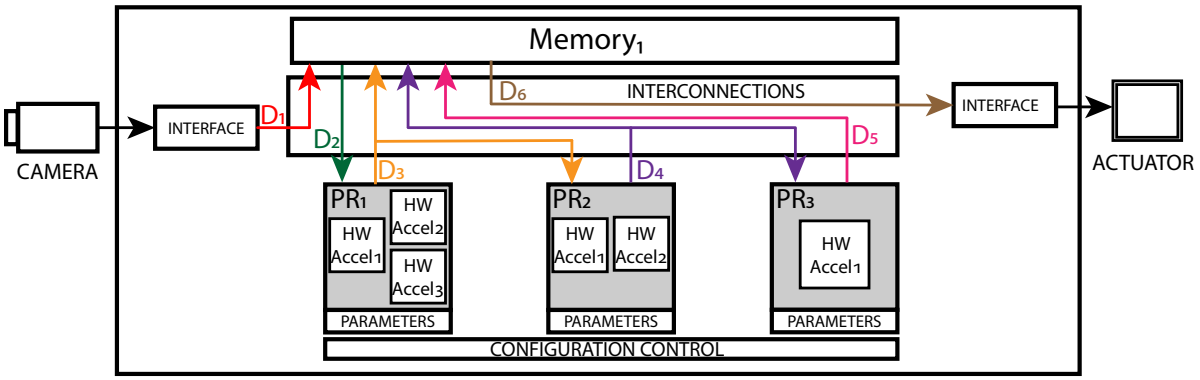


Figure 3.25 – SPS-CGRA example 3

Notice that the processing resource PR_1, modeled by r_3^P, reads and writes data to the memory module. Recall that each processing resource can be connected directly to one or more communication resources (See Section 3.4.5). Thanks to r_2^{RD} and an r_7^{WR}, we can model this behavior. In this example r_3^P, r_4^P and r_5^P can write independently their data in r_1^M, thanks to r_7^{WR}, r_8^{WR}, r_6^{WR} respectively. This correspond to the D_3, D_4 and D_5 datapaths available on this architecture (Figure 3.25).

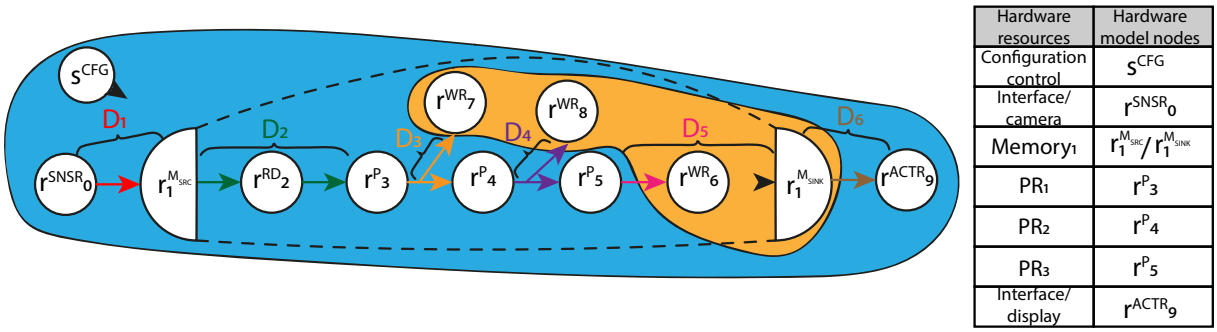


Figure 3.26 – Hardware model of the SPS-CGRA example 3

3.5.3 Memory Resources R^M

In this section, we present four examples that detail the modeling process of the memory resources. The first two examples illustrate the use of a memory resource as the source and sink of a datapath [12, 11]. The third example illustrates the usage of two memory resources within the same architecture. The fourth example shows a memory resource as a frame buffer within a processing pipeline [133].

A Memory Resource as a Source and Sink of a Datapath

Consider the simple hardware example of Figure 3.27. It consists of one camera, a memory block, a processing element (PR_1), and a single actuator. In this architecture, the processing element reads the input data from the memory block and stores its result in the same memory. The actuator then reads this memory.

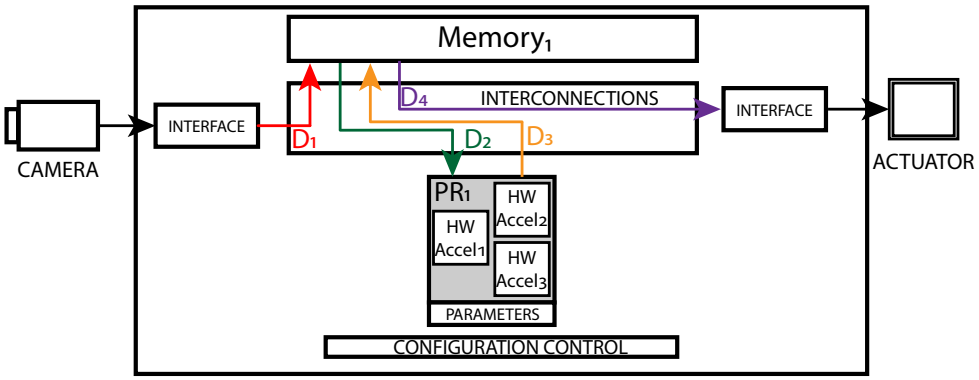
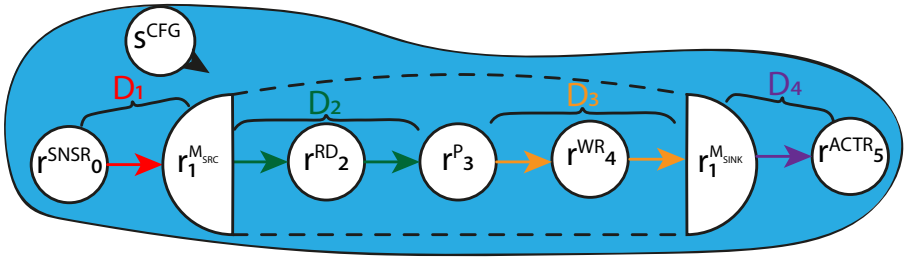


Figure 3.27 – SPS-CGRA example 4

Figure 3.28 shows the hardware model of SPS-CGRA example 4. Notice the graphical representation of r_1^M. Recall that the hardware model does not allow cycles (See Section 3.4.6). We split r_1^M into $r_1^{M_{SRC}}$ and $r_1^{M_{SINK}}$. Furthermore, notice the dotted lines in Figure 3.28. The dotted lines show that the datapath highlighted is connected in both ends with the same r_1^M.



Hardware resources	Hardware model nodes
Configuration control	s^{CFG}
Interface/ camera	$r^{SNSR}{}_0$
Memory1	$r_1^{M_{SRC}}$/ $r_1^{M_{SINK}}$
PR1	$r^{P}{}_3$
Interface/ display	$r^{ACTR}{}_5$

Figure 3.28 – Hardware model of example 4

Consider now the SPS-CGRA example 5, depicted in Figure 3.29. Like the preceding example, the input data from the camera goes directly to the memory module of the system. The stored data is read and processed, in this example by two processing resources connected consecutively. Finally, the data is stored again in the memory module.

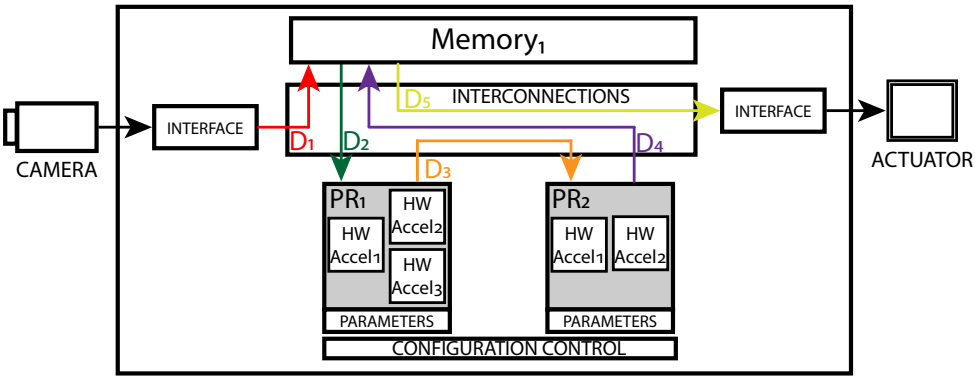
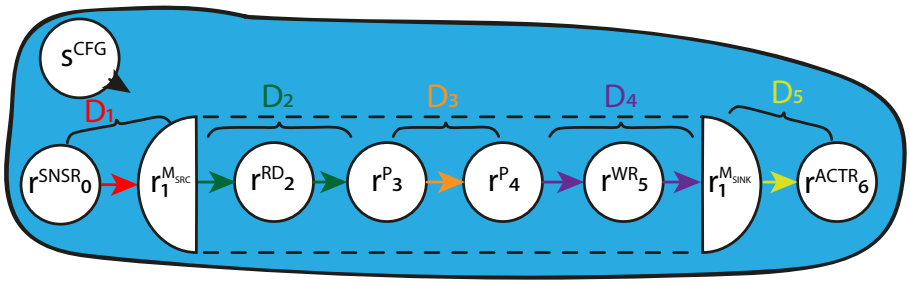


Figure 3.29 – SPS-CGRA example 5

Figure 3.30 shows the G_{HW} of hardware example 5. We again split r_1^M. One r^{RD} (r_2^{RD}) and one r^{WR} (r_5^{WR}) represent the internal read and write of data to the memory module. r^{SNSR} represents the external write to the memory module, and r^{ACTR} the external read of the same memory module. Notice the dotted lines enclosing the datapath, in this case, of two processing

resources. This means that both ends of the datapath belong to the same memory module, r_1^M.



Hardware resources	Hardware model nodes
Configuration control	s^{CFG}
Interface/ camera	$r^{SNSR}{}_0$
Memory$_1$	$r_1^{M_{SRC}}$/ $r_1^{M_{SINK}}$
PR$_1$	$r^{P}{}_3$
PR$_2$	$r^{P}{}_4$
Interface/ display	$r^{ACTR}{}_6$

Figure 3.30 – Hardware model of example 5

Two Memory Resources in the Same Architecture

The SPS-CGRA example depicted in Figure 3.31 has two memory modules. In this architecture, the input data goes directly to one memory module ($Memory_1$) of the system, then the stored data is read in parallel by two processing elements (PR_1 and PR_3). These two processing elements are the first elements of two independent datapaths. One datapath stores its result in the same memory module. The second datapath stores its result in a different memory module ($Memory_2$). Then a pair of actuators reads the processed data from each memory module.

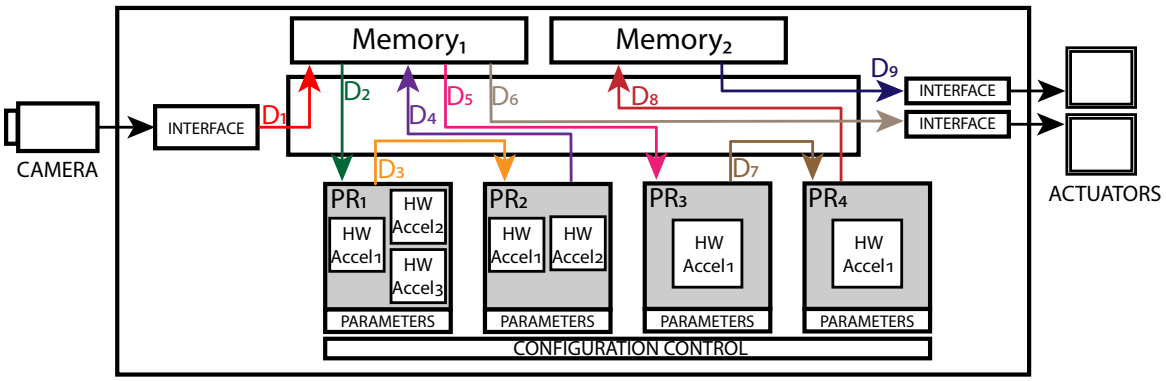
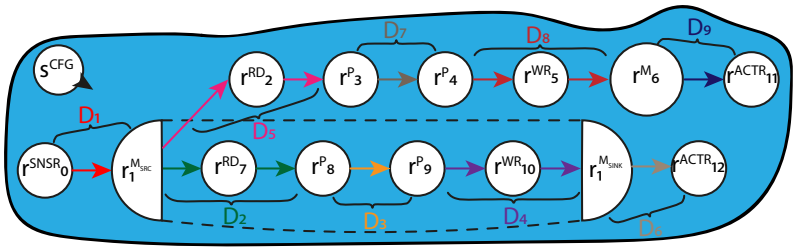


Figure 3.31 – SPS-CGRA example 6

In this case, we introduce two r^M. r_1^M models $Memory_1$. We split r_1^M to avoid creating a cycle within the hardware model. Notice the dotted lines only highlight the datapath, in which r_1^M is the source and the sink. r_2^M models the second memory module ($Memory_2$). This memory module only stores the processed data of r_4^P (PR_4), and we do not need to split it. Finally, one r^{SNSR} and two r^{ACTR} represent the external write and read of the memory modules. Figure 3.32 shows the hardware model of SPS-CGRA example 6.



Hardware resources	Hardware model nodes	Hardware resources	Hardware model nodes
Configuration control	s^{CFG}	PR_2	r^{P}_{9}
Interface/ camera	r^{SNSR}_{0}	PR_3	r^{P}_{3}
$Memory_1$	$r^{M_{SRC}}_{1}$/$r^{M_{SINK}}_{1}$	PR_4	r^{P}_{4}
$Memory_2$	r^{M}_{6}	Interface/ display	r^{ACTR}_{11}
PR_1	r^{P}_{8}	Interface/ display	r^{ACTR}_{12}

Figure 3.32 – Hardware model of example 6

A Memory Resource in the Middle of a Datapath

The SPS-CGRA example of Figure 3.33 makes use of an internal memory module, which serves to synchronize the processed data of PR_2 and PR_4.

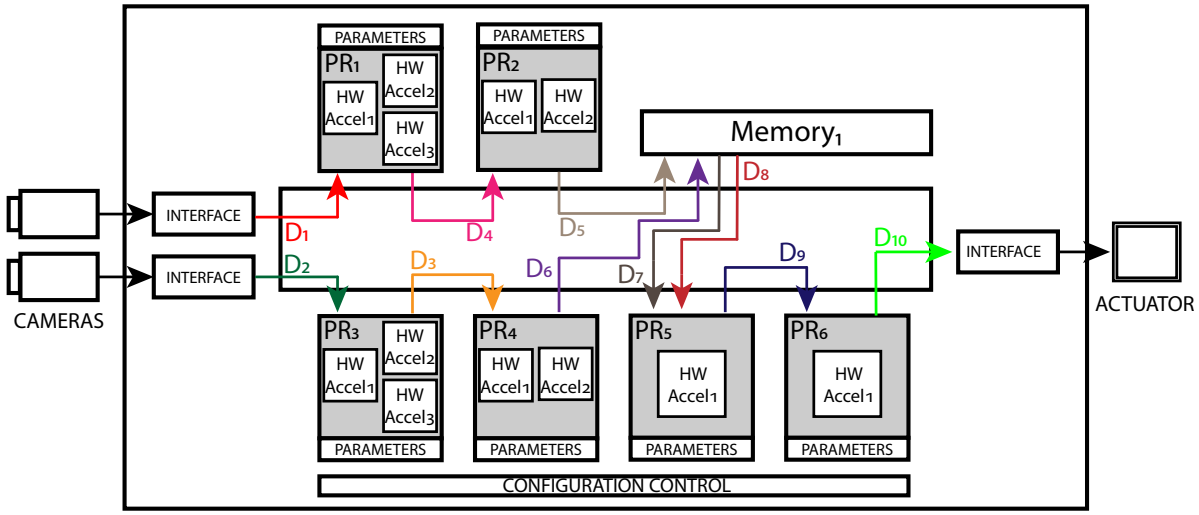
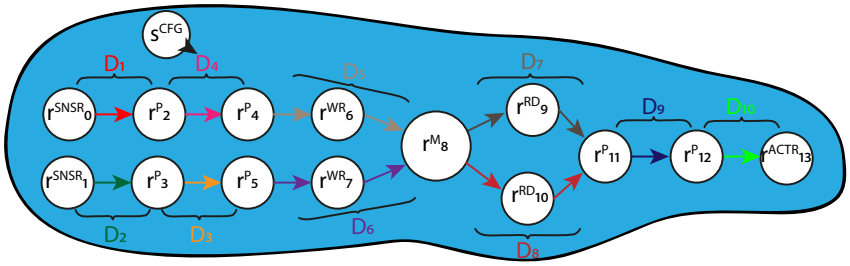


Figure 3.33 – SPS-CGRA example 7

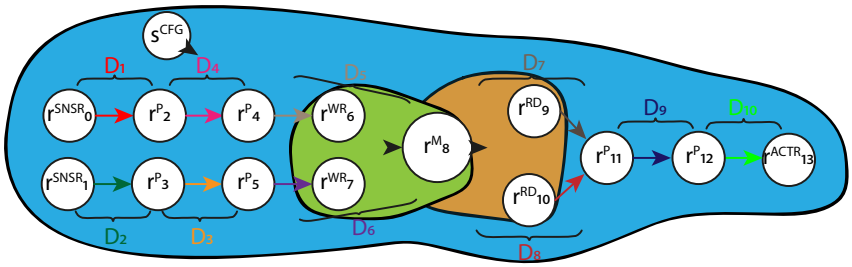
Figure 3.34 shows the hardware model. r^{M}_{8}, which models $Memory_1$, does not require to be split. Notice the interconnections between $r^{RD}_{9,10}$ and $r^{WR}_{6,7}$ and r^{M}_{8}. We make use of simple edges for graphical representation.



Hardware resources	Hardware model nodes	Hardware resources	Hardware model nodes
Configuration control	s^{CFG}	PR_3	r^{P}_{4}
Interface/ camera	r^{SNSR}_{0}	PR_4	r^{P}_{5}
Interface/ camera	r^{SNSR}_{1}	PR_5	r^{P}_{11}
$Memory_1$	r^{M}_{8}	PR_6	r^{P}_{12}
PR_1	r^{P}_{2}	Interface/ display	r^{ACTR}_{13}
PR_2	r^{P}_{3}		

Figure 3.34 – hardware model of example 7

Figure 3.35 shows a different graphical representation, where an hyperedge is used (See Section 3.4.2). The graphical representation of Figure 3.34 and 3.35 are equivalents.



Hardware resources	Hardware model nodes	Hardware resources	Hardware model nodes
Configuration control	s^{CFG}	PR_3	r^{P}_4
Interface/ camera	r^{SNSR}_0	PR_4	r^{P}_5
Interface/ camera	r^{SNSR}_1	PR_5	r^{P}_{11}
$Memory_1$	r^{M}_8	PR_6	r^{P}_{12}
PR_1	r^{P}_2	Interface/ display	r^{ACTR}_{13}
PR_2	r^{P}_3		

Figure 3.35 – An equivalent hardware model of SPS-CGRA example 7

3.5.4 Fork-Join Special Nodes

In the architecture depicted in Figure 3.36, the camera connected to $Memory_1$ produces a burst of n pixels. These n pixels will be processed by the processing elements PR_1 and PR_2, pixel by pixel. The processed pixels will be stored in the same $Memory_1$, and they will be read in sets of m pixels by the display.

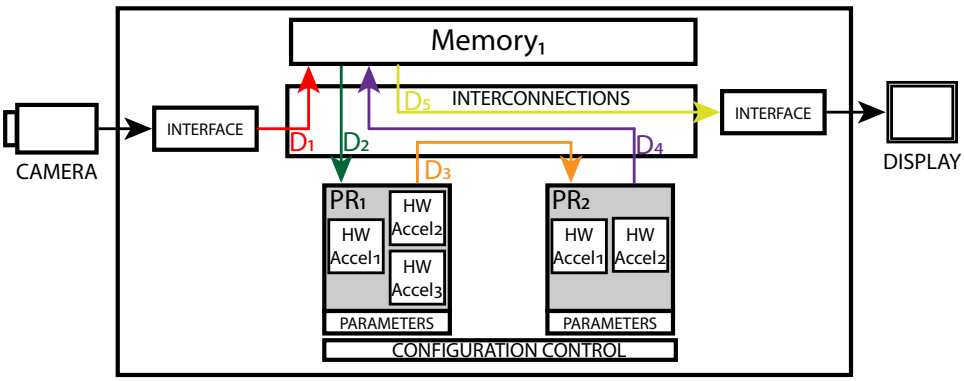
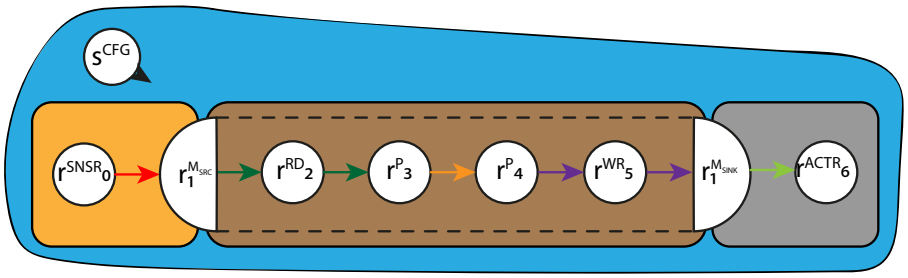


Figure 3.36 – SPS-CGRA example 8

We can see in Figure 3.37 the hardware model of the system described before. In the Figure, we notice the three regions of the system. The orange section represents the production of pixels made by the camera (r_0^{SNSR}), in this region, we will wait until the arrival of n samples to move them to the memory resource (r_1^{M}). The brown section represents the processing and communication elements. In this region, r_2^{rd} will read pixel by pixel and transfer them to the processing elements in a stream-like way. At the end of this section, all the processed pixels will be stored again in the same memory resource. We need to wait for m samples to be stored to move to the following region. After this procedure, the display (r_6^{ACTR}) will start consuming the bursts of m data samples.



Hardware resources	Hardware model nodes
Configuration control	s^{CFG}
Interface/ camera	r^{SNSR}_0
$Memory_1$	$r^{M_{SRC}}_1 / r^{M_{SINK}}_1$
PR_1	r^{P}_3
PR_2	r^{P}_4
Interface/ display	r^{ACTR}_6

Figure 3.37 – Division of the regions of processing in example 8

To describe this behavior accurately, we use the join and fork nodes introduced in Section 3.4.7. The join node r_7^J models the burst of n pixels of the camera. Similarly, the join node

r_9^J models the write of m samples to $r_1^{M_{SINK}}$, which is the number of pixels that the camera r_6^{ACTR} requires. On the other hand, the fork node r_8^F models the read of pixel by pixel of the communication resource r_2^{RD}, and r_{10}^F models the same behavior for r_6^{ACTR}. Figure 3.38 depicts the hardware model that includes the fork/join nodes.

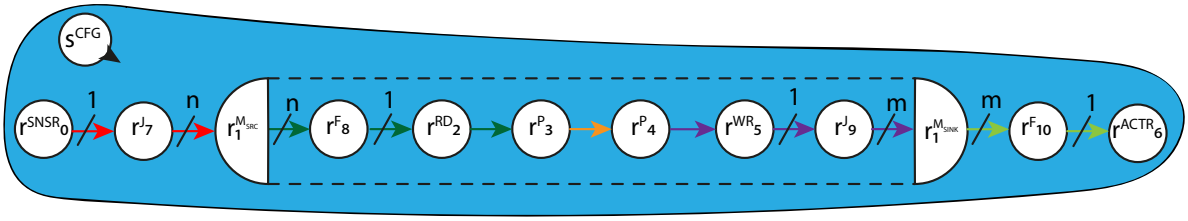


Figure 3.38 – Hardware model of example 8 with the fork and join special nodes

3.5.5 The Morphological Co-Processor Unit

The MCPU was developed by Bartovsky et al. [12]. It is integrated as a coprocessor in an FPGA-based platform. The MCPU assembles several efficient units to support a large collection of morphological operations. Figure 3.39 layout its architecture. As we can notice, the MCPU follows the principles of an SPS-CGRA with its pipeline-based array of processing resources (Large SE pipeline and Geodesic Pipeline), and it has a loose coupling(i.e., no resource sharing with its host processor). Notice the direct connection of the processing pipelines to the memory, and the configuration control is depicted as configuration registers connected to all the hardware resources.

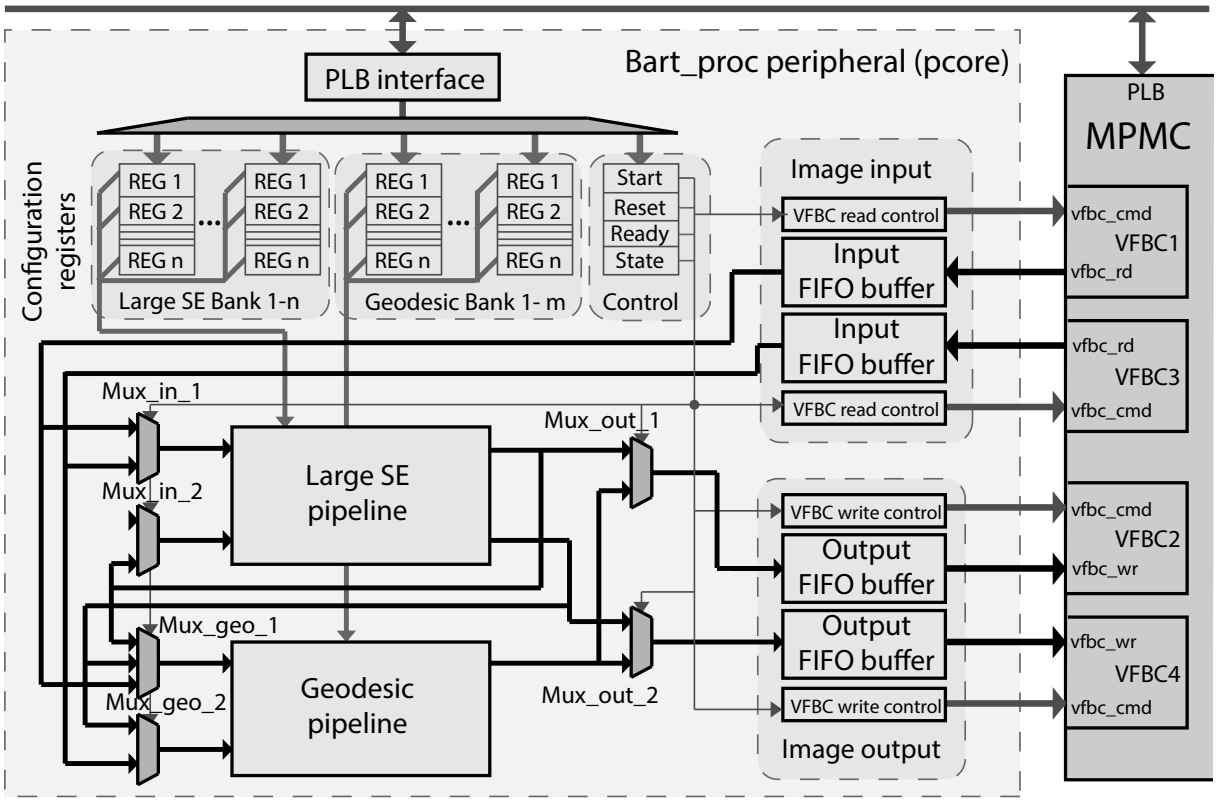


Figure 3.39 – Architecture of the Morphological Co-processor Unit [12]

The MCPU features two pipelines, a Large SE pipeline (detailed in Figure 3.40) and a Geodesic Pipeline. Both with similar characteristics. They consist of an array of processing

resources, ALUs, and multiplexers. All resources can be configurated through the configuration registers, and as a result, the type of operations and the datapaths can be reconfigured. The granularity is mixed, from fine-grained (multiplexer) up to coarse-grained (dilation/erosion). This is a particularity of the SPS-CGRAs, the broad range of granularity that it can contain.

In this example, we focus on the Large SE pipeline detailed in Figure 3.40. Notice that two pipeline stages are shown and need to be modeled. Other stage quantities and the Geodesic pipeline can be modeled; however, for didactic purposes, we only focus on two pipeline stages. From Figure 3.39, the configuration registers may be modeled as a sequencer node s^{cfg} where the configuration cost and their associated functions are described. The "Image input" and "Image output" modules may be model as a communication memory resources: respectively nodes $r^{RD}_{3,4}$ and $r^{WR}_{18,19}$. The resulting modelization is given in Figure 3.41. The corresponding external producers and consumers are model as $R^{INTERFACE}$, either as sensors ($r^{SNSR}_{0,1}$) or actuators ($r^{ACTR}_{21,22}$).

Each Large SE Pipeline basic stage consists of: *1)* two processing resources called Large SE erosion/dilation, *2)* one processing resource able to perform ALU-like operations, *3)* two multiplexers, *4)* two processing elements able to compute the infimum, maximum and accumulator of the image pixels. The modeling of these elements is straightforward. We use r^P for the processing elements and r^{MUX} for the multiplexers.

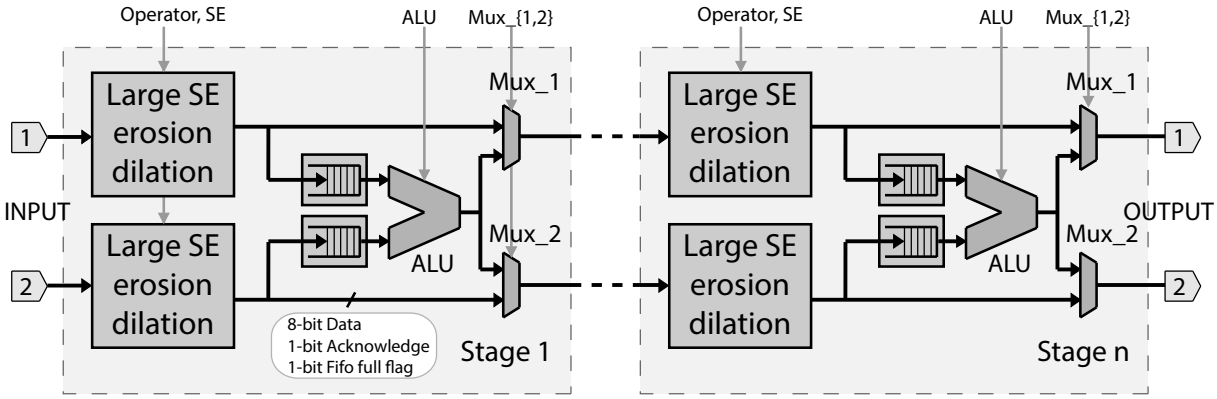


Figure 3.40 – Large SE pipeline architecture

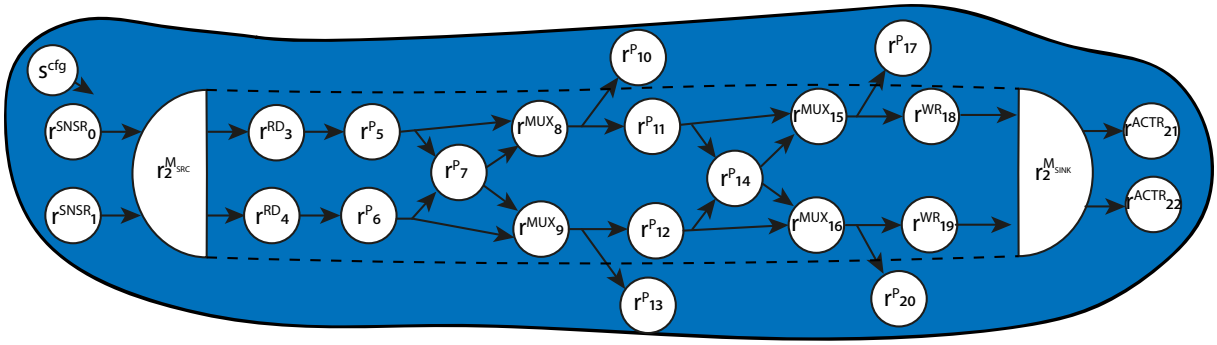


Figure 3.41 – Hardware model of the MCPU

Latency Features

Each resource $r_j \in R^P \cup R^C$ has a specification for the computing latency specific for the hardware. For the resources $r_j \in R^P$ we also define the input latency. Table 3.2 summarizes these specifications, where KS means the size of the input structuring element, l is equal to the size of structuring element (a line) in the orientated erosion/dilation, *width* is the width of the input image, and β is the input angle of the orientated erosion/dilation. Finally, N/A means not applicable.

Table 3.2 – Latency features of the resources

Resource	$\mathcal{T}_i$	$\mathcal{L}_i$	
		$\mathcal{L}_i^{IN}$	$\mathcal{L}_i^{CL}$
$r_{0,1}^{SNSR}$	interface	N/A	1
$r_{21,22}^{ACTR}$	interface	N/A	1
$r_{3,4}^{RD}$	read	N/A	1
$r_{18,19}^{RW}$	write	N/A	1
$r_{8,9,15,16}^{MUX}$	multiplexer	N/A	1
$r_{5,6,11,12}^{P}$	erosion/ dilation	((KS-1)/2)*width+ ((KS-1)/2)	3
	orientated erosion/ dilation	$l * sin(\beta) * width$	3
$r_{10,13,17,20}^{P}$	acummulator, maximum, minimum	1	1
$r_{7,14}^{P}$	bit-wise operations, addition, substraction	1	1

3.6 Conclusions

In this chapter, we introduced a generic hardware model for SPS-CGRA. It allows modeling the memories, processing resources, and all kinds of datapaths connecting them. Additionally, this hardware model provides the means to describe the latency features of all the hardware resources accurately. Also, for the processing resources, the model can describe two kinds of latencies: input latency and computing latency. These two latencies are specifically considered in a stream-based processing system. However, the concept of different sources of latency may be extended to other types of processing.

The presented hardware model includes the means to model the configuration control resources. This modeling focuses on the configuration cost through the description of its functions and parameters. This feature will provide the means to accurately compute the upper bound of the computing cost of implementing an application onto an SPS-CGRA (See Section 4.4).

We describe, through several virtual hardware, the modeling process of various structures and systems. We divide the examples into the type of hardware resources and, we explain in detail how we can model each one of these types. Furthermore, we presented a real-life example and shown how the hardware model can abstract the important characteristics.

Finally, the hardware model can be used for most of the SPS-CGRAs, regardless of its application field, due to its generic descriptors and the consideration of several types of hardware resources.

Chapter 4

Implementation Model

4.1 Introduction

Generally, the sole output product of a mapping and scheduling algorithm, in the literature, is the configuration context or executable (a hardware implementation description) [134, 135, 136]. Few works, such as the Algorithm Architecture Matching (AAM) methodology [35, 85], include an implementation model into their framework. An implementation model abstracts the result of the mapping and scheduling of an application onto the hardware system (Figure 4.1). It is the result of graph transformations between the application graph and the hardware graph. The mapping and scheduling produce an implementation graph based on a given implementation model from a couple of hardware and application graphs.

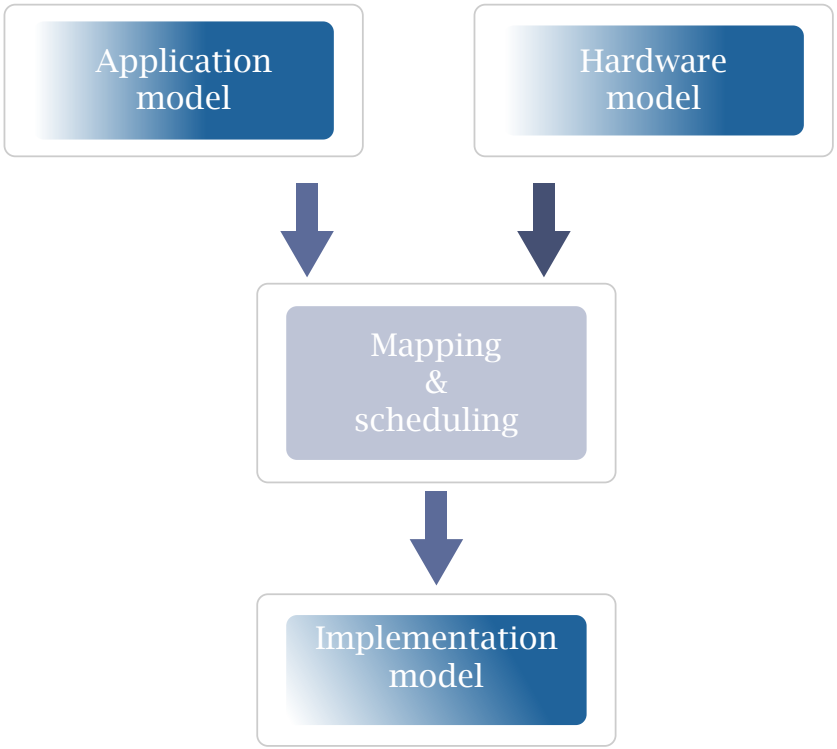


Figure 4.1 – Y-chart of a mapping and scheduling framework

An implementation model has the information of the configuration context and, in a few cases, information about the implementation's performance. Thus another step may be needed to generate the missing information about the implementation's performance. Furthermore, the features of an implementation model usually are only suitable to its target platform. Hence an

implementation model may be applicable only for its target platform and not usable for others.

To solve these issues, we propose to use a graph-based implementation model. It is built with at least one instance of the hardware model but with fixed parameters (descriptors). To represent the scheduling of the tasks over time periods, we replicate these instances and only change the values of the descriptors. These last mentioned are fixed during the matching process of the mapping and scheduling algorithm. Thus we take advantage of the descriptors of the nodes from the implementation graph to include the required information for the configuration context. Additionally, we include and process latency information required for performance analysis. With this model, we can produce the configuration context and, in parallel, produce the results of the performance evaluation. Furthermore, the entire framework may apply to other platforms.

We organize the remaining part of the chapter as follows. In Section 4.3, we introduce the formalization of the model. Section 4.4 describes the performance evaluation methodology. Finally, Section 4.5 makes a summary of the chapter.

4.2 Proposed Implementation Model

The implementation model provides information of both temporal and spatial allocation of the application tasks onto the system hardware resources. Additionally, the model integrates latency information needed for performance analysis. The implementation model allows describing the result of the matching process during the mapping and scheduling algorithm. The implementation model is derived from the hardware model.

Definition 4.2.1 (Implementation graph)**.** G_{MAP} is a directed weighted hypergraph. Several different instances G_{MAP_i} (called *time slot*) may construct the implementation graph when the available resources are not sufficient.

Definition 4.2.2 (Time slot)**.** A time slot is a subset of hardware resources configured for a computed duration to perform a subset of the application's tasks. Thus, a time slot allows reusing temporally the resources configured through different parameters.

The hardware reconfiguration delimits the duration of a time slot. Each time slot starts with hardware reconfiguration and ends before the following hardware reconfiguration. Recall that the reconfiguration is represented by node s^{CFG}. Within a time slot, an instance of the hardware executes a subset of tasks or the whole application. A time slot fulfills two purposes, a resource occupation representation and a representation of resource latencies. Firstly, a time slot details the task allocation on both temporal and spatial dimensions. By parsing the implementation graph, we obtain information on the mapping and scheduling of the tasks. Secondly, our implementation model includes latency information on both nodes and edges. Using this latency information, we can execute a performance evaluation over the implementation graph. In Section 4.4 we describe a performance evaluation method.

4.3 Formal Implementation Model

We define our implementation model as a directed weighted hypergraph $G_{MAP}(S', K')$, where S' represents the nodes, and K' represents the weighted hyperedges. It consists of a set of subgraphs (G_{MAP_i}) that may be or not disjoint, and each subgraph is an instance of the hardware graph G_{HW}. In this setting, each subgraph represents a time slot.

The set of weighted hyperedges $K' = K'_1, K'_2, \ldots, K'_m$, where $m = \mid K' \mid$, represents the interconnections of the programmed hardware resources.

Definition 4.3.1 (Weight of a hyperedge)**.** We define the input latency of the head node [1] as the weight of each hyperedge.

[1] Given a directed edge $E(x, y)$, y is called the tail node, and x is called the head node. Furthermore, node y is a direct successor of x, and x is the direct predecessor of y.

The set of nodes S' represents the hardware resources with fixed parameters. These parameters are fixed during the mapping and scheduling. The set S' has similar structure of the set S (See Section 3.4.1), but with the addition of the subsets $R^{sn} \in R'$ (See Section 4.3.5) and S^{cfg} (See Section 4.3.1) . Thus, S' consist of

$$S' = S^{cfg} \cup R' \tag{4.1}$$

where

$$R' = R^p \cup R^c \cup R^m \cup R^{sn} \tag{4.2}$$

We systematically use lowercase superscripts for the implementation model resources to differentiate them from the hardware model resources. We describe each subset in the following sections.

4.3.1 Configuration Control Nodes S^{cfg}

The subset S^{cfg} represents the configuration control nodes of each time slot i. Each sequencer node s^{CFG} (See Section 3.4.2), from the hardware model, is in charge of the generation of this subset. Recall that s^{CFG} integrates the configuration cost functions and their parameters. During the mapping and scheduling process, this information will be processed to obtain the configuration cost of each time slot i ($T\mathrm{C}_i$). Therefore, there will be the same number of $s_i^{cfg} \in S^{cfg}$ as the number of time slots that integrates G_{MAP}. Finally, each s_i^{cfg} is described as a value of the time (in clock cycles) needed to configure all the resources, and this value is a function of the parameters needed to configured.

4.3.2 Processing Resources R^p

The subset R^p represents the processing resources with fixed parameters. Similarly, as in Section 3.4.4, we describe each $r_i^p \in R^p$.

$$r_i^p = (\tau_i, \pi_i, l_i, cfg_i) \tag{4.3}$$

where $\tau_i \in \mathcal{T}_i$, represents the selected type of task. The descriptor $\pi_i \in \Pi_i$ represents the set of parameters that correspond to τ_i. The computing latency of the processing resource is l_i and corresponds to the type of task and its parameters. The configuration cost of the resource is cfg_i and corresponds to the type of task and its parameters.

4.3.3 Communication Resources R^c

The subset R^c represents the communication resources with fixed parameters. This subset has the same structure of R^C (See Section 3.4.5), which is

$$R^c = R^{mux} \cup R^{rd} \cup R^{wr} \cup R^{interface} \tag{4.4}$$

The subsets of R^c have the same descriptors as their conterparts of R^C. Accordingly, each $r_i^{mux} \in R^{mux}$ is represented as

$$r^{mux} = (i_i^{port}, o_i^{port}, l_i, cfg_i) \tag{4.5}$$

where i_i^{port} is the input port selected from I_i^{port}, o_i^{port} is the output port selected from O_i^{port}, l_i is the latency value and cfg_i is the configuration cost of r_i^{mux}.

For the subset R^{rd} and R^{wr}, we describe each $r_i^{wr} \in R^{wr}$ and $r_i^{rd} \in R^{rd}$ as

$$r_i^{wr}, r_i^{rd} = (a_i, l_i, cfg_i) \tag{4.6}$$

where a_i is the assigned address space, l_i is the cost of the read/write operation and cfg_i is the configuration cost.

For the subset $R^{interface}$, we describe each $r^{interface} \in R^{interface}$ as

$$r_i^{interface} = (\pi_i, l_i) \tag{4.7}$$

where $\pi_i \in \Pi_i$ are the assigned parameters and l_i the latency value.

4.3.4 Memory Resources R^m

The subset R^m refers to the memory resources of the system. Similarly, as in Section 3.4.6, each $r^m \in R^m$ is described as

$$r_i^m = (a_i, c_i^{rd}, c_i^{wr}) \tag{4.8}$$

Where a_i is the address space available, c_i^{rd} is the number of read channels and c_i^{wr} the number of write channels.

4.3.5 Data Dependency Resources R^{sn}

R^{sn} is a subset of virtual nodes that allows representing the data dependency between time slots and between datapaths. Data dependency between time slots occurs when the number of tasks to allocate exceeds the number of processing resources available. Data dependency between datapaths occurs when the number of tasks to allocate exceeds the number of processing resources available in a single datapath. However, other independent datapaths are available.

The addition of these resources provides the correct information to generate the configuration context and evaluate the performance of the implementation of the application onto the hardware. These resources do not have any descriptor, and their latency is considered as zero (bypass). If the represented data dependency is between time slots, we add two $r_i^{sn} \in R^{sn}$. The first one (r_i^{sn}) will be in time slot i. Its predecessor will be the $r^{m_{sink}}$ of time slot i and its successor will be r_{i+1}^{sn}. The second, r_{i+1}^{sn}, will be in time slot $i+1$ and its successor will be the $r^{m_{source}}$ of time slot $i+1$. The connection between these two data dependency resources will be through an inter-slot hyperedge $K'_j \in K'$.

Definition 4.3.2 (Inter-slot hyperedge)**.** An inter-slot hyperedge $K'_j \in K'$ is an hyperedge that connects two data dependency resources r_i^{sn} and r_{i+1}^{sn}, which are in two consecutive time slots.

If the represented data dependency is between datapaths, we add one r_i^{sn}. Its predecessor will be the sink node of the datapath that will process the input data first. Its successor will be the source node of the datapath that will process the input data secondly. The addition of the data dependency resources is subject to the capabilities of the SPS-CGRA. Examples of the addition of the data dependency resources and the use of the inter-slot hyperedge are given in the next section.

4.3.6 Examples

In this section, we present simple experimental graphs for both application and hardware to detail the main characteristics of the implementation graph. Additionally, we describe the utility of the data dependency resources.

Implementation Graph

Consider the implementation graph shown in Figure 4.2. For didactic purposes, let's assume that this implementation graph results from mapping the application graph onto the hardware graph, both shown in Figure 4.2. The hardware graph and the implementation graph are very similar. The only difference is that each resource of the implementation graph is configured (r_3^P is configured to execute t_1, r_4^P is configured to execute t_2, etc.)

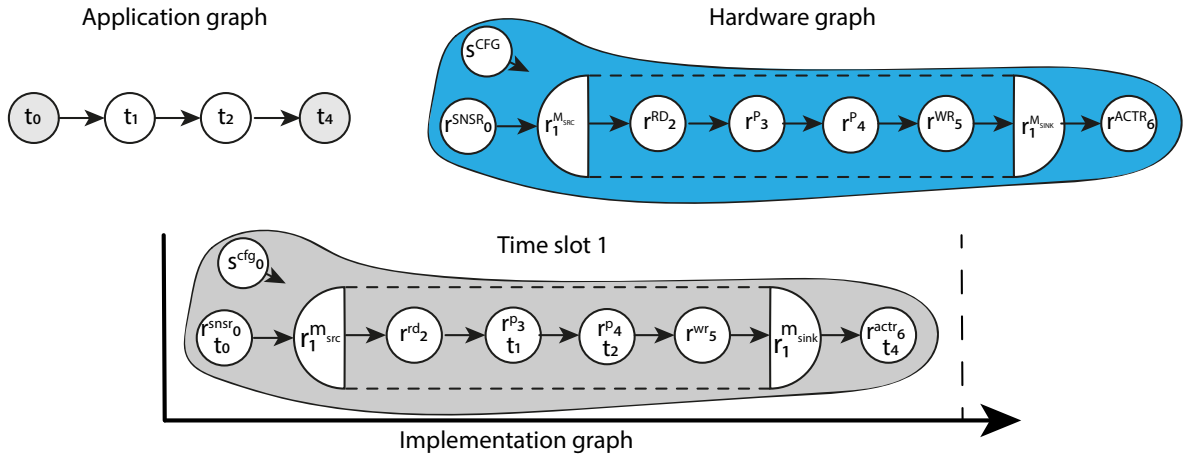


Figure 4.2 – Example 1 of a generic implementation graph

Now, let study the more complex example depicted in Figure 4.3. In this example, the tasks of the application graph exceed the number of processing resources of the hardware graph. Hence, there is a need for 3 processing resources (to allocate t_1, t_2 and t_3) but only two (r_3^P and r_4^P) are available. Thus, we need to execute the whole application through two steps, using reconfiguration of the hardware between these two steps. During the first step we may execute t_1 on r_2^p and t_2 on r_3^p. Then, we store the t_2 data into the memory ($r_1^{m_{sink}}$). Next, in a second step we can reconfigure the hardware so that we read t_2 data from memory ($r_6^{m_{src}}$) and reuse r_8^p to execute t_3. In that case r_9^p is configured as a bypass (will copy its input to its output) thanks to the **copy** operation (See Section 3.4.4).

Formally, each hardware configuration is called a time slot (as introduced in Section 4.3). So, depending on the number of time slots required to execute an application, the implementation graph may be made of several instances of the hardware graph. Notice that in time slot 1, the actuator node (r^{actr}) is missing, and in time slot 2, the sensor node (r^{snsr}) is also missing. This is due to two reasons. The first is because we can not consume the data in time slot 1, as the entire processing of the application is not yet finished. The second reason is because we do not need to produce data in time slot 2 since it stores the partially processed data in time slot 1 and rereads it in time slot 2. To complete, notice the inter-slot hyperedge, in green, (presented in Section 4.3.5) that connects r_{12}^{sn} to r_{13}^{sn}. As stated before, the processing of the whole application is not finished in time slot 1. Therefore the processed data of this time slot will be reuse in time slot 2. By adding both the inter-slot hyperedge and the data dependency resources, we model this behavior. Examples of the use of the data dependency resources are given later.

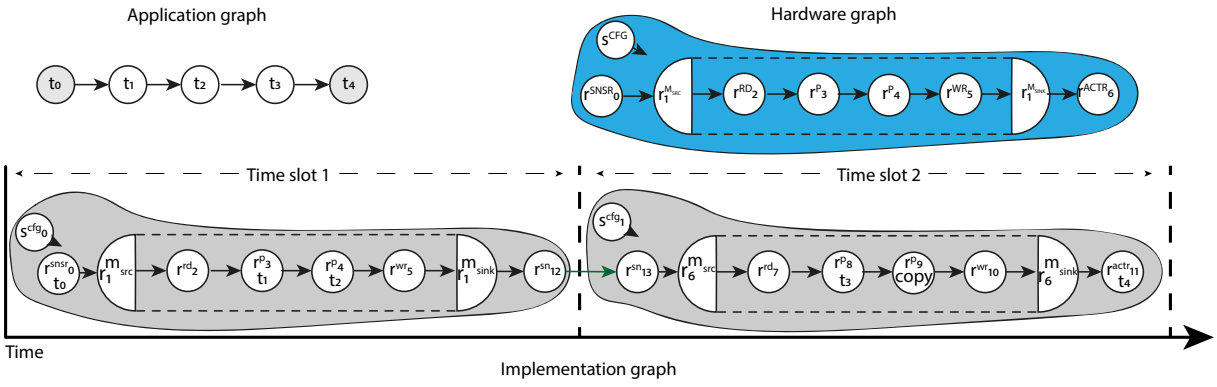


Figure 4.3 – Example 2 of a generic implementation graph

Data Dependency Resources R^{sn}

Between Time Slots

The number of tasks of the application graph exceeds the number of processing resources of the hardware graph. Thus, the whole application will be executed in two time slots. In Figure 4.4 we show the implementation graph. As previously stated, in time slot 1, we execute t_1 on r_2^p and t_2 on r_3^p and then, in time slot 2 we execute the remaining part of the application (t_3 on r_8^p). This behavior creates a data dependency between time slots. The t_2 data is stored during time slot 1, and it will be reuse in time slot 2. To represent this behavior, we add the resources r_{12}^{sn} for time slot 1 and r_{13}^{sn} for time slot 2. Moreover, we connect both through an inter-slot hyperedge to connect both time slots.

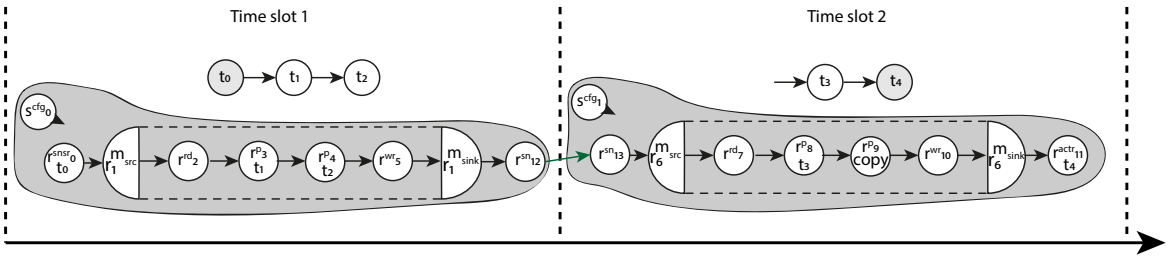


Figure 4.4 – Inclusion of the data dependency resources between time slots

Between Datapaths

The hardware graph of Figure 4.5 consists of two independent datapaths of processing resources. Each datapath has two processing resources. Consider the application graph shown in the same Figure 4.5. The total number of tasks exceeds the number of available processing resources of a single hardware datapath. Let's assume that the hardware can perform recomputation of data between datapaths in the same time slot. To execute the whole application, we may execute t_1 on r_5^p and t_2 on r_7^p. Then we store the t_2 data into the memory ($r_2^{m_{sink}}$). During the same time slot we read t_2 data from $r_2^{m_{sink}}$, to execute t_3 on r_6^p. This behavior creates a data dependency between datapaths. To model this behavior, we add r_{13}^{sn}. The predecessor of this node will be the sink hardware resource of the first datapath (r_9^{rw}). The successor of r_{13}^{sn} will be the source hardware resource of the second datapath (r_4^{rd}). This resource will also be useful for the performance evaluation because it allows one to construct the critical path correctly.

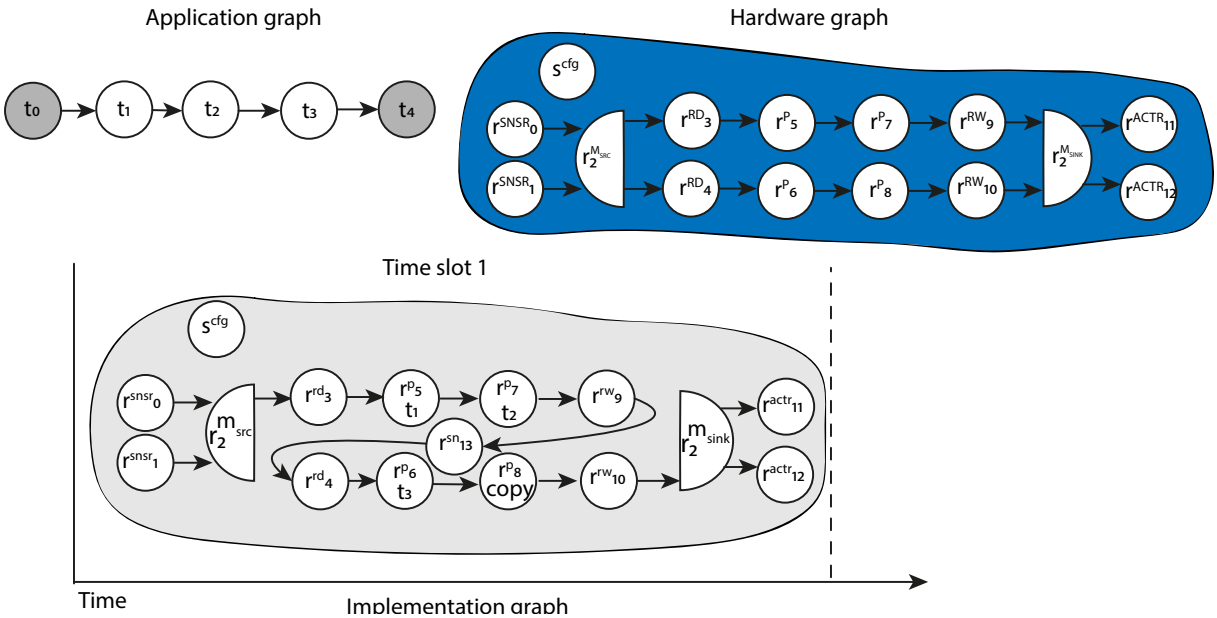


Figure 4.5 – Inclusion of the data dependency resources between data-paths

4.4 Performance Evaluation

In this section, we introduce a method to compute an upper bound estimation of the computing cost of an implementation. The estimation is based on the critical path of the evaluation graph. The evaluation graph is the resulting product of a series of transformations on the implementation graph. We present the performance evaluation in the next subsections.

We divide the rest of the section as follows. In Section 4.4.1 we present a brief description of the state of the art of timing analysis and argue the gaps in the current approaches. In Section 4.4.2 we introduce our performance evaluation methodology. Finally, in Section 4.4.3 we present some examples on the build of the evaluation graph and an experimental example that shows the use of our proposed equation.

4.4.1 State of the Art

A timing analysis refers to computing the execution time bounds or estimates [137]. It is a crucial step in the design and prototyping of real-time systems. Two main approaches exist for timing analysis, static methods and measurement-based methods.

Measurement-based methods combine static program analysis with a dynamic part, the execution time measurements [138]. This means that a task or parts of a task are executed in a given hardware or simulator. Then, the timing is measured for a given input. Static methods analyze the task itself and its possible implementations using a hardware model and compute the upper bounds of this analysis. This method is highly dependable on the model's accuracy. However, it is appropriate for fast prototyping and development.

Several static-based methods have been proposed to compute the execution time of an application. Melani et al. [89] presented a method to calculate the response time of a task. The response time or make-span is the longest possible time any instance of the task requires to complete its execution [93]. The authors consider the sum of the WCET of all critical path nodes and the interference time, which is the time that each node consumes, from its ready state until its executed. This method is applied to a DAG model that considers conditional parallel tasks and focuses in homogeneous multiprocessors platforms. Risat et al. [93] proposed an improvement to the work of Melani et al. to decrease the pessimistic results of their method. However, their approach also considers the same factors.

Frid et al. [139] proposed to compute the computing cost of an application using the elementary cost of each basic operation. The authors propose a method to identify a set of possible elementary operations of a platform and compute their costs individually. Next, divide the applications into those elementary operations and obtain the computing cost by adding the previously calculated elementary operations cost.

Grandpierre and Sorel [35] presented a performance prediction based on their implementation model. Based on the methodology AAM [85], the implementation model is a graph that describes real-time behavior, and it may be used to verify real-time constraints. The method is based on the critical path and takes into account the operation and the communication cost. Additionally, the method provides a diagram of the memory allocation.

Zadrija and Sruk [94] presented a method to compute the cost of mapping a task t_i onto a processing resource r_j, which is the base of their scheduling algorithm. This approach particularly considers the different costs of a task depending on where it is implemented.

Hamann et al. [140] presented a method to compute the end-to-end latency where the implicit communication cost is taken into account. This approach helps to optimize the communication overheads and reducing event-chain latencies.

Lu et al. [110] introduced an equation to calculate the computing cost of an application implemented on NoC. They propose to use the longest path weighted Manhattan distance (LPWMD). This method uses the critical path of the application after it is implemented and considers the communication cost as the weight of the edges.

Topcuoglu et al. [91] presented a method to obtain the computation cost of a task recursively. They consider the computation cost of the task and the maximum value of the computation cost plus the communication cost on the critical path set to a sink task (upward rank).

SPS-CGRA applications are time-critical. This aspect defines the requirements of the accuracy of the performance evaluation. Several conditions need to be considered. The heterogeneity of the material realizations is an important characteristic of the SPS-CGRA, and this feature is not commonly considered on static-based timing analysis. In [94, 139] they consider this heterogeneity however only using different WCET. Another important aspect is the communication cost, which is considered in [91, 110, 35, 140]. Also, some path-based works tend to overestimate the execution time [89, 93]. And significantly, most of the current approaches neglect the configuration cost and may not be directly applied to the timing analysis of SPS-CGRA applications.

Our proposal is to compute the upper bound of the computing cost, which encompasses the configuration cost and the execution time. It covers the time from the hardware configuration up to the output of the last processed sample. It allows taking into account different material realizations, and it also considers two different latencies per resource, input and computing latency. Overall, it can give accurate cycle upper bounds of the computing cost.

4.4.2 Methodology

The performance evaluation methodology consists of two steps:

1. Evaluation graph building.
2. Estimation of the computing cost upper bound over the critical path of the evaluation graph.

The input of the methodology is the implementation graph from the mapping and scheduling algorithm. Through graph transformations of the implementation graph, we obtain the evaluation graph. Next, we use our proposed equation to compute the upper bound of the computing cost.

Evaluation Graph

An implementation graph (defined in Section 4.3) may be connected across its time slots through the data dependency resources introduced in Section 4.3.5. Otherwise, each time slot will be disjoint for each other. Regardless of the implementation graph is connected, the configuration control resources may not be considered a part of the critical path because they are source nodes and are not connected to the previous time slot. To accurately compute the critical path of the implementation graph, we perform some transformations over it and build the evaluation graph.

The transformations done over the implementation graph to build an evaluation graph are:

- Removal of the disabled nodes (See Section 3.4.4, Complementary operations).
- Removal of the R^m resources.
- If two consecutive time slots i and $i+1$ are connected through an inter-slot hyperedge (See Section 4.3.5), substitute the hyperedge with the configuration control resource of time slot $i+1$. The configuration control resource will have as a predecessor the r_i^{sn} of time slot i and as a successor the r_{i+1}^{sn} of time slot $i+1$.
- If two consecutive time slots i and $i+1$ are disjoint. We insert the configuration control resource (s^{cfg}) of time slot $i+1$ in sequence between time slots i and $i+1$. All the sink resources of the hardware instance of time slot i will be the predecessors of the configuration control resource of time slot $i+1$. All the source resources of the hardware instance of time slot $i+1$ will be the successors of the configuration control resource of time slot $i+1$.

Some examples of the building process of the evaluation graph are given in Section 4.4.3. After building the evaluation graph, we compute the estimation of the upper bound of the computing cost on the critical path.

Latency-based Performance Evaluation

Computing cost is the time elapsed from the first configuration of the hardware until the consumption of the last output sample. We compute an estimation of the upper bound of the computing cost (CC) as follows:

$$CC = \sum_{i=1}^{N} (T_{\text{IN}_i} + T_{\text{EX}_i} + T_{\text{C}_i}) \tag{4.9}$$

where N is the number of time slots of the implementation graph.

We define **overall input latency time interval** T_{IN_i} of time slot i as the number of clock cycles from the arrival of the first data sample until the start of the computing of the first result. We call T_{EX_i} **the execution duration** of time slot i, and it represents the execution time for processing the entire set of data samples after the input pipeline is full and can provide a continuous stream of data. T_{C_i} is the configuration cost of time slot i.

We propose to bound the problem to the identification of the critical path of the implementation graph and compute the computing cost on it. T_{IN_i} depends on both the computing and input latency of each resource. These latencies propagate through the processing pipeline. Also, T_{IN_i} considers the type of resource the type of the task to execute on the given resource. To obtain an accurate measurement of T_{IN_i} we use the following equation:

$$T_{\text{IN}_i} = \sum_{j=1}^{|CP_i|-1} (\mathcal{L}_j^{IN})(\omega_j) + \mathcal{L}_j^{CL} \tag{4.10}$$

Where CP_i is the set of resources that belong to the critical path of time slot i. $\mathcal{L}_j^{IN}$ is the input latency of the resource, $\mathcal{L}_j^{CL}$ is the computing latency of the resource. Finally, ω_j

is a variable allowing to express the propagation of the impact of the computing latency from predecessor to successor resources. We call this variable latency propagation parameter.

Latency Propagation Parameter

Recall that an SPS-CGRA consists of a set of heterogeneous processing resources. Thus, we expect that, within a path, several different values of latencies exist. Accordingly, the critical path will be imbalanced in terms of latency, and we need to model this phenomenon. We propose to use the variable ω that will carry the information about the worse computing latency throughout the critical path.

Let ω_j be given by:

$$\omega_j = max(\omega_{j-1}, \mathcal{L}^{CL}_{j-1}) \tag{4.11}$$

where ω_{j-1} is the ω of the predecessor and $\mathcal{L}^{CL}_{j-1}$ is the computing latency of the predecessor. At the beginning of the evaluation process, ω will be initialized as zero, and it will be updated at each resource that belongs to the critical path. The final value of ω (ω value of the last resource in the critical path) will be used as the worse computing latency of time slot i.

Consider Figure 4.6, where two linear sub-graphs of processing resources (t_1/r^p_1, t_2/r^p_2 and t_3/r^p_3) are given (notation t_i/r^p_j represents that t_i has been allocated onto r^P_j). From the values of the computing and input latencies, we can see that both paths are imbalanced. The corresponding timing diagrams are shown on the right. In the first linear sub-graph (top), the first resource (t_1/r^p_1) is the one with the largest computing latency, compared to (t_2/r^p_2 and t_3/r^p_3), therefore it will give the cadence to the rest of the resources in the pipeline. In other words, it will provide the timing for the processing of the remaining resources in the critical path. The impact of its computing latency will result in a waiting time between the output of two consecutive processed samples of t_2/r^p_2 and t_3/r^p_3. This largest computing latency and the associated delays are considered thanks to ω_1. In the second linear sub-graph (bottom), the second resource (t_2/r^p_2) is the one that has the largest computing latency, and it will only affect the processing of the next resource. The variable ω_2 will provide the means to propagate its computing latency to the following resources.

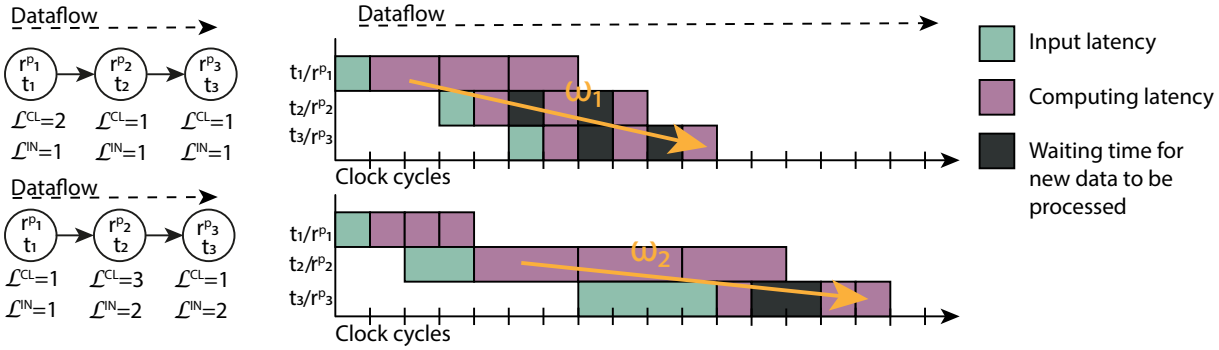


Figure 4.6 – Effects of the computing latency of a resource in the critical path

The execution duration $T_{\text{EX}i}$is computed as follows

$$T_{\text{EX}i} = (CL_i)(TS) \tag{4.12}$$

where CL_i is the worse computing latency of the critical path (final value of ω) and TS is the total amount of input samples of time slot i. Finally, recall that s^{cfg}_i carries the value of $T_{\text{C}i}$ (See Section 4.3.1).

4.4.3 Examples

Example 1 : Evaluation Graph and Memory

Let's use some examples with experimental implementation graphs to explain the building process of the evaluation graph. In Figure 4.7 a generic implementation graph with only one time slot. We notice that all the processing resource have their corresponding task.

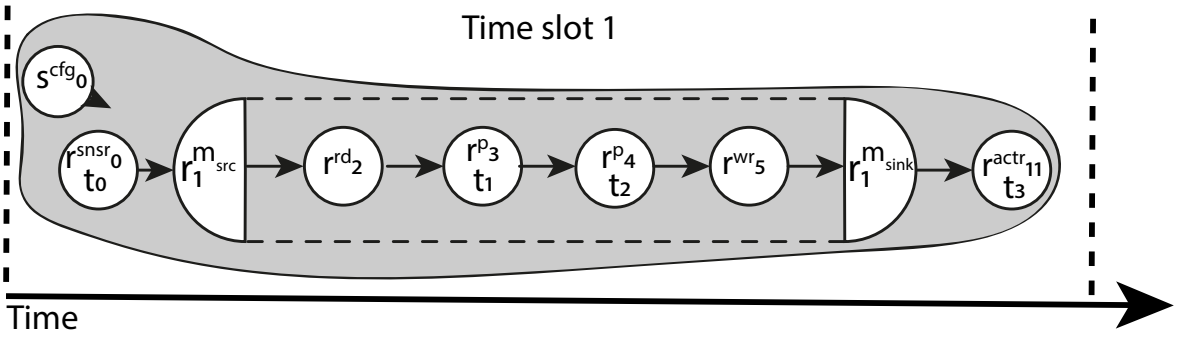


Figure 4.7 – First implementation graph example.

In this first example, the transformation is the removal of r_1^m. In Figure 4.8 we show the evaluation graph of this first example. As we can notice, with this change, we can obtain the critical path, which will also correspond to the unique simple path of the evaluation graph, and compute the upper bound of the computing cost.

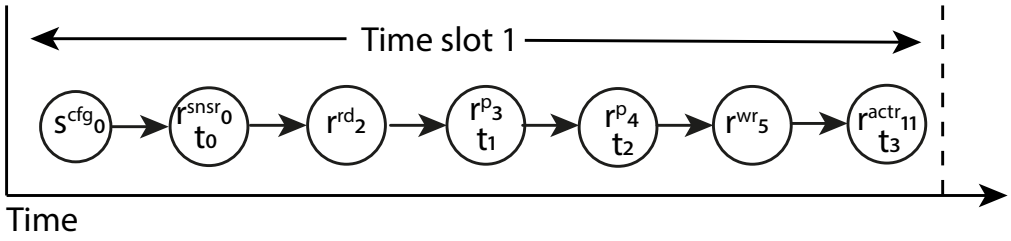


Figure 4.8 – First evaluation graph example

Example 2 : Evaluation Graph and Two Time Slots

For the second example, consider the implementation graph of Figure 4.9. This implementation graph has two time slots, and all the processing resources have a task (t_1,t_2 and t_3) or a copy ($r_9^p/copy$) operation assigned. Also, notice the inclusion of two data dependency resources (r_{12}^{sn} and r_{13}^{sn}), which are connected through an inter-slot hyperedge. This means that the partially processed data from time slot 1 needs to be reread for further processing in time slot 2.

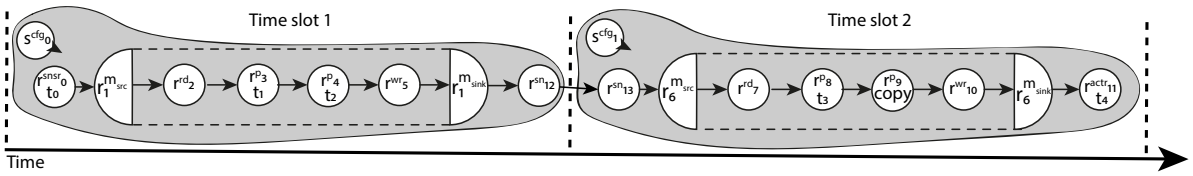


Figure 4.9 – Second implementation graph example

Figure 4.10 shows the evaluation graph of the second implementation graph example. In this example, the transformation includes the connection of both configuration control resources for the time slot 1 to r_0^{snsr}. In time slot 2, the configuration control resource s_1^{cfg} will be the successor of the resource r_{12}^{sn} and the predecessor of resource r_{13}^{sn}. We include s_1^{cfg} in this way to be able to get the correct critical path of the implementation. The last change is removing r_i^m in both time slots. The upper bound of the computing cost includes the latency of processing the entire input image. This is the main reason to perform this transformation to the implementation graph to produce the evaluation graph.

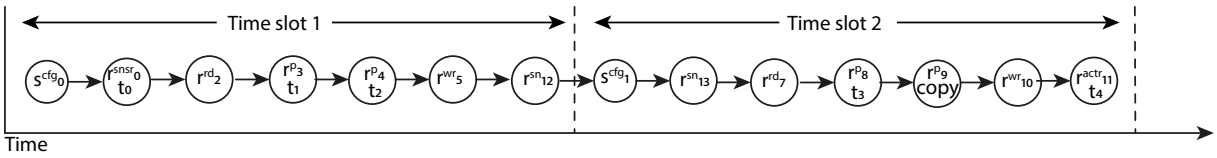


Figure 4.10 – Second evaluation graph example

Example 3 : Evaluation Graph and One Time Slot

We can see in Figure 4.11 a third implementation graph example. In this case, the implementation graph is built only with one time slot. Also, all the processing resources have a task assigned.

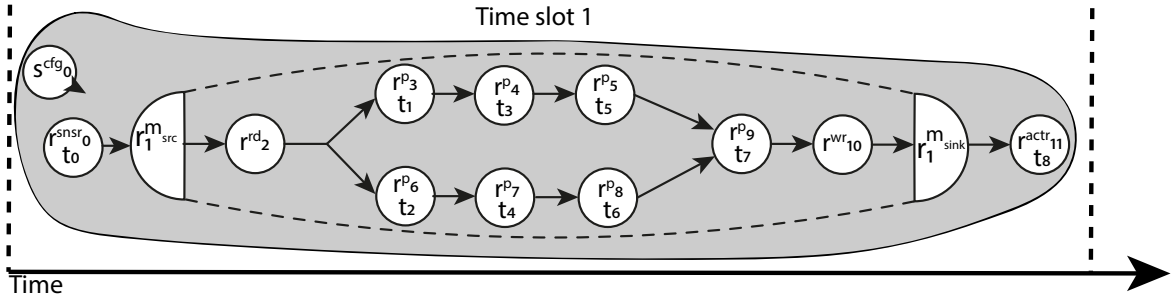


Figure 4.11 – Third implementation graph example

In Figure 4.12 the resulting evaluation graph is shown. We can see that the transformation corresponds to the connection of the configuration control resource to the hardware resource r_0^{snsr} and removal of r_1^m.

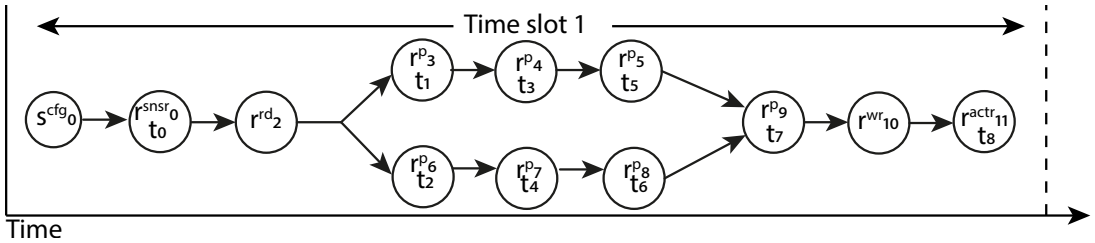


Figure 4.12 – Third evaluation graph example

Example 4 : Evaluation Graph and Copy Resource

In Figure 4.13 we can see a similar implementation graph example as in the Third example. The difference is that we include copy operations. But as the copy operations should be included in the performance analysis, we need to keep them in the evaluation graph, which is shown in Figure 4.14.

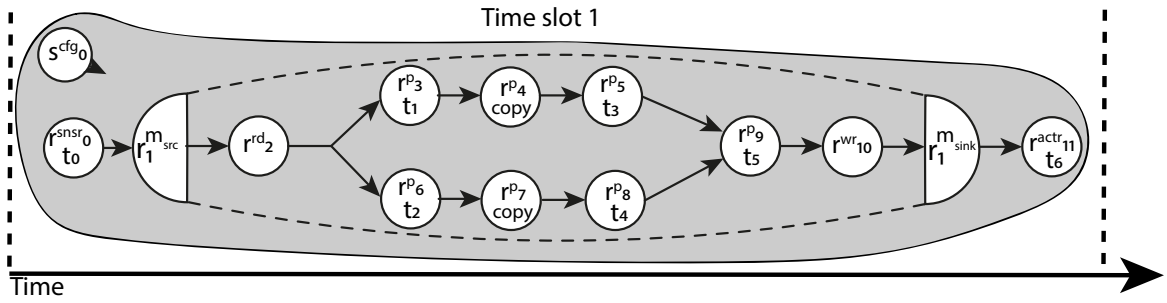


Figure 4.13 – Fourth implementation graph example

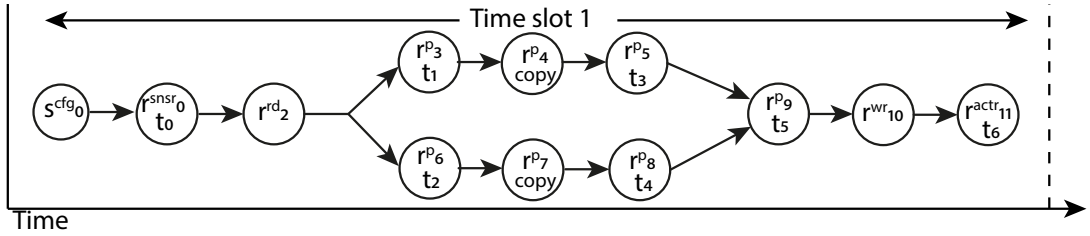


Figure 4.14 – Fourth evaluation graph example

Example 5 : Evaluation Graph and Disable Resource

This last example of an evaluation graph corresponds to an implementation graph that includes disable resources depicted in Figure 4.15. Notice that the resources colored in blue are disable resources.

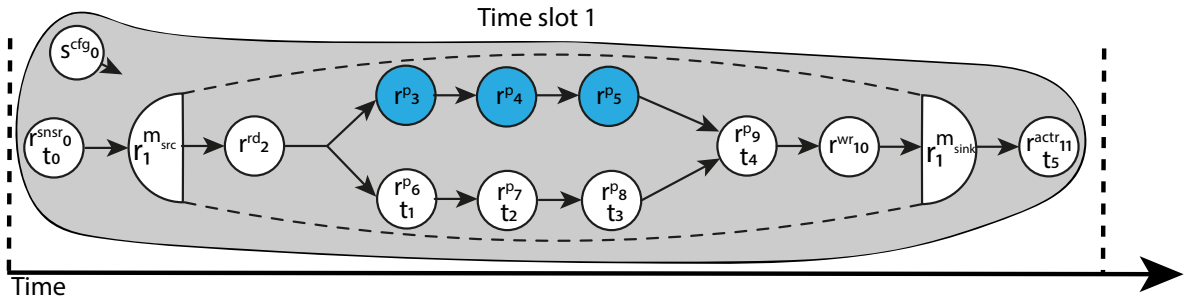


Figure 4.15 – Fifth implementation graph example

As the fifth implementation graph includes disabled nodes, we need to exclude them as well as connect the configuration control resource to r_0^{snsr} and remove r_1^m. In Figure 4.16 the resulting evaluation graph is shown.

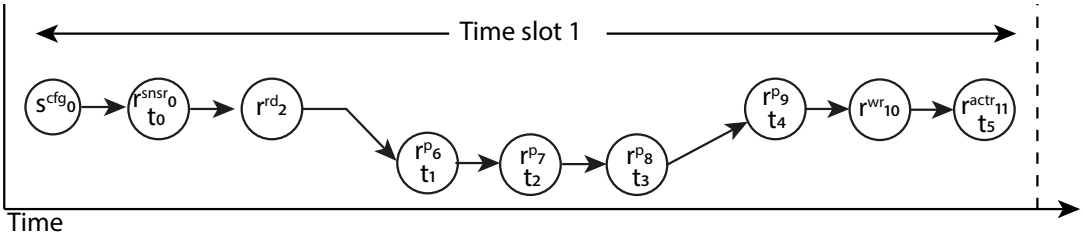


Figure 4.16 – Fifth evaluation graph example

Example 6 : Performance Evaluation

To exemplify the performance evaluation, let's use a simple, intuitive example. This example consists of three different sets of parameters for the hardware resources. These different values will produce different critical paths. The purpose of this example is to detail the performance evaluation methodology and show the accuracy of the Equation 4.9 presented in Section 4.4.2.

Figure 4.17 shows the experimental application graph that we will use. And, Figure 4.18 depicts the experimental hardware graph. Finally, Figure 4.19 we showed the resulting implementation graph.

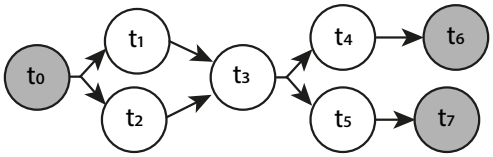


Figure 4.17 – Application graph for the performance evaluation example.

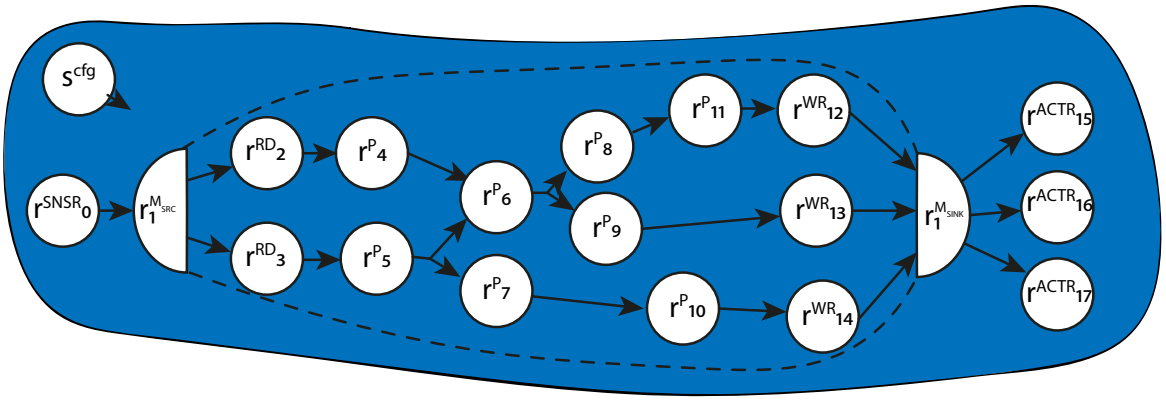


Figure 4.18 – Hardware graph for the performance evaluation example.

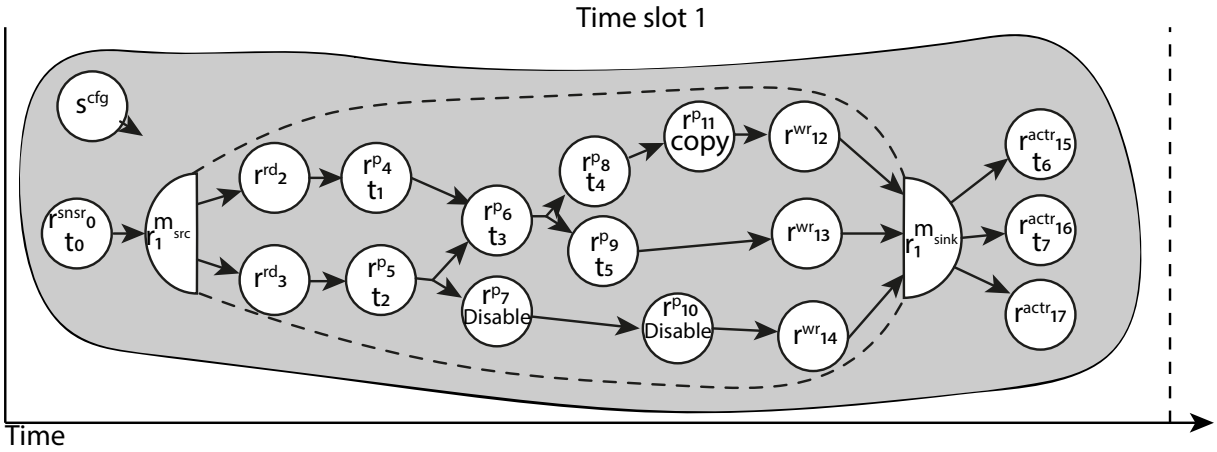


Figure 4.19 – Implementation graph for the performance evaluation example.

In order to get the evaluation graph we apply the transformations described in Section 4.4.2 (memory node removal, disable node removal, etc.). The resulting evaluation graph presented in Figure 4.20.

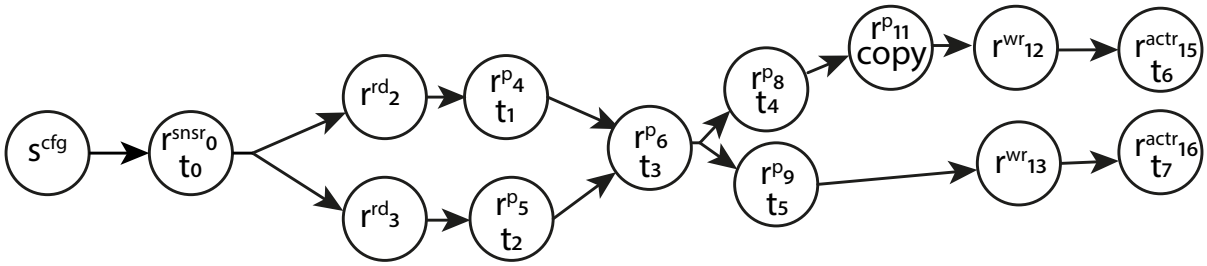


Figure 4.20 – Evaluation graph for performance evaluation example.

To compute the critical path of the graph from Figure 4.20, we have to compute the length of 4 possible paths represented in Figure 4.21 (each path with different colors):

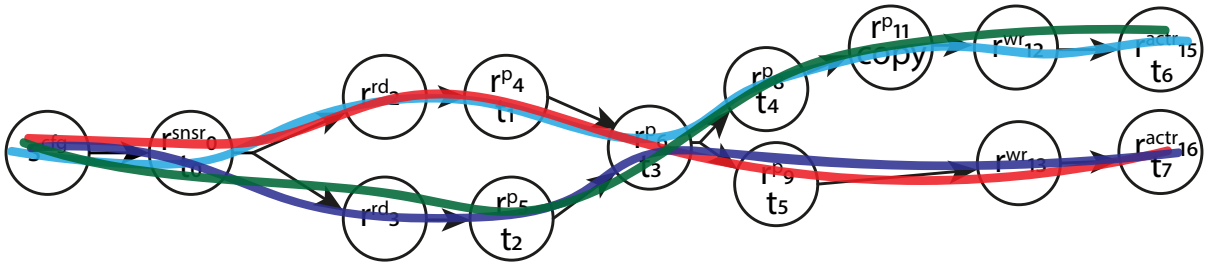


Figure 4.21 – Paths of the evaluation graph.

- Path 1 (blue) : $(s^{cfg}, r_0^{snsr}, r_2^{rd}, t_1/r_4^p, t_3/r_6^p, t_4/r_8^p, copy/r_{11}^p, r_{12}^{wr}, t_6/r_{15}^{actr})$
- Path 2 (red) : $(s^{cfg}, r_0^{snsr}, r_2^{rd}, t_1/r_4^p, t_3/r_6^p, t_5/r_9^p, r_{13}^{wr}, t_7/r_{16}^{actr})$
- Path 3 (green) : $(s^{cfg}, r_0^{snsr}, r_3^{rd}, t_2/r_5^p, t_3/r_6^p, t_4/r_8^p, copy/r_{11}^p, r_{12}^{wr}, t_6/r_{15}^{actr})$
- Path 4 (purple) : $(s^{cfg}, r_0^{snsr}, r_3^{rd}, t_2/r_5^p, t_3/r_6^p, t_5/r_9^p, r_{13}^{wr}, t_7/r_{16}^{actr})$

First Set of Parameters

In the whole example, we will use the parameters of Table 4.1 for the application graph. Also, for the rest of the section, N/A means *Not Applicable*.

Table 4.1 – Parameters of the application graph of the performance evaluation example.

Task	*type*	p
t_0	interface	N/A
t_1	$task1$	fix
t_2	$task2$	fix
t_3	$task3$	fix
t_4	$task2$	fix
t_5	$task1$	fix
t_6	interface	N/A
t_7	interface	N/A

In this first part of the example, we will use the parameters of the hardware resources presented in 4.2.

Table 4.2 – First set of parameters of the hardware graph.

	$\mathcal{T}_i$	$\mathcal{L}_i$		Cfg_i
		Input latency	Computing latency	
$r^P_{4,9,10}$	$task1, task5$	2	2	1
$r^P_{5,8,11}$	$task2, task6$	2	2	1
r^P_7	$task5, task6$	2	2	1
r^P_6	$task3, task4$	2	2	1
r^{SNSR}_0	N/A	N/A	1	N/A
$r^{ACTR}_{15,16,17}$	N/A	N/A	1	N/A
$r^{RD}_{2,3}$	N/A	N/A	1	1
$r^{RW}_{12,13,14}$	N/A	N/A	1	1

Among the paths established above, we have to identify the critical according to the first set of parameters. For that purpose, we compute the lengths using a timing diagram:

Figure 4.22 presents the path 1. We compute the computing cost over this path (if we compute the computing cost of the three other paths, we will obtain a lower value; hence it will not be the critical path).

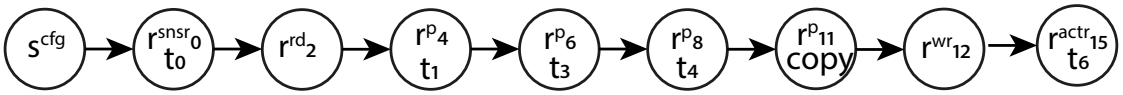


Figure 4.22 – Critical path of first set of parameters.

Using the values of Table 4.2 and 4.1. Consider an input of 100 samples and a configuration cost of 1 clock cycle. Also, suppose the duration of the copy operation for all processing resources has a value of 1 clock cycle. Using Equation 4.9, we compute the computing cost as follows:

$$
\begin{aligned}
CC = & (\overbrace{(1)}^{\mathbf{Tc}_1} + \overbrace{(2*100)}^{\mathbf{Tex}_1} + \\
& \overbrace{\underbrace{(1)}_{r^{snsr}_0} + \underbrace{(1)}_{r^{rd}_2} + \underbrace{(2*1+2)}_{r^p_4} + \underbrace{(2*2+2)}_{r^p_6} + \underbrace{(2*2+2)}_{r^p_8} + \underbrace{(1)}_{r^p_{11}} + \underbrace{(1)}_{r^{wr}_{12}}}^{\mathbf{Tin}_1}) \\
CC = & (\overbrace{(1)}^{\mathbf{Tc}_1} + \overbrace{(200)}^{\mathbf{Tex}_1} + \overbrace{20}^{\mathbf{Tin}_1}) = 221 \; clock \; cycles
\end{aligned}
\tag{4.13}
$$

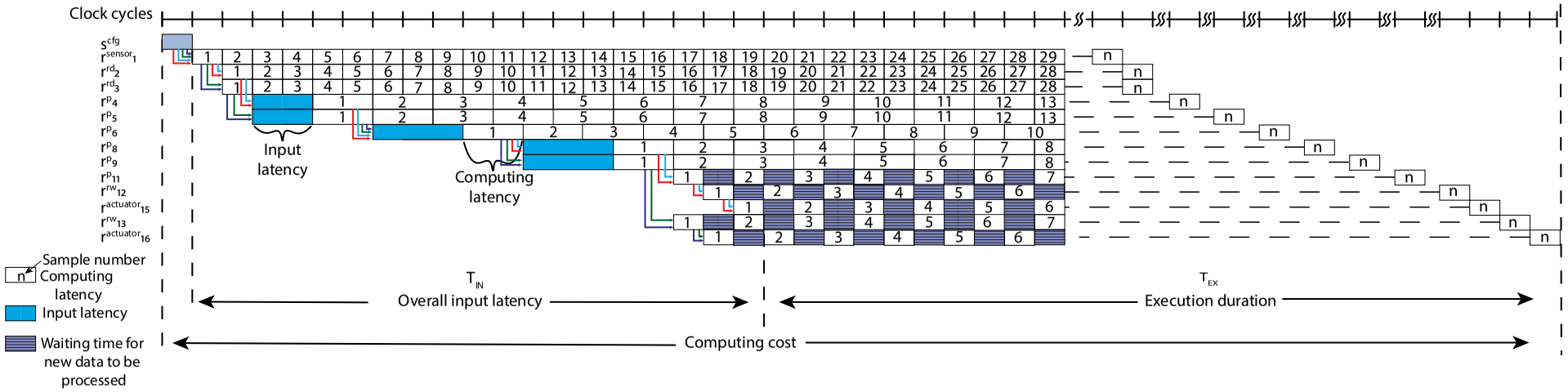


Figure 4.23 – Timing diagram of the first set of parameters.

Second Set of Parameters

Now, let's use a different set of latency features. We change the parameters for r_5^P. Let's assume that for the implementation of *task2*, the input latency changes to 3 samples and the computing latency to 3 clock cycles. These changes are shown in Table 4.3. The critical path is no longer the same. Figure 4.24 shows the new critical path and Figure 4.25 the new timing diagram.

Table 4.3 – Second set of parameters of the hardware graph of performance evaluation example.

	$\mathcal{T}_i$	$\mathcal{L}_i$		Cfg_i
		Input latency	Computing latency	
$r_{4,9,10}^P$	$task1, task5$	2	2	1
r_5^P	$task2, task6$	3	3	1
$r_{8,11}^P$	$task2, task6$	2	2	1
r_7^P	$task5, task6$	2	2	1
r_6^P	$task3, task4$	2	2	1
r_0^{SNSR}	N/A	N/A	1	N/A
$r_{15,16,17}^{ACTR}$	N/A	N/A	1	N/A
$r_{2,3}^{RD}$	N/A	N/A	1	1
$r_{12,13,14}^{RW}$	N/A	N/A	1	1

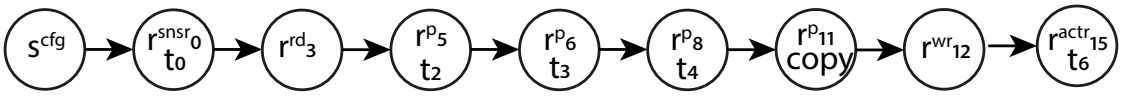


Figure 4.24 – Critical path for the second set of parameters

With the new values described in Table 4.3, we compute the same Equation 4.9 as follows:

$$
\begin{aligned}
CC = & \overbrace{(1)}^{\mathbf{Tc}_1} + \overbrace{(3*100)}^{\mathbf{Tex}_1} + \\
& \overbrace{\underbrace{(1)}_{r_0^{snsr}} + \underbrace{(1)}_{r_3^{rd}} + \underbrace{(3*1+3)}_{r_5^p} + \underbrace{(2*3+2)}_{r_6^p} + \underbrace{(2*3+2)}_{r_8^p} + \underbrace{(1+1)}_{r_{11}^p} + \underbrace{(1)}_{r_{12}^{wr}}}^{\mathbf{Tin}_1} \\
CC = & \overbrace{(1)}^{\mathbf{Tc}_1} + \overbrace{(300)}^{\mathbf{Tex}_1} + \overbrace{(27)}^{\mathbf{Tin}_1} = 328 \; clock \; cycles
\end{aligned}
\tag{4.14}
$$

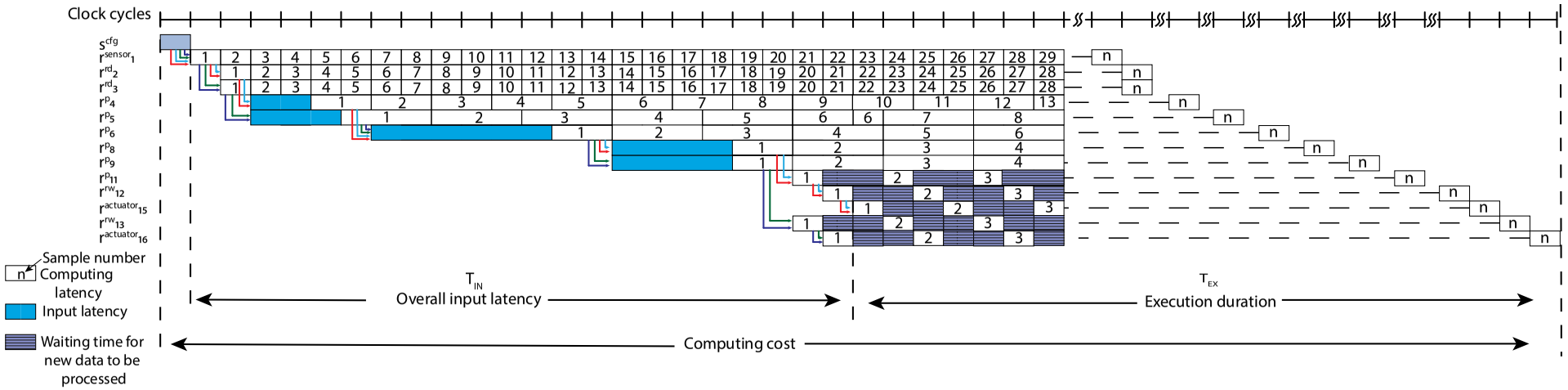


Figure 4.25 – Timing diagram for the second set of parameters.

Third Set of Parameters

Finally, for the third considered latency features, the changes are considerable. For r_5^P, the implementation of $task2$ considers 3 samples as input latency and 2 clock cycles as computing latency. For r_6^P, the implementation of $task3$ considers 4 samples as input latency and 3 clock cycles as computing latency. For r_9^P, the implementation of $task1$ assumes 1 sample as input latency and 3 clock cycles as computing latency. These changes are summarized in Table 4.4. We get a new critical path shown in Figure 4.26, and the latest timing diagram is shown in Figure 4.27.

Table 4.4 – Third set of parameters of the hardware graph of performance evaluation example.

	$\mathcal{T}_i$	$\mathcal{L}_i$		Cfg_i
		Input latency	Computing latency	
$r_{4,9,10}^P$	$task1, task5$	2	2	1
r_5^P	$task2, task6$	3	2	1
r_9^P	$task1, task5$	3	3	1
$r_{8,11}^P$	$task2, task6$	2	2	1
r_7^P	$task5, task6$	2	2	1
r_6^P	$task3, task4$	4	3	1
r_0^{SNSR}	N/A	N/A	1	N/A
$r_{15,16,17}^{ACTR}$	N/A	N/A	1	N/A
$r_{2,3}^{RD}$	N/A	N/A	1	1
$r_{12,13,14}^{RW}$	N/A	N/A	1	1

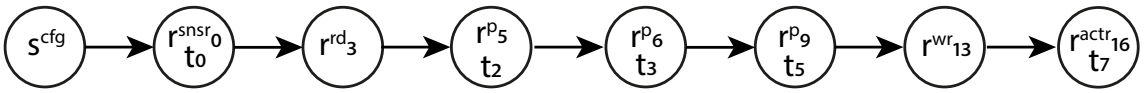


Figure 4.26 – Critical path for the third set of parameters.

We compute the computing cost with the new values as follows

$$
\begin{aligned}
CC = & \overbrace{(1)}^{\mathbf{Tc}_1} + \overbrace{(3*100)}^{\mathbf{Tex}_1} + \\
& \overbrace{\underbrace{(1)}_{r_0^{snsr}} + \underbrace{(1)}_{r_3^{rd}} + \underbrace{(3*1+2)}_{r_5^{p}} + \underbrace{(4*2+3)}_{r_6^{p}} + \underbrace{(3*3+3)}_{r_9^{p}} + \underbrace{(1)}_{r_{12}^{wr}}}^{\mathbf{Tin}_1} \\
CC = & \overbrace{(1)}^{\mathbf{Tc}_1} + \overbrace{(300)}^{\mathbf{Tex}_1} + \overbrace{(31)}^{\mathbf{Tin}_1} = 332 \; clock \; cycles
\end{aligned}
\tag{4.15}
$$

As we can see with these examples, the proposed equation for the computing cost produces cycle-accurate results. It is helpful to determine the estimation of the upper bound of the computing cost.

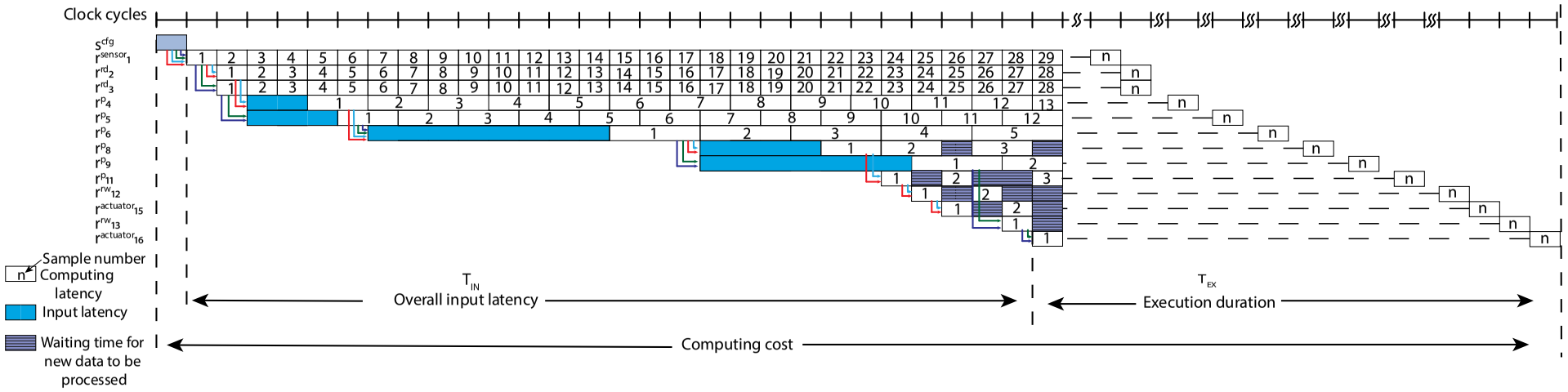


Figure 4.27 – Timing diagram for the third set of parameters.

4.5 Conclusions

In this chapter, we have introduced our graph-based implementation model. It can represent both the spatial (mapping) and the temporal (scheduling) allocation of the application's tasks onto the hardware resources. The implementation model also carries information about the resource's latencies. This feature is inherited from the hardware model. Additionally, we introduce a subset of virtual nodes that allows one to model data dependency between time slots and between datapaths. This subset enables to describe recomputation of data within a time slot and between time slots if the capabilities of the hardware architecture allow it.

We presented a performance evaluation methodology. Based on graph transformations of the implementation graph, we build what we call an evaluation graph. From the evaluation graph, we compute an estimation of the upper bound of the computing cost over the critical path. We evaluate our proposed equation over an example and three different sets of parameters. We demonstrate the accuracy and the benefits our the performance evaluation methodology.

Part II

Mapping Algorithms

Chapter 5

Introduction

The mapping algorithm seeks to allocate, by matching process, an application's tasks to the available hardware resources. The scheduling algorithm establishes the order of (sequence) execution of each task on each hardware resource [141]. Both processes can be performed separately, although they impact each other efficiency. Both the mapping and scheduling algorithms play a critical role in today's systems. They are in charge of allocating the tasks efficiently and optimizing metrics such as latency, resource utilization, energy consumption, and data throughput, among others.

Generally, the mapping and scheduling problem is considered NP-complete [142]. This means that the optimal solution can only be obtained by exploring all the possible solutions. The execution time of algorithms for solving NP-complete problems increases exponentially with respect to the size of the input data (the number of tasks and resources). An efficient heuristic may be used to decrease the exploration time and obtain a reasonable solution.

Common approaches to solve this problem are list-scheduling, clustering scheduling, and task duplication-based scheduling. List-scheduling is the most used approach. List-based scheduling algorithms usually are single-shot algorithms that focus on systems with limited hardware resources. This approach defines priorities to the tasks according to a metric (latency, types, data dependence) and schedules them in a topological order in decreasing priority [143]. The task's priority can be computed statically, before the scheduling, or dynamically, interleaving with the scheduling.

List-based scheduling algorithms are widely used for heterogeneous systems and represent an efficient approach for SPS-CGRA-based systems. This approach provides the means to consider the complex structure of an SPS-CGRA and the limited availability of hardware resources. It also allows one to define the priorities according to both the topological order and the realization efficiency.

Given today's applications complexity and demands, new approaches for the scheduling problem have been proposed. Machine learning, a subset of artificial intelligence, has grown in popularity because of its simple recipes and algorithms that can learn patterns, behaviors, models, and functions [144]. In particular, a type of machine learning called reinforcement learning has been the focus of several current works. In reinforcement learning, an agent is given the task to explore and interact with an environment. During this interaction, the agent will learn how to act given some state. This approach relaxes the scheduling problem's complexity and increases the universe of possible solutions given the learn-by-trial method used in reinforcement learning.

In this part of the thesis, we explore two approaches to solve the mapping and scheduling problem for SPS-CGRA. In Chapter 6 we present three different types of list-based algorithms. Chapter 7 presents a Q-Learning mapping algorithm. Finally, in Chapter 8, we evaluate and compare our solutions and briefly discuss the results.

We organize the remaining part of this chapter as follows. Section 5.1 reviews state of the art. Section 5.2 presents a brief discussion of state of the art. Finally, Section 5.3 summarizes

this chapter.

5.1 State of the art

As mentioned in the previous paragraphs, mapping and scheduling problems are defined as an NP-complete problem [142]. The best solution can only be obtained by brute force search. A brute force algorithm will find the optimal implementation of an application on the hardware, but it explores and evaluates all possible solutions in a considerable exploration time. To deal with this problem, several heuristics and techniques have been proposed to reduce exploration space and exploration time.

In this section, we review the state of the art of mapping algorithms. We start with list-based algorithms, continue with linear programming algorithms, following reinforcement learning planning algorithms. Finally, in the last section, we group together several interesting approaches. Although these approaches are far from our work, we can benefit from some of their characteristics.

5.1.1 List-Based Scheduling Algorithms

One of the families of algorithms widely represented in the literature are list-based mapping and scheduling algorithms. Their principle is based on a list of priorities, in which resources and tasks are ordered according to a fixed metric. This metric is chosen according to the type of material and its characteristics.

Qin et al. [87] introduced the Heuristic task based on the Critical Path and Task replication scheduling algorithm (HCPTD) for distributed systems. It is a list-based algorithm with two phases. First, it orders the tasks with the earliest start time. Next, the resources are ordered using the earliest completion time and the shortest distance to the exit. The task is scheduled according to the previous priority lists. If the task's predecessors are more than two, the task may need to wait until their completion, using the processor's space time slot (idle time).

Topcuoglu et al. [91] presented the Heterogeneous Earliest Finish Time (HEFT) scheduling algorithm for Multiprocessors. Based on the minimization of the computation cost of the task and the upward rank of the task. The authors also present the critical path on the processor (CPOP) algorithm based on the task's computation cost and both the downward and upward rank. The upward rank is the computing cost value, which is the sum of the task's computing cost plus the maximum value of the successor's upward rank plus the communication cost to that task. The downward rank is the maximum value of the set of sums of the following items: the computing cost of the task plus the communication cost to the predecessor plus the upward rank of the predecessor.

Bhatti et al. [145] presented Noodle (No Node is Left Behind), a list-based scheduling algorithm for MPSoCs. The priority mechanism aims to maintain proportionate fairness among all ready (to be executed) tasks belonging to all paths within a task graph.

Sih and Lee [81] presented a dynamic level scheduling algorithm. It is a compile-time heuristic for heterogeneous and homogeneous processors. At each step, the algorithm considers the interprocessor communication overhead and incorporates information about the processor interconnection topology.

Grandpierre and Sorel [35] presented a greedy [1] list-based heuristic algorithm targeting heterogeneous multicomponent systems based on the critical path and schedule flexibility. It takes into account the execution duration of operations but also each communication overhead as a cost function. It is computed from the worst-case execution time.

Wang et al. [88] presented the Heterogeneous Scheduling with Improved Task Priority for heterogeneous computing systems. It is a list-based scheduling algorithm that orders the tasks,

[1] An algorithm that always takes the best immediate, or local, solution while finding an answer [146].

and their predecessors, concerning their computing cost. The available processors are ordered using the Earliest Finish Time depending on the type of task that it will be allocated.

Youness et al. [98] presented a partitioning algorithm based on the A-start algorithm [147] for mixed hardware/software systems. It is a list-based algorithm with computing cost as the priority. The algorithm first schedules all the tasks to processors. Then, it identifies the critical path and tries to convert it into hardware to reduce the scheduling length.

Kaida et al. [148] presented a mapping algorithm for embedded many-core SoCs. It aims to maximize the gain of mapping task i onto core j, including energy consumption, execution duration, and other factors. We authors introduce the concept of tile, a set of cores where a single task might be mapped. A single many-core SoC may have several tiles distributions which increase the possibilities of scheduling.

Frid and Sruk [95] presented a modification of the Critical Path method [149] targeting MPSoCs. The algorithm identifies the critical path of the application and assigns it to the fastest processor available. Next, two stages are performed for the remaining tasks. First, they order the processors concerning their computation speed in ascending order. Then the remaining tasks are scheduled to the slowest possible processor. Finally, if there are still tasks, the algorithm inverts the processors' order and schedule the remaining tasks.

Zadrija and Sruk [94] presented a list-based mapping algorithm for MPSoCs that uses the longest processing time as the priority.

Kota et al. [106] presented two algorithms targeting Reconfigurable Logic Units (RLU). The first algorithm uses a greedy heuristic based on the minimization of the task finish time. The second approach is based on dynamic programming. The difference between the two algorithms is that the greedy algorithm is based on local decisions (the best allocation for a particular task) and the dynamic programming approach on a sequence of decisions. A novelty of both methods is that it considers heterogeneous RLUs, both in terms of size of the chip area and multiple physical hardware implementations.

Lu et al. [101] presented a mapping algorithm for CGRA. It is a list-based algorithm that uses the topological order and the number of available resource neighbors as priorities. The reasoning behind the latter's use as a part of the priorities is that the successors of the current task will have more possibilities to be mapped. The algorithm is equipped with backtracking to avoid mapping deadlock, and it can split the application into time slots.

Sun and Zhang [134] presented an Energy-aware mapping algorithm for NoCs. It is a list-based algorithm that uses a greedy heuristic based on Rent's rule-based communication probability function. The priority list is constructed according to the smallest communication value.

Sun and Zhang [134] presented an Energy-aware mapping algorithm for NoCs. It is a list-based algorithm that uses a greedy heuristic based on Rent's [2] rule-based communication probability function [152]. The priority list is constructed according to the smallest communication value, which yields the lowest energy consumption.

Jiang et al. [153] presented the Testing-Aware Mapping Algorithm (TAMA) that targets NoCs. It is a list-based algorithm that orders the tasks in a first instance using the number of edges (maximum first), then after the selection of the first task, the ordering continues using a Breadth-first search algorithm [154]. The processors are ordered according to the maximum number of available neighbors, the most significant number of tested links, and finally, using the smallest Manhattan distance [3]. The test of links refers to the iterative test of idle links to define if their processors can allocate the tasks and be able to comply with the deadlines of the tasks. The testing occurs meanwhile mapping other tasks and increase the performance of the mapping.

Lu et al. [110] presented a Greedy mapping algorithm based on the Manhattan distance. It

[2] In VLSI designs, Rent's rule relates the number of terminals in a boundary, to the number of blocks within that boundary by a power-law relation: $T = \kappa G^{\beta}$. Where T is the number of terminals, G is the number of blocks (gates), κ is the average number of terminals for each block, and β is the Rent's exponent [150, 151].

[3] The Manhattan distance between two points $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ in n-dimensional space is the sum of the distances in each dimension: $d(x, y) = \sum_{i=1}^{n} \mid x_i - y_i \mid$ [155].

is a list-based algorithm that uses the breadth-first search[4] to traverse the application graph. It is suitable for both regular or irregular NoCs.

Mehran et al. [130] presented a Dynamic Spiral Mapping (DSM) heuristic that targets 2-D mesh topologies and aims to minimize the Manhattan distance. The Spiral Mapping is a list-based algorithm that uses data transfer, Manhattan distance, and connection degrees as priorities [129]. The authors introduce two versions, Full Dynamic Spiral Mapping, which tries to minimize the reconfiguration of the hardware resources without stopping the execution, and Partial Dynamic Spiral Mapping, which stops the application's execution tries to execute the Spiral Mapping again to improve the performance.

5.1.2 Linear Programming

Another approach is represented by linear programming. We can cite the example of Chin and Anderson [104] who presented an integer linear programming approach for CGRA mapping. The designer gives a set of constraints such as one operation per functional unit, the functional unit needs to be able to implement the operation, a route can only be used once, and disallowing multiplexer inputs from having the same value. Then, the solver tries to find a solution that improves the performance of the mapping.

Kim et al. [72] introduced a linear programming scheduling algorithm for heterogeneous multi-core architectures. The algorithm takes into account the different physical hardware implementations and latencies of the hardware resources.

Yoon et al. [105] introduced Graph Drawing Based Spatial Mapping (SPKM) Algorithm for CGRA. It is an integer linear programming approach based on a split-push kernel algorithm [156]. It tries to decrease the number of rows used, thus minimizing the resource occupation.

5.1.3 Reinforcement Learning

In recent years, Machine Learning techniques have been successfully applied to the scheduling problem. We focus on those based on Q-learning. Let's recall that reinforcement learning (RL) is a subset of Machine Learning, where an agent interacts within an environment. From these interactions, the agent aims to improve its actions towards the maximization of a reward. RL algorithms are known for its generality and adaptability [157, 158, 159] and are used in several fields [160, 161, 162, 163, 164, 165, 166]. Adapted to the scheduling problem, RL algorithms have helped to decrease exploration time [167], decrease the complexity of the scheduling algorithm [160], generalization of cases [168] and optimization of some metric [169].

Recently, RL algorithms have been adapted to the task scheduling problem. From a single machine to hardware platforms, RL algorithms have shown improvements to the current state of the art. However, the main focus has been Cloud-based [170, 171, 172] or High Performance Computing (HPC) [173, 174]. Wu et al. [175] proposed an RL-based solution of the DAG tasks scheduling problem for HPC. The authors used a policy gradient-based REINFORCE agent. Each action's reward is obtained by calculating the increment of the current schedule length after a task is scheduled. Liu et al. [176] proposed a variant of the work of [175] using Monte Carlo Tree Search. They showed improvements in the Schedule Length Ratio, however, with an increase of the exploration time. Grinsztajn et al. [177] presented an Actor-Critic algorithm for HPC scheduling. The reward policy is based on the final makespan of the implementation. Lee et al. [178] presented Panda, a Reinforcement Learning-Based Priority Assignment for Multi-Processor Real-Time Scheduling. The authors use the REINFORCE algorithm with a custom reward policy. The reward policy includes a schedulability evaluation and response time analysis. Their evaluation shows robust performance in terms of schedulability ratio and adaptability for non-trivial large-scale settings. Luley and Qiu [179] presented a Deep Q-learning scheduler for GPUs. They used a reward policy based on a combination of the device utilization

[4] Any search algorithm that explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level [146].

and turnaround time of waiting tasks. Their evaluation showed an improvement in resource utilization and throughput. Liu et al. [180] presented a Deep RL algorithm for CGRA mapping. The authors proposed a Deep Q-learning method where an application is randomly mapped to the architecture, and then the agent will relocate the tasks according to a greedy policy. After each change, the agent will receive a reward that considers power consumption, area, and performance.

5.1.4 Others

Finally, we recall here again other interesting approaches which allow us to illustrate both the wealth of research in the field and at the same time the importance of the problem treated.

Honorat et al. [181] presented a scheduling algorithm for Cyber-Physical Systems (CPSs). It aims to provide an efficient scheduler for partially periodic CPSs which is modelled as a Synchronous Data Flow (SDF) graphs [83].

Yang et al. [182] presented a scheduling algorithm based on Bayesian optimization algorithm (BOA) [183] for a heterogeneous computing environment. The algorithm consists of two steps. At first, BOA assigns tasks to processors based on the computing cost. Next, a list-scheduling algorithm establishes the sequence of tasks. This process is iteratively repeated until the makespan is optimal.

Biswas et al. [184] introduced a bayesian optimization-based approach for task scheduling of heterogeneous multiprocessor systems. This algorithm aims to learn the structure and the parameters of both the task graph and the multiprocessor system.

Namazi et al. [109] presented a majority-based reliability-aware mapping algorithm for NoCs. It is a task duplication-based algorithm that uses reliability as its criteria for task duplication. Reliability is a quantitative parameter that shows the probability of a system being operational after some time, considering it was operating initially. The mapping technique takes into consideration the reliability as well as the execution time.

Yuan et al. [92] presented the fairness-aware single DAG scheduling algorithm (FASS) and fairness-aware multiple DAGs scheduling algorithm (FAMS), both for multiprocessors. The algorithms use task replication to improve the reliability, and the upward rank of the task [91] as the priority. FAMS is a dynamic algorithm capable of scheduling multiple DAGs.

Jiang et al. [112] presented the Path Clustering Heuristic (PCH) and the Gap Search algorithm for high-performance computing systems. The PCH clusters the tasks into groups depending on the communication cost. Next, the tasks are scheduled using a list scheduling algorithm. After the first schedule, the Gap Search algorithm identifies gaps in the schedule and maps entire clusters to increase the performance.

Liu and Shen [185] presented the dependency-aware and resource-efficient scheduling algorithm for cloud frameworks. The algorithm can verify the dependency of the tasks, identify the independent tasks, and schedule them to run in parallel and improve the performance. The algorithm also identifies different hardware resources (e.g., CPU, memory, GPU), which allows assigning the tasks to the best suitable resource.

Mei et al. [100] introduced the modulo scheduling algorithm that targets CGRA. It is based on simulated annealing. The algorithm iteratively reallocates an operation randomly and computes the next execution time. Next, a simulated annealing strategy evaluates the new execution time and decides if the new schedule is accepted or rejected. If rejected, a new iteration occurs, where a minor change is done to see if the next iteration would be better.

Wang et al. [121] presented a genetic algorithm for mapping application tasks to high computing environments. The algorithm uses a mapping list and topological sorting of the application as chromosomes, and the fitness value is the execution time of the implementation.

Sinaei and Fatemi [97] presented a tree-based algorithm for mapping application onto MP-SoCs. It uses two methods, an exhaustive and a genetic algorithm. The algorithm iterative searches for the implementation with the lowest energy consumption.

Zhao et al. [186] introduced a task scheduling algorithm with resource attribute selection for Many task computing. Resource attribute selection uses a gene expression programming-based approach [187] and work-stealing algorithm [5]. It can define several characteristics of the task, such as type, storage space needed, start time, execution time, and use them as a fitness function to iterative search for the best candidate to allocate the task.

Selvameena and Prasath [189] introduced a modified score boarding algorithm that targets MPSoC. It is used along with the particle swarm optimization technique. It aims to improve the communication cost of the system.

Frid and Sruk [96] presented an evolutionary mapping algorithm for multicore systems with two approaches. In the first approach, the mapping considers both memory and processing resources as chromosomes. The second approach divides the mapping process into two phases. One phase considers only the processing resources as chromosomes, and the second phase uses only the memory resources.

Emeretlis et al. [190] presented a mapping algorithm based on benders decomposition [191] for multicore systems. The Logic-Based Benders decomposition approach is an iterative process that aims at reducing the solution time of complex optimization problems. The main idea is to create a sequence of two sub-problems, where the second sub-problem uses the solution of the first one. The process terminates if no better solution can be found for the first sub-problem or the whole solution space has been examined [190, 191]. The authors propose to divide the mapping and scheduling problem into two. The first problem is the mapping and relaxation of the scheduling. The second problem is the scheduling.

Hamzeh et al. [103] introduced the Register-Aware Application Mapping (REGIMap) for CGRA. The authors exploit the processing resources' register files to decrease the initiation interval *II*. They reduce the mapping problem to find the maximal weighted clique in the product graph of the time-extender CGRA and the data dependence graph.

Ferreira et al. [192] presented a mapping and scheduling tool based on the modulo scheduling with a heuristic for CGRA. The algorithm uses a mix of asap/alap (as soon as possible/as late as possible) to determine each operation's scheduling range. It is custom-designed for their in-house hardware.

Lai and Yeh [107] presented a mapping algorithm that converts a data flow graph into a reconfigurable architecture rDPA [193]. The algorithm is based on a set of templates of processing resources called data path units, and the goal is to convert the nodes of the data flow to a type of data path unit.

Pathan et al. [93] proposed a two-level preemptive global fixed-priority scheduling for multicore systems. The first level is a task-level scheduler. Next, a subtask-level scheduler is implemented.

Qamhieh et al. [90] presented a stretching algorithm that aims to transform the application into a set of independent sequential threads. This algorithm is used as a previous step for scheduling algorithms like Global Earliest Deadline First from the fixed-job priority family and Global Deadline Monotonic from the fixed task priority family. It can improve the performance of the scheduling.

5.2 Discussion

As illustrated above, there is an extensive quantity of works related to mapping and scheduling different platforms. Usually, a mapping and scheduling algorithm is designed to perform well for a particular platform, and it is not always made to allow a migration to another. A significant drawback of most algorithms is that the hardware is considered a regular structure [87, 91, 88]. However, approaches like [35, 81, 148] capable of dealing with irregular structures may not be directed to our problem. This represents an issue for SPS-CGRA mapping, as usually, this type of platform features an irregular heterogeneous structure. It is also important to mention the

[5] In work-stealing, underutilized processors attempt to "steal" threads from other processors [188].

approach of the algorithm. Cluster-based [112] and task duplication algorithms [109, 92] may work fine with multicore systems. Still, it is not directly applicable to our targeted hardware due to the limited amount of resources and the granularity of the types of tasks they can implement.

List-based algorithms [35, 101] are suitable for our purposes, as they aim to limit resource platforms and the designer who chooses the appropriate priority. Combined with a graph representation of the hardware and application, we can use standard graph theory algorithms, which provide high performance [181, 182, 184, 100]. Furthermore, a heuristic may increase the performance of such a mapping algorithm [110, 153]. Characteristics like computing cost, use of resources, and memorization may be considered for the heuristic. Moreover, well-established techniques such as the Bayes theorem [182, 184] and its combination with a list-based scheduling algorithm may be good candidates to solve the SPS-CGRA mapping problem.

The exploitation of RL algorithms for the scheduling problem is hasn't spread yet. However, it has shown attractive advantages in contrast to classical approaches. In particular, for hardware accelerators and specifically CGRA, there are no so many RL-based works. However, RL algorithms' inclusion may solve some of the current issues, such as non-optimal mapping and lack of support for complex structures.

5.3 Conclusions

In this chapter, we presented a brief state of the art of mapping and scheduling algorithms. We divide the algorithms into list-based, linear programming, RL, and others. Given the characteristics of the SPS-CGRA and its applications, we propose to use the list-based approach and the reinforcement approach to solve its mapping problem. The list-based approach is primarily used on platforms with a limited amount of resources. Moreover, the custom priority list helps manage the data dependence of the application and the unidirectional processing data of the hardware. Additionally, an RL approach will help us deal with the complex structure of the SPS-CGRA and its applications. Using the trial-and-error methodology of this approach, we may increase the locality of the list-based algorithms' mapping decision and improve its results.

In the following chapters, we will introduce our proposed solutions for the SPS-CGRA mapping problem.

Chapter 6

List-Based Mapping Algorithms

In the previous chapter, we saw that the mapping and scheduling problem is considered NP-complete. This means that the optimal solution can only be obtained by exploring all the possible solutions. Moreover, SPS-CGRA increases the exploration space because of its different physical hardware implementation for a given task. Effectively, an efficient hardware resource may compensate for the memorization and configuration cost. Thus a possible final mapping would have a single task allocated per time slot. Hence, the design space and exploration time increase exponentially according to the number of tasks and hardware resources due to combinatorial explosion. This is why we propose an efficient heuristic that may be used to decrease the exploration time and obtain a reasonable solution.

In this chapter we present a basis algorithm that we transform into two optimization heuristics.

- **Single-Shot mapping algorithm**, based on the topological order of the input graphs (application and hardware graph). This algorithm is able to build a mapping without considering the performance of this mapping.
- **Topology-Aware mapping heuristic**, based on look-ahead techniques. It relies initially on the single-shot algorithm but considers topological distances, the probability of mapping success, and the computing latency in order to build an optimized solution.
- **Bayes-Based mapping heuristic**, based on the formalization of the previous algorithm into a Bayes problem. It includes several parameters that the user can use to tune the mapping process.

We organize the remaining part of the chapter as follows. Section 6.1 introduces the Single-Shot mapping algorithm. Section 6.2 presents the Topology-Aware mapping algorithm. Section 6.3 presents the Bayes-Based mapping algorithm. Finally, Section 6.4 summarizes this chapter.

6.1 Single-Shot Mapping Algorithm

In this section, we introduce the Single-Shot mapping algorithm (SS-MAP). It is the starting point of the optimization heuristic presented later. As previously stated, the SS-MAP is based on both the application and the hardware's topological order. It is capable of accepting one or all topological sortings of the hardware graph. In the latter case, the algorithm will compute the mapping for all the topological sortings and select the one with the lowest computation cost. We select this approach because we target an irregular platform, and there may be several topological sortings of its hardware resources. Given that we choose one topological sorting, but we may not obtain a sub-optimal mapping because the firsts tasks may be allocated to resources that are not the best allocation. Using all possible topological sortings, we ensure that we test all possible allocations. In both cases, one or all topological sortings, the methodology described in the following sections holds.

6.1.1 Methodology

The SS-MAP mapping algorithm may be divided in two steps : **initialization** and **allocation process**.

Initialization

In this first step, we can decide whether the mapping algorithm will use one or all of the hardware graph's topological sortings. Subsequently, the allocation process can be adapted to this choice quite directly.

From a couple of application (G_{APP}) and hardware (G_{HW}) graphs we build two subgraphs $G'_{APP} = (T', D')$ where $T' = \{t_i \in G_{APP} \mid type(i) \neq interface\}$ and $G'_{HW}(S', K')$ with $S' = R^P$. Thus, the algorithm starts to compute one random topological sorting of the application subgraph G'_{APP} and one random or all topological sortings of a subgraph G'_{HW} (lines 1 to 2 of Algorithm 1, that depicts the processing flow).

This is done using Kahn's Algorithm [194]. This step produces two lists, L_{HW} and L_{APP}. Each of them contains ordered numbers (indexes) of the input subgraphs nodes. The first, L_{HW}, represents the processing resources organization. If the mapping algorithm has to use all the topological sortings, at each mapping iteration, one will be stored in L_{HW}. The second, L_{APP}, represents the data dependence between tasks defined by the application model.

Figure 6.1 illustrates the initialization process on a pair of application graph (top left) and hardware graph (bottom left). In this example, several topological sortings of the hardware graph exist. One is selected randomly. On the contrary, only one topological sorting exist for the application graph.

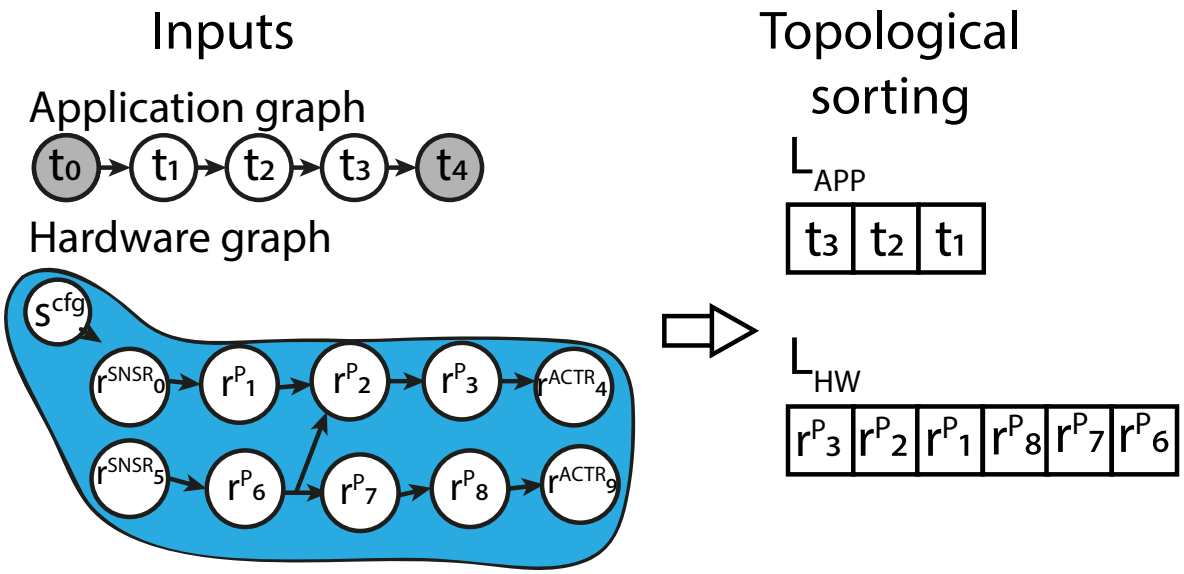


Figure 6.1 – Flow diagram of initialization.

Allocation Process

The allocation process aims to map all the tasks of an application on the available and compatible hardware resources. It explores both L_{APP} and L_{HW}, using one element of each list at a time and verifies if the current $t_i \in L_{APP}$ can be allocated on $r_j^P \in L_{HW}$. If r_j^P does not match, we dequeue another element of L_{HW}.

Figure 6.2 illustrates the principle behind the allocation process. For this illustration, let's consider a homogeneous application graph, where all the application tasks belong to the same *type* and have the same parameters (p). Consequently, we use a homogeneous hardware graph, where all the processing resources can implement the same $\mathcal{T}$ with the same parameters (Π).

We proceed to order both the application and hardware graphs topologically. Next, we start the allocation process. We dequeue the first task (t_1) and map it to the first resource in L_{HW} (r_6^P). Since it is possible to allocate t_1 on r_6^P, we move to the next task. We dequeue t_2 and map it to the next resource in L_{HW} (r_7^P). Finally, we dequeue t_3 and we allocate it to the next item in L_{HW} (r_8^P). The result is the implementation graph.

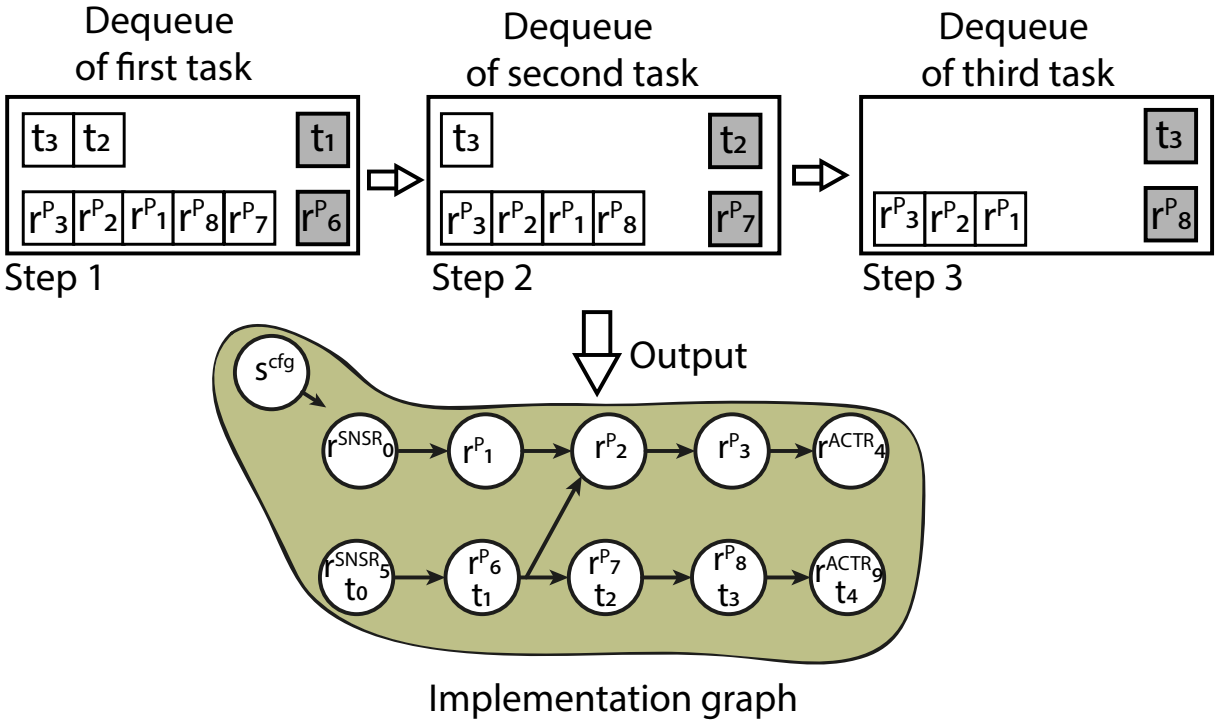


Figure 6.2 – Allocation process.

The main part of the allocation process algorithm (line 3 to end of Algorithm 1) is the function ASSIGNING in line 15 (the other two functions REALLOCATION, in line 10, and PARTITION, in line 13, are required to escape some allocation issues, they will be explained later). It aims to find a matching between a current application task $t_i \in T^{'}$ and a processing resource $r_j^P \in S^{'}$. The pseudo-code of the function is shown in Function ASSIGNING (page 104). Given that we want to allocate $t_i = (type_i, p_i)$ (see Section 2.4) on $r_j^P = (\mathcal{T}_j, \Pi_j, \mathcal{L}_j, Cfg_j)$ (see Section 3.4.4), the function in Line 3 verifies the following conditions:

- Resource can be assigned to execute the task, stated by:

$$type_i \in \mathcal{T}_j \tag{6.1}$$

- Parameters of the task matches with the set of resource parameters:

$$p_i \in \Pi_j \tag{6.2}$$

- Source of data is valid. If t_i is the successor of t_l and $type_l = interface$ (t_l is a sensor), there must exist a path where $\mathcal{P} = \{r^{sensor}, \ldots, r_j^P\}, \mathcal{P} \in G_{HW}$, where :

$$t_l \text{ is assigned to } r^{sensor} \ \wedge \ \{\forall r_k \in S | r_k \in \mathcal{P}\} \text{ no task has been allocated} \tag{6.3}$$

- Predecessors are correctly reachable. Considering the set of t_i predecessors $t_{pre} = Pred(t_i)$ already mapped to a subset of processing resources $r_{pre} \subset S^{'}$, there must exist a path $\mathcal{P}(r_{pre}, r_j^P)$ verifying:

$$\forall r_k \in \mathcal{P}(r_{pre}, r_j^P) \text{ no task has been allocated} \tag{6.4}$$

Algorithm 1 SS-MAP mapping algorithm

Input: $G'_{APP},\ G_{APP},\ G'_{HW},\ G_{HW}$
Output: L_{MAP}

```
1: New list L_APP, with the random topological sorting of G'_APP
2: New list L_HW, with the random topological sorting of G'_HW
3: New empty list L_MAP
4: New counter of failed attempts cnt_attempts = 0
5: while L_APP ≠ {∅} do
6:     dequeue t_i from L_APP
7:     done = False
8:     while not done do
9:         if cnt_attempts == |G'_HW| then
10:            L_MAP, L_APP, L_HW, done =
               REALLOCATION(t_i, L_MAP, G_APP, G_HW, L_HW, L_APP)
11:        else
12:            if L_HW == {∅} then
13:                L_HW, L_MAP = PARTITION(L_MAP, G_APP, G_HW, L_HW)
14:            else
15:                cnt_attempts, done, L_MAP =
                   ASSIGNING(t_i, r_j^P, L_MAP, G_APP, G_HW, cnt_attempts)
```

- Can successors of the current task be allocated on the descendants [1] of the candidate resource? If yes, the task is mapped to the given resource, if not, we check if there is an alternative path to a sink node from the candidate resource: let's consider $t_{suc} = Succ(t_i)$ the set of successors of t_i and $r_{des} = Desc(r_j^P)$ the set of descendants of r_j^P, then

$$\forall t_l \in t_{suc}\ \exists r_k \in r_{des} \mid type_l \in \mathcal{T}_k \wedge p_l \in \Pi_k \tag{6.5}$$

  If equation 6.5 does not hold: let's consider $r^{act} \subset R^C$ where $r^{act} = \forall r_l^{actuator}$ (it means the set of actuators of G_{HW}). Then we verify:

$$\exists \mathcal{P}_k = (r_j^P, \ldots, r_k) \mid r_k \in r^{act} \tag{6.6}$$

- Is the input degree of the candidate resource compatible with the input degree of the task or zero?

$$deg^-(t_i) = deg^-(r_j^P) \vee deg^-(r_j^P) = 0 \tag{6.7}$$

  If Equation 6.7 does not hold and if the Input degree of the processing resource is higher: let's consider $t_{pre} = Pred(t_i)$ the set of predecessors of t_i and $r_{map} \subset S'$ a subset of processing resources where $t_{pre} \rightarrow r_{map}$. Additionally, $r_{pre} = Pred(r_j^P)$ is the set of predecessors of r_j^P, then we verify:

$$\nexists r_k^P \in r_{pre} \mid allocates\ a\ t_l \notin t_{pre} \tag{6.8}$$

- Is the output degree of the candidate resource equal or higher than the output degree of the current task or zero?

$$deg^+(t_i) \leq deg^+(r_j^P) \vee deg^+(r_j^P) = 0 \tag{6.9}$$

[1] Given a node v, the set of its descendants is composed of all the nodes that are reachable from v.

If Equation 6.9 does not hold and if the output degree of the candidate resource is lower, we check that the sum of output degrees of its successors plus the output degree of the successors of the successors is higher or equal than the output degree of the current task. To implement this, let's consider $r_{des} = N^+(N^+(r_j^P))$[2].

$$deg^+(t_i) \leq \sum_{k=1}^{|r_{des}|-1} deg^+(r_k^P \in r_{des}) \tag{6.10}$$

If the above conditions are validated, we consider that the allocation of t_i on r_j^P is valid and proceed to the allocation (Lines 4 to 7).

```
1:  function ASSIGNING(t_i, L_HW, L_MAP, G_APP, G_HW, cnt_attempts )
2:      dequeue r_j^P from L_HW
3:      if t_i can be mapped on r_j^P then
4:          map t_i on r_j^P
5:          done = True
6:          cnt_attempts = 0
7:          Store mapping in L_MAP
8:      else
9:          done = False
10:         cnt_attempts = cnt_attempts + 1
11:     return cnt_attempts, done, L_MAP
```

The allocation process goes through all the tasks and tries to use all the resources available. However, some issues may appear during the mapping, we present now two functions that will prevent these issues.

Algorithm Inconveniences

During the mapping, we have to deal mainly with three issues: *Sub-optimal correspondence between L_{HW} and L_{APP}*, *Availability of the Hardware resources* and *Matching fails*. The first two issues are solved using the function PARTITION. The pseudo-code of this function is presented below.

```
1:  function PARTITION (L_MAP, G_APP, G_HW, L_HW )
2:      New list Paths_HW, that consists of the independent datapaths in G_HW
3:      New empty list nodes_available
4:      for each path ∈ Paths_HW do
5:          if none r_j^P ∈ path allocates a task or copy operation then
6:              Store all r_j^P ∈ path in nodes_available
7:      if nodes_available == {∅} then
8:          Creation of a new time slot
9:      else
10:         Store the topological ordering of nodes_available in L_HW
11:     return L_HW, L_MAP
```

a) *Sub-optimal correspondence between L_{HW} and L_{APP}*. This issue comes from the multiplicity of the topological sorting results and appears as a false lack of resources. Consider the application and hardware graphs of Figure 6.3 as inputs. The pipelined application consists of

[2] $N^+(r_j^P)$ is the set of out-going neighbors (successors) of r_j^P.

three heterogeneous tasks (t_1, t_2, t_3). The hardware architecture consists of two possible independent datapaths of three resources each (r_1^P, r_2^P, r_3^P and r_6^P, r_7^P, r_8^P). Following the Table 6.1 assume that t_1 can be allocated only on r_1^P, t_3 on r_7^P, and t_2 on $r_2^P, r_3^P, r_6^P, r_8^P$. During the initialization stage, we produce randomly the topological sorting of the hardware graph showed in Figure 6.3.

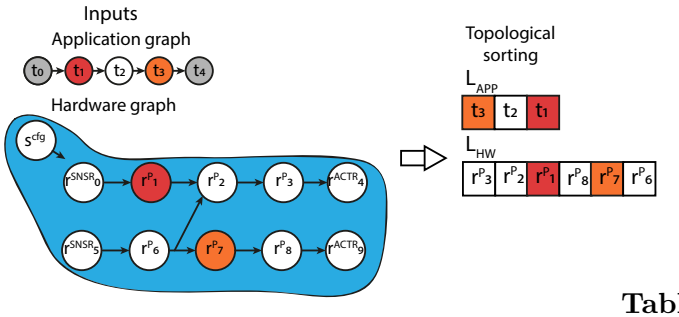


Figure 6.3 – Initialization process.

Task	Processing resource able to execute the task
t_1	r_1^P
t_2	$r_2^P, r_3^P, r_6^P, r_8^P$
t_3	r_7^P

Table 6.1 – Correspondence between tasks and resources

Then we proceed with the allocation process, depicted in Figure 6.4. The first task to map is t_1. The algorithm goes through the elements of L_{HW} until it reaches r_1^P (the only one possible thanks to Table 6.1) and allocates the task to this resource. Notice the datapath r_6^P, r_7^P, r_8^P has been dismissed, but no task has been allocated to any of its resources. The next task to allocate is t_2. The algorithm map it to r_2^P, which is the first item of L_{HW}. Finally, we try to allocate t_3. At this moment the last item of L_{HW} is r_3^P which is not a valid allocation for t_3. This is an issue, the algorithm fails. This is why at this moment, the mapping algorithm calls the Function Partition (lines 4 to 10). The function verifies if there is any datapath without a task mapped. If there is a datapath available, the function re-add its nodes to L_{HW} and continues with the mapping. If the function cannot find available data, it will split G_{APP} into sub-graphs and allocate the remaining tasks to a different time slot. In Figure 6.4, we can see that after the call of Partition, L_{HW} has resources again, and the mapping process can restart. In the end, t_3 is allocated to r_7^P, and we get the final implementation graph.

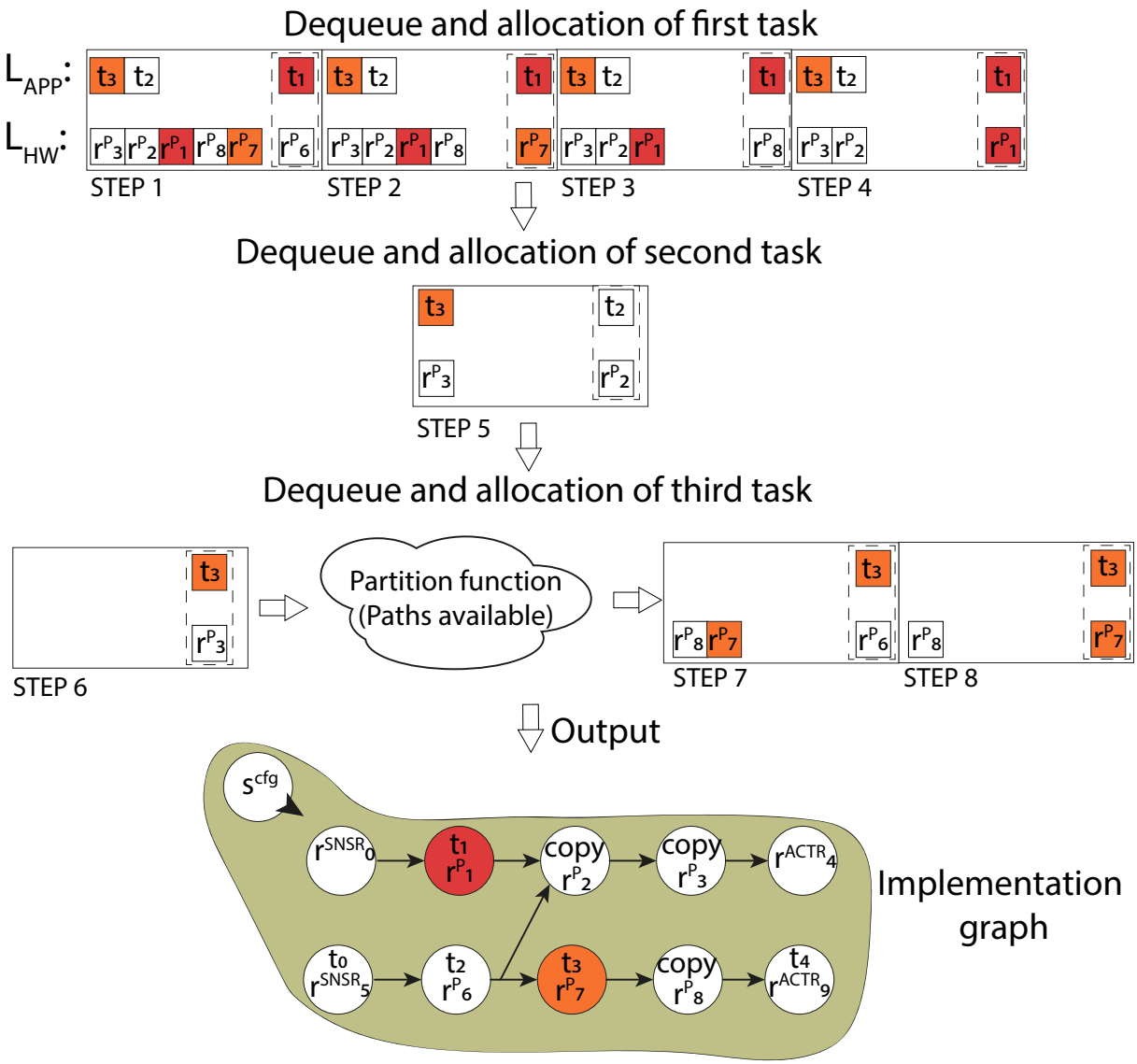


Figure 6.4 – Allocation process.

b) *Availability of the Hardware resources.* The application mapping requires more resources than the available in the hardware model. Consider the application and hardware graphs of Figure 6.5. These inputs are similar to the previous example. However, now the application graph has one more task (t_4), which can only be executed in r_1^P (Table 6.2).

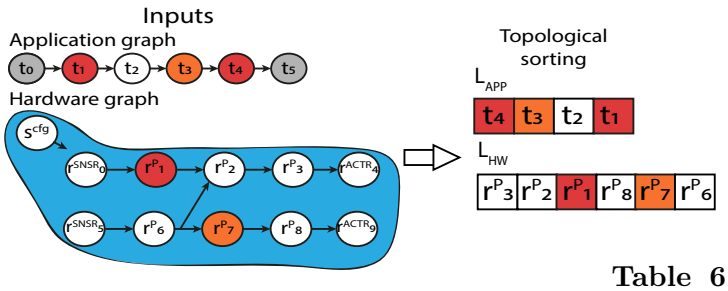


Figure 6.5 – Initialization process.

Task	Processing resource able to execute the task
t_1	r_1^P
t_2	r_2^P, r_3^P
t_3	r_7^P
t_4	r_1^P

Table 6.2 – Correspondence between tasks and resources

The firsts steps (1 to 8) are the same as stated for the *Sub-optimal correspondence between L_{HW} and L_{APP}* issue. After the allocation of t_3, we continue with t_4. In L_{HW}, the only element left is r_8^P, which is not compatible with the current task. At this moment, there are no more resources to use. To solve the problem we call the function PARTITION. Again this function verifies the datapaths available, and because all of them are occupied (Line 7), the function splits G_{APP} (line 8) into sub-graphs and duplicate the hardware graph (Step 10). Next, the mapping algorithm will try to schedule them into time slots. This process is depicted in Figure 6.6.

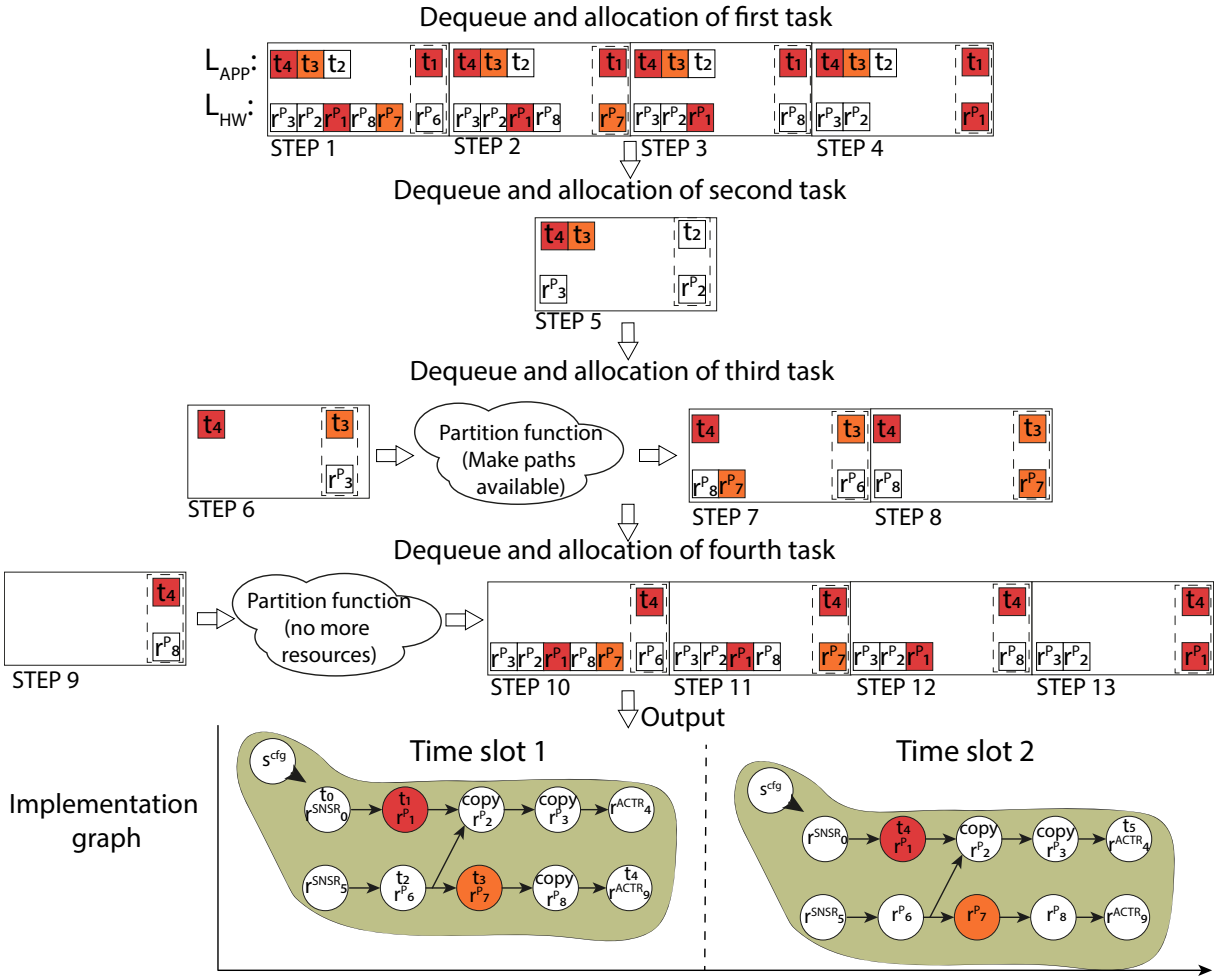


Figure 6.6 – Allocation process.

c) *Matching fails.* We observe this issue by an unsuccessful search of a resource for a particular task. We solve this issue using the function REALLOCATION. The function REALLOCATION is a modification of the backtracking algorithm presented by Lu et al. [195]. The function removes the mapping of the conflicting task's predecessor, re-add the task and the resource to their respective list and restart the mapping algorithm.

1: **function** REALLOCATION(t_i, L_{MAP}, G_{APP}, G_{HW}, L_{HW}, L_{APP})
    /*t_i is the conflicted node of G_{APP} being mapped*/
2: *New list Predecessors, predecessors of* t_i
3: **for** *each task* $t_k \in Predecessor$ **do**
4: *Locate the resource* r_j^P *that allocates* t_k
5: *Remove the allocation* (t_k, r_j^P) *from* L_{MAP}
6: *Reintegrate* t_k *into* L_{APP}
7: *Reintegrate* r_j^P *into* L_{HW}
8: $done = True$
9: **return** L_{MAP}, L_{APP}, L_{HW}, *done*

The partial results of the mapping and the overall mapping are stored in a list called L_{MAP}. This list contains the parameters assigned to each resource during the mapping. L_{MAP} follows similar structure as G_{MAP}, because it consists of lists (subgraphs in the case of G_{MAP}) and each list represents a time slot. Each time slot elements are equal to the resources available ($|S^{'}|$). The methodology's final step is the creation of G_{MAP}, which is obtained by parsing L_{MAP}. G_{MAP} will collect all the information contained in L_{MAP}.

6.1.2 Discussion

The SS-MAP algorithm searches for a feasible mapping of an application onto hardware. It provides a reasonable mapping with very little exploration time. It is equipped with a meta-heuristic, in which we use all the topological sortings of the subgraph $G_{HW}^{'}$ in a bigger loop with the performance evaluation (Figure 6.7). Then the best mapping with the lowest computing cost is selected.

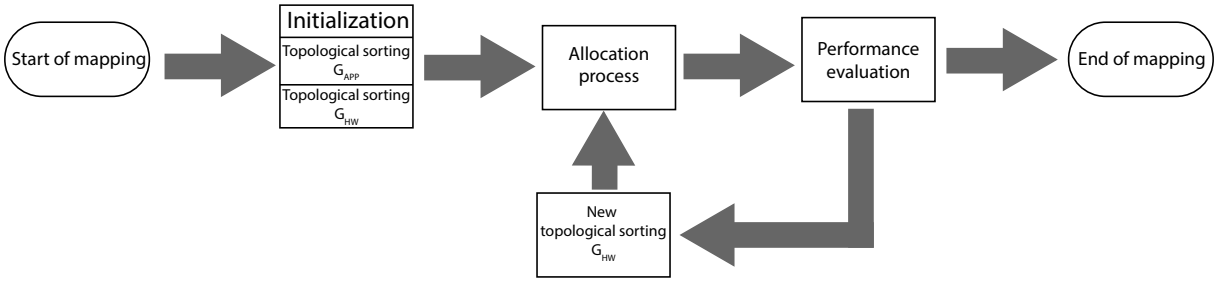


Figure 6.7 – Flow diagram of the SS-MAP algorithm (all topological sortings of G_{HW}).

Even if we provide the algorithm with the tools to overcome some mapping inconveniences described in Section 6.1.1, there is still a need for a better approach. This algorithm may not work well with multisensor systems. It may encounter a deadlock during the allocation process that may need to make use of the backtracking feature, representing an increment in the exploration time. To overcome the SS-MAP algorithm's problems, we propose to use a variants heuristic that considers the latency of the resources and the resources' topological distance. Also, the decision of mapping a task to a resource is local. We require to increase the magnitude of this decision. In the following section, we present a heuristic that considers the task's successors and the computing latency of the processing resources. This will improve the mapping results.

6.2 Topology-Aware Mapping Algorithm

After the SS-MAP algorithm presented previously, we now introduce the Topology-Aware mapping algorithm (TA-MAP), an enhancement to the last algorithm in the form of a heuristic optimization.

The proposed heuristic optimization is based on look-ahead techniques. The purpose of a look-ahead approach is to foresee the effects of a particular decision. The main goal is to evaluate the allocation of a specific task onto a particular resource. The heuristic evaluates not only the allocation of the current task but also the allocation of its successors. The heuristic choices are based on uses elements such as the topological distance, the computing latency, and the number of possible matching resources.

6.2.1 Methodology

In this section, we explain the general principle of the algorithm. It is based on list scheduling and integrates a heuristic optimization. This heuristic is used when we have to choose between two or more resource candidates. As for the SS-MAP, the inputs of the algorithm are two hypergraphs representing the application (G_{APP}) and the hardware model (G_{HW}). The flow process is also divided in two stages: **initialization** and **allocation process** (Figure 6.8).

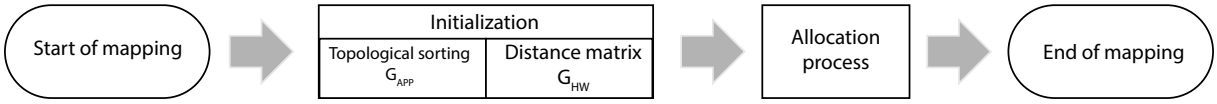


Figure 6.8 – Flow diagram of the TA-MAP algorithm.

Initialization

Similar to the SS-MAP algorithm (see Section 6.1), we define two subgraphs. An application subgraph $G'_{APP} = (T', D')$ where $T' = \{t_i \in G_{APP} \mid type(i) \neq interface\}$ and a hardware subgraph $G'_{HW}(S', K')$ with $S' = R^P$. We start this stage creating a **Distance matrix** which is going to be used for the heuristic equation. The **Distance matrix** represents the distances, according to the number of nodes, between each couple of nodes of S'. When there is no simple path between a couple of nodes we represent this with an infinite value. We use the Floyd-Marshall algorithm [196] to compute the **Distance matrix**. We illustrate this computation in Section 6.2.2.

Next, we compute the topological sorting of G'_{APP} and store the result in a list (L_{APP}).

Allocation Process

The goal of the allocation process is to allocate all tasks onto the available compatible resources. The complete TA-MAP pseudo-code is presented in Algorithm 2. Recall that L_{MAP} is a list where the partial results of the mapping and the overall mapping are stored. **MS** is the probability of mapping success function and MAP is a function that allocates t_i onto r_k^P and stores this allocation in L_{MAP}, both functions are explained in detail in the following section.

Algorithm 2 TA-MAP algorithm

Input: G'_{APP}, G_{APP}, G'_{HW}, G_{HW}
Output: L_{MAP}
1: *New array distance_matrix*
2: *New node list* L_{APP} *sorted topological*
3: **while** $L_{APP} \neq \{\emptyset\}$ **do**
4: *dequeue* t_i *from* L_{APP}
5: *New list* $w^{HW} \leftarrow$ *candidates* $S'(G'_{HW}, t_i)$
6: *For all resources* r_j^P *in* w^{HW} : *compute* $\mathbf{MS}(r_j^P)$
7: $r_k^P \leftarrow max(\mathbf{MS}(w^{HW}))$
8: $MAP(t_i, r_k^P, L_{MAP})$

The first step of the allocation process is to dequeue the first task of L_{APP} (line 4). Following this, we select the possible resources candidates (line 5). This selection is according to the following rules (Equations 6.11 and 6.22 implemented in function *candidates* of line 5):

- At the first time, the set of resources candidates (w^{HW}) will be the source nodes of $S^{'}$.

$$w^{HW} = \{r_k^P \in S^{'} \mid deg^{-}(r_k^P) = 0\} \tag{6.11}$$

- Following, the resources candidates will be the successors of the resource or resources that allocate the predecessor or predecessors of the current task. Let's consider $t_{pre} = Pred(t_i)$ the set of predecessors of the current task t_i and r_{pre} the set of processing resources where $t_{pre} \rightarrow r_{pre}$, then:

$$w^{HW} = N^{+}(r_{pre}) \tag{6.12}$$

Recall that $N^{+}(r_{pre})$ is the set of out-going neighbors (successors) of r_{pre}. After the selection of the set of resource candidates (w^{HW}) we prune the list. For each candidate $r_j^P \in w^{HW}$, we verify the compatibility with the current task t_i. We use the same rules as the ones presented in Section 6.1.1:

- Resource can be assigned to execute the task, stated by:

$$type_i \in \mathcal{T}_j \tag{6.13}$$

- Parameters of the task matches with the set of resource parameters:

$$p_i \in \Pi_j \tag{6.14}$$

- Source of data is valid. If t_i is the successor of t_l and $type_l = interface$ (t_l is a sensor), there must exist a path where $\mathcal{P} = \{r^{sensor}, \ldots, r_j^P\}, \mathcal{P} \in G_{HW}$, where :

$$t_l \text{ is assigned to } r^{sensor} \ \wedge \ \{\forall r_k \in S | r_k \in \mathcal{P}\} \text{ no task has been allocated} \tag{6.15}$$

- Predecessors are correctly reachable. Considering the set of t_i predecessors $t_{pre} = Pred(t_i)$ already mapped to a subset of processing resources $r_{pre} \subset S^{'}$, there must exist a path $\mathcal{P}(r_{pre}, r_j^P)$ verifying:

$$\forall r_k \in \mathcal{P}(r_{pre}, r_j^P) \text{ no task has been allocated} \tag{6.16}$$

- Successors of the current task can be allocated on the descendants of the candidate resource. If yes, the task is mapped to the given resource, if not, we check if there is an alternative path to a sink node from the candidate resource: let's consider $t_{suc} = Succ(t_i)$ the set of successors of t_i and $r_{des} = Desc(r_j^P)$ the set of descendants of r_j^P, then

$$\forall t_l \in t_{suc} \ \exists r_k \in r_{des} \mid type_l \in \mathcal{T}_k \wedge p_l \in \Pi_k \tag{6.17}$$

If Equation 6.17 does not hold: let's consider $r^{act} \subset R^C$ where $r^{act} = \forall r_l^{actuator}$ (it means the set of actuators of G_{HW}). Then we verify:

$$\exists \mathcal{P}_k = (r_j^P, \ldots, r_k) \mid r_k \in r^{act} \tag{6.18}$$

- Input degree of the candidate resource is compatible with the input degree of task or zero.

$$deg^{-}(t_i) = deg^{-}(r_j^P) \vee deg^{-}(r_j^P) = 0 \tag{6.19}$$

If Equation 6.19 does not hold and if the Input degree of the processing resource is higher: let's consider $t_{pre} = Pred(t_i)$ the set of predecessors of t_i and $r_{map} \subset S^{'}$ a subset of processing resources where $t_{pre} \rightarrow r_{map}$. Additionally, $r_{pre} = Pred(r_j^P)$ is the set of predecessors of r_j^P, then we verify:

$$\nexists r_k^P \in r_{pre} \mid allocates\ a\ t_l \notin t_{pre} \tag{6.20}$$

- Is the output degree of the candidate resource equal or higher than the output degree of the current task or zero?

$$deg^{+}(t_i) \leq deg^{+}(r_j^P) \vee deg^{+}(r_j^P) = 0 \tag{6.21}$$

If Equation 6.21 does not hold and if the output degree of the candidate resource is lower, we check that the sum of output degrees of its successors plus the output degree of the successors of the successors is higher or equal than the output degree of the current task. Let's consider $r_{des} = N^{+}(N^{+}(r_j^P))$[3].

$$deg^{+}(t_i) \leq \sum_{k=1}^{|r_{des}|-1} deg^{+}(r_k^P \in r_{des}) \tag{6.22}$$

After the verification, we have left a set of feasible candidates or an empty set. For the first case, we obtain the mapping success probability of each feasible candidate according to Equation 6.23. Next, we select the candidate with the greatest probability of mapping success and map the task to it. In the second case, if we don't have any feasible candidate, we need to check which path of G_{HW} is suitable for the current task. This means the path where we will have a higher probability to map the current task. For this, we use the same Equation 6.23, but instead of the successor tasks, we use the current task as input. After this computation, we select the path with the higher probability value and obtain the possible candidates from it. Also, if the task to allocate is a sink task, we compute the mapping success with the task as input.

An interesting characteristic of the algorithm is that it allows recomputation and creates the time slots automatically. Effectively, if the resulting list w^{HW} is an empty set, the algorithm enters into a function where it checks if there is any datapath available in G_{HW}. If there is a datapath available, the function will compute w^{HW} from the resources of that datapath. If there is no datapath available, the algorithm creates a time slot and computes again w^{HW}. For these purposes we use the function PARTITION introduced in Section 6.1.1.

Heuristic Optimization

Given $t_i \in T^{'}$ that represents the task to map. Let $w^{APP} = (t_1, t_2, \ldots, t_x)$, be the set of successors of t_i.

Let $w^{HW} \in S^{'}$ represent a subset of n nodes $(r_1^P, r_2^P, ..., r_n^P)$ of interest (window of possible candidates resources). For each node of interest, we compute the set of its descendants F_n. Let $F_n \in S^{'}$ and $\mid F_n \mid = m$.

The selection of the best allocation for a particular task is determined by our probability of mapping success function **MS** (line 6). This probability is based on the value of the computing latency, the number of resources that may be used to allocate a particular task and the topological distance to the very next processing resource that can allocate a successor task:

[3] $N^{+}(N^{+}(r_j^P))$ is the set of out-going neighbors (successors) of the successors of r_j^P.

$$MS_j = \sum_{b=1}^{|w^{APP}|} \frac{CL^w \cdot C \cdot \mid Q_b \mid}{CL^a \cdot d \cdot \sum_{k=1}^{|w^{HW}|} \mid F_k \mid} \tag{6.23}$$

where $1 \leq j \leq \mid w^{HW} \mid$, CL^w is the worse computing latency of the set of possible allocations in all the descendants of all candidates. Computing latency is the one that has a higher impact on the computing cost. With the consideration of one sample, we can assess the performance of a given resource when allocating a given task. C is the length of the critical path of the sub-graph $G^F(S^F, K^F) \subseteq G'_{HW}(S', K')$, where $S^F = F_1 \cup F_2 \cup \cdots \cup F_n$ and $K^F \subseteq \{(r_i^P, r_l^P) | (r_i^P, r_l^P) \in S^F \wedge r_i^P \neq r_l^P\}$. Q_b is given by:

$$Q_b = \{r_i^P \in F_j \mid \ type(t_b) \in \mathcal{T}(r_i^P)\} \tag{6.24}$$

CL^a is the average computing latency of Q_b and d is the shortest distance (from the distance matrix, see Section 6.2.1) to the next node $r_k^P : type(t_b) \in \mathcal{T}(r_k^P)$.

After the computation of all mapping success probability onto each candidate resource, we select the one that maximizes **MS** (line 7) and map the task t_i on this best resource (line 8, function **MAP**). Finally, we loop again while there is a task to map in the list L_{APP}. A graphical representation of the heuristic optimization is shown in Figure 6.9. A simple application and hardware graphs are used. The subgraph G^F is depicted in the bottom right with its critical path C highlighted.

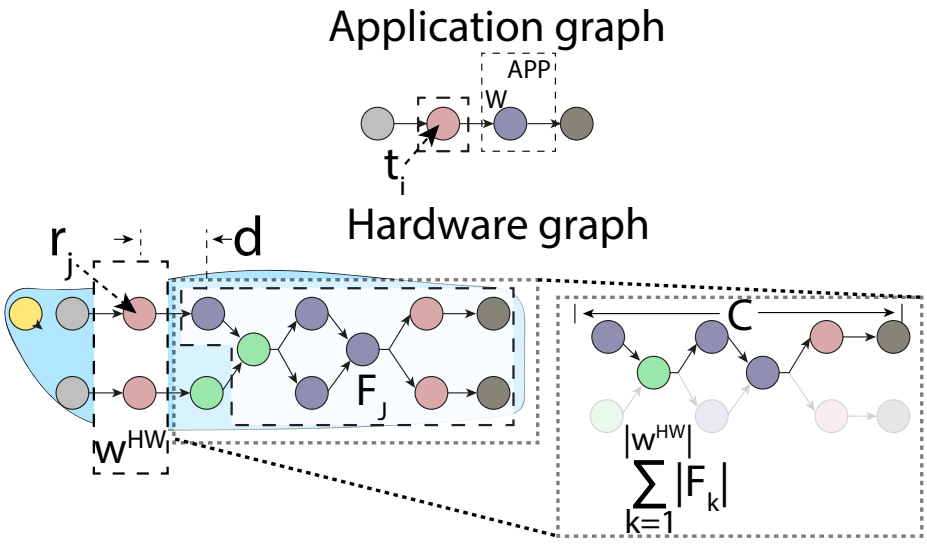


Figure 6.9 – Graphical representation of the heuristic optimization.

6.2.2 Illustrations of the TA-MAP Principle

Example 1

Consider the application graph depicted in Figure 6.10. It consists of one sensor, one actuator, and three tasks. Table 6.3 shows the description of each node. For didactic purposes, we neglect the parameters of each task in these illustrations.

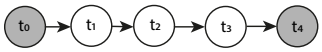


Figure 6.10 – Example application graph 1.

Name of task	$type_i$
t_0	interface
t_1	task0
t_2	task1
t_3	task2
t_4	interface

Table 6.3 – Application example 1 parameters

Consider the hardware graph depicted in Figure 6.11. Table 6.4 makes a summary of the features of the processing elements. $\mathcal{L}_i^{IN}$ and $\mathcal{L}_i^{CL}$ are in clock cycles.

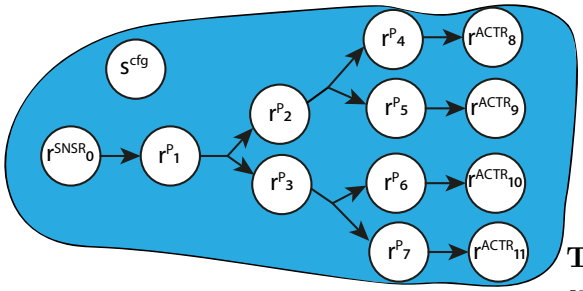


Figure 6.11 – Example hardware graph 1.

	$\mathcal{T}_i$	$\mathcal{L}_i$	
		$\mathcal{L}_i^{IN}$	$\mathcal{L}_i^{CL}$
r_1	task0	2	2
r_2^P	task1	2	2
	task2	2	2
r_3^P	task1	2	2
	task2	2	2
r_4^P	task1	2	3
	task2	2	3
r_5^P	task1	2	1
	task2	2	1
r_6^P	task0	2	3
	task2	2	3
r_7^P	task1	2	3
	task2	2	3

Table 6.4 – Processing resources features for example 1, for readability the Π of r^P has been removed.

The first step of the algorithm is the *initialization*. We compute the topological sorting of G'_{APP}. The result is stored in L_{APP}. The resulting topological sorting is

$$L_{APP} = (t_1, t_2, t_3)$$

Notice that the topological sorting does not consider the tasks with $type = interface$, which are not included in the set T'. Next, we use the Floyd-Marshall algorithm to build a matrix of the shortest paths between all the processing resources in G'_{HW}. Table 6.5 shows the resulting matrix.

Table 6.5 – Distance matrix for example 1

	r_1^P	r_2^P	r_3^P	r_4^P	r_5^P	r_6^P	r_7^P
r_1^P	0	1	1	2	2	2	2
r_2^P	∞	0	∞	1	1	∞	∞
r_3^P	∞	∞	0	∞	∞	1	1
r_4^P	∞	∞	∞	0	∞	∞	∞
r_5^P	∞	∞	∞	∞	0	∞	∞
r_6^P	∞	∞	∞	∞	∞	0	∞
r_7^P	∞	∞	∞	∞	∞	∞	0

Next, we look for the first group of possible candidates, which are the nodes with zero input degree, $w^{HW} = \{r_1^P\}$ (Figure 6.12).

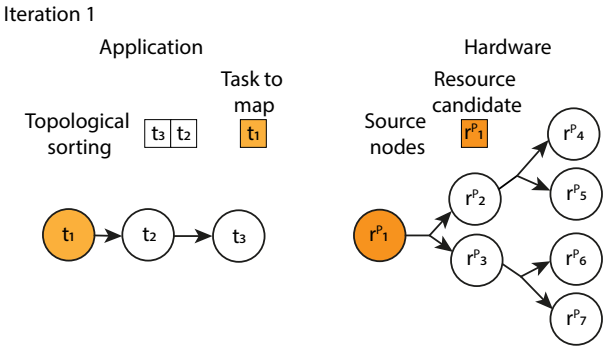


Figure 6.12 – First group of possible candidates for example 1.

Now, we start the mapping algorithm. The first task to map is t_1. As there is only one candidate, we verify if the resource candidate is able to implement the task, and as it is, we map the task onto this resource.

The second iteration starts with selecting the next task to map, which is t_2. Next (line 5 of Algorithm 2), we obtain the resource candidates: $w^{HW} = \{r_2^P, r_3^P\}$. After the selection of the resource candidates (line 6), we also compute the descendants of all of them. The resulting descendants are $F_2 = \{r_4^P, r_5^P\}$ and $F_3 = \{r_6^P, r_7^P\}$. Figure 6.13 shows a graphical representation of these lists.

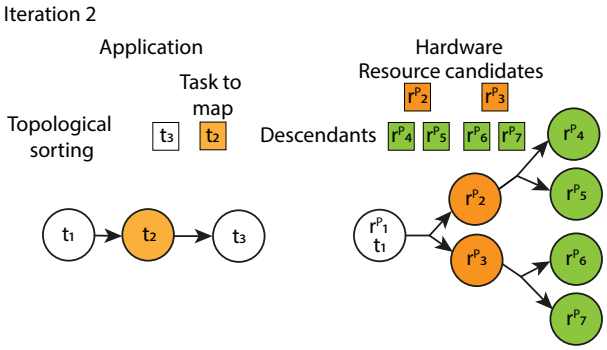


Figure 6.13 – Second group of possible candidates for example 1.

The next step is to compute the probability of mapping success for each resource candidate.

$$\text{for } r_2^P : MS_2 = \frac{(CL^w = 3) \times (C = 1) \times (|Q_b| = 2)}{(CL^a = 2) \times (d = 1) \times (\sum |F| = 2 + 2)} = 0.75$$

$$\text{for } r_3^P : MS_3 = \frac{CL^w = 3) \times (C = 1) \times (|Q_b| = 1)}{(CL^a = 3) \times (d = 1) \times (\sum |F| = 2 + 2)} = 0.25$$

The selected resource candidate is r_2^P because it obtains the greatest probability of mapping success. We map the task to this resource.

The following and last task to map is t_3 (Figure 6.14). The resource candidates for the mapping of this task are $w^{HW} = \{r_4^P, r_5^P\}$.

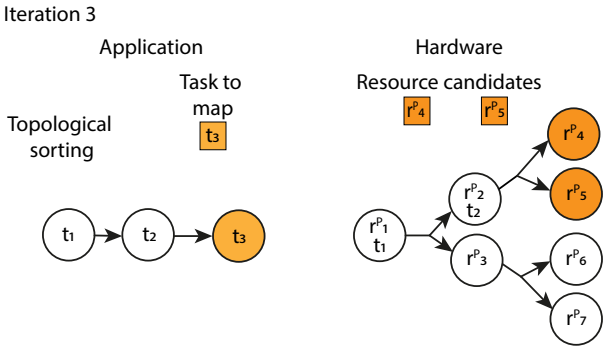


Figure 6.14 – Third group of possible candidates for example 1.

Because there are no more tasks to map, we do not compute the descendants of these group of candidates. We compute the probability of mapping success with these same candidates as input. The result will help us identify the best candidate for this particular task in both latency and topological distance terms. The computations are as follows.

$$\text{for } r_4^P : MS_4 = \frac{3 \cdot 1 \cdot 1}{3 \cdot 1 \cdot 2} = 0.5$$

$$\text{for } r_5^P : MS_5 = \frac{3 \cdot 1 \cdot 1}{1 \cdot 1 \cdot 2} = 1.5$$

With the computation of each resource candidates' mapping success, we can see that the best candidate is r_5^P. We select this resource to allocate the task, and we end the mapping.

Example 2

Let's consider a different example. Figure 6.15 shows an application graph that consists of two parallel pipelines of tasks. Each pipeline consists of one sensor, one actuator, and three tasks. Table 6.6 makes a summary of the characteristics of all the nodes of the application graph.

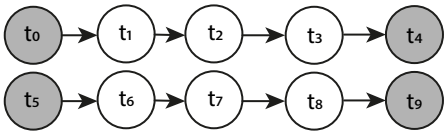


Figure 6.15 – Application graph of example 2.

Name of task	$type_i$
t_0	interface
t_1	task1
t_2	task1
t_3	task1
t_4	interface
t_5	interface
t_6	task2
t_7	task2
t_8	task2
t_9	interface

Table 6.6 – Application example 2 parameters.

For this example 2, consider the hardware graph exposed in Figure 6.16. It consists of two parallel independent pipelines of processing resources. Each pipeline has its pair of communication resources (a sensor and an actuator). Table 6.7 shows the characteristics of the processing resources of the hardware graph for example 2.

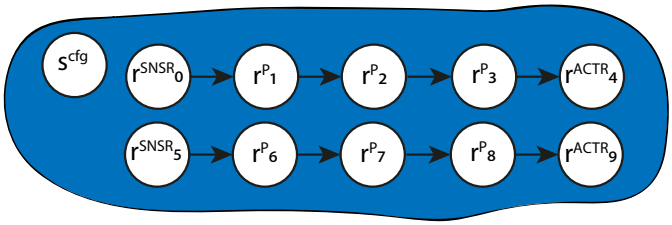


Figure 6.16 – Hardware graph of example 2.

	$\mathcal{T}_i$	$\mathcal{L}_i$	
		$\mathcal{L}_i^{IN}$	$\mathcal{L}_i^{CL}$
r_1^p	task1	3	2
	task2	3	4
r_2^P	task1	3	2
	task2	3	4
r_3^P	task1	3	2
	task2	3	4
r_6^P	task1	3	4
	task2	3	2
r_7^P	task1	3	4
	task2	3	2
r_8^P	task1	3	4
	task2	3	2

Table 6.7 – Processing resources features for example 2.

We start the mapping with the *initialization* stage. We compute the topological sorting of G_{APP}, and the resulting list is

$$L_{APP} = (t_1, t_2, t_3, t_6, t_7, t_8)$$

And the distance matrix of the processing resources is the following.

Table 6.8 – Distance matrix for example 2

	r_1^P	r_2^P	r_3^P	r_6^P	r_7^P	r_8^P
r_1^P	0	1	2	∞	∞	∞
r_2^P	∞	0	1	∞	∞	∞
r_3^P	∞	∞	0	∞	∞	∞
r_6^P	∞	∞	∞	0	1	2
r_7^P	∞	∞	∞	∞	0	1
r_8^P	∞	∞	∞	∞	∞	0

Next, we start the *allocation process* of the algorithm. We select the first task to map, t_1 and the first group of resource candidates (Figure 6.17). Because we are starting the *allocation process* we select the source nodes of G_{HW} without considering the communication resources. The resulting list is $w^{HW} = \{r_1^P, r_6^P\}$. Next we obtain the descendants of each resource candidates. The descendants are $F_1 = \{r_2^P, r_3^P\}$ and $F_6 = \{r_7^P, r_8^P\}$.

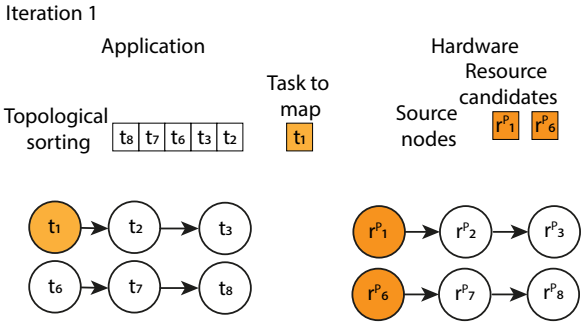


Figure 6.17 – First iteration for example 2.

The next step is compute the probability of mapping success for each candidate.

$$\text{for } r_1^P : MS_1 = \frac{4 \cdot 3 \cdot 2}{4 \cdot 1 \cdot 4} = 1.5$$

$$\text{for } r_6^P : MS_6 = \frac{4 \cdot 3 \cdot 2}{2 \cdot 1 \cdot 4} = 3$$

Iteration 2: the selected resource is r_6^P. We allocate the task to this resource and continue with the mapping. This iteration (Figure 6.18) only considers one resource candidate, r_7^P we verify that we can use it and map the task onto it.

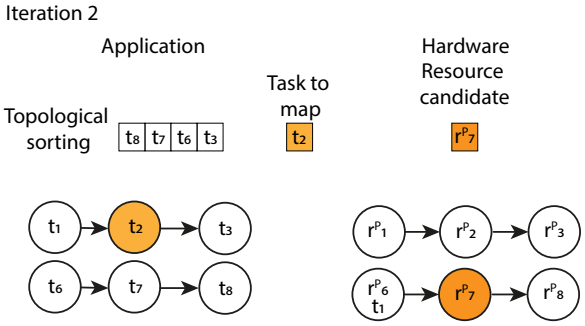


Figure 6.18 – Second iteration of example 2.

Iteration 3: the following iteration (Figure 6.19) also has one candidate, r_8^P. We perform the same operation as previously. We verify that we can use the resource and map the task to it.

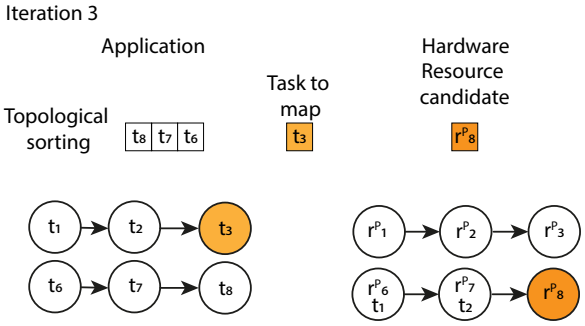


Figure 6.19 – Third iteration of example 2.

Iteration 4: the next task to map is t_6 (Figure 6.20), and as it is a source node of its pipeline, the selected candidates will be the available source nodes of the subgraph composed only by the processing resources. In this case, the resource candidate will be $w^{HW} = r_1^P$. As there is only one candidate, we verify the resources' characteristics and map the task to it.

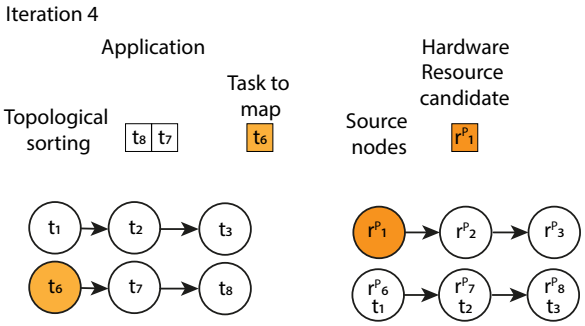


Figure 6.20 – Fourth iteration of example 2.

Iteration 5: the following iteration (Figure 6.21) consider also one single candidate $w^{HW} = r_2^P$. We also verify that we can use it and map the task to it.

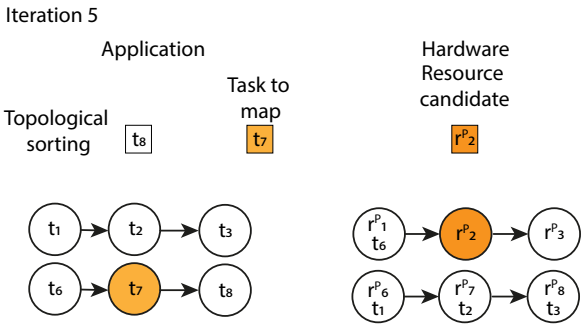


Figure 6.21 – Fifth iteration of example 2.

Iteration 6: finally, for the last task to map (Figure 6.22), we use the only remaining resource. We verify that we can use it and map the task to it.

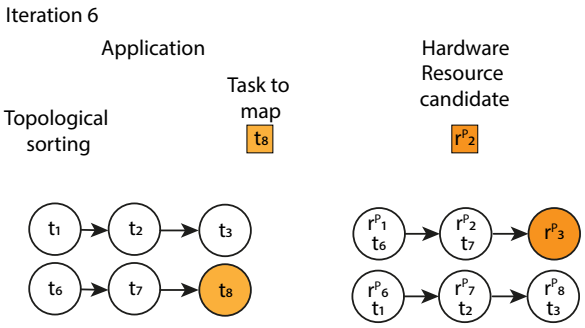


Figure 6.22 – Sixth iteration of example 2.

6.2.3 Discussion

In this section, we introduced our first heuristic, the topology-aware algorithm. It improves the decision-making of the SS-MAP by adapting look-ahead techniques to the mapping problem. The purpose of a look-ahead approach is to foresee the effects of a particular decision. It provides the possibility to weight characteristics such as latency and topological distances in the mapping and scheduling process.

6.3 Bayes-Based Heuristic Mapping Algorithm

During the mapping of an application onto a hardware platform, there is a level of uncertainty concerning the chosen resources optimality since we face to a NP-complete problem. This uncertainty may be solved using an exhaustive brute-force algorithm. However, this will possible lead to a considerable exploration time. We saw that, if we use a heuristic-based algorithm, the exploration time decreases significantly, but there is a high probability that an optimal mapping maybe not be achieved.

Any heuristic-based mapping algorithm aims to maximize the probability of choosing the best set of resources for a given application. Although an application consists of several tasks, we can not consider all of them once because we will end up with an approach similar to a brute force algorithm with a huge exploration time. We need to concentrate on a smaller set of tasks, find the best allocation, and then move dynamically to the next set of tasks until we allocate all the application's tasks.

An interesting approach is to use a conditional probability-based heuristic [182]. Conditional probability allows computing how likely an event will occur given that one or more related events had happened. Using this approach, we may analyze if a particular task's successors may be allocated to a specific resource's descendants. This allows to chose that specific resource to allocate the task. With this analysis, we may maximize the probability of obtaining an optimal or sub-optimal mapping without increasing the exploration time. A successful conditional probability method is the Bayes Theorem which is extensively used in learning methods, cloud computing, communications, medicine. The Bayes Theorem was developed by Rev. Thomas Bayes and published post-mortem by his friend Richard Price [197]. The Bayesian approach is a modeling methodology that provides a principled approach to reason and act in the context of uncertainty, and a dynamic environment [198]. This method will analyze dynamically during the mapping the allocation of each task and its successors and increase the probability of obtaining an optimal or sub-optimal mapping.

In this section, we introduce the Bayes-based heuristic mapping algorithm (BB-MAP). We propose to use a Bayes approach to enhance our previously presented heuristic. This enhancement will provide higher performance results.

6.3.1 Bayes Theorem

The canon equation of the Bayes theorem is:

$$P(H \mid E) = \frac{\overbrace{P(E \mid H)}^{Likelihood} \bullet \overbrace{P(H)}^{Posterior\ Probability}}{\underbrace{P(E)}_{Prior\ Probability}} \tag{6.25}$$

Where **P(H|E)** reads as, given the *evidence* **E**, what is the probability of the *hypothesis* **H** to happen. Bayes' theorem gives a method of revising probability estimates as additional information becomes available. The additional information is the information that is being conditioned on. The probability before additional information becomes available is referred to as the prior probability, and the revised probability using the additional information is called the posterior probability [199].

A way to describe the Bayes theorem graphically is through a bayesian belief network. A Bayesian belief network is a directed acyclic graph whose nodes are the model variables and whose links represent local causal dependencies. The network topology can be thought of as an abstract knowledge base that holds independently of the numerical assignment of conditional probabilities. A fully specified Bayesian network can be used as a probabilistic inference engine, which computes the posterior probability distribution for a set of query variables given the probability distribution for some evidence variables [200].

6.3.2 Bayes Theorem Applied to the Mapping Problem

In this section, we explain how the Bayes theorem can be used to solve the SPS-CGRA mapping problem. Similar to the topology-aware methodology presented in Section 6.2.1, the Bayes-based methodology is divided in two parts : **Initialization** and **Allocation process**. The complete pseudo-code is presented in Algorithm 3.

Algorithm 3 BB-MAP algorithm

Input: $G_{APP},\ G_{HW}, \phi, \psi$
Output: L_{MAP}
1: *New array* B, *where each* $B_b = \{r_j^P \in S^{'} \mid t_b \rightarrow r_j^P\}$ *and* $\mid B_b \mid = \phi$
2: *New array* κ, *where each* $\kappa_j = \{N^{+\psi}(r_j^{'})\}$
3: *New node list* L_{APP} *sorted topological*
4: **while** $L_{APP} \neq \{\emptyset\}$ **do**
5: *dequeue* t_i *from* L_{APP}
6: *New list* w^{HW} *with the possible candidates resources from* $S^{'}$
7: *Compute the bayesian probability for all elements in* w^{HW}
8: *Map* t_i *to the resource that maximizes the bayesian probability*
9: *Store the mapping in* L_{MAP}

Initialization

As in the previous two sections, we build two subgraphs from a couple of application and hardware graphs (G_{APP} and G_{HW}). The application subgraph $G_{APP}^{'} = (T^{'}, D^{'})$ where $T^{'} = \{t_i \in G_{APP} \mid type(i) \neq interface\}$ and the hardware subgraph $G_{HW}^{'}(S^{'}, K^{'})$ with $S^{'} = R^P$.

Given an application graph $G_{APP}^{'}$ and a hardware graph $G_{HW}^{'}$ as inputs, we compute B_b, which is the set of processing resources $r_j^P \in S^{'}$ that can allocate task t_b and provides the best performance. To obtain B_b, the algorithm computes the task's performance t_b over all the processing resources. Then an ordered list is produced, where the order corresponds to the decreasing performance. The parameter ϕ, provided by the user, will select a number ϕ of the resources of B_b, starting from the best in terms of latency performance.

$$\mid B_b \mid = \phi \tag{6.26}$$

Next, we compute κ_j, which is given by:

$$\kappa_j = \{N^{+\psi}(r_j^P)\} \tag{6.27}$$

where $\psi \in \mathbb{Z}$ is a second parameter provided by the user. It indicates the size of the neighborhood of r_j^P to be explored by the algorithm. For example, a $\psi = 1$ represents a set that includes all the successors of r_j^P. A $\psi = 2$ represents a set that includes all the successors of r_j^P and the successors of these successors.

Finally, we compute the topological sorting of $G_{APP}^{'}$ and store the result in L_{APP}.

Allocation Process

The BB-MAP algorithm's allocation process follows the same methodology as the allocation process of the Topology-Aware algorithm described in Section 6.2.1. The only difference is the part devoted to select the resource or the path. It is described in the following section.

Bayes-based Heuristic

Given a task $t_i \in T^{'}$, where $T^{'}$ is the set of vertices of $G^{'}_{APP}$, and a set of possible candidates $w^{HW} = (r_1^P, r_2^P, ..., r_n^P)$, where $w^{HW} \in S^{'}$, we aim to find which is the $r_j^P \in w^{HW}$ that maximizes the probability of allocating the successors of t_i, denoted by w^{APP}, onto the set of descendants F_j of r_j.

We solve this problem using the Bayes theorem. Hence, this probability can be formulated as: given t_i allocated onto r_j^P, what is the probability of allocate w^{APP} onto F_j. This is done by computing this probability for all the possible candidates and select the resource that maximizes the probability.

Let's formalize the principle. For each possible candidate, we compute the following equation (coming from Equation 6.25):

$$P(w^{APP} \rightarrow F_j \mid t_i \rightarrow r_j^P) = \frac{\overbrace{P(t_i \rightarrow r_j^P \mid w^{APP} \rightarrow F_j)}^{Likelihood} \bullet \overbrace{P(w^{APP} \rightarrow F_j)}^{Posterior\ probability}}{\underbrace{P(t_i \rightarrow r_j^P)}_{Prior\ probability}} \tag{6.28}$$

Where the denominator $P(t_i \rightarrow r_j^P)$ is the probability of allocating success of t_i onto r_j^P, and this is equal to:

$$P(t_i \rightarrow r_j^P) = \frac{1}{n} \tag{6.29}$$

Where n is the number of candidates ($|w^{HW}|$). We consider that all the candidates have the same probability to be selected because we prioritize the probability of allocating w^{APP} onto F_j.

The first element of the numerator $P(t_i \rightarrow r_j^P \mid w^{APP} \rightarrow F_j)$ is computed as follows:

$$P(t_i \rightarrow r_j^P \mid w^{APP} \rightarrow F_j) = \frac{P(w^{APP} \rightarrow F_j)}{n \sum_{b=1}^{n} P(w^{APP} \rightarrow F_b)} \tag{6.30}$$

Where each element of the numerator and the denominator is calculated as Equation 6.31. The second element of the numerator $P(w^{APP} \rightarrow F_j)$ is the probability of allocating w^{APP} onto F_j and it is calculated as follows:

$$P(w^{APP} \rightarrow F_j) = P(CL^B) \bullet P(d) \bullet P(Q) \tag{6.31}$$

Where $P(CL^B)$ is the probability of finding the best allocation ($r_l^P \in F_j$) for each successor of t_i, and it is calculated as follows:

$$P(CL^B) = \sum_{b=1}^{x} \frac{\mid CL \mid}{\mid F_j \mid} \tag{6.32}$$

where

$$CL = \{r_i^P \in F_j \mid r_i^P \in B_b\} \tag{6.33}$$

$P(d)$ is the probability that we can allocate each successor of t_i onto the successors of r_j^P, in other words, that the topological distance from the allocation of t_i and its successors will be the minimum. $P(d)$ is calculated as follows:

$$P(d) = \sum_{b=1}^{x} \frac{\mid d_b \mid}{\mid F_j \mid} \tag{6.34}$$

where

$$d_b = \{r_i^P \in F_j \mid type(t_b) \in \mathcal{T}(r_i^P) \wedge r_i^P \in \kappa\} \tag{6.35}$$

Finally, $P(Q)$ is the probability of finding an allocation of all the successors of t_i from the descendants of r_j^P, and it is calculated as:

$$P(Q) = \sum_{b=1}^{x} \frac{\mid Q_b \mid}{\mid F_j \mid} \tag{6.36}$$

where

$$Q_b = \{r_i^P \in F_j \mid type(t_b) \in \mathcal{T}(r_i^P)\} \tag{6.37}$$

Back to the allocation heuristic, we compute Equation 6.28 for each candidate and select the one that maximizes the probability. In Figure 6.23 we can see the Bayesian network of the SPS-CGRA mapping problem, recall that such a network can be seen as a probability inference engine (see Section 6.3.1). We highlight each part of the Equation 6.28. The first level of the network concerns the the prior probability, recall that we consider the same probability for all the candidates ($\frac{1}{|w^{HW}|}$). The following levels represent both the likelihood and the posterior probability. The main features that we consider are the compatibility between a task and a processing resource, the topological distance and the resulting latency cost. The principle is that we look for a candidate that can be compatible, be the nearest processing resource and provide the lowest latency possible.

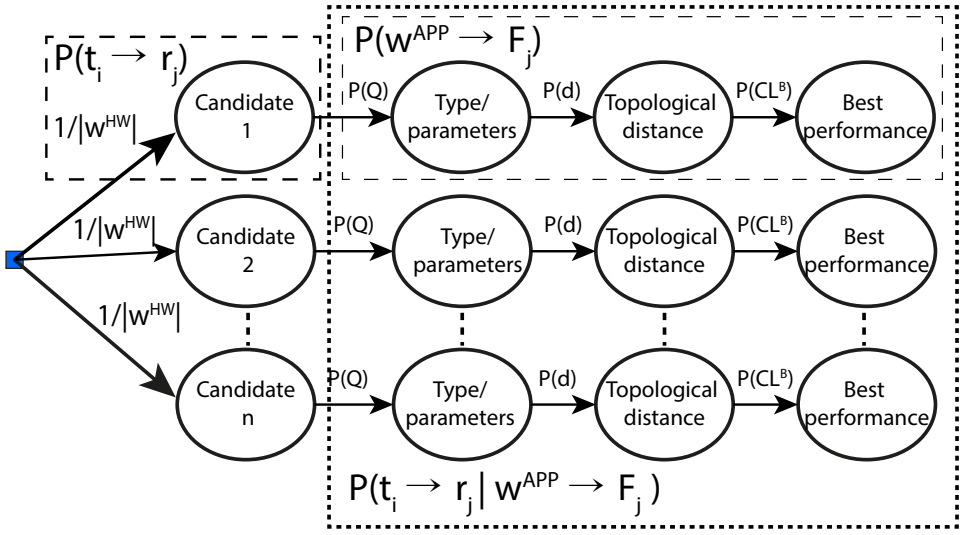


Figure 6.23 – Bayesian network.

6.3.3 Illustration of the BB-MAP Principle

Example 1

At first, we present the principle on an easy example. Consider the same example 1 given in Section 6.2.2. The application graph is depicted again in Figure 6.24. Table 6.9 shows the description of each node. For didactic purposes, we neglect the parameters of each task. Consider the hardware graph depicted in Figure 6.25. Table 6.10 makes a summary of the characteristics of the processing elements. $\mathcal{L}_i^{IN}$ and $\mathcal{L}_i^{CL}$ are given in clock cycles.

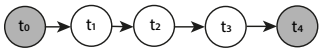


Figure 6.24 – Example application graph 1.

Name of task	$type_i$
t_0	interface
t_1	task0
t_2	task1
t_3	task2
t_4	interface

Table 6.9 – Application example 1 parameters

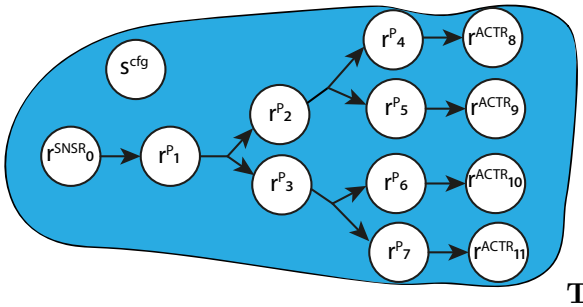


Figure 6.25 – Example hardware graph.

	$\mathcal{T}_i$	$\mathcal{L}_i$		Corresponding task
		$\mathcal{L}_i^{IN}$	$\mathcal{L}_i^{CL}$	
r_1^P	task0	2	2	t_1
r_2^P	task1	2	2	t_2
	task2	2	2	t_3
r_3^P	task1	2	2	t_2
	task2	2	2	t_3
r_4^P	task1	2	3	t_2
	task2	2	3	t_3
r_5^P	task1	2	1	t_2
	task2	2	1	t_3
r_6^P	task0	2	3	t_1
	task2	2	3	t_3
r_7^P	task1	2	3	t_2
	task2	2	3	t_3

Table 6.10 – Processing resources features for example 1

Initialization

1) The initial step is to fix ϕ and ψ arbitrarily (see Equations 6.26 and 6.27). Let's choose $\phi = 1$ and $\psi = 1$ in this example, but the user could change these parameters if not satisfied with the obtained results.

2) The second step is to extract G'_{APP} from G_{APP} and compute the topological sorting L_{APP} of the subgraph G'_{APP}. In this example this gives:

$$L_{APP} = (t_1, t_2, t_3)$$

3) Next, we compute κ (definition on Equation 6.27) and B_b (6.26) for each *type* of task. To compute each B_b we list first the number of task *types* in the application: in this example they are $task0$, $task1$ and $task2$.

For each of them we search the list of processing resources able to execute them. Among this list we select the best resource given (because we use $\phi = 1$ only one resource is selected).

$$B_{task0} = r_1^P$$
$$B_{task1} = r_5^P$$
$$B_{task2} = r_5^P$$

Now we need to compute the sets κ for each resource r_j^P, we consider $\psi = 1$, the results are

$$\kappa_{r_1^P} = r_2^P, r_3^P$$
$$\kappa_{r_2^P} = r_4^P, r_5^P$$
$$\kappa_{r_3^P} = r_6^P, r_7^P$$
$$\kappa_{r_4^P} = \emptyset$$
$$\kappa_{r_5^P} = \emptyset$$
$$\kappa_{r_6^P} = \emptyset$$
$$\kappa_{r_7^P} = \emptyset$$

Allocation Process

After the completion of the initialization stage we can start the allocation process. We define the first set of possible candidates, which is going to be r_1^P because is the only source node of the hardware graph. This resource will be used to allocate t_1, so we verify the requirements of t_1 ($type(t_1) = task0$) (Table 6.9) and we allocate t_1 onto r_1^P. Next, we take the successors of r_1^P as the next set of possible candidates (r_2^P, r_3^P), we verify that both of these processing resources are able to implement the next task (t_2). At this point to select the best processing resource we evaluate Equation 6.28 which compute the bayesian probability of mapping success for a set of resources defined by ϕ and ψ. We need to get the successor of t_2 which is t_3. We need the *type* of t_3 (which is $task2$), to identify the possible processing resources able to execute $task2$: these are r_2^P and r_3^P. So we compute $F_{r_2^P}$ and $F_{r_3^P}$ as follow: $F_{r_2^P} = (r_4^P, r_5^P)$ and $F_{r_3^P} = (r_6^P, r_7^P)$. First, we present the evaluation process of r_2^P.

$$P(t_i \rightarrow r_j^P) = P(t_2 \rightarrow r_2^P) = \frac{1}{2}$$

$$P(w^{APP} \rightarrow F_j) = P(w_2^{APP} \rightarrow F_{r_2^P}) = (P(CL^B) = \frac{1}{2}) \bullet (P(d) = \frac{2}{2}) \bullet (P(Q) = \frac{2}{2}) = \frac{1}{2}$$

$$P(t_i \rightarrow r_j^P \mid w^{APP} \rightarrow F_j) = P(t_2 \rightarrow r_2^P \mid w_2^{APP} \rightarrow F_{r_2^P}) = \frac{P(w_2^{APP} \rightarrow F_{r_2^P}) = \frac{1}{2}}{(n=2)(\sum P(w^{APP} \rightarrow F_b) = \frac{1}{2} + 0)} = \frac{1}{2}$$

$$P(w^{APP} \rightarrow F_j \mid t_i \rightarrow r_j^P) = P(w_2^{APP} \rightarrow F_{r_2^P} \mid t_2 \rightarrow r_2^P) = \frac{(\frac{1}{2})((\frac{1}{2})(\frac{2}{2})(\frac{2}{2}))}{\frac{1}{2}} = 0.5 \tag{6.38}$$

Second, the evaluation of r_3^P is given as follows:

$$P(w_2^{APP} \rightarrow F_{r_3^P} \mid t_2 \rightarrow r_3^P) = \frac{(0)((\frac{0}{2})(\frac{2}{2})(\frac{2}{2}))}{\frac{1}{2}} = 0 \tag{6.39}$$

After, the evaluation we choose r_2^P for the allocation of t_1 since the probability of r_2^P gave a better change to success (50%) than on r_3^P (0%). Next, we try to find the best allocation resource for execution of t_3 of type $task2$. The possible candidates are r_4^P, r_5^p. We evaluate the same Equation 6.28, however with the current task as input, because there is no remaining task to map.

$$P(w_3^{APP} \rightarrow F_{r_4^P} \mid t_3 \rightarrow r_4^P) = \frac{(0)((\frac{0}{1})(\frac{1}{1})(\frac{2}{2}))}{\frac{1}{2}} = 0 \tag{6.40}$$

$$P(w_3^{APP} \rightarrow F_{r_5^P} \mid t_3 \rightarrow r_5^P) = \frac{(\frac{1}{2})((\frac{1}{1})(\frac{1}{1})(\frac{1}{1}))}{\frac{1}{2}} = 1 \tag{6.41}$$

Since the probability of r_5^P (100 %) is higher than on r_4^P (0%). We allocate the last task to r_5^P and finalize the mapping.

6.4 Conclusions

In this chapter, we have introduced three list-based approaches for the SPS-CGRA mapping problem. A simple list-based algorithm (SS-MAP) can be used for less complex applications and hardware, mainly full pipelined hardware. Due to its simplicity, the exploration time is not considerable. The SS-MAP allows to explore all the topological sortings of the hardware graph and, among the set of results, to select the best using the performance evaluation equation (Equation 4.9). Using all topological sortings increases the quality of the final mapping for complex applications, but, the exploration time will increase significantly.

This is why we proposed the second solution in which we added a heuristic based on look-ahead techniques (TA-MAP). This heuristic principle is to foresee the outcome of allocating one task to a given resource in terms of allocating the successors of such task onto the descendant of the given resource. This approach targets platforms that have several datapaths and interconnection between them. It can select the best datapath to use for the allocation of a task and its successors.

The third proposed solution, a Bayes-based mapping algorithm, may be recommended for complex structures. It considers two hyperparameters that can be used for fine-tuning the mapping process by the user. These hyperparameters define the preferred maximum topological distance to explore and the number of elements of the set of best allocations to explore, in terms of latency, for a given task.

These solutions can deal with more or less complex applications and hardware. However, their decision policy remains local. Both of the heuristics approaches can increase this range of decisions to include the successors of the task. Nonetheless, they still may not be optimal. In this regard, we propose to use a reinforcement learning approach that can solve this problem. This approach will be presented in the next chapter.

Chapter 7

Q-learning Mapping Algorithm

In the previous chapter, we presented three list-based mapping algorithms. Each of these algorithms combines different techniques to increase the probability of obtaining an optimal mapping. The SS-MAP (see Section 6.1) is a single shot mapping algorithm that targets less complex hardware platforms. Both TA-MAP (see Section 6.2) and BB-MAP (see Section 6.3) algorithms feature a heuristic that is capable of selecting the best resource for a given task. However, these approaches use a greedy policy, where the decision is taken locally (the allocation of a given task). The heuristics of TA-MAP and BB-MAP consider the mapping of a task's successors, improving the locality of the decision. However, it is still not enough for complex hardware platforms with multisensor capabilities.

We propose to transpose the SPS-CGRA mapping problem into a reinforcement learning problem. In this chapter, we introduce a Q-learning mapping algorithm. This mapping algorithm features an agent capable of identifying a suitable processing resource for a given task. Its learning process is based on information about the type/parameters of the processing resource and task, interconnectivity of the resource, and data dependence, among other data.

We organize the remaining part of the chapter as follows. Section 7.1 introduces reinforcement learning and illustrates the basic reinforcement learning problem. Section 7.2 describes the Q-learning algorithm and its application to the scheduling problem. Section 7.3 presents our Q-learning mapping algorithm and its methodology. Section 7.4 presents an illustrative example of the use of different reward policies and their impact on the total reward. Finally, Section 7.5 summarizes this chapter.

7.1 Reinforcement Learning

Reinforcement learning (RL) is a branch of Machine Learning (ML) that aims to learn what to do by maximizing a numerical reward signal [157]. It involves the construction of an agent that interacts within a dynamic environment and learns from it. The agent will deal with different states and take actions according to some policy. Each action that the agent takes will be rewarded either positively or negatively according to some performance measure which will grade the "goodness" of the current action [201]. The goal of the agent is to improve its performance and maximize the cumulative reward.

RL contrasts with Unsupervised Learning (UL). The second does not depend on any guidance or teacher. The training data is not tagged with the correct output, and the learning process relies on an unsupervised model that will learn freely. Similar to RL, Supervised Learning (SL) requires guidance for its learning method. However, the approach of SL is instructional and it learns by minimizing some loss. On the contrary, the approach of RL is evaluative, where the guidance or teacher provides feedback to the agent according to some performance measure [201]. These differences allow RL to support custom performance evaluation functions which help to accelerate the learning process. The evaluative approach of RL helps to discover solutions to problems where there is a lack of information and structure. The agent will compensate for

these problems by randomly exploring the universe of possible solutions and obtaining a reward related to its actions.

RL algorithms may be divided according to the used model: model-based [202] and model-free. Model-based algorithms use the transition probability distribution[1] and reward function for learning a model of the environment. They may be able to predict the outcomes (rewards) of actions. Model-free algorithms do not use the transition probability distribution, and reward function related to the Markov decision process (MDP) [203]. A model-free algorithm will not try to understand the environment. It will learn through an iterative process of trial and error. Model-free algorithms may be further divided into policy-based and value-based. Policy-based algorithms try to improve a policy function directly without using a value function. The policy function selects the best action which should be considered in a particular state to increase the reward without calculating the value function. Some examples of this type of algorithms are Deep Deterministic Policy Gradient [204], Trust Region Policy Optimization [205] and Proximal Policy Optimization [206]. In value-based algorithms, an agent makes its decisions based on the value function, which is the representation of the expected maximum reward, collected using some policy (greedy or random). Examples of value-based algorithms are Q-learning [207, 208], State-Action-Reward-State-Action (SARSA) [209, 157], Deep Q-Network (DQN) [210], Double DQN [211, 212], Dueling DQN [213].

Description of an RL Problem

An RL problem consists of a finite set of states $\mathbf{S}$[2] and a finite set of actions $\mathbf{A}$. During the learning (training) phase, the agent interacts with the environment and observes one state. In this state, the agent is fed with some information, from the environment, about the state and the available actions. Then, the agent is required to choose some action according to some policy function [157]. This policy may explore or exploit the environment. Following the former, the agent will randomly choose any action. According to the latter, the agent will choose the best-observed action (so far) related to that particular state. After the decision, the agent will be transferred to the following state and receives a reward according to the action taken (Figure 7.1).

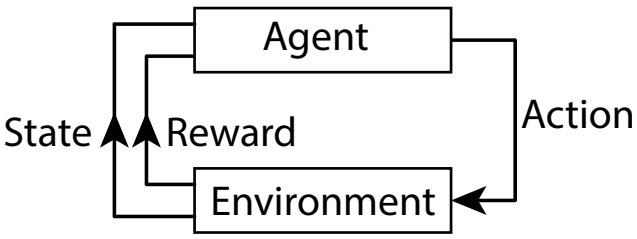


Figure 7.1 – Basic principle of RL.

The transition between states follows the Markov Decision Process (MDP)consisting in five components [214]:

- A finite set of states $\mathbf{S}$.
- A finite set of actions $\mathbf{A}$.
- A transition function $\mathbf{T} : (\mathbf{S} \times \mathbf{A}) \rightarrow \mathbf{S}^{'}$ that maps a state to its successor according to the action taken.

[1] In a Markov decision process, given the states i and j, the transition probability is the probability of transition from the state i in time n to the state j in time $n + 1$.

[2] We systematically differentiate the variables related to the reinforcement learning approach from the models and mapping and scheduling algorithms variables, with bold letters.

- A reward function $\mathbf{R} : (\mathbf{S}, \mathbf{A}) \rightarrow \mathbb{R}$.
- A discount factor $0 \leq \gamma \leq 1$ to assess value over future actions compared to the current ones.

By following the Markov property, we can assume that the next state and the received reward depend upon only the previous state and the action taken. The result of the learning phase is a surjective policy function, which maps states to actions. The agent will use this policy to decide which action to take when observing a state[214].

7.2 Q-learning

Q-learning is a value-based RL algorithm developed by Watkins [207, 208] that uses the Bellman optimal equation and an ϵ-greedy policy [3] to select the action for a given state. Its efficiency has made it the foundation of many other reinforcement learning algorithms [215, 216]. The adaptability of the reward function can be used to suit different problems [171, 217, 218, 216, 219]. The basic Q-learning algorithm takes as inputs a q-table ($\mathbf{Q}$), the set of actions ($\mathbf{A}$), the set of states ($\mathbf{S}$), and a possible terminal state that belongs to the set of states. The q-table consists of the action and the state space, and it is arbitrarily initialized. The process will iterate a number of episodes defined by the user. During each episode, we initialize the list of states $\mathbf{S}$ (Algorithm 4, line 2). Next, for each state $\mathbf{s} \in \mathbf{S}$, we choose an action ($\mathbf{a} \in \mathbf{A}(\mathbf{s})$) using a policy defined by the user (Algorithm 4, line 4). We observe the reward and update the q-table (Algorithm 4, line 5 to 6) using some learning rate (α) and some discount factor (γ) policies, update the state ($\mathbf{S}^{'}$), and iterate again until we reach the terminal state. The terminal state defines the end of the episode.

Algorithm 4 Q-learning algorithm

Input: *Initialize* $\mathbf{Q}(\mathbf{s}, \mathbf{a}), \forall \mathbf{s} \in \mathbf{S}, \mathbf{a} \in \mathbf{A}(\mathbf{s})$, *arbitrarily and* $\mathbf{Q}(\text{terminal-state}, \cdot) = 0$

1: *Repeat* (*for each episode*) :
2: *Initialize* $\mathbf{S}$
3: *Repeat* (*for each step of episode*) :
4: *Choose* $\mathbf{A}$ *from* $\mathbf{S}$ *using policy derived from* $\mathbf{Q}$(*e.g.*, ϵ-greedy)
5: *Take action* $\mathbf{A}$, *observe* $\mathbf{R}, \mathbf{S}^{'}$
6: $\mathbf{Q}(\mathbf{S}, \mathbf{A}) \leftarrow \mathbf{Q}(\mathbf{S}, \mathbf{A}) + \alpha[\mathbf{R} + \gamma max_{\mathbf{a}} \mathbf{Q}(\mathbf{S}^{'}, \mathbf{a}) - Q(\mathbf{S}, \mathbf{A})]$
7: $\mathbf{S} \leftarrow \mathbf{S}^{'}$;
8: *until* $\mathbf{S}$ *is terminal*

Q-learning Applied to Mapping and Scheduling Problems

Applied to mapping and scheduling problems, Q-learning has been proved to solve some of the issues that classical and other machine learning algorithms may not be able to solve, such as stochastic arrival of tasks [220], online single-machine scheduling [221] or resource allocation for vehicular systems [222]. In a Q-learning algorithm, the agent may learn to identify the best allocations for each particular task considering the entire application without the expense of high exploration time and memorization needs. And compared to other learning approaches, Q-learning does not need a huge data set to learn. The learn-by-trial of Q-learning allows searching for the optimal implementation exploring possible solutions that with classical algorithms we may not be able to explore.

[3] ϵ-greedy policy is a simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly (see Section 7.3.4).

7.3 Q-learning Mapping and Scheduling Algorithm

In this section, we introduce our Q-learning mapping and scheduling algorithm. The inputs are a couple of application (G_{APP}) and hardware (G_{HW}) graphs. The goal for the agent is to learn the best allocation for each task of the application. The agent will receive information about two subgraphs (Figure 7.2). The first one will comprise the task and its successors and predecessors. The second subgraph will be a resource and its successors and predecessors. For both subgraphs, the uprank and the downrank will be also given. With this information, the agent will learn to select the optimal allocation.

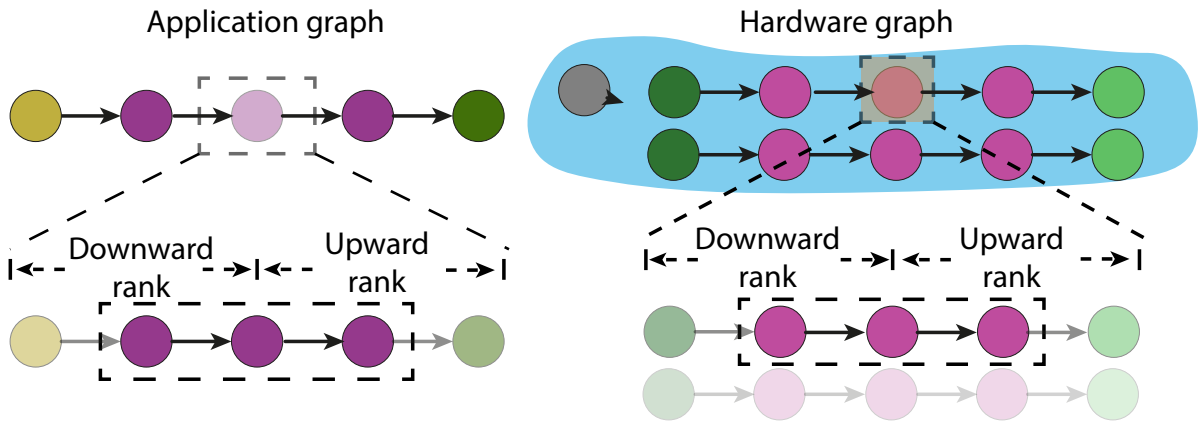


Figure 7.2 – Subgraphs and information given to the agent from the environment.

7.3.1 Agent

The agent $\mathscr{A}$ is a mapping function that maps the application to the hardware.

$$\mathscr{A} : G_{APP} \rightarrow G_{HW} \tag{7.1}$$

The agent will traverse the application allocating each task (**State**) to a suitable resource (**Action**). It will obtain rewards according to the policy described in Section 7.3.3. The terminal **State** will be the last task to allocate. The decisions that take the agent are local. However, the information that is fed to it is semi-global. The environment provides information about the successors and predecessors of a given task and resource (Figure 7.2). Additionally, information about the location (downward and upward rank) of the resource and task within their own graph is also given.

7.3.2 Environment

We propose to use the hardware graph (G_{HW}) as an environment. It will provide the possible set of actions and the reward feedback. We consider that the agent will traverse the environment, and it will be allocating (mapping) the tasks to the available resources. The environment after each allocation of a task will provide the reward to the agent, and this reward will be used to update the q-table.

State Space

We propose to define the state space (**S**) with all the possible combinations of the processing resources of the hardware graph. Hence, the state space will consist of the combinations of the following characteristics:

- Type of task ($\mathcal{T}_j$)

- Parameters of the task (Π_j)
- Input degree of the resource ($deg^-(r_j^P)$)
- Output degree of the resource ($deg^+(r_j^P)$)
- Type of tasks that the successors of the resource ($r_{suc} = Succ(r_j^P)$) may implement ($\mathcal{T}(r_{suc})$)
- Type of tasks that the predecessors of the resource ($r_{pre} = Pred(r^P j)$ may implement ($\mathcal{T}(r_{pre})$)
- Upward rank of r_j^P
- Downward rank of r_j^P

These state characteristics can be interpreted as explained in the following lines: Given a hardware graph G_{HW}, we gather the information about all the possible tasks that each resource can implement, without special operations: *disable* and *copy*. As we deal with heterogeneous hardware on both types of tasks and physical hardware implementation, we need to ensure that the parameters are considered. On this matter, the final list of type of tasks will discard only the type of tasks that have the same parameters, also considering ranges and values. Also, we gather information about the degree, both the input and output.

The upward rank is the largest topological distance from a given resource to a sink resource (a resource with an output degree equal to zero). The downward rank is the largest topological distance from a given resource to a source resource (a resource with an input degree equal to zero). However, we consider the number of resources that compose G_{HW} ($|G_{HW}|$) as the upward and downward rank. Though this number will exceed any actual rank, it is used because some pipelined applications exceed the number of resources, and we need an item to identify this situation.

Given an application G_{APP}, the number of states for a single episode will be given by $|G_{APP}|$. Some states of the complete state space may never be visited during the learning process or the inference. However, the logic behind it is to create a state space that is the complete universe of possible hardware combinations. Using this idea, we can discard any application with a task that does not correspond to any state. If any task can not be related to a state, the application can not be implemented on the hardware. Furthermore, the complete state space allows one to implement different applications to the same hardware.

Action Space

The action space ($\mathbf{A}$) is the set of available resources. The set $\mathbf{A}$ changes dynamically depending on the previous mappings. Considering the subgraph $G'_{HW}(S', K')$ with $S' = R^P$, the action space consists of all the $r_j^P \in S'$. After each allocation, the used resources will be removed from $\mathbf{A}$.

7.3.3 Rewards Policy

The rewards policy ($\mathbf{R_T}$) for each *State-Action* pair is given by:

$$\mathbf{R_T}(\mathbf{S_i}, \mathbf{A_j}) = \mathbf{R_G} + \mathbf{R_L}(\mathbf{S_i}, \mathbf{A_j}) \tag{7.2}$$

Where $\mathbf{R_G}$ represents the global reward and $\mathbf{R_L}(\mathbf{S_i}, \mathbf{A_j})$ the local reward.

Local Reward Policy

The local reward policy $\mathbf{R_L(S_i, A_j)}$, evaluates the allocation of a task (t_i), of the application subgraph $G'_{APP} = (T', D')$ where $T' = \{t_i \in G_{APP} \mid type(i) \neq interface\}$, onto some resource ($r_j^P \in S'$). This evaluation is based on an adaptation for the Q-learning approach of the rules given in Section 6.1.1. It is given by the following equation:

$$\mathbf{R_L(S_i, A_j)} = \mathbf{M_{valid}} + \mathbf{DD_{valid}} + \mathbf{Src_{valid}} + \mathbf{Snk_{valid}} + \mathbf{D_{valid}} + \mathbf{L_{opt}} + \mathbf{Pre_{valid}} + \mathbf{Suc_{valid}} \tag{7.3}$$

where $\mathbf{M_{valid}}$ represents the reward assigned by the environment and represents the goodness of the allocation considering the type of task and its parameters. $\mathbf{M_{valid}}$ is given by:

$$\mathbf{M_{valid}} = \begin{cases} \mathbf{M_{value}}, & \text{if } type(t_i) \in \mathcal{T}(r_j^P) \wedge p(t_i) \in \Pi(r_j^P) \\ -\mathbf{M_{value}}, & \text{otherwise} \end{cases} \tag{7.4}$$

Where $\mathbf{M_{value}}$ is a user defined value.

$\mathbf{DD_{valid}}$ (data dependence validation) is the reward provided by the environment given that the predecessors of t_i are correctly reachable. $\mathbf{DD_{valid}}$ is given by Equation 7.5. Considering the set of t_i predecessors $t_{pre} = Pred(t_i)$ already mapped to a subset of processing resources $r_{pre} \subset S'$, and the path $\mathcal{P}(r_{pre}, r_j^P)$ exist.

$$\mathbf{DD_{valid}} = \begin{cases} \mathbf{DD_{value}}, & \text{if } \forall r_k \in \mathcal{P}(r_{pre}, r_j^P) \text{ no task has been allocated} \\ -\mathbf{DD_{value}}, & \text{otherwise} \end{cases} \tag{7.5}$$

Where $\mathbf{DD_{value}}$ is a user defined value.

$\mathbf{Src_{valid}}$ is the reward provided by the environment given that the source of data is valid. If t_i is the successor of t_l and $type_l = interface$ (t_l is a sensor), we validate $\mathbf{Src_{valid}}$ by:

$$\mathbf{Src_{valid}} = \begin{cases} \mathbf{Src_{value}}, & \text{if } t_l \text{ is assigned to } r^{sensor} \ \wedge \ \{\forall r_k \in S | r_k \in \mathcal{P}\} \text{ no task has been allocated} \\ -\mathbf{Src_{value}}, & \text{otherwise} \end{cases} \tag{7.6}$$

where $\mathcal{P} = \{r^{sensor}, \ldots, r_j^P\}, \mathcal{P} \in G_{HW}$ and $\mathbf{Src_{value}}$ is a user defined value.

$\mathbf{Snk_{valid}}$ is the reward provided by the environment given that the sink of data is valid. If t_i is the predecessor of t_l and $type_l = interface$ (t_l is a sensor), we validate $\mathbf{Snk_{valid}}$ by:

$$\mathbf{Snk_{valid}} = \begin{cases} \mathbf{Snk_{value}}, & \text{if } t_l \text{ is assigned to } r^{sensor} \ \wedge \ \{\forall r_k \in S | r_k \in \mathcal{P}\} \text{ no task has been allocated} \\ -\mathbf{Snk_{value}}, & \text{otherwise} \end{cases} \tag{7.7}$$

where path $\mathcal{P} = \{r_j^P, \ldots, r^{sensor}\}, \mathcal{P} \in G_{HW}$ and $\mathbf{Snk_{value}}$ is a user defined value.

$\mathbf{D_{valid}}$ is the reward provided by the environment due to the *out degree* and *in degree* of both the task and resource. $\mathbf{D_{valid}}$ is given by:

$$\mathbf{D_{valid}} = \begin{cases} \mathbf{D_{value}}, & \text{if } \mathbf{I_{valid}} = True \wedge \mathbf{O_{valid}} = True \\ -\mathbf{D_{value}}, & \text{otherwise} \end{cases} \tag{7.8}$$

Where $\mathbf{I_{valid}}$ represents the evaluation of the *in degree* characteristics and is given by:

$$\mathbf{I_{valid}} = \begin{cases} True, & \text{if } deg^-(t_i) = deg^-(r_j^P) \vee deg^-(r_j^P) = 0 \\ True, & \text{if } \nexists r_k^P \in r_{pre} \mid allocates\ a\ t_l \notin t_{pre} \\ False, & \text{otherwise} \end{cases} \tag{7.9}$$

Consider $t_{pre} = Pred(t_i)$ the set of predecessors of t_i and $r_{map} \subset S'$ a subset of processing resources where $t_{pre} \rightarrow r_{map}$. Additionally, $r_{pre} = Pred(r_j^P)$ is the set of predecessors of r_j^P.

$\mathbf{O_{valid}}$ represents the evaluation of the *out degree* characteristics and is given by:

$$\mathbf{O_{valid}} = \begin{cases} True, & \text{if } deg^+(t_i) \leq deg^+(r_j^P) \vee deg^+(r_j^P) = 0 \\ True, & \text{if } deg^+(t_i) \leq \sum_{k=1}^{|r_{des}|-1} deg^+(r_k) \\ False, & \text{otherwise} \end{cases} \tag{7.10}$$

Where $r_{des} = N^+(N^+(r_j^P))$[4] and $r_k \in r_{des}$.

$\mathbf{L_{opt}}$ represents the reward assigned by the environment that means the optimally of the allocation of the task t_i onto r_j^P in terms of latency. This value depends on a table of rewards where the lowest latency will receive the entire value $\mathbf{L_{value}}$, which is an input value defined by the user. This value will decrease depending on the increase of the latency of the allocation. If the resource can not implement the type of task, the reward will be $-\mathbf{L_{value}}$.

$$\mathbf{L_{opt}} = \begin{cases} (\mathbf{L_{value}})(\frac{max(L_i^{CL}) - \mathcal{L}_j^{CL}}{max(L_i^{CL}) - min(L_i^{CL})}), & \text{if } type_i \in \mathcal{T}_j \wedge p_i \in \Pi_j \\ -\mathbf{L_{value}}, & \text{otherwise} \end{cases} \tag{7.11}$$

where L_i^{CL} is the list of computing latencies of all the resources that can implement t_i.

$\mathbf{Pre_{valid}}$ is the reward assigned by the environment that represents if the predecessors ($r_{pre} = Pred(r_j^P)$) of r_j^P can allocate the predecessors ($t_{pre} = Pred(t_i)$) of t_i. If they able to allocate them the reward is $\mathbf{Pre_{value}}$ and if not $-\mathbf{Pre_{value}}$.

$$\mathbf{Pre_{valid}} = \begin{cases} \mathbf{Pre_{value}}, & \text{if } t_{pre} \rightarrow r_{pre} \\ -\mathbf{Pre_{value}}, & \text{otherwise} \end{cases} \tag{7.12}$$

$\mathbf{Pre_{value}}$ is an input value defined by the user.

$\mathbf{Suc_{valid}}$ is a reward provided by the environment that represents if the successors($r_{suc} = Succ(r_j^P)$) of r_j^P are able to allocate the successors($t_{suc} = Succ(t_i)$) of t_i. If they are able to allocate them, the reward is $\mathbf{Suc_{value}}$ and if not $-\mathbf{Suc_{value}}$.

$$\mathbf{Suc_{valid}} = \begin{cases} \mathbf{Suc_{value}}, & \text{if } t_{suc} \rightarrow r_{suc} \\ -\mathbf{Suc_{value}}, & \text{otherwise} \end{cases} \tag{7.13}$$

$\mathbf{Suc_{value}}$ is an input value defined by the user.

Global Reward Policy

The global reward policy $\mathbf{R_G}$ evaluates the entire mapping, if all the tasks are allocated in some resource and if the data dependency is respected. We have two cases if we evaluate the performance or not. This depends on the type of training that we are executing. During the offline training, we don't consider the performance. Otherwise, during the online training, we evaluate the performance. If the computing cost of the episode ($CC_{episode}$) is better than the previous episode ($CC_{episode-1}$), we provide a reward. The reward of the performance evaluation needs to be minimal. Otherwise, its value will exceed the other rewards as it may increase severely during the entire training. In this sense, we divide the value by the number of episodes, and we take advantage of the exploitation of the q-table made by the epsilon policy. In other words, the greatest reward will be given during the last episodes of the online training, where we are converting to the optimal mapping.

For the offline training the value of $\mathbf{R_G}$ is given by:

$$\mathbf{R_G} = \begin{cases} \mathbf{R_{value}}, & \text{if } \forall t_i \in G'_{APP}, \exists r_j^P \in G'_{HW} : t_i \rightarrow r_j^P \\ -\mathbf{R_{value}}, & \text{otherwise} \end{cases} \tag{7.14}$$

[4] $N^+(r_j^P)$ is the set of out-going neighbors (successors) of r_j^P.

where $t_i \rightarrow r_j^P$ is subject to the same mapping constrains given in Section 6.1.1.
For the online training the equation changes to

$$\mathbf{R_G} = \begin{cases} \mathbf{R_{value}} + \dfrac{CC_{value}}{episodes}, & \text{if } \forall t_i \in G'_{APP}, \exists r_j^P \in G'_{HW} : t_i \rightarrow r_j^P \\ & CC_{episode-1} > CC_{episode} \\ \mathbf{R_{value}}, & \text{if } \forall t_i \in G'_{APP}, \exists r_j^P \in G'_{HW} : t_i \rightarrow r_j^P \\ & CC_{episode-1} \geq CC_{episode} \\ -\mathbf{R_{value}}, & \text{otherwise} \end{cases} \tag{7.15}$$

Where $t_i \rightarrow r_j^P$ is subject to the same mapping constrains given in Section 6.1.1. $\mathbf{R_{value}}$ and CC_{value} are input values defined by the user, r_{pre} is the subset of processing resources that allocate the predecessors of the task t_i, *episode* is the current episode number and *episodes* is total number of episodes of the training.

7.3.4 Training

The Q-learning algorithm is based on the dynamic update of a table (q-table) made by the action space and the states space. The q-table is a look-up table that stores the q values obtained after each episode that composes the learning process.

The learning process is based on two different types of training: offline training and online training. The set of applications of a given hardware platform may not be large enough to be used in a training process. This is particularly true for SPS-CGRAs, as the universe of applications for each one is limited. We solve this issue using randomly generated graphs. The offline training is used to create a seed q-table (bootstrap table) that may be used directly for inference if all the application states are already evaluated and the resulting mapping is a correct mapping. We initialize the training with the offline training, which will provide the above mentioned seed q-table, afterwards we proceed with the online training. During the online training, we use the target application that we want to map onto the hardware platform. The online training is used to increase the inference performance and the obtention of the optimal mapping. The idea of both types of training is to generalize the q-table to map other applications than the current one.

We use two policies that control the balance of exploitation-exploration and future-present rewards for both types of training. The ϵ-policy allows one to balance the exploitation-exploration. We propose to at the beginning explore as much as possible. As the number of episodes increases, we start exploiting the q-table to converge faster, as the agent will know most of the environment and it will be able to select intelligently a suitable processing resource for a given task. The gamma policy allows one to balance future and present rewards. We propose prioritizing future rewards at the start of the training, and as the number of episodes increases, change the policy to present rewards. This will help us provide better rewards to more mature solutions at the end of the training. Both policies will help the agent to learn more efficiently.

Q-Table

The q-table is made by the states space and the action space. Where the *y-axis* is the state space and the *x-axis* is the action space. It is initialized all zeros, and after each episode, it is updated.

Epsilon Policy

The epsilon-greedy policy or ϵ-greedy policy allows one to manage the exploitation and exploration of the environment. We propose using an epsilon-greedy policy where we explore the environment randomly at the beginning of the training. After a given number of episodes, we

allow the agent to exploit its knowledge. Therefore, ϵ will be initialized with a value near or equal to 1. After a given number of episodes (*start_decay_epsilon*), its value will decrease until zero. The epsilon policy is provided by:

$$\epsilon = \begin{cases} \epsilon_{init}, & \text{if } episodes \geq start\_decay\_epsilon \\ \epsilon_{init} - \dfrac{\epsilon_{init}}{episodes - start\_decay\_epsilon}, & \text{otherwise} \end{cases} \tag{7.16}$$

where *start_decay_epsilon* is an input value defined by the user, and ϵ_{init} is the initial value of ϵ, also given by the user.

Learning Rate

The learning rate (α) controls how fast the learning process modifies our q-values given new evidence [223]. One expects to start with a high learning rate, which allows fast changes, and lowers the learning rate as time progresses. However, this policy will easily discard old values and replace them with new knowledge, which may mean that the agent will not learn what is needed to find the optimal mapping. We use a fixed learning rate value to allow the agent to learn only with epsilon and gamma variation.

Gamma Policy

The discount factor or gamma (γ) allows balancing future and present rewards. With the learning rate, both control the agent learning and define future rewards' value over present rewards. Recall that we use a fixed learning rate (see Section 7.3.4). Therefore we use only γ to control the balance. We propose using a dynamic gamma policy, where the future rewards are prioritized (gamma value equal or near to 1) at the beginning of the training. After a given number of episodes (*start_decay_gamma*), gamma starts to decrease until the present rewards are the priority (gamma value equal or near to 0). We propose a gamma policy given by:

$$\gamma = \begin{cases} \gamma_{init}, & \text{if } episodes \geq start\_decay\_gamma \\ \gamma_{init} - \dfrac{\gamma_{init}}{episodes - start\_decay\_gamma}, & \text{otherwise} \end{cases} \tag{7.17}$$

where *start_decay_gamma* is an input value defined by the user, and γ_{init} is the initial value of γ, also given by the user.

Off-line Training

We use the off-line training to create a seed q-table. The training uses as inputs the subgraph $G'_{HW}(S', K')$ with $S' = R^P$ and N, a user-input variable that defines the number of randomly generated application graphs that we will produce. The first step is the creation of the q-table (Algorithm 5, line 1), which consists of the combination of the state space and the action space.

Next, we create the set APP_SET, that consist of N randomly generated application graphs (Algorithm 5, line 2). We enter the main cycle, where we will iterate according to the number of training episodes. Inside of the main cycle, we start with choosing an application graph from APP_SET (Algorithm 5, line 4). Then we create the list of states of the current episode (Algorithm 5, line 5). This list of states will correspond to the tasks of the chosen application. Also, we create a new list L_{HW} that consists of all the processing resources of the hardware subgraph (G'_{HW}) (Algorithm 5, line 6).

Next, we enter a sub-cycle where we iterate over the list of states created previously. The first step in this sub-cycle is choosing a processing resource from L_{HW}. This decision will be based on the *ϵ-policy* introduced in Section 7.3.4. Then we allocate the current task (state) to the selected resource, observe the reward of this allocation and update the q-table accordingly

(Algorithm 5, lines 9 to 12). Finally, we update the state and remove the selected resource from L_{HW} (Algorithm 5, lines 12 to 13) and iterate again until there are no more states.

Algorithm 5 Offline training algorithm

Input: $G'_{HW}, N(Number\ of\ G_{APP}\ for\ training)$
Output: **Q**

1: *New* **Q** *with the universe of states given by* G'_{HW}
2: *New array APP_SET, which consists of N randomly created* G_{APP}
3: **for** *each episode* **do**
4: *Choose a* G_{APP} *from APP_SET*
5: *New list of states given by* G_{APP}, *each state represents a* $t_i \in G_{APP}$
6: *New list* L_{HW} *of all* $r^P \in S'$
7: **repeat**$(for\ each\ state\ i\ of\ the\ episode)$:
8: *Choose a* r_j^P *from* L_{HW} *using* ϵ*-policy*
9: *Map* t_i *to* r_j^P
10: *Observe the Reward* **R** *and the next state* t_{i+1}
11: *Update* $\mathbf{Q}(t_i, r_j^P) \leftarrow \mathbf{Q}(t_i, r_j^P) + \alpha[\mathbf{R} + \gamma max_{\mathbf{a}}\mathbf{Q}(t_{i+1}, \mathbf{a}) - \mathbf{Q}(t_i, r_j^P)]$
12: $t_i \leftarrow t_{i+1}$
13: *Remove* r_j^P *from* L_{HW}
14: **until** t_i *is terminal*

On-line Training

The online training uses the hardware graph, the application graph, and the Q-table obtained from the offline training. After the offline training, we attempt to allocate the application onto the hardware without performing the online training (Inference, Section 7.3.5). If the allocation is successful, we end the entire process. If the allocation is not successful, we perform the online training. Usually, the inference process fails after the offline training due to the following reasons:

- A state that belongs to the application graph has not been visited yet.
- The inference process produced a not valid complete mapping.

The first step of the Inference algorithm is to create the list of states that belongs to the application graph. **If any of these states had not been visited yet**, the q-values will be all zeros. In this case, the inference process will be finished, and we will proceed to the online training. If all the states had been visited, we attempt to allocate the tasks to the resources exploiting the Q-Table. Since in the inference, we only exploit the q-values, we may not produce a mapping that respects the data dependence or shows errors on the source of data. As a result, the **complete mapping will be not valid** and we proceed to the online training.

The online training helps to evaluate the states from the target application graph that were not covered during the offline training and improve the inference process results. The steps are the same as the offline training. The difference is that the application graph used is the target application graph, and the use of randomly generated application graphs is removed. Additionally, as introduced in Equation 7.15, during the online training we also evaluate the performance of the final mapping. Algorithm 6 shows the pseudo-code of the online training.

Algorithm 6 Online training algorithm

Input: G'_{HW}, G'_{APP}, Q
Output: **Q**

1: *New list of states given by* G'_{APP}, *each state represents a* $t_i \in G'_{APP}$
2: **for** *each episode* **do**
3: *New list* L_{HW} *of all* $r^P \in S'$
4: **repeat**(*for each state i of the episode*) :
5: *Choose a* r_j^P *from* L_{HW} *using* ϵ*-policy*
6: *Map* t_i *to* r_j^P
7: *Observe the Reward* $\mathbf{R}$ *and the next state* t_{i+1}
8: *Update* $\mathbf{Q}(t_i, r_j^P) \leftarrow \mathbf{Q}(t_i, r_j^P) + \alpha[\mathbf{R} + \gamma max_{\mathbf{a}}\mathbf{Q}(t_{i+1}, \mathbf{a}) - \mathbf{Q}(t_i, r_j^P)]$
9: $t_i \leftarrow t_{i+1}$
10: *Remove* r_j^P *from* L_{HW}
11: **until** t_i *is terminal*

7.3.5 Inference

The inference provides the final allocation of G_{APP} onto G_{HW}. In this step, we exploit the q-table, which is the result of the training. The process starts with the creation of the list of states and the list of processing resources (Algorithm 7, line 1 and 2). Next, we select the processing resource for each state of the application that has the higher q-value. If we can not use the first M (M is a user-input value, and $1 \leq M \leq |S'|$) processing resources with the higher q-values, we create a new time slot. We consider that this operation will not increase the latency significantly, and we will still produce a sub-optimal mapping. After the selection of the processing resource, we allocate the task to it (Line 5) and store the partial mapping (Line 6. The process ends when we allocate all the tasks to the resources. Then we validate the mapping in terms of data dependence. If the mapping is valid, according to the rules given in Section 6.1.1, we finish the entire process. If not, we perform another iteration of the online training.

Algorithm 7 Inference algorithm

Input: G'_{APP}, $G'_{HW}, \mathbf{Q}$
Output: L_{MAP}

1: *New list of states given by* G'_{APP}, *each state represents a* $t_i \in G'_{APP}$
2: *New list* L_{HW} *of all* $r^P \in S'$
3: **for** *each state i of the list of states* **do**
4: *Choose the* r_j^P *from* L_{HW} *with the maximum q-value for* t_i
5: *map* $t_i \rightarrow r_j^P$
6: *Store the mapping in* L_{MAP}

7.4 Illustration of the Q-learning Principle

7.4.1 General Parameters and Policies

In this section, we present two easy examples to show the principle of the Q-learning mapping algorithm. We will show that we can achieve different results by using different reward policies and even fail to converge to a good mapping. Let's consider the set of rewards values of Table 7.1. In this table, we present three different reward policies. Policy 1 prioritize the final mapping ($\mathbf{R_G}$), Policy 2 made emphasize the parameters verification ($\mathbf{M_{valid}}$), Policy 3 prioritize the data dependence ($\mathbf{DD_{value}}$).

Table 7.1 – Rewards policies

	Rewards								
	$\mathbf{R_{value}}$	$\mathbf{CC_{value}}$	$\mathbf{M_{value}}$	$\mathbf{DD_{value}}$	$\mathbf{Src_{value}}$	$\mathbf{Snk_{value}}$	$\mathbf{L_{value}}$	$\mathbf{D_{value}}$	$\mathbf{Pre_{value} + Suc_{value}}$
Policy 1	0.8	0.2	0.5	0.5	0.1	0.1	0.1	0.1	0.1
Policy 2	0.5	0.5	0.9	0.1	0.1	0.1	0.1	0.1	0.1
Policy 3	0.5	0.5	0.1	0.9	0.1	0.1	0.1	0.1	0.1

The general parameters for these examples are shown in Table 8.2.

Table 7.2 – Q-learning mapping general parameters

Episodes		Epsilon			Gamma			Learning rate
Offline training	Online training	Initial value	Decrement start episode	Decrement end episode	Initial value	Decrement start episode	Decrement end episode	Value
10000	10000	1.0	1500	7000	0.9	1	10000	0.1

7.4.2 Example 1

Let's consider the hardware and application graphs showed in Figure 7.4.2 and 7.4.2 accordingly. These simple graphs represent homogeneous resources and tasks.

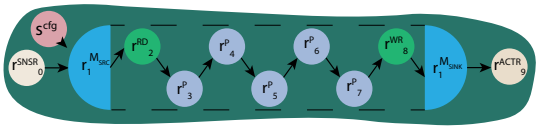


Figure 7.3 – Hardware graph of example 1.

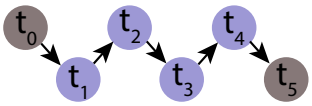


Figure 7.4 – Application graph of example 1.

In Figure 7.5 we depict the average reward composition for both offline and online training. We separate the rewards for local and global rewards policies. In blue, we can see the results for Policy 1, in red for Policy 2, and in green for Policy 3. In this example, we can notice that the best results from the offline training are obtained with Policy 2. This may be because we use randomly generated graphs, and the agent uses the parameters as a foundation because the structures of the graphs are changing, but the parameters remain the same. The worse results are obtained using Policy 1, which uses the entire mapping of the application onto the hardware as a metric. Again, as we are using randomly generated graphs, it is difficult for the agent to learn the entire structure and features of a complete application and hardware graph. Moreover, the total reward, which considers $\mathbf{R_G}$ shows no improvement during the episodes, and it may need more episodes to achieve a good value. In the online training (bottom two figures), we can see a similar behavior for all the sets. Moreover, as we already perform the offline training, this helps as a bootstrap, and the agent will learn faster during the online training.

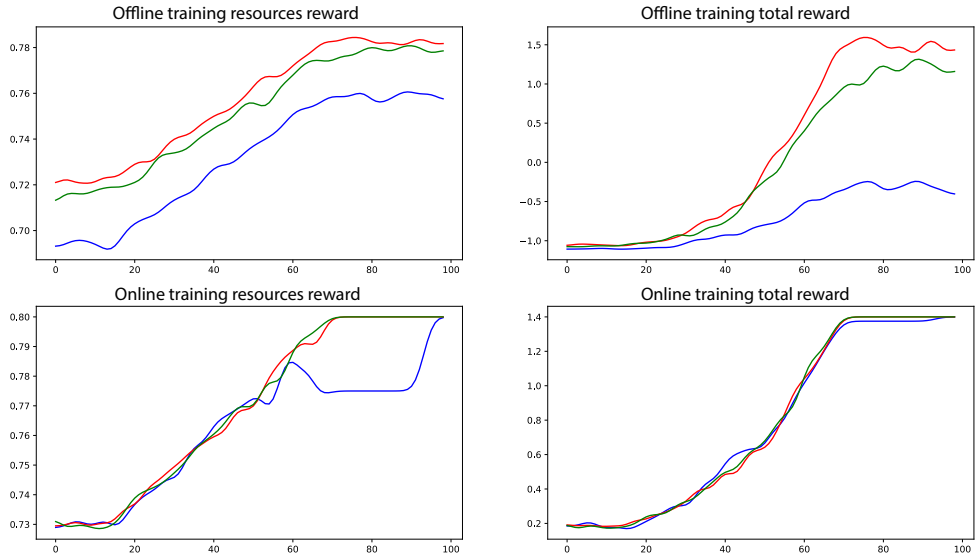


Figure 7.5 – Average rewards composition for example 1.

7.4.3 Example 2

Now, lets consider the hardware application graphs of Figure 7.4.3 and 7.4.3. Again we are using homogeneous resources and tasks, but the number of them are different from the previous example.

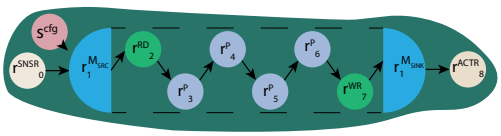


Figure 7.6 – Hardware graph of example 2.

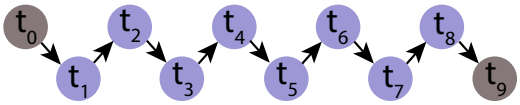


Figure 7.7 – Application graph of example 2.

In Figure 7.8 we depict the average reward composition of this example. In this example, the input graphs are bigger than the ones of the previous example. The lines coloring are the same as in Section 7.4.2. Policy 2 (blue) shows better performance in the offline training. This may result from the number of processing resources of the hardware, which is lower than the previous example. The reward $\mathbf{DD_{valid}}$ refers to the data dependence validation. This means that we verify that all the predecessors can reach the task being mapped. If the number of processing resources is low, the probability that this verification results in a valid value is higher than if the number is high. Regarding the online training, the behavior of Policy 3 shows that for the local reward, that set can achieve better results than the others but fails to achieve a valid mapping (total reward). This is maybe due to the number of tasks of the application and its pipelined structure. It is difficult for the algorithm to consider this aspect, as most of the structures (see Section 7.3) of the application are the same. For this, the algorithm may rely on the other sets of parameters to achieve a good mapping.

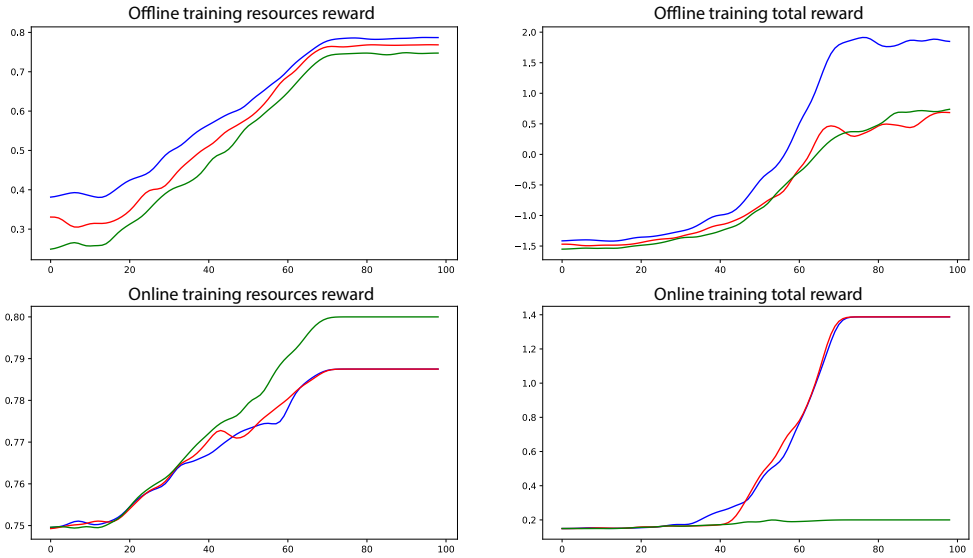


Figure 7.8 – Average rewards composition for example 2.

Another aspect to considering is the randomness of the ϵ-greedy policy. This may result in difficulty to converge to a valid mapping. As the agent chooses its actions, it may not choose at the beginning the best actions. Thus the agent will not be able to learn.

7.5 Conclusions

This chapter has introduced a formalization of the SPS-CGRA mapping problem to a reinforcement learning problem. We describe the environment in terms of the hardware graph. We propose considering the agent as a mapping function that aims to allocate the application onto the hardware. We feed the agent with information about the structures that we can produce from the hardware graph. We consider a structure as the association of a resource with its predecessors and successors and the position in which it concerns the hardware graph (downward and upward rank). Furthermore, we consider the descriptors of all the resources that belong to such structure as part of the information that we feed to the agent. With this information, the agent can decide where the best allocation for a given task is.

We describe a reward policy composed of a global and a local reward. This policy aids to converge quickly, given that it not only considers local decisions. Additionally, we propose to solve the lack of examples for training using a random graph generator and a two-step training process. This approach allows one to generalize the Q-Table, and it may be subject to exploitation from others applications, not only the targeted one.

Finally, this approach is expected to be the best option for complex applications, where the nodes' degree is considerable.

Chapter 8

Experimental Evaluation

In the last two chapters, we presented two different approaches to solve the SPS-CGRA mapping problem. Hence, in Chapter 6 we presented three list-based mapping algorithms, and in Chapter 7 we presented a Q-learning mapping algorithm. In this chapter, we present an experimental evaluation of these mapping algorithms. The mapping algorithms are evaluated in terms of exploration time and computing cost of the final mapping (i.e., the execution time of the resulting implementation). We divide the evaluation into three experiments:

- Experimental graphs.
- Randomly generated graph.
- Case study.

The experimental graphs are a set of defined graphs with generic parameters that characterize the mapping algorithms. Used as a benchmark, they represent the typical structures of an SPS-CGRA and its applications, pipeline, and parallel configurations. The set includes both homogeneous and heterogeneous organizations of resources.

In order to enlarge our evaluations, we present a pseudo-random graph generator that can create virtual SPS-CGRA-alike hardware. Additionally, it can generate a set of applications for the built virtual hardware. The generator includes the means to define the degree of interconnectivity of the hardware organization and the number and types of resources. Also, it provides several parameters that target the set of applications, which help to define the number of tasks and structure of the application (pipeline or parallel). We use a set of randomly generated SPS-CGRA-alike hardware platforms to evaluate the mapping algorithms. These artificially generated hardware platforms' structure and resource organization may be more complex than a normal SPS-CGRA and represent the worst-case scenario for the list-based and the Q-learning mapping algorithms. We present the results on one couple of hardware and application graphs.

Finally, we evaluate our mapping algorithms using a real-life hardware platform. In this experiment, we use the Morphological Co-processor Unit (MCPU) [12]. It has the main features that we can find in other SPS-CGRA hardware and serve as a mean to provide a proof of concept to the mapping algorithms.

For the experiments of Section 8.2 and 8.4, we evaluate our algorithms against an exhaustive mapping algorithm (named exhaustive algorithm in the rest of the chapter) that we also developed. We chose to use an exhaustive algorithm because no other framework or mapping and scheduling algorithm can be used as a direct comparison to the extent of our knowledge. The exhaustive mapping algorithm is a brute-force algorithm that aims to construct all the possible mappings of an application onto hardware. It systemically tries to find the optimal allocation to a task. The exhaustive mapping algorithm also considers the possible different physical realizations of each hardware resource. Hence, a possible mapping may be the allocation of one task per time slot, where the efficiency of the chosen resource compensates the configuration and memorization cost. This exhaustive algorithm is used as a golden reference for our proposed

mapping and scheduling algorithms. In Section 8.3 we present an experimental evaluation only considering our proposed algorithms. This is because the use of the exhaustive algorithm is prohibitive due to combinatorial explosion.

8.1 Experimental Setup

The evaluation presented in this chapter is done using a software tool developed from scratch using Python 3.6 (more than 10,000 lines of code without comments). We use the library Networkx [224] to handle graph-based processes. For visualization we use Graphviz [225] and Matplotlib [226]. For the experiments in this section we use a personal computer with an 8-core Intel® Core™ i7-7700HQ CPU @ 2.80GHz and 7,7 GB of RAM and 1 TB of disk, running Ubuntu 16.04 LTS. The complete code is available in `https://github.com/ebarbudo/MappingSPSCGRA`.

8.2 Experimental Graphs

In this section, we present the first evaluation of the mapping algorithms. We use different sets of typical experimental graphs. These graphs will help us to characterize the behavior of the algorithms. They represent the typical structure of an SPS-CGRA and its applications. We make use of four sets of experimental application graphs:

- Pipeline of homogeneous tasks
- Parallel structure of homogeneous tasks
- Pipeline of heterogeneous tasks
- Parallel and hybrid (parallel-pipeline) structures of heterogeneous tasks

These are the basic and representatives types of application structures and organizations. We consider a set of application graphs covering the same characteristics, pipeline and parallel structures, and homogeneous and heterogeneous resources. For each set of experimental application graphs, corresponding experimental hardware graphs are defined.

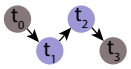
The evaluation analyses the following features:

- Exploration time.
- Computing cost of the resulting application.
- Training time (Q-learning algorithm).

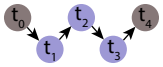
In the following sections, we will refer to the version of the SS-MAP that uses all topological sortings of the hardware graph as SS-MAP T. Also, the version that only uses one random topological sorting as SS-MAP U.

8.2.1 Pipeline of Homogeneous Tasks

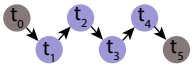
The first set of application graph examples represent a group of pipelined homogeneous applications. The difference between each other is the number of tasks. Figure 8.1 shows this set. All tasks belong to the same *type* of task, which is in this case a generic $task0$. The parameter for this task is fixed. We identify the tasks with type $task0$ with the color purple and in brown the ones with type $interface$.



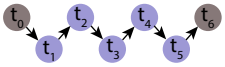
(a) Application graph example APP1 (2 homogeneous tasks).



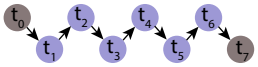
(b) Application graph example APP2 (3 homogeneous tasks).



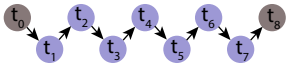
(c) Application graph example APP3 (4 homogeneous tasks).



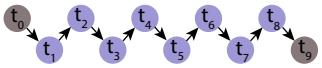
(d) Application graph example APP4 (5 homogeneous tasks).



(e) Application graph example APP5 (6 homogeneous tasks).



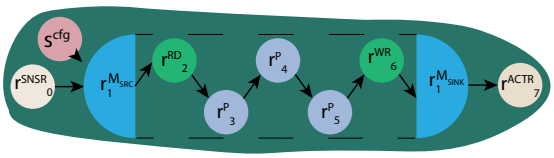
(f) Application graph example APP6 (7 homogeneous tasks).



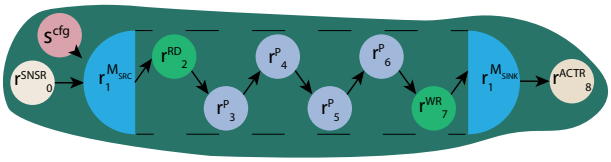
(g) Application graph example APP7 (8 homogeneous tasks).

Figure 8.1 – First set of application examples

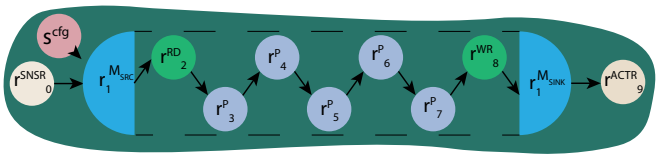
We will use this first set of application graphs over four sets of hardware graphs. The first set of hardware graphs represents a linear pipeline of homogeneous resources. Consider that all the resources can implement the $task0$. Figure 8.2 shows this first set, notice that the number of resources increases by one for all the graphs. Notice the color purple of the processing resources (r^P) that matches the color of the tasks (t_i) of the application graphs. This represents that the processing resources are able to allocate those tasks. We evaluate the mappings in terms of the computing cost and the exploration time.



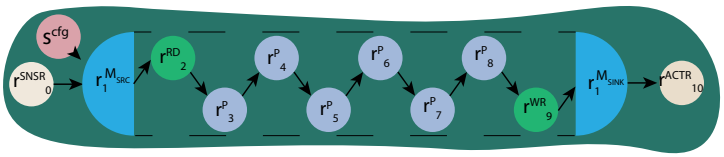
(a) Hardware graph example HW1 (3 homogeneous tasks).



(b) Hardware graph example HW2 (4 homogeneous tasks).



(c) Hardware graph example HW3 (5 homogeneous tasks).



(d) Hardware graph example HW4 (6 homogeneous tasks).

Figure 8.2 – First set of hardware examples

We consider three reward policies for the Q-learning approach (see Section 7.3.3), each one emphasizing an important aspect, the final mapping ($\mathbf{R_G}$), the data dependence verification ($\mathbf{DD_{valid}}$) and the parameters verification ($\mathbf{M_{valid}}$). The values of the each policy are showed in Table 8.1. For clarity, we will refer to the Q-learning algorithm subject to Policy 1 as Q-L POL 1, the one subject to Policy 2 as Q-L POL 2, and the one subject to Policy 3 as Q-L POL 3.

Table 8.1 – Reward policies

	Rewards								
	$\mathbf{R_{value}}$	$\mathbf{CC_{value}}$	$\mathbf{M_{value}}$	$\mathbf{DD_{value}}$	$\mathbf{Src_{value}}$	$\mathbf{Snk_{value}}$	$\mathbf{L_{value}}$	$\mathbf{D_{value}}$	$\mathbf{Pre_{value} + Suc_{value}}$
Policy 1	0.8	0.2	0.5	0.5	0.1	0.1	0.1	0.1	0.1
Policy 2	0.5	0.5	0.9	0.1	0.1	0.1	0.1	0.1	0.1
Policy 3	0.5	0.5	0.1	0.9	0.1	0.1	0.1	0.1	0.1

The general parameters for all the Q-learning mapping approaches are shown in Table 8.2. Both the reward policies and the general parameters showed above are used for all the sets of experimental graphs.

Table 8.2 – Q-learning mapping general parameters

Episodes		Epsilon			Gamma			Learning rate
Offline training	Online training	Initial value	Decrement start episode	Decrement end episode	Initial value	Decrement start episode	Decrement end episode	Value
10000	10000	1.0	1500	7000	0.9	1	10000	0.1

Firstly we study the exploration time. Figure 8.3 gives the resulting exploration times of the mapping and scheduling algorithms. We only consider the inference and the performance evaluation time for the Q-learning approaches. We can observe that there are no substantial differences between the other heuristics mappings nor the Q-learning mapping algorithm. The behavior of the algorithms is not that influenced by the number of nodes, both tasks and resources. This means that there is no combinatorial explosion, as we can see with the behavior of the exhaustive algorithm.

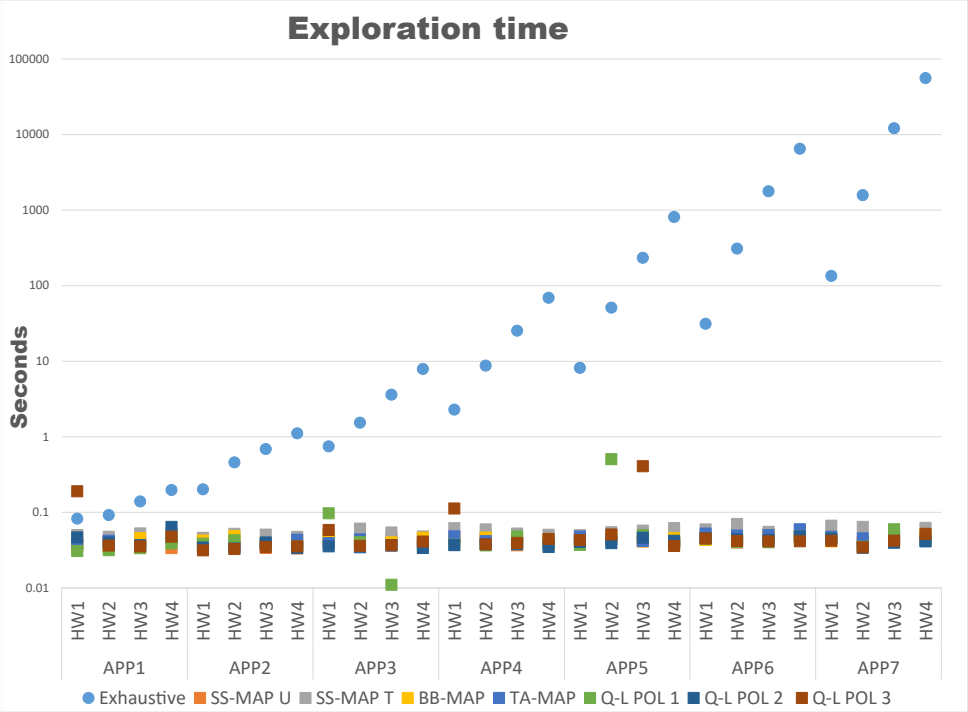


Figure 8.3 – Comparative study of the exploration time of the first set of applications and the first set of hardware graphs.

With the Q-learning mapping algorithm, the offline and online training time depends on the number of tasks and resources. This phenomenon is more evident in offline training. We use randomly generated applications for the offline training, and these applications may be larger than the original target application. Figure 8.4 shows the timings for both offline and online training for this first set of experimental application graphs. The offline training time is considerably more significant than the online training.

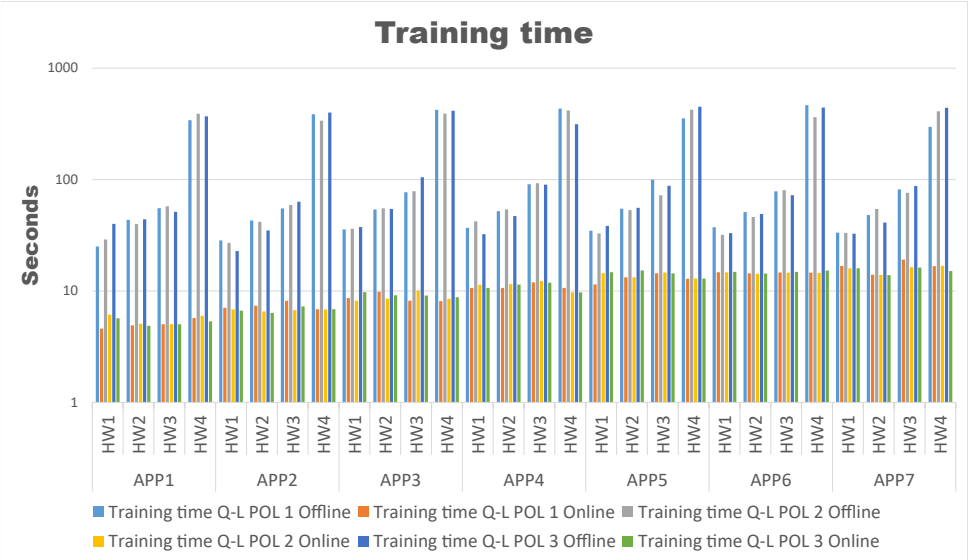


Figure 8.4 – Training time of the Q-learning approaches.

Finally, the resulting computing costs (the execution time of the implementation) and the error percentage with regard to the optimal are shown in Table 8.3. We consider as optimal the results of the exhaustive algorithm, and in the rest of this chapter, we calculate the error or difference with regard to this value as follows

$$\text{ERROR} = \frac{|optimal\ value - obtained\ value|\ \times\ 100}{optimal\ value}\% \tag{8.1}$$

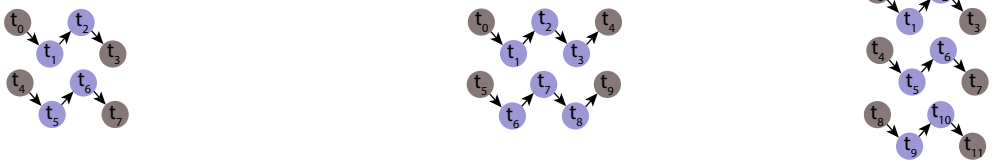
Thus, a value of 0% will mean that the resulting computing cost is equal or close to the optimal value. We can notice that the results of all the mapping algorithms are optimal since they are identical to the ones obtained by the exhaustive algorithm. This first set of simple experimental graphs shows the impact of increasing the number of tasks and resources for the mapping algorithms. We can see that the difference between exploration time is not considerable, and in most cases, it may be neglected. The mapping algorithms met the expectations of achieving the exact computing cost as the exhaustive approach.

Table 8.3 – Comparative between mapping approaches according to the computing cost measured in clock cycles (pipeline of homogeneous tasks)

Application	Hardware	Exhaustive algorithm	SS-MAP U		SS-MAP T		TA-MAP		BB-MAP		Q-L POL 1		Q-L POL 2		Q-L POL 3	
		COMPUTING COST	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)
APP1	HW1	307212	307212	0 %	307212	0%	307212	0%	307212	0%	307212	0%	307212	0%	307212	0%
	HW2	307214	307214	0%	307214	0%	307214	0%	307214	0%	307214	0%	307214	0%	307214	0%
	HW3	307216	307216	0%	307216	0%	307216	0%	307216	0%	307216	0%	307216	0%	307216	0%
	HW4	307218	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%
APP2	HW1	307213	307213	0%	307213	0%	307213	0%	307213	0%	307213	0%	307213	0%	307213	0%
	HW2	307215	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%
	HW3	307217	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%
	HW4	307219	307219	0%	307219	0%	307219	0%	307219	0%	307219	0%	307219	0%	307219	0%
APP3	HW1	614422	614422	0%	614422	0%	614422	0%	614422	0%	614422	0%	614422	0%	614422	0%
	HW2	307216	307216	0%	307216	0%	307216	0%	307216	0%	307216	0%	307216	0%	307216	0%
	HW3	307218	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%
	HW4	307220	307220	0%	307220	0%	307220	0%	307220	0%	307220	0%	307220	0%	307220	0%
APP4	HW1	614423	614423	0%	614423	0%	614423	0%	614423	0%	614423	0%	614423	0%	614423	0%
	HW2	614427	614427	0%	614427	0%	614427	0%	614427	0%	614427	0%	614427	0%	614427	0%
	HW3	307219	307219	0%	307219	0%	307219	0%	307219	0%	307219	0%	307219	0%	307219	0%
	HW4	307221	307221	0%	307221	0%	307221	0%	307221	0%	307221	0%	307221	0%	307221	0%
APP5	HW1	614424	614424	0%	614424	0%	614424	0%	614424	0%	614424	0%	614424	0%	614424	0%
	HW2	614428	614428	0%	614428	0%	614428	0%	614428	0%	614428	0%	614428	0%	614428	0%
	HW3	614432	614432	0%	614432	0%	614432	0%	614432	0%	614432	0%	614432	0%	614432	0%
	HW4	307222	307222	0%	307222	0%	307222	0%	307222	0%	307222	0%	307222	0%	307222	0%
APP6	HW1	921633	921633	0%	921633	0%	921633	0%	921633	0%	921633	0%	921633	0%	921633	0%
	HW2	614429	614429	0%	614429	0%	614429	0%	614429	0%	614429	0%	614429	0%	614429	0%
	HW3	614433	614433	0%	614433	0%	614433	0%	614433	0%	614433	0%	614433	0%	614433	0%
	HW4	614437	614437	0%	614437	0%	614437	0%	614437	0%	614437	0%	614437	0%	614437	0%
APP7	HW1	921634	921634	0%	921634	0%	921634	0%	921634	0%	921634	0%	921634	0%	921634	0%
	HW2	614430	614430	0%	614430	0%	614430	0%	614430	0%	614430	0%	614430	0%	614430	0%
	HW3	614434	614434	0%	614434	0%	614434	0%	614434	0%	614434	0%	614434	0%	614434	0%
	HW4	614438	614438	0%	614438	0%	614438	0%	614438	0%	614438	0%	614438	0%	921652	50%

8.2.2 Parallel Structure of Homogeneous Tasks

The second set of experimental application graphs are based on a parallel organization, where several independent task branches build an application. The tasks' type is again generic *task*0, and the parameters are fixed. We consider that all the tasks are of the same type. Figure 8.5 gives the set of experimental application graphs. Again we use the same coloring as the previous example.



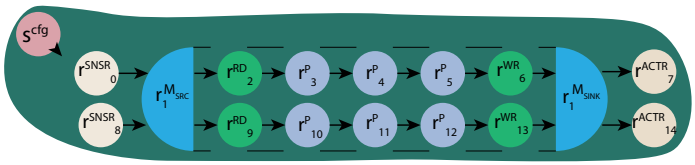
(a) Application graph example APP8 (4 homogeneous tasks).

(b) Application graph example APP9 (6 homogeneous tasks).

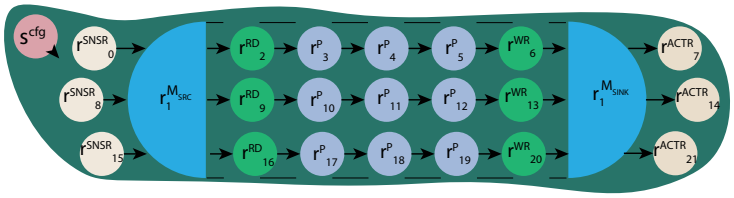
(c) Application graph example APP10 (6 homogeneous tasks).

Figure 8.5 – Second set of application examples

The second set of hardware graphs are structured with parallel connectivity. Assume that all the processing resources can execute *task*0. Figure 8.6 shows the hardware graphs that we consider for this experiment.



(a) Hardware graph example HW5 (homogeneous, 6 processing resources).



(b) Hardware graph example HW6 (homogeneous, 9 processing resources).

Figure 8.6 – Second set of hardware examples

Figure 8.7 shows the comparison between the mapping algorithms base on the exploration time. Notice that, except for the SS-MAP that uses all the topological sortings of the hardware graph, the other algorithms remain with similar exploration times. This shows that the impact of a parallel structure can be neglected, and the algorithms' behavior remains constant without regard to the structure of the application and hardware graphs.

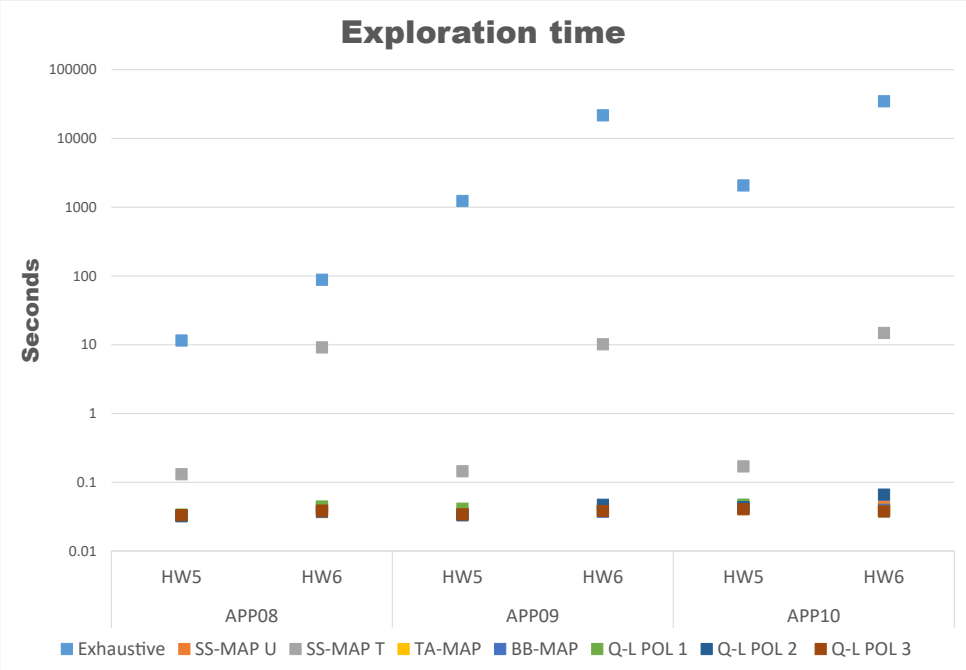


Figure 8.7 – Comparative of the exploration time of the second set of applications and the second set of hardware graphs.

Regarding the training times of the Q-learning approach, we can see in Figure 8.8 that the changes in the reward policies do not affect the training time. Also, notice that the online training is almost the same for all the reward policies.

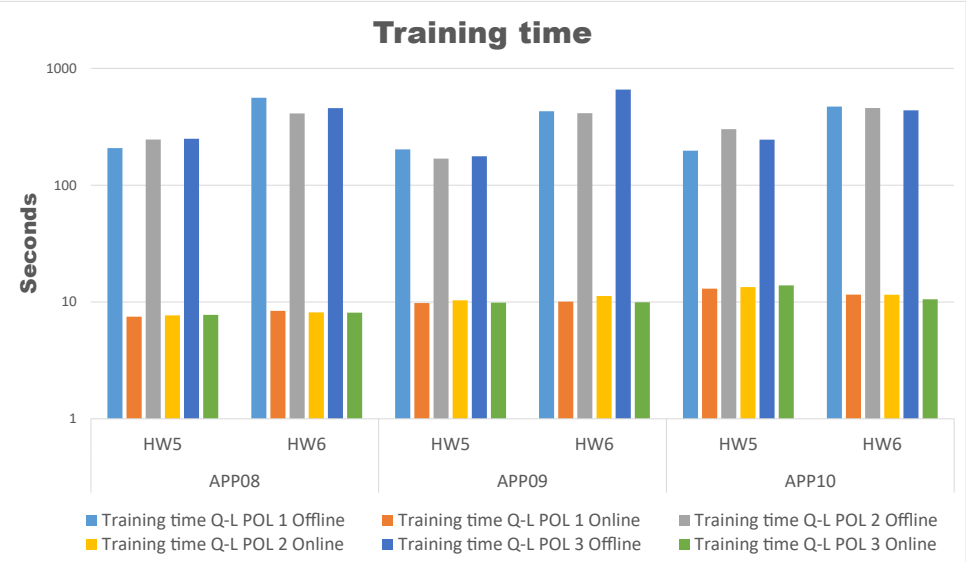


Figure 8.8 – Training time of the Q-learning approaches.

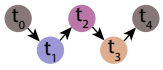
Table 8.4 shows the resulting computing cost for the mapping algorithms. Notice that the heuristics and the Q-learning approaches shown better performance than the SS-MAP algorithms, as they can obtain the exact optimal value. Recall that the optimal value is the one obtaining from the exhaustive algorithm. As in the previous set of experimental graphs, the algorithms can obtain near or optimal results (compared to the exhaustive algorithm).

Table 8.4 – Comparative between mapping approaches according to the computing cost measured in clock cycles (parallel structure of homogeneous tasks).

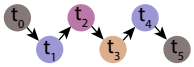
Application	Hardware	Exhaustive algorithm	SS-MAP U		SS-MAP T		TA-MAP		BB-MAP		Q-L POL 1		Q-L POL 2		Q-L POL 3	
		COMPUTING COST	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)
APP08	HW5	307207	307215	0%	307215	0%	307215	0%	307215	0%	614426	100%	307215	0%	307215	0%
	HW6	307207	307218	0%	307218	0%	307218	0%	307218	0%	307223	0%	307223	0%	307223	0%
APP09	HW5	307214	307214	0%	307214	0%	307214	0%	307214	0%	307214	0%	307214	0%	307214	0%
	HW6	307217	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%
APP10	HW5	614426	614427	0%	614427	0%	614426	0%	614426	0%	614427	0%	614426	0%	614426	0%
	HW6	307217	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%

8.2.3 Pipeline of Heterogeneous Tasks

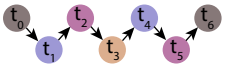
The third set of application graphs (Figure 8.9) represents an heterogeneous pipeline of tasks. We increase the number of tasks for each example and we consider three type of tasks, $task0$, $task1$ and $task2$. Each type of task is colored in a different way, $task0$ is colored in dark violet, $task1$ in light violet and $task2$ in green. We consider that the parameters of the task are fixed.



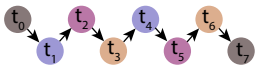
(a) Application graph example APP11 (3 heterogeneous tasks).



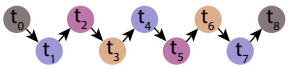
(b) Application graph example APP12 (4 heterogeneous tasks).



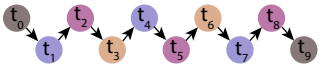
(c) Application graph example APP13 (5 heterogeneous tasks).



(d) Application graph example APP14 (6 heterogeneous tasks).



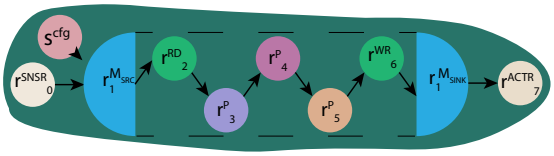
(e) Application graph example APP15 (7 heterogeneous tasks).



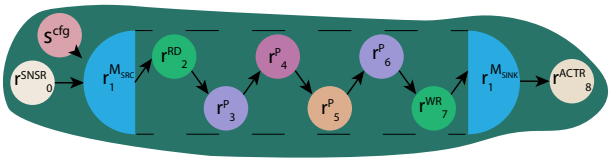
(f) Application graph example APP16 (8 heterogeneous tasks).

Figure 8.9 – Third set of application examples

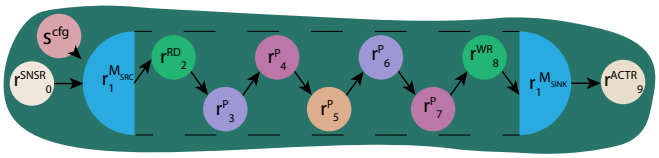
The third set of hardware examples (Figure 8.10d) represent a group of heterogeneous resources. Each resource is able to implement one single task, complying to the second application set of examples they can implement $task0$, $task1$ and $task2$. We identify each type of resource by its color, blue for $task0$, brown for $task1$ and orange for $task2$.



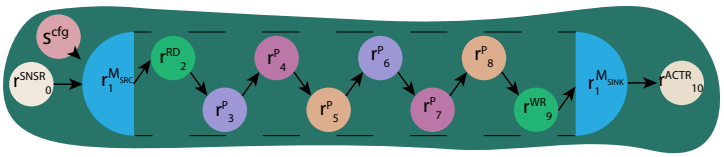
(a) Hardware graph example HW7 (3 heterogeneous resources).



(b) Hardware graph example HW8 (4 heterogeneous resources).



(c) Hardware graph example HW9 (5 heterogeneous resources).



(d) Hardware graph example HW10 (6 heterogeneous resources).

Figure 8.10 – Third set of hardware examples

Figure 8.11 shows the exploration time for this set of experimental graphs. The exploration time remains uniform for the mapping algorithms, with only a few exceptions, mainly the Q-learning algorithm.

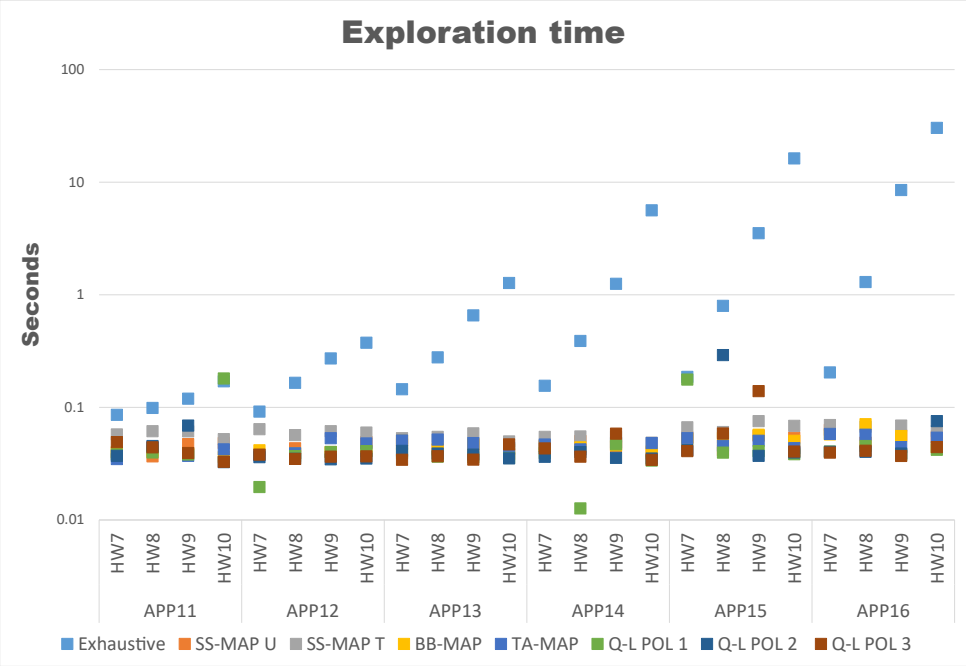


Figure 8.11 – Comparative of the exploration time of the third set of applications and the third set of hardware graphs.

As seen in Figure 8.12, the training time showed some differences, mainly for the offline training. The online training remains with a uniform behavior.

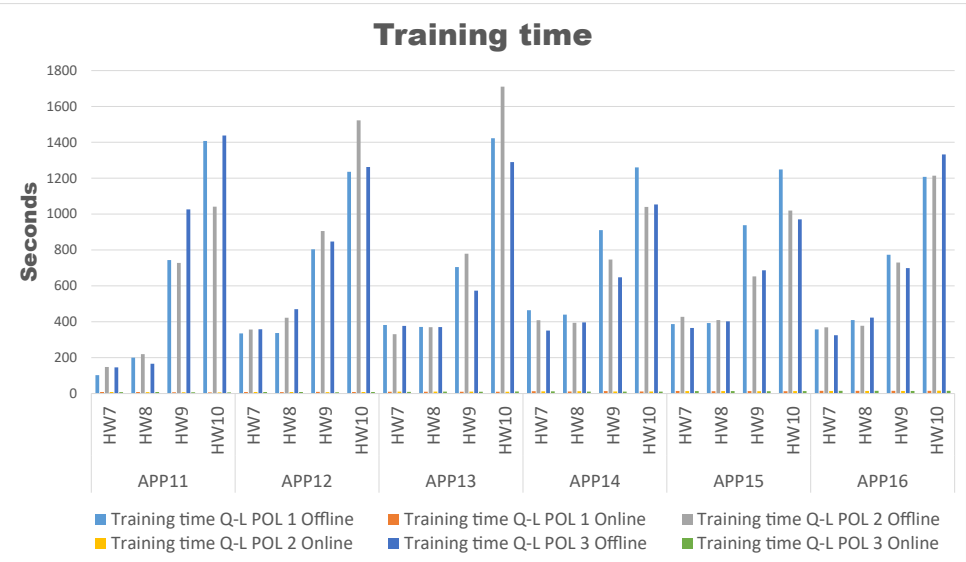


Figure 8.12 – Training time of the Q-learning approaches.

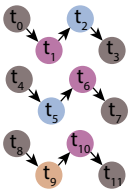
Because of the simplicity of these experimental graphs, we can see in Table 8.5 that all the mapping algorithms could meet a sub-optimal computing cost.

Table 8.5 – Comparative between mapping approaches according to the computing cost measured in clock cycles (pipeline of heterogeneous tasks)

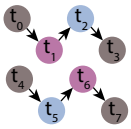
Application	Hardware	Exhaustive algorithm	SS-MAP U		SS-MAP T		TA-MAP		BB-MAP		Q-L POL 1		Q-L POL 2		Q-L POL 3	
		COMPUTING COST	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)
APP11	HW7	307213	307213	0%	307213	0%	307213	0%	307213	0%	307213	0%	307213	0%	307213	0%
	HW8	307215	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%
	HW9	307215	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%
	HW10	307215	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%	307215	0%
APP12	HW1	614422	614422	0%	614422	0%	614422	0%	614422	0%	614422	0%	614422	0%	614422	0%
	HW2	307216	307216	0%	307216	0%	307216	0%	307216	0%	307216	0%	307216	0%	307216	0%
	HW3	307218	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%
	HW4	307220	307220	0%	307220	0%	307220	0%	307220	0%	307220	0%	307220	0%	307220	0%
APP13	HW7	614423	614423	0%	614423	0%	614423	0%	614423	0%	614423	0%	614423	0%	614423	0%
	HW8	614427	614427	0%	614427	0%	614427	0%	614427	0%	614427	0%	614427	0%	614427	0%
	HW9	307219	307219	0%	307219	0%	307219	0%	307219	0%	307219	0%	307219	0%	307219	0%
	HW10	307221	307221	0%	307221	0%	307221	0%	307221	0%	307221	0%	307221	0%	307221	0%
APP14	HW7	614424	614424	0%	614424	0%	614424	0%	614424	0%	614424	0%	614424	0%	614424	0%
	HW8	314428	314428	0%	314428	0%	314428	0%	314428	0%	314428	0%	314428	0%	314428	0%
	HW9	614432	614432	0%	614432	0%	614432	0%	614432	0%	614432	0%	614432	0%	614432	0%
	HW10	307222	307222	0%	307222	0%	307222	0%	307222	0%	307222	0%	307222	0%	307222	0%
APP15	HW7	921633	921633	0%	921633	0%	921633	0%	921633	0%	921633	0%	921633	0%	921633	0%
	HW8	614429	614429	0%	614429	0%	614429	0%	614429	0%	614429	0%	614429	0%	614429	0%
	HW9	614433	614433	0%	614433	0%	614433	0%	614433	0%	614433	0%	614433	0%	614433	0%
	HW10	614437	614437	0%	614437	0%	614437	0%	614437	0%	614437	0%	614437	0%	614437	0%
APP16	HW7	921634	921634	0%	921634	0%	921634	0%	921634	0%	921634	0%	921634	0%	921634	0%
	HW8	921640	921640	0%	921640	0%	921640	0%	921640	0%	921640	0%	921640	0%	921640	0%
	HW9	614434	614434	0%	614434	0%	614434	0%	614434	0%	614434	0%	614434	0%	614434	0%
	HW10	614438	614438	0%	614438	0%	614438	0%	614438	0%	614438	0%	614438	0%	614438	0%

8.2.4 Parallel and Hybrid Structures of Heterogeneous Tasks

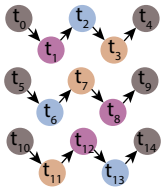
The last set of experimental graphs (Figure 8.13) represent both parallel and hybrid organizations. We use four application graphs that represents parallel organizations and the remaining two applications are used for hybrid hardware organizations. All the application graphs are heterogeneous as in the previous example.



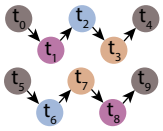
(a) Application graph example APP17 (6 heterogeneous tasks).



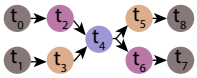
(b) Application graph example APP18 (8 heterogeneous tasks).



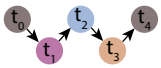
(c) Application graph example APP19 (9 heterogeneous tasks).



(d) Application graph example APP20 (12 heterogeneous tasks).



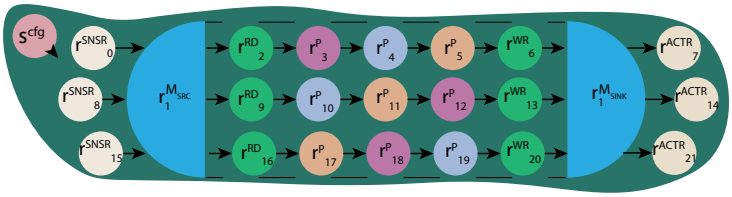
(e) Application graph example APP21 (5 heterogeneous tasks).



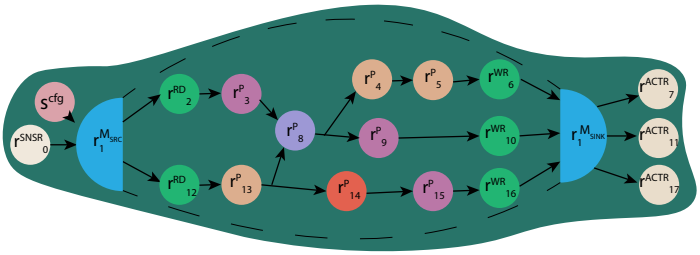
(f) Application graph example APP22 (3 heterogeneous tasks).

Figure 8.13 – Fourth set of application examples

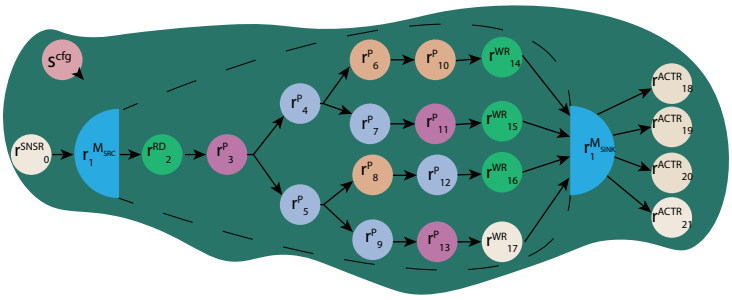
We use three hardware graphs (Figure 8.14). One is used for the parallel application graphs, and the other two are used for the hybrid application graphs (one for each application). We make use of these hardware graphs to highlight the features of our mapping algorithms.



(a) Hardware graph example HW11 (9 heterogeneous resources).



(b) Hardware graph example HW12 (8 heterogeneous resources).



(c) Hardware graph example HW13 (11 heterogeneous resources).

Figure 8.14 – Fourth set of hardware examples

Contrarily to the previous sets of experimental graphs, the exploration times of the mapping algorithms are different. We can see in Figure 8.15 that the exhaustive algorithm and the SS-MAP T, which uses all topological sortings of the hardware graph, shown the worst performance in terms of exploration time. Followed by the Q-learning approaches. Still, the TA-MAP and BB-MAP have shown better performance than the other approaches.

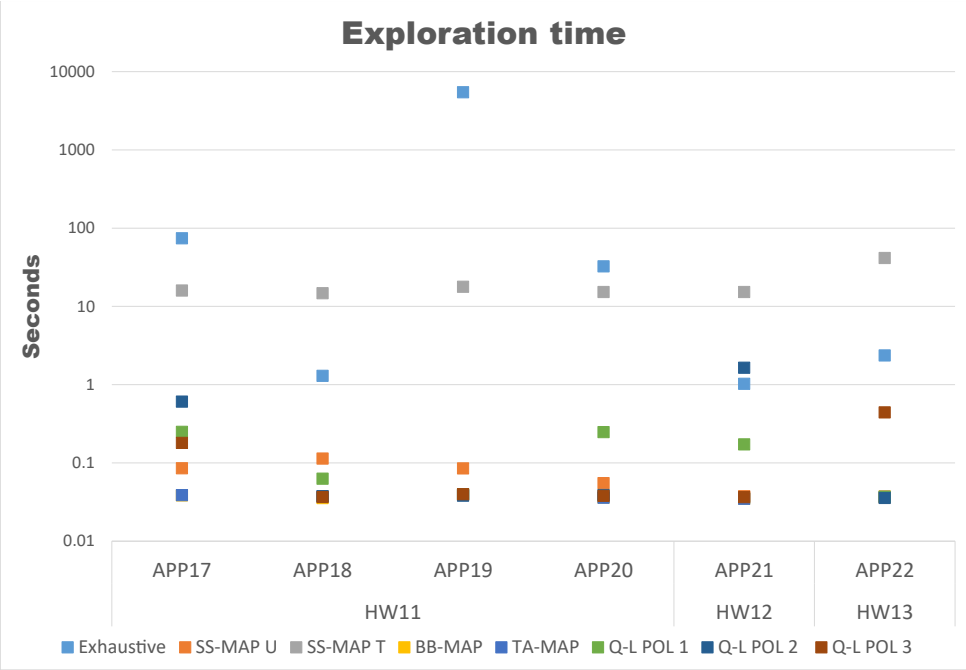


Figure 8.15 – Comparative of the exploration time of the fourth set of applications and the fourth set of hardware graphs.

The behavior of the training time remains the same, with slight differences. In Figure 8.16, we can notice that the offline training takes almost the same time for all the applications and the different reward policies.

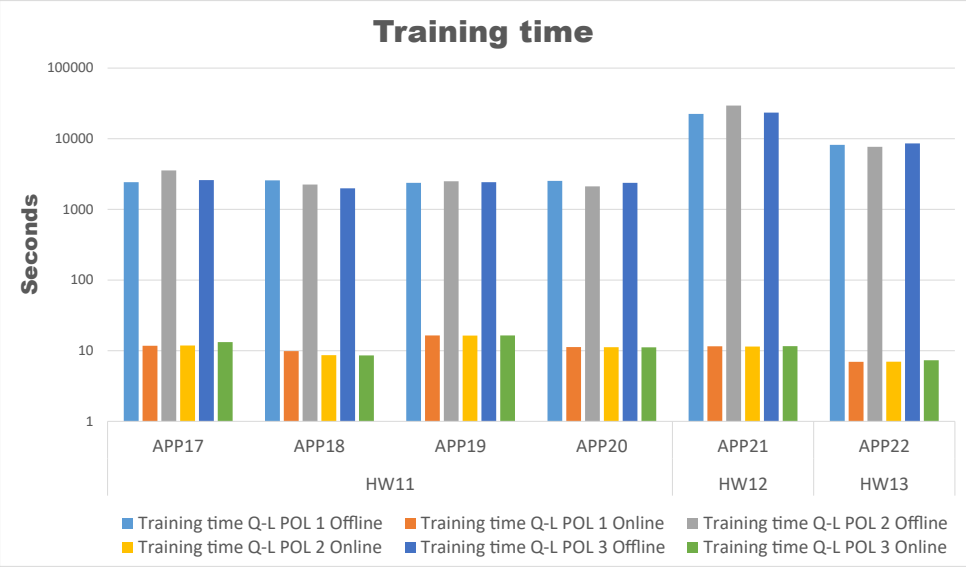


Figure 8.16 – Training time of the Q-learning approaches.

Table 8.6 shows the resulting computing cost from the mapping algorithms. We want to highlight that for hybrid organizations, the Q-learning approach can obtain results close to the exhaustive algorithm, while the SS-MAP (both SS-MAP U and SS-MAP T) and TA-MAP results are 50% and sometimes 100% worse. This shows the great adaptability of the Q-learning approach for complex structures.

Table 8.6 – Comparative between mapping approaches according to the computing cost measured in clock cycles (hybrid structures of heterogeneous tasks)

Application	Hardware	Exhaustive algorithm	SS-MAP U		SS-MAP T		TA-MAP		BB-MAP		Q-L POL 1		Q-L POL 2		Q-L POL 3	
		COMPUTING COST	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)	COMPUTING COST	ERROR (%)
HW11	APP17	307217	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%
	APP18	307217	614436	100%	614436	100%	307217	0%	307217	0%	307217	0%	307217	0%	307217	0%
	APP19	307218	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%
	APP20	307218	614438	100%	614438	100%	307218	0%	307218	0%	307218	0%	307218	0%	307218	0%
HW12	APP21	921638	921638	0%	921638	0%	921639	0%	921639	0%	921638	0%	921638	0%	921638	0%
HW13	APP22	614430	921634	50%	921634	50%	921634	50%	921634	50%	614430	0%	614430	0%	921633	50%

8.3 Pseudo-Random Graph Generator

Random graphs are extensively used to experiment complex systems. For example, the phone network, the internet, the gene network or the cell, which is a network of chemicals linked by chemical reactions, may be described using random graphs [227]. Random graphs are built through an evolutionary process based on models such as Erdős-Rényi [228], Watts-Strogatz [229], and Barabási-Albert [230], which are the main models.

The Erdős-Rényi model defines a random graph as N labeled nodes connected by n edges, which are chosen randomly given a probability P. The decision is made individually for each edge that connects two given nodes. The Watts-Strogatz model starts with N nodes placed regularly in a ring and each of these nodes are connected to its $K/2$, where K is an even number, neighbors on both side. Next, in a clockwise loop, for every node $v \in N$, the edge that connects to its clockwise $i-th$ next node is rewired with a probability P.

Both previous models establish a priory the number of nodes and do not modify it during the evolutionary process. Additionally, the attachment of an edge is given by a probability P, which is independent of the edges that are already attached. On the contrary, most real-world networks tend to grow through time by adding new nodes. Furthermore, the addition of an edge that connects two nodes, shows a direct relation with the already existent edges. The Barabási-Albert model or Scale-free model allows the growth of the random graph and also a preferential attachment. It starts with a small number of nodes M, where $1 \leq M < N$. Next, the method sequentially adds a new node with M new edges. For a node to be added, it will be connected to an existing node v with probability proportional to its degree. The new node repeatedly adds non-duplicate edges in this way until it has M edges. Then this is iterated until the graph has N nodes [231].

Since the universe of SPS-CGRAs is considerably broad, in our work, we have developed from scratch a random graph generator to evaluate our mapping algorithms. This random graph generator, coded in Python 3.6, allows to create complex structures that include a wide range of input and output degrees and manage the heterogeneity of the artificially created hardware. We base our generator on the Barabási-Albert model. This will enable us to manage the input and output degree of the hardware, define the number of nodes and their parameters and functions. It should be mentioned that the described models target undirected graphs. Consequently, we include a mechanism to produce only directed graphs without loops.

8.3.1 Methodology

The pseudo-random graph generator allows one to produce synthetic graphs to verify the performance of the mapping algorithms. The input of the generator is a specification file (text file), and the output products are a hardware graph and a set of application graphs (text files). It is specifically designed to build SPS-CGRA a-like systems, although it can be used to build pseudo-random DAGs.

The process starts with the specification file written by the user, where the following fields have to be filled:

- Number of
  - Internal nodes
  - Input and output degree of the nodes
  - Sensors
  - Actuators
  - Memory resources
- Name, type and parameters of the tasks that each processing resource may perform.
- Parameters of the memory and the communication resources.

- Configuration cost functions.

Next, we build the hardware graph and from it we create the application set.

Hardware Graph

The generator uses the above parameters to build the hardware graph, starting with the basic structure of the graph. For this purpose, the generator will create n number of nodes according to the specification file. Next, it will create the interconnections (edges). For each created node, the generator defines the input and output connections. For the input connection, the generator will randomly decide if the node will be connected to a source node (sensor) or to internal node. For both decisions the generator will randomly decide to which sensor node or internal node will be connected. For the output connection, the generator will also randomly select if the node will be connected to a sink node (actuator) or to a internal node. The specific sink node or internal node will be selected randomly by the generator. Before the addition of every edge (interconnection) the generator checks if a cycle will be created. If we do not create a cycle, the edge is added, otherwise the edge is dismissed.

After the creation of the basic structure, the generator performs a pruning of the nodes without any interconnection, both input and output interconnections. This means that the nodes with input and output degree equal to zero are removed. At the end of this step we will ended up with a pseudo-random dag, that can be used for other purposes other than mapping. Figure 8.17 shows an example of a complex pseudo-random generated dag (some simpler random generated graphs will be presented in details in Section 8.3.2). Even with the pruning the expected number of nodes are similar to the real number of nodes as we can see in Figure 8.18. The relative error between the expected and the real number of nodes is minimal.

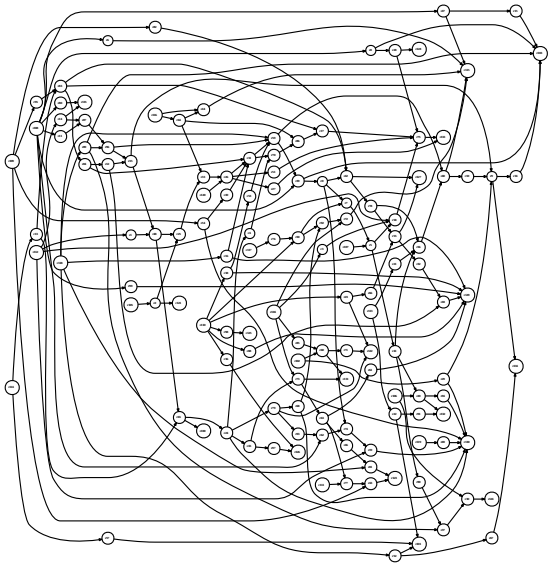


Figure 8.17 – Example of a random generated graph

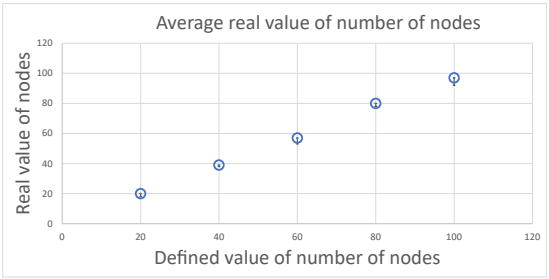


Figure 8.18 – Comparison between expected values and real values for the number of nodes in the graph

As for the degree distribution, both input and output degree, we can see in Figure 8.19 (input degree) and Figure 8.20 (output degree) that second and third degree takes the predominance.

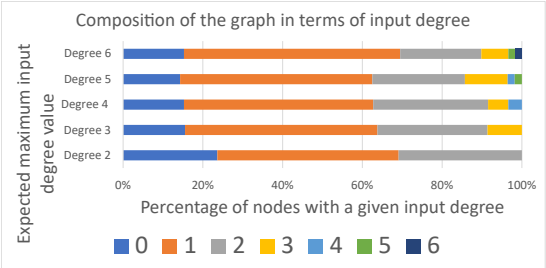


Figure 8.19 – Comparison of the input degree of the nodes in a graph

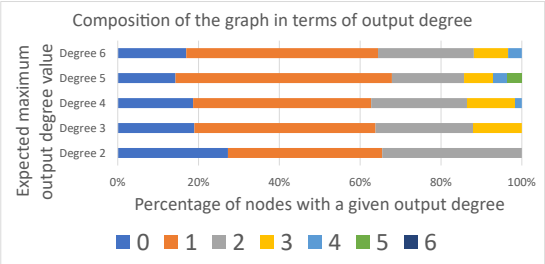


Figure 8.20 – Comparison of the input degree of the nodes in a graph

The next step is to deal with memory resources integration. The generator will randomly decide the addition of memory resources. If the generator chooses to integrate memory resources, the number of this resources will be according to the limit of memory resources defined by the specification file. To add the memory resource, we need to also add communication resources, specifically R^{RW} and R^{RD}. This is done automatically by the generator. If the generator choose not to add any memory block, the sensors and actuators will be directly connected to the processing resources.

The following step is the assignment of the parameters and labels of each resource. The generator will randomly select the parameters of each node, according to its type. Another feature is the inclusion of communication resources of type R^{MUX}. This inclusion is enabled through the specification file. The generator will assign this type of resource to any node with input degree above 1 and output equal or above 1. This assignment is randomly decided.

Application Graph

After the finalization of the build of the hardware graph we continue with the construction of the application graph set. The application graph is the result of the modification (addition and removal of nodes) of the hardware graph. Also, the generator allows one to define some aspects of the application graph through the specification file which have to contain:

- Number of applications.
- Type of application.
- Nodes to remove.
- Number of parallel and serial instances.

The generator uses the number of applications to create accordingly the set of applications. The type of application is used to define the type of input that the application will require. Two types of application are considered, *signal* and *image*. The type of application *image* considers an image as an input, and parameters such as *width* of the image and *height* of the image as main parameters for the processing. The type of application *signal* considers a signal as an input, and the main parameter is *input samples*.

The structure of the application depends on the number of nodes to remove and the number of parallel and serial instances. The number of nodes to remove defines how many nodes of the hardware graph will be removed to create the application graph. As stated before, the application graph is build after the hardware graph by removing randomly nodes from it (Figure 8.21). After the removal we get the main structure of the application graph.

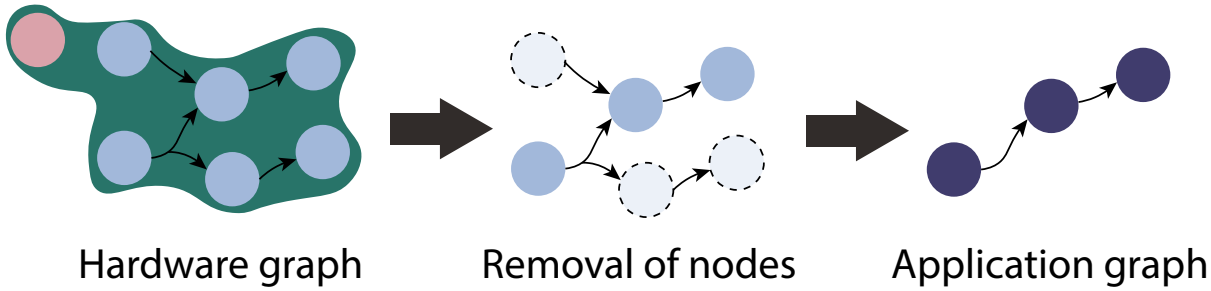


Figure 8.21 – Process flow of the generation of an application graph.

The number of parallel and serial instances represents how many instances will be added to the main structure. The parallel instances are disjoint graphs that make up the application graph. The serial instances are instances connected in series that make up the application graph. Both type of instances are shown in Figure 8.22.

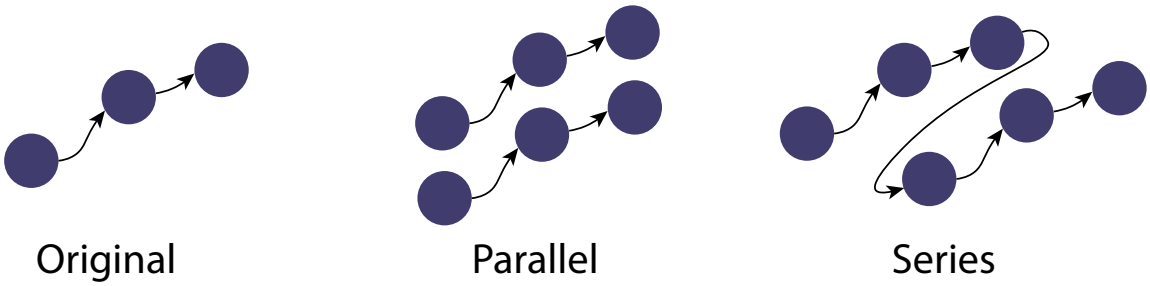


Figure 8.22 – Example of the different structures.

8.3.2 Evaluation with Randomly Generated Graphs

This section presents an evaluation of our mapping algorithms using one couple of application and hardware graphs. These graphs are randomly generated using our pseudo-random generator. Both graphs exhibit a hybrid structure and are more complex, in the organization, than a regular SPS-CGRA. The purpose of this evaluation is to measure the performance of each mapping algorithm against a possible worst-case scenario for them. A detailed example of a randomly generated pair of graphs is given in Figure 8.23 (hardware graph) and Figure 8.24 (application graph).

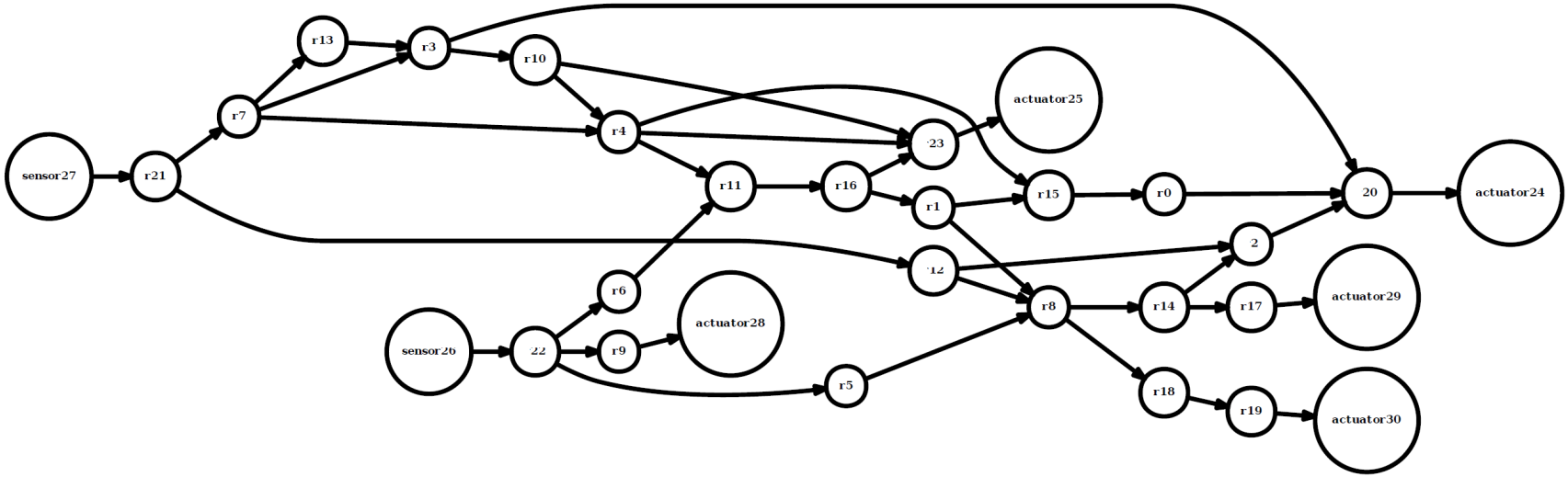


Figure 8.23 – Randomly generated hardware.

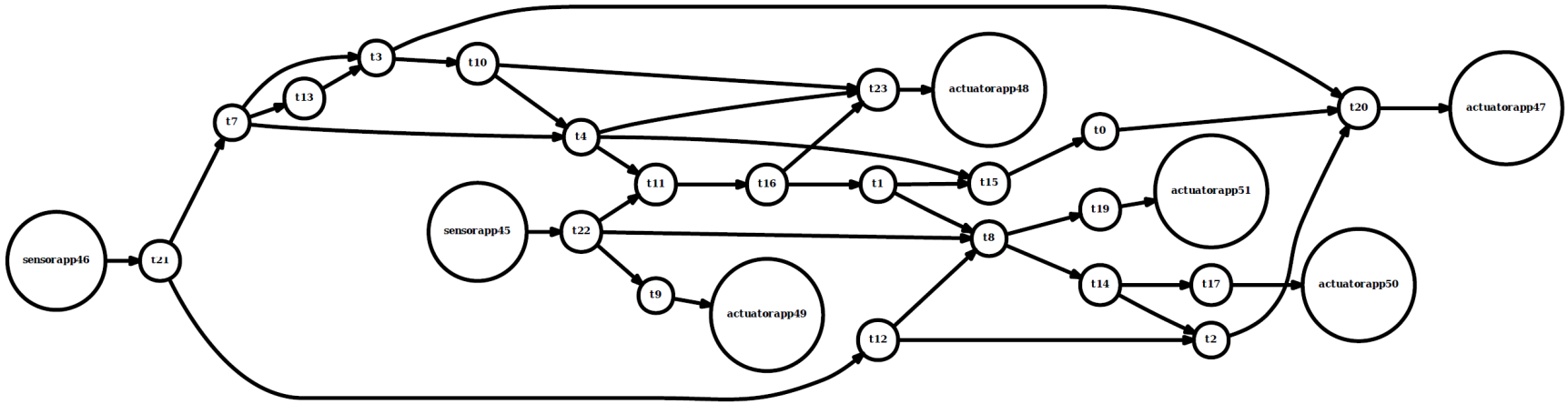


Figure 8.24 – Randomly generated application.

Figure 8.25 shows the comparison of exploration time between the mapping algorithms. For this evaluation, we only use 10000 random topological sortings of the hardware graph as input of the SS-MAP T algorithm. As in the previous evaluations, the difference between mapping algorithms is almost negligible, except for the SS-MAP that uses all topological sortings of the hardware graph. In that case, we can see that the exploration time is more than 1000 seconds. Also, the Q-learning that uses reward policies 2 and 3 takes more time than using reward policy 1.

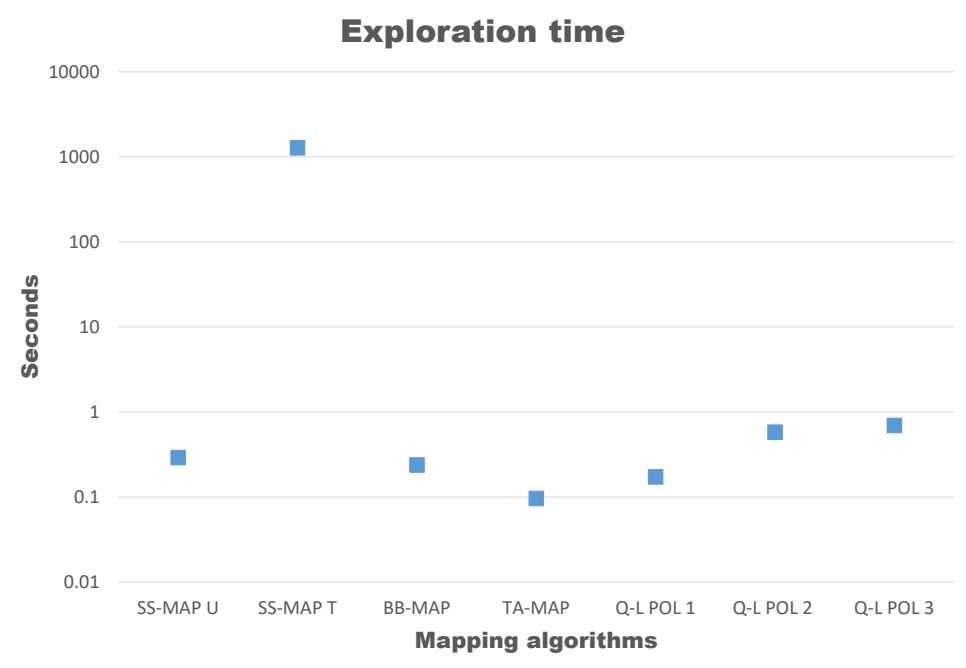


Figure 8.25 – Comparative of the exploration time of the first set of applications and the first set of hardware graphs.

Figure 8.26 shows the training time. Notice the lower training time for reward policy 2. Recall, that this policy emphasizes the parameters verification ($\mathbf{M_{valid}}$).

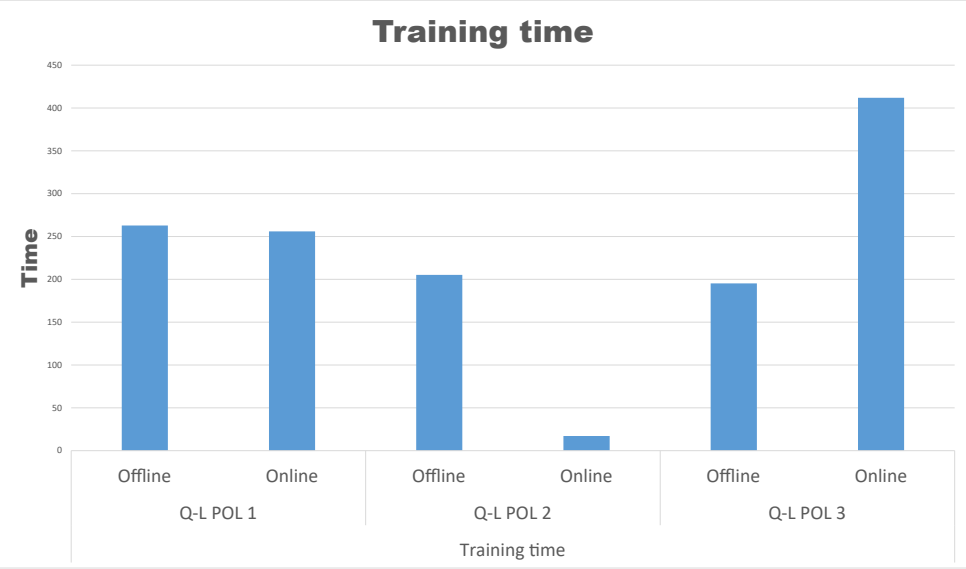


Figure 8.26 – Training time of the Q-learning approaches.

Table 8.7 shows the resulting computing cost of all the mapping algorithms. TA-MAP and

BB-MAP can achieve lower computing costs than both SS-MAP versions. As the complexity of the hardware and application graph increases, the impact of a heuristic is more evident, and in this case, the effect is shown through the computing cost. On the other hand, the Q-learning mapping algorithm can obtain the same results as TA-MAP and BB-MAP. This indicates that the agent can learn and the approach can match the performance of the list-based methods. The lower training time of reward policy 2 is because the agent is focused on neither the complete mapping nor the data dependence of all the tasks. This decreases the number of negative rewards during training, and it can converge faster.

Table 8.7 – Comparative between mapping approaches according to the computing cost (randomly generated graphs)

	Number of tasks	Algorithm	Computing cost (clock cycles)	Exploration time (seconds)
Randomly generated example 1	21	SS-MAP U	9216262	0.291
		SS-MAP T	9216214	1277.667
		TA-MAP	3072211	0.096
		BB-MAP	3072211	0.237
		Q-L POL 1	3072211	0.172
		Q-L POL 2	3072211	0.578
		Q-L POL 3	3072211	0.693

Lastly, in the case of randomly generated graphs, we could not perform the exhaustive algorithm comparison since the exploration time became too long (months).

8.4 Real SPS-CGRA Example

In this section we present the last evaluation of the mapping and scheduling algorithms. We use a real-life SPS-CGRA, the Morphological Co-Processor Unit (MCPU) [12] introduced in Section 1.4. The MCPU assembles several efficient dilation/erosion units with geodesic units and ALUs to support a large collection of morphological operations. It is integrated as a coprocessor in an FPGA-based platform. The MCPU follows the principles of an SPS-CGRA, with its pipeline-based array of processing resources (Large SE pipeline and Geodesic Pipeline).

8.4.1 Hardware Model

Recall that in Section 3.5.5 we presented the modeling methodology of the MCPU as an example. Here, we again depict in Figure 8.27 the hardware model of the MCPU. This is the hardware graph that we will use in the evaluation.

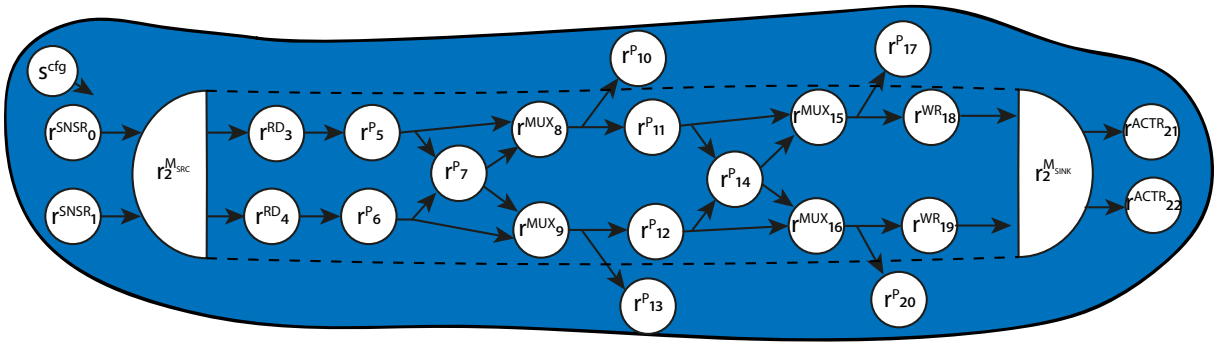


Figure 8.27 – Hardware model of the MCPU

8.4.2 Application Graphs

In Section 2.5, we presented two applications and their modeling methodology. In this Section, we use these two applications and show the obtained results of the mapping and scheduling algorithms. We use the same metrics as the previous evaluations, computing cost and exploration time.

Alternated Sequential Filter

The first example is an Alternated Sequential Filter (ASF) [118], presented in Section 2.5. The ASF is extensively used to smooth objects in images, preserving the topology characteristics. It is known for its computing cost. In our context, it represents a long linear pipeline of tasks with the possibility to overpass the length of the hardware resources (Figure 8.28).

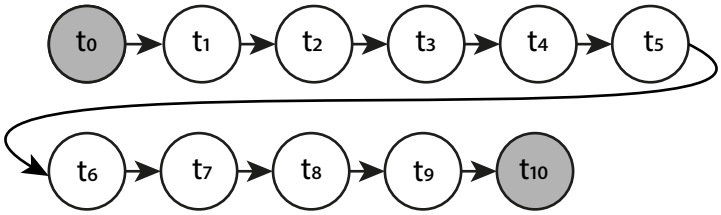


Figure 8.28 – Application model of the example ASF^4.

In this example we consider an ASF with $\lambda = 4 : ASF^4$. Table 8.8 presents the evaluation of the implementation of the ASF^4 onto the MCPU. In this first example, the pipeline of tasks exceeds the number of processing resources of a single time slot, so the necessary time slots had been added automatically. As we can see in Table 8.8, the exploration time of the proposed mapping and scheduling algorithms are significantly lower than the one of the exhaustive algorithm. In Table 8.8, N/A means not applicable. In terms of computing cost, the list-based algorithms can achieve the resulting value of the exhaustive. The Q-learning mapping algorithm only achieves a near to optimal value with the second reward policy (see Table 8.1), which prioritize the verification of the parameters. A possible reason that the Q-learning algorithms do not achieve good results is that the application is a pipeline. Thus it is formed by several subgraphs with the same characteristic. This is the worst-case scenario for the Q-learning because the agent cannot differentiate the tasks and find the best allocation for them.

Table 8.8 – Algorithms evaluation for the ASF application

	Number of tasks	Algorithm	Computing cost (clock cycles)	Exploration time (seconds)	Error (%)
Alternated Sequential Filter (Pipeline of tasks)	9	Exhaustive algorithm	2790499	47 hours	N/A
		SS-MAP U	2790499	0.0707	0%
		SS-MAP T	2790499	122.059	0%
		TA-MAP	2790499	0.0832	0%
		BB-MAP	2790499	0.08319	0%
		Q-L POL 1	5555313	0.269	100%
		Q-L POL 2	2790500	2.99	0%
		Q-L POL 3	5555307	0.109	100%

Road Line Detection

The second example in this evaluation is the road line detection application introduced in Section 2.5.2. This second application represents a highly parallel task organization. The principle is

the computing of oriented linear openings of the input. As presented in 2.5.2, we only focus on the part of the application that can be allocated onto the MCPU. Again, we present, in Figure 8.29 the application model that we will use in this evaluation.

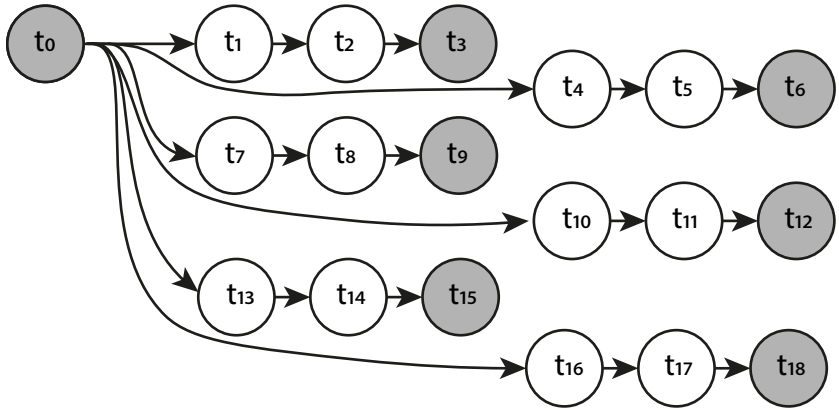


Figure 8.29 – Application model of the example of the road line orientation detection.

The large number of tasks and structure of the road line detection application represents a challenge for the mapping algorithms. The exploration time for the complete application is too huge using the exhaustive algorithm. But, the parallel structure of each linear subgraph allows us to map each subgraph separately and search for the best combination of subgraphs in a single time slot to achieve the lowest computing cost. Making these considerations, we manually divide the application into a set of fourth linear subgraphs. For each set, we evaluate the resulting mappings. Table 8.9 summarizes the results (N/A means not applicable). Notice that, in this example, the Q-learning approaches work better than the other algorithms, especially for the complete application (six subgraphs). The Q-learning algorithm can obtain results near to the optimal. On the other hand, the results from the list-based algorithms are also good, approaching by less than two digits percentage to the results of the exhaustive algorithm. In terms of exploration time, the Q-learning approach can obtain good results. In some cases, it is able to decrease the exploration time of other approaches by half.

Table 8.9 – Algorithms evaluation for the Road line detection application

	Number of tasks	Algorithm	Computing cost (clock cycles)	Exploration time (seconds)	Error (%)
Three subgraphs t_0, t_1, t_2, t_3 t_4, t_5, t_6 t_7, t_8, t_9	6	Exhaustive	1853468	141.447	N/A
		SS-MAP U	1859869	0.082	0.345%
		SS-MAP T	1859869	71.9235	0.345%
		TA-MAP	1859868	0.117	0%
		BB-MAP	1859869	0.0869	0.345%
		Q-L POL 1	1853469	0.48866	0.345%
		Q-L POL 2	1853469	0.4560	0.345%
		Q-L POL 3	1859869	0.4377	0.345%
Four subgraphs t_{10}, t_{11}, t_{12}	8	Exhaustive	1859868	7058.679	N/A
		SS-MAP U	2794281	0.079	50%
		SS-MAP T	1972669	98.09	6%
		TA-MAP	1872668	0.096	0.68%
		BB-MAP	1872669	0.0727	0.68%
		Q-L POL 1	1859869	0.468	0%
		Q-L POL 2	1859869	0.058	0%
		Q-L POL 3	1872669	0.053	0.68%
Five subgraphs t_{13}, t_{14}, t_{15}	10	Exhaustive	2791721	192396.421	N/A
		SS-MAP U	2798121	0.126	0.22%
		SS-MAP T	2798121	154.133	0.22%
		TA-MAP	2800681	0.152	0.32%
		BB-MAP	2800683	0.0809	0.32%
		Q-L POL 1	2798123	0.072	0.22%
		Q-L POL 2	2798123	0.068	0.22%
		Q-L POL 3	2794283	0.071	0.09%
Six subgraphs t_{16}, t_{17}, t_{18}	12	Exhaustive	2791721	+30 days	N/A%
		SS-MAP U	3732534	0.142	33.7%
		SS-MAP T	2804523	162.216	0.45%
		TA-MAP	2810921	0.144	0.68%
		BB-MAP	2810923	0.086	0.68%
		Q-L POL 1	3737660	0.087	33.88%
		Q-L POL 2	2798123	0.069	0.22%
		Q-L POL 3	2798123	0.08	0.22%

Figure 8.30 shows graphically the exploration time for both applications, where we can see that the average exploration time for all algorithms, except SS-MAP T, is less than one second.

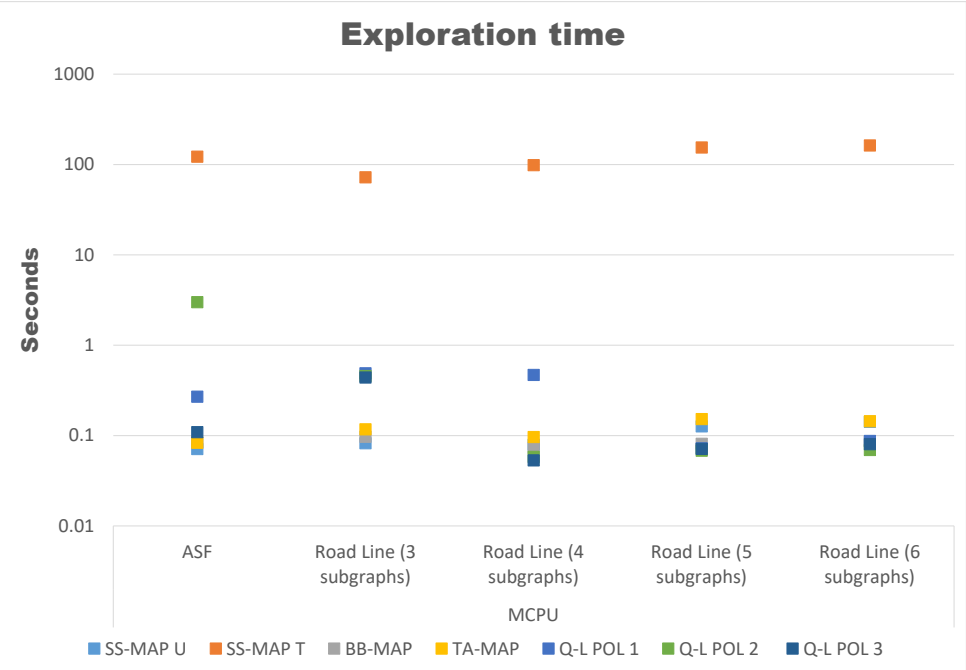


Figure 8.30 – Comparative of the exploration time of the first set of applications and the first set of hardware graphs.

Figure 8.31 shows the training time of the Q-learning approaches. Given that the complexity of the structure of the MCPU, the resulting randomly generated applications are usually bigger than the target application graphs. Clearly, the offline training time is longer than the online training.

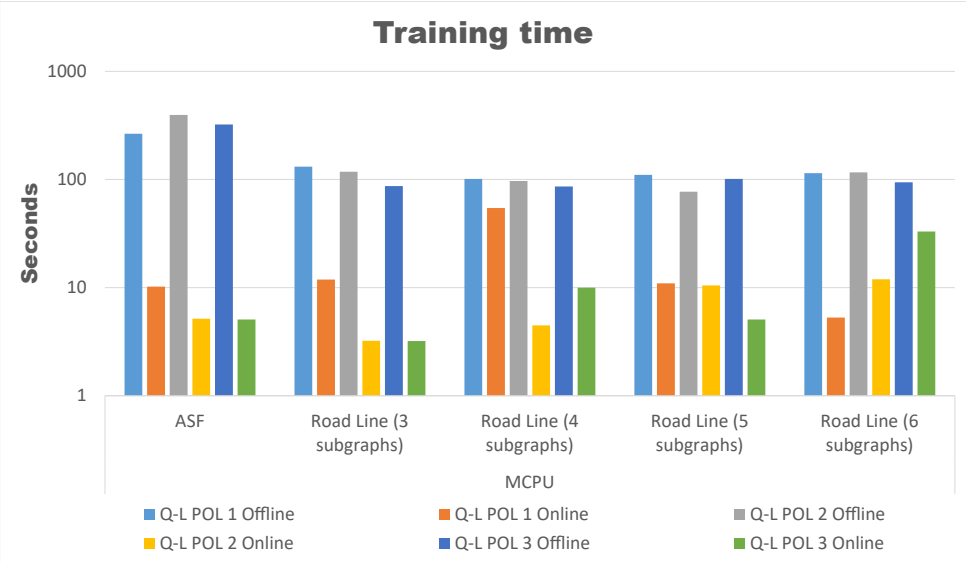


Figure 8.31 – Training time of the Q-learning approaches.

8.5 Conclusions

In this experimental evaluation, we demonstrate the benefits of using our mapping and scheduling algorithms. We present three types of experiments that took several months to be performed. In the first one, we use several sets of experimental graphs to exhibit the performance and exploration time of the mapping algorithms. Next, we present an evaluation using a randomly generated couple of hardware and application graphs. This evaluation demonstrates that our

algorithms can be used for complex hardware interconnectivity and highly data-dependent applications. Finally, we use a real-life SPS-CGRA to show that our methodology can be used for real-life cases.

Chapter 9

Conclusions and Perspectives

In this manuscript, we presented a complete framework for easy reuse of SPS-CGRA. Effectively, we propose to name SPS-CGRA a spatially-configured overlay [23], which may be based on an FPGA, ASICs or System on Programmable Chip (SoC). More precisely, such SPS-CGRA is a systolic, highly pipelined array of heterogeneous hardware resources that provides high computing performance while decreasing computing latency. This hardware platform may be reconfigured through software parameters programming without low-level reconfiguration. We consider them as a highly pipelined hardware platform specialized in stream-based processing. We presented a taxonomy of SPS-CGRA based on the works of Liu et al. [18], Wijtvliet et al. [22], Hartenstein [58] and Chattopadhyay [60]. Our effort was to situate the SPS-CGRA and its features in a known context, the one of CGRAs, since most of the SPS-CGRAs are within the CGRA domain. Moreover, this taxonomy helps to exhibit the main features of an SPS-CGRA that can also be found in an FPGA, ASIC, or SoC contexts. We presented several representative examples of SPS-CGRA from various domains (deep learning, machine vision, and others). We demonstrate that the main features of an SPS-CGRA are shared between all the examples. Moreover, a generic framework of SPS-CGRA is an important contribution given that no generic mature tool is available to reuse them easily.

The works realized during this manuscript have been presented into two main parts: modeling and mapping algorithms. In addition, a list of papers and communications published during this thesis is given on page 174.

9.1 Modeling

In Part I, we presented a new unified graph-based modeling framework for SPS-CGRA. This modeling framework is based on hypergraphs and covers application, hardware, and implementation models. The use of hypergraphs allows the broadcast of data (application model) and multiple types of interconnections (hardware and implementation model). The use of F-hyperedges models the broadcast or interconnection of one origin to several successors, and the use of an E-hyperedge that models the data dependence of one task with several predecessors or the interconnection between a hardware resource and several predecessors.

In Chapter 2, we presented the application model, a specilization of the PiSDF model and the AAM methodology. We presented the current state of the art of application models. Within this study, we considered models that apply to other platforms such as multicore processors and MPSoCs. We argue that a graph-based model, and specifically an extension of the PiSDF model, could fulfill our requirements. We provide the means to model heterogeneous tasks with different types of parameters (numerical constant, a boolean value, a string of characters). Additionally, we include two subsets of tasks called **sensor** and **actuators**. These subsets allow modeling the source and sink of data. This information is matched to their counterparts in the hardware model. It ensures that the data to be processed is originated from the expected source and consumed from the expected sink. We presented two real-life examples from the image

processing domain. These two examples are well-known computing cost demanding and targets SPS-CGRA based platforms. We illustrated the modeling process and demonstrated that our proposed model is able to abstract both examples accurately.

In Chapter 3, we presented the hardware model for SPS-CGRA. It focuses on the modeling of the latency properties of the hardware resources, all possible datapaths, and the memories. We also proposed an original modelization of the configuration control resources, usually a set of registers. The result is a new type of sequencer node that we named s^{CFG}. This sequencer node includes information on the configuration control function of all the hardware resources. This important contribution allows to consider the configuration cost during the performance evaluation of the whole application. We also showed that this is a feature that most of the current work in this field do not include. Since the SPS-CGRA process the data in a stream-like fashion, we model the latency of the processing resources through a composition of two functions that we developed. Hence, we proposed to use an input latency function and a computing latency function to compute the latency of each processing resource. Both latency functions depend on the physical realizations of the hardware resource and the parameters of the allocated task. Through several examples, we demonstrated how this contribution allows to compute accurate estimations of the computing cost in terms of execution duration. Additionally, we include the means to model the memories, the processing resources, and possible datapaths connecting them. We illustrate the global modeling process through several experimental examples and one real-life SPS-CGRA from the image processing domain.

After the definition of the new models of application and hardware architecture, in Chapter 4 we introduced the implementation model, together with a performance evaluation methodology to compute an estimate of the upper bound of the computing cost of the implementation of an application onto an SPS-CGRA. In line with the application and hardware graph, the implementation model consists of a weighted directed hypergraph, where the weight of the hyperedges is the input latency of the head node. The implementation graph consists of instances of the hardware graph configured to execute a subset of tasks. We named time slot each instance of the hardware graph. We also introduced the notion of data dependency resources. This subset of virtual hardware resources allows to model data dependency between datapaths, within the same time slot, and data dependencies between time slots. This contribution helps to identify a possible recomputation or reuse of data between time slots. Each hardware resource of the implementation graph is described with the parameters and latency values defined during the mapping and scheduling process. The implementation graph carries information on spatial and temporal allocation of tasks, and also about latency values. This contribution allows not only to execute a performance evaluation efficiently over the implementation graph using well-known graph-based algorithms but also allows to optimize the resource usage by reuse/reprogramming it into the same application. We presented a performance evaluation methodology that aims to estimate the upper bound of the computing cost. This methodology consists of building the evaluation graph, which results from graph transformations of the implementation graph, and the compute of the critical path over this evaluation graph. The critical path is computed using an original equation presented in this work. We demonstrated how this contribution allows to compute the estimated upper bound of the computing cost at cycle accuracy. We presented several experimental examples to illustrate the building process of the evaluation graph and one example that shows the entire performance evaluation methodology.

9.2 Mapping Algorithms

After the definition of three new models in the first part of the manuscript, we need to exploit them to build an optimized implementation of an application onto an architecture. This NP-complete allocation and scheduling problem is covered in the second part of the manuscript.

Hence, in part II, we presented four mapping algorithms. We divide Part II into three parts: list-based mapping algorithms, Q-learning mapping algorithm, and evaluation of mapping

algorithms.

In Chapter 6, we presented three mapping algorithms of increasing complexity but also of increasing efficiency. The list-based algorithms are differentiated upon the heuristic that they use to select the processing resource that will execute a certain task. Firstly, we introduce SS-MAP (SS-MAP stands for Single-Shot Mapping algorithm). This simple and relatively efficient mapping algorithm takes as priority the topological order of both the hardware resources and tasks. SS-MAP is defined into two versions, one with a single topological sorting of the hardware graph and the second version exploring all possible topological sortings. The reasoning behind this is that there may be several topological sortings of the hardware graph, and using only one will impact the optimality of the final mapping. Using all topological sortings and selecting the best mapping decreases the probability of failure to achieve a sub-optimal mapping. Since SS-MAP does not include any heuristic and the outcome of this algorithm is based only on the topological sorting it is a slow algorithm. This is why we developed and experimented with two new different heuristics to improve its performance.

Our first heuristic, named TA-MAP (TA-MAP stands for Topology-Aware Mapping algorithm), uses look-ahead techniques to select a resource, from a set of candidates, that will execute a certain task. The heuristic is based on information about latency (computing latency) and topological distances of the candidates. Then, this heuristic focuses on the possible allocation of the successors of the task onto the descendants of the resource. This contribution allows to select the best resource to execute a task, and furthermore, define if the successors of the task will be able to be allocated onto the descendants of the selected resource. This feature allows to eliminate the backtracking process used in the SS-MAP.

In order to get better results, we developed and implemented a second heuristic that we named BB-MAP (BB-MAP stands for Bayes-Based Mapping algorithm). The BB-MAP is a mapping algorithm based on the Bayes Theorem. We formulated the SPS-CGRA mapping problem as: given that a task is allocated to a resource, what is the probability that the successors of the task will be allocated onto the descendants of the resource. This formulation is based on the previous heuristic, however, formalized into a Bayes problem. Although the results obtained with BB-MAP are better (in some cases, a 90% decrease in exploration time without impact to the computing cost) than SS-MAP and TA-MAP, we wanted to go further.

This is why in Chapter 7, we studied a Q-learning mapping algorithm. We formalize the SPS-CGRA mapping problem as a reinforcement learning problem. This contribution allows the use of the agent, environment, reward policy, and training policy, which are part of the formalization, for other types of reinforcement learning algorithms. We define the mapping function as the agent and the hardware graph as the environment. The agent will traverse the hardware graph allocating tasks, and for each allocation, he will get a reward depending upon the optimality of the allocation. We introduce a hierarchical reward policy. This contribution helps the agent's learning process, as it includes particular rewards for matching parameters, degree matching, computing latency, and others. Also, we proposed a mix-nature training, where an offline and an online training are performed. This contribution reduces the need for a big data set. We used a random graph generator to create experimental graphs that will be used for the offline training. During the online training, we use the application that we wanted to map.

Finally, the last chapter is devoted to the experimental results. Effectively, all the models presented in this manuscript have been implemented in a software tool developed from scratch using the Python language. For that purpose, we also developed a new format of description-configuration file. Then we implemented each allocation and scheduling algorithm variant (SS-MAP, TA-MAP, BB-MAP, Q-Learning) progressively to compare them. The comparison is made on two axes: the quality of the resulting mapping and scheduling, but also in terms of the execution duration of each optimization algorithm. In order to validate our results on the largest possible set of application and hardware couples, we developed a parametric random graph generator. We also experimented with our tools on a real-life example taken from the

image processing area, which is computing power demanding. Finally, all the comparative results show the effectiveness of our approach: the BB-MAP/TA-MAP are well adapted for large graphs that are usually computed in less than 200 milliseconds. BB-MAP should be preferred for highly data-dependent applications (application graphs with an input/output degree higher than 3). At the same time, TA-MAP seems to be more adapted to highly pipelined applications. Q-Learning behaves better but requires many more resources (q-table memory usage) for its execution on large applications.

9.3 Perspectives

We define the hardware graph as $G_{HW}(S, K)$, where the set of nodes (S) models the hardware resources and the set of oriented hyperedges (K) models the hardware resources interconnections. In this formalization, the hyperedges do not include any information of interconnection nor standard. The hardware model may be enhanced by including the type of interconnection (direct, bus) and standard of such interconnection (AXI, Avalon, Wishbone, etc.). We detail that the latency of the processing resource consists of two functions, input latency function, and computing latency function. This concept is well suited to stream-based processing. However, it may not be suitable for others. A possible enhancement would be the concept of several latencies functions, where different physical hardware realizations and different kinds of processing may coexist.

In this work, we proposed a new format of configuration files. These configuration files are the inputs of the complete framework. They describe both the application and the hardware. A possible enhancement of our tool would be to use a well-known standard for the description of both the application and hardware. IP-XACT is a format that defines and describes reusable electronic circuit designs [232]. While IP-XACT can not be directly applied to describe an SPS-CGRA, an extension may describe the latency features. Several works propose extensions to IP-XACT to exploit it, as in [233].

We proved that, the list-based mapping algorithms presented in this work are efficient and able to achieve a sub-optimal final mapping. Despite these features, our proposed q-learning mapping algorithm may have better potential. It is able to learn different structures and achieve better results on more complex structures than the list-based algorithms. However, as number of nodes of both application and hardware graph increases, the Q-table increases as well. A bottleneck arises when the Q-table starts to grown and saturates the RAM memory of the system where the mapping is running. A possible enhancement is to use a deep q-learning approach. With the use of a neural network, the required RAM usage decreases, thus allowing to use the mapping algorithm for bigger graphs (both application and hardware). Additionally, the formalization of the SPS-CGRA mapping problem into a reinforcement learning problem may be suitable for other hardware platforms such as MPSoCs or multicore processors.

The presented evaluation validate our proof of concept of a complete framework able to support an emerging class of architecture: the SPS-CGRA.

Publications and Communications

International Peer-Reviewed Journal

1) Elias Barbudo, Thierry Grandpierre, Eva Dokladalova, *"Modelling and Mapping Framework for Software Programmable Streaming Coarse Grained Reconfigurable Architectures",* **IEEE Transactions on Computers (in preparation for submission May 2021)**

International Peer-Reviewed Conferences

2) Elias Barbudo, Eva Dokladalova, Thierry Grandpierre, Laurent George, *"A New Mapping Methodology for Coarse-Grained Programmable Systolic Architectures",* **22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES 2019), May 2019, St Goar, Germany (https://hal-upecupem.archives-ouvertes.fr/hal-02013560v1).**

3) Elias Barbudo, Eva Dokladalova, Thierry Grandpierre, Laurent George, *"Mapping Methodology for Coarse-Grained Pipelined Configurable Architectures",* **14th Workshop on Models and Algorithms for Planning and Scheduling Problems (MAPSP 2019), Jun 2019, Renesse, Netherlands (https://hal-upecupem.archives-ouvertes.fr/hal-02104162v1).**

International Seminar

4) Elias Barbudo, Eva Dokladalova, Thierry Grandpierre, *"Modelling and Mapping Framework for Coarse-Grained Programmable Architectures",* **14th Summer School on Modelling and Verification of Parallel Process (MOVEP 2020), June 2020, Virtual Summer School.**

National Peer-Reviewed Colloquiums and Conferences

5) Elias Barbudo, Eva Dokladalova, Thierry Grandpierre, *"A New Modelling Framework for Coarse-Grained Programmable Architectures",* **Conférence Francophone D'informatique en Parallélisme, Architecture et Système (COMPAS 2020), July 2020, Lyon, France (Accepted, conference cancelled due to health conditions).**

6) Elias Barbudo, Eva Dokladalova, Thierry Grandpierre, Laurent George, *"A mapping tool for configurable pipeline co-processors",* **Colloque National du GDR SoC-SiP, June 2018, Paris, France (https://hal-enpc.archives-ouvertes.fr/hal-01801018).**

Others

7) Elias Barbudo *"Towards Software Programmable Streaming Coarse-Grained Reconfigurable Architectures Efficient Re-use",* **Atelier des doctorants, Laboratoire d'Informatique Gaspard Monge, March 2021.**

8) Elias Barbudo, Eva Dokladalova, Thierry Grandpierre, Laurent George, *"A mapping tool for configurable pipeline co-processors",* **Journée des Doctorants ESIEE, June 2018, Paris, France.**

Bibliography

[1] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Epimap: Using epimorphism to map applications on cgras," in *DAC Design Automation Conference 2012*, pp. 1280–1287, 2012.

[2] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings, "A reconfigurable arithmetic array for multimedia applications," in *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, (New York, NY, USA), p. 135–143, Association for Computing Machinery, 1999.

[3] G. Lu, H. Singh, M.-h. Lee, N. Bagherzadeh, F. Kurdahi, and E. M. C. Filho, "The morphosys parallel reconfigurable system," in *Euro-Par'99 Parallel Processing* (P. Amestoy, P. Berger, M. Daydé, D. Ruiz, I. Duff, V. Frayssé, and L. Giraud, eds.), (Berlin, Heidelberg), pp. 727–734, Springer Berlin Heidelberg, 1999.

[4] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix," in *Field Programmable Logic and Application* (P. Y. K. Cheung and G. A. Constantinides, eds.), (Berlin, Heidelberg), pp. 61–70, Springer Berlin Heidelberg, 2003.

[5] B. Mei, A. Lambrechts, J. . Mignolet, D. Verkest, and R. Lauwereins, "Architecture exploration for a reconfigurable architecture template," *IEEE Design Test of Computers*, vol. 22, no. 2, pp. 90–101, 2005.

[6] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "Piperench: a coprocessor for streaming multimedia acceleration," in *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, pp. 28–39, 1999.

[7] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "Piperench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[8] J. Tetu, L. Trudeau, M. Van Beirendonck, A. Balatsoukas-Stimming, and P. Giard, "A standalone fpga-based miner for lyra2rev2 cryptocurrencies," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 4, pp. 1194–1206, 2020.

[9] N. Ngan, E. Dokladalova, and M. Akil, "Dynamically adaptable noc router architecture for multiple pixel streams applications," in *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1006–1009, 2012.

[10] A. Isavudeen, N. Ngan, E. Dokladalova, and M. Akil, "Highly Scalable Monitoring System on Chip for Multi-Stream Auto-Adaptable Vision System," in *Research in Adaptive and Convergent Systems 2017, RACS 2017*, ACM Proceedings Research in Adaptive and Convergent Systems 2017, (Krakow, Poland), ACM SIGAPP, ACM, Sept. 2017.

[11] P. Possa, N. Harb, E. Dokládalová, and C. Valderrama, "P2ip: A novel low-latency programmable pipeline image processor," *Microprocessors and Microsystems*, vol. 39, no. 7, pp. 529 – 540, 2015.

[12] J. Bartovsky, P. Dokládal, M. Faessel, E. Dokladalova, and M. Bilodeau, "Morphological Co-Processing Unit for Embedded Devices," *Journal of Real-Time Image Processing*, pp. pp. 1–12, July 2015.

[13] J. Chang, K. Kang, and S. Kang, "An energy-efficient fpga-based deconvolutional neural networks accelerator for single image super-resolution," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 1, pp. 281–295, 2020.

[14] D. Piyasena, M. Thathsara, S. Kanagarajah, S. K. Lam, and M. Wu, "Dynamically growing neural network architecture for lifelong deep learning on the edge," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 262–268, 2020.

[15] J. Bartovský, P. Dokládal, E. Dokládalová, and M. Bilodeau, "One-scan algorithm for arbitrarily oriented 1-d morphological opening and slope pattern spectrum," in *2012 19th IEEE International Conference on Image Processing*, pp. 133–136, 2012.

[16] M. Keating and P. Bricaud, *Reuse Methodology Manual: For System-on-a-Chip Designs.* USA: Kluwer Academic Publishers, 1998.

[17] J. Morris Chang and S. Kagan Agun, "On design-for-reusability in hardware description languages," in *Proceedings IEEE Computer Society Workshop on VLSI 2000. System Design for a System-on-Chip Era*, pp. 103–108, 2000.

[18] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surv.*, vol. 52, Oct. 2019.

[19] M. S. Mansureh, J.-M. Cho, and K. Choi, *Reconfigurable Architectures*, pp. 1–42. Dordrecht: Springer Netherlands, 2017.

[20] T. Mitra, *Application-Specific Processors*, pp. 377–409. Dordrecht: Springer Netherlands, 2017.

[21] M. W. et al., "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 235–244, July 2016.

[22] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pp. 235–244, 2016.

[23] A. K. Jain, D. L. Maskell, and S. 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)*, pp. 586–593, 2016.

[24] K. Vipin and S. A. Fahmy, "Mapping adaptive hardware systems with partial reconfiguration using copr for zynq," in *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 1–8, 2015.

[25] H. Singh, Ming-Hau Lee, Guangming Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[26] P. Haghi, T. Geng, A. Guo, T. Wang, and M. Herbordt, "Fp-amg: Fpga-based acceleration framework for algebraic multigrid solvers," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 148–156, 2020.

[27] T. Yiyu, Y. Inoguchi, Y. Sato, M. Otani, Y. Iwaya, H. Matsuoka, and T. Tsuchiya, "Design of a fpga-based timing sharing architecture for sound rendering applications," in *2012 Ninth International Conference on Information Technology - New Generations*, pp. 484–489, 2012.

[28] Y. Tan and T. Imamura, "An fpga-based sound field rendering system," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 414–415, 2020.

[29] N. Ngan, E. Dokladalova, M. Akil, and F. Contou-Carrerè, "Dynamically adaptable architecture for real-time video processing," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 4125–4128, 2010.

[30] E. Dokladalova, R. Schmit, S. Pajaniradja, and S. Amadori, "Carvision: Soc architecture for dynamic vision systems from image capture to high level image processing," in *MEDEA+ Design Automation Conference*, June 2006.

[31] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *CVPR 2011 WORKSHOPS*, pp. 109–116, 2011.

[32] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 553–564, 2017.

[33] R. Kuramochi and H. Nakahara, "An fpga-based low-latency accelerator for randomly wired neural networks," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 298–303, 2020.

[34] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. D. Dinechin, "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources," *Real-Time Syst.*, vol. 52, p. 399–449, July 2016.

[35] T. Grandpierre and Y. Sorel, "From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings.*, pp. 123–132, 2003.

[36] L. Kaouane, M. Akil, and Y. Sorel, "An automated design flow for optimized implementation of real-time image processing applications onto fpga," in *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, vol. 1, pp. 71–75 vol.1, 2003.

[37] L. Kaouane, M. Akil, T. Grandpierre, and Y. Sorel, "A methodology to implement real-time applications onto reconfigurable circuits," *The journal of supercomputing*, vol. 30, no. 3, pp. 283–301, 2004.

[38] P. Niang, T. Grandpierre, and M. Akil, "Implementing real-time algorithms by using the aaa prototyping methodology," in *Embedded System Design: Topics, Techniques and Trends* (A. Rettberg, M. C. Zanella, R. Dömer, A. Gerstlauer, and F. J. Rammig, eds.), (Boston, MA), pp. 27–36, Springer US, 2007.

[39] Y. Elloumi, M. Akil, T. Grandpierre, and M. Bedoui, "Latency and power optimization in aaa methodology for integrated circuits," in *2010 17th IEEE International Conference on Electronics, Circuits and Systems*, pp. 639–642, 2010.

[40] C. Jalier, D. Lattard, and G. Sassatelli, "A flexible modeling and simulation framework for design space exploration," in *2008 International Symposium on System-on-Chip*, pp. 1–4, 2008.

[41] J. Castrillon, R. Leupers, and G. Ascheid, "Maps: Mapping concurrent dataflow applications to heterogeneous mpsocs," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 527–545, 2013.

[42] O. Jovanovic, P. Marwedel, I. Bacivarov, and L. Thiele, "Mamot: Memory-aware mapping optimization tool for mpsoc," in *2012 15th Euromicro Conference on Digital System Design*, pp. 743–750, 2012.

[43] B. Dauphin, R. Pacalet, A. Enrici, and L. Apvrille, "Odyn: Deadlock prevention and hybrid scheduling algorithm for real-time dataflow applications," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pp. 88–95, 2019.

[44] L. Suriano, A. Rodriguez, K. Desnos, M. Pelcat, and E. de la Torre, "Analysis of a heterogeneous multi-core, multi-hw-accelerator-based system designed using preesm and sdsoc," in *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pp. 1–7, 2017.

[45] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. Nezan, and S. Aridhi, "Preesm: A dataflow-based rapid prototyping framework for simplifying multicore dsp programming," in *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pp. 36–40, 2014.

[46] K. Bruneel, F. Abouelella, and D. Stroobandt, "Automatically mapping applications to a self-reconfiguring platform," in *2009 Design, Automation Test in Europe Conference Exhibition*, pp. 964–969, 2009.

[47] F. Fricke, A. Werner, K. Shahin, F. Werner, and M. Hübner, "Automatic tool-flow for mapping applications to an application-specific cgra architecture," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 147–154, 2019.

[48] T. Peyret, G. Corre, M. Thevenin, K. Martin, and P. Coussy, "An automated design approach to map applications on cgras," in *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, GLSVLSI '14, (New York, NY, USA), p. 229–230, Association for Computing Machinery, 2014.

[49] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "Cgra-me: A unified framework for cgra modelling and exploration," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 184–189, 2017.

[50] M. Canesche, M. Menezes, W. Carvalho, F. S. Torres, P. Jamieson, J. A. Nacif, and R. Ferreira, "Traversal: A fast and adaptive graph-based placement and routing for cgras," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 8, pp. 1600–1612, 2021.

[51] Z. Li, D. Wijerathne, X. Chen, A. Pathania, and T. Mitra, "Chordmap: Automated mapping of streaming applications onto cgra," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2021.

[52] C. Tan, C. Xie, T. Geng, A. Marquez, A. Tumeo, K. Barker, and A. Li, "Arena: Asynchronous reconfigurable accelerator ring to enable data-centric parallel computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 12, pp. 2880–2892, 2021.

[53] M. Shen, G. Luo, and N. Xiao, "Coarse-grained parallel routing with recursive partitioning for fpgas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 884–899, 2021.

[54] M. H. Sargolzaei and S. Mohammadi, "Energy efficient configuration unification and compression for cgras," *Microprocessors and Microsystems*, vol. 62, pp. 1–11, 2018.

[55] T. Stansfield, "Using multiplexers for control and data in d-fabrix," in *Field Programmable Logic and Application* (P. Y. K. Cheung and G. A. Constantinides, eds.), (Berlin, Heidelberg), pp. 416–425, Springer Berlin Heidelberg, 2003.

[56] H. Singh, Ming-Hau Lee, Guangming Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[57] M. Jo, Dongwook Lee, and K. Choi, "Chip implementation of a coarse-grained reconfigurable architecture supporting floating-point operations," in *2008 International SoC Design Conference*, vol. 03, pp. III–29–III–30, 2008.

[58] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pp. 642–649, 2001.

[59] V. Tehre and R. Kshirsagar, "Article: Survey on coarse grained reconfigurable architectures," *International Journal of Computer Applications*, vol. 48, pp. 1–7, June 2012. Full text available.

[60] A. Chattopadhyay, "Ingredients of adaptability: A survey of reconfigurable processors," *VLSI Des.*, vol. 2013, Jan. 2013.

[61] C. L. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle, "A comparison of heuristics for scheduling dags on multiprocessors," in *Proceedings of 8th International Parallel Processing Symposium*, pp. 446–451, 1994.

[62] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pp. 404–411, 2011.

[63] B. Bhattacharya and S. Bhattacharyya, "Parameterized dataflow modeling for dsp systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, 2001.

[64] J. Piat, S. S. Bhattacharyya, and M. Raulet, "Interface-based hierarchy for synchronous data-flow graphs," in *2009 IEEE Workshop on Signal Processing Systems*, pp. 145–150, 2009.

[65] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya, and S. Aridhi, "Pimm: Parameterized and interfaced dataflow meta-model for mpsocs runtime reconfiguration," in *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 41–48, 2013.

[66] S. C. Lo and S. N. Jean, "Mapping algorithms to vlsi array processors," in *ICASSP-88., International Conference on Acoustics, Speech, and Signal Processing*, pp. 2033–2036 vol.4, 1988.

[67] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, p. 46–61, Jan. 1973.

[68] J. Marinho, V. Nélis, S. M. Petters, M. Bertogna, and R. I. Davis, "Limited pre-emptive global fixed task priority," in *2013 IEEE 34th Real-Time Systems Symposium*, pp. 182–191, 2013.

[69] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese, "A generalized parallel task model for recurrent real-time processes," in *2012 IEEE 33rd Real-Time Systems Symposium*, pp. 63–72, 2012.

[70] J. Marinho and S. M. Petters, "Timing analysis for dag-based and GFP scheduled tasks," *CoRR*, vol. abs/1406.1133, 2014.

[71] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *2010 31st IEEE Real-Time Systems Symposium*, pp. 259–268, 2010.

[72] D. Kim, J. Lee, J. Lee, I. Shin, J. Kim, and S. Yoon, "Scheduling in heterogeneous computing environments for proximity queries," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 9, pp. 1513–1525, 2013.

[73] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *2011 IEEE 32nd Real-Time Systems Symposium*, pp. 217–226, Nov 2011.

[74] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 71–80, 2011.

[75] S. Baruah, "The non-cyclic recurring real-time task model," in *2010 31st IEEE Real-Time Systems Symposium*, pp. 173–182, 2010.

[76] B. Peng and N. Fisher, "Parameter adaption for generalized multiframe tasks and applications to self-suspending tasks," in *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 49–58, 2016.

[77] J. Chen, "Task set synthesis with cost minimization for sporadic real-time tasks," in *2013 IEEE 34th Real-Time Systems Symposium*, pp. 350–359, 2013.

[78] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 269–279, 2011.

[79] R. Tabish, R. Mancuso, S. Wasly, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A reliable and predictable scratchpad-centric os for multi-core embedded systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 377–388, 2017.

[80] C. Maia, G. Nelissen, L. Nogueira, L. M. Pinho, and D. G. Pérez, "Schedulability analysis for global fixed-priority scheduling of the 3-phase task model," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 1–10, 2017.

[81] G. C. Sih and E. A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175–187, 1993.

[82] P. Derler, K. Ravindran, and R. Limaye, "Specification of precise timing in synchronous dataflow models," in *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pp. 85–94, 2016.

[83] M. Pelcat, A. Mercat, K. Desnos, L. Maggiani, Y. Liu, J. Heulot, J. Nezan, W. Hamidouche, D. Ménard, and S. S. Bhattacharyya, "Reproducible evaluation of system efficiency with a model of architecture: From theory to practice," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 10, pp. 2050–2063, 2018.

[84] M. Ammar, M. Baklouti, M. Pelcat, K. Desnos, and M. Abid, "Marte to pisdf transformation for data-intensive applications analysis," in *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing*, pp. 1–8, 2014.

[85] Y. Sorel, "Massively parallel computing systems with real time constraints: the "algorithm architecture adequation" methodology," in *Proceedings of the First International Conference on Massively Parallel Computing Systems (MPCS) The Challenges of General-Purpose and Special-Purpose Computing*, pp. 44–53, 1994.

[86] O. Feki, T. Grandpierre, M. Akil, N. Masmoudi, and Y. Sorel, "Syndex-mix: A hardware/software partitioning cad tool," in *2014 15th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*, pp. 247–252, 2014.

[87] L. Qin, F. Ouyang, and G. Xiong, "Dependent task scheduling algorithm in distributed system," in *2018 4th International Conference on Computer and Technology Applications (ICCTA)*, pp. 91–95, 2018.

[88] G. Wang, H. Guo, and Y. Wang, "A novel heterogeneous scheduling algorithm with improved task priority," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 1826–1831, 2015.

[89] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo, "Response-time analysis of conditional dag tasks in multiprocessor systems," in *2015 27th Euromicro Conference on Real-Time Systems*, pp. 211–221, 2015.

[90] M. Qamhieh, L. George, and S. Midonnet, "Stretching algorithm for global scheduling of real-time dag tasks," *Real-Time Syst.*, vol. 55, p. 32–62, Jan. 2019.

[91] 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*, vol. 13, no. 3, pp. 260–274, 2002.

[92] L. Yuan, S. Wei, M. Wang, and H. Yu, "Fairness-aware scheduling algorithm for multiple dags based on task replication," in *2018 9th International Conference on Information and Communication Systems (ICICS)*, pp. 169–174, 2018.

[93] R. Pathan, P. Voudouris, and P. Stenström, "Scheduling parallel real-time recurrent tasks on multicore platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 915–928, 2018.

[94] V. Zadrija and V. Sruk, "Mapping algorithms for mpsoc synthesis," in *The 33rd International Convention MIPRO*, pp. 624–629, 2010.

[95] N. Frid and V. Sruk, "Critical path method based heuristics for mapping application software onto heterogeneous mpsocs," in *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1030–1034, 2014.

[96] N. Frid and V. Sruk, "Memory-aware multiobjective design space exploration of heteregeneous mpsoc," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 0861–0866, 2018.

[97] S. Sinaei and O. Fatemi, "Tree-based algorithm for design space exploration and mapping application onto heterogeneous platforms," in *2017 19th International Symposium on Computer Architecture and Digital Systems (CADS)*, pp. 1–6, 2017.

[98] H. Youness, A. Wahdan, M. Hassan, A. Salem, M. Moness, K. Sakanushi, Y. Takeuchi, and M. Imai, "Efficient partitioning technique on multiple cores based on optimal scheduling and mapping algorithm," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 3729–3732, 2010.

[99] *IEEE Standard for Standard SystemC Language Reference Manual - Redline*, 2012.

[100] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Dresc: a retargetable compiler for coarse-grained reconfigurable architectures," in *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.*, pp. 166–173, 2002.

[101] L. Ma, W. Ge, and Z. Qi, "A graph-based spatial mapping algorithm for a coarse grained reconfigurable architecture template," in *Informatics in Control, Automation and Robotics* (D. Yang, ed.), (Berlin, Heidelberg), pp. 669–678, Springer Berlin Heidelberg, 2012.

[102] S. Das, K. J. M. Martin, P. Coussy, D. Rossi, and L. Benini, "Efficient mapping of cdfg onto coarse-grained reconfigurable array architectures," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 127–132, 2017.

[103] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras)," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–10, 2013.

[104] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to cgra mapping," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2018.

[105] J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Yunheung Paek, "Spkm : A novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures," in *2008 Asia and South Pacific Design Automation Conference*, pp. 776–782, 2008.

[106] S. R. Kota, C. Shekhar, A. Kokkula, D. Toshniwal, M. V. Kartikeyan, and R. C. Joshi, "Parameterized module scheduling algorithm for reconfigurable computing systems," in *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*, pp. 473–478, 2007.

[107] Yen-Tai Lai, Hsin-Ya Lai, and Chia-Nan Yeh, "Placement for the reconfigurable datapath architecture," in *2005 IEEE International Symposium on Circuits and Systems*, pp. 1875–1878 Vol. 2, 2005.

[108] J. J. Cong and Songjie Xu, "Performance-driven technology mapping for heterogeneous fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 11, pp. 1268–1281, 2000.

[109] A. Namazi, M. Abdollahi, S. Safari, and S. Mohammadi, "A majority-based reliability-aware task-mapping in high-performance homogenous noc architectures," in *2016 Euromicro Conference on Digital System Design (DSD)*, pp. 479–486, 2016.

[110] C. Lu, Y. Zhang, and J. Jiang, "A breadth-first greedy mapping algorithm for reducing internal congestion in noc," in *2017 4th International Conference on Information Science and Control Engineering (ICISCE)*, pp. 1336–1341, 2017.

[111] Y. Ma, B. Gong, and L. Zou, "Energy-efficient scheduling algorithm of task dependent graph on dvs-unable cluster system," in *2009 10th IEEE/ACM International Conference on Grid Computing*, pp. 217–224, 2009.

[112] H.-J. Jiang, K.-C. Huang, H.-Y. Chang, D.-S. Gu, and P.-J. Shih, "Scheduling concurrent workflows in hpc cloud through exploiting schedule gaps," in *Algorithms and Architectures for Parallel Processing* (Y. Xiang, A. Cuzzocrea, M. Hobbs, and W. Zhou, eds.), (Berlin, Heidelberg), pp. 282–293, Springer Berlin Heidelberg, 2011.

[113] A. GamatiÃ©, X. An, Y. Zhang, A. Kang, and G. Sassatelli, "Empirical model-based performance prediction for application mapping on multicore architectures," *Journal of Systems Architecture*, vol. 98, pp. 1 – 16, 2019.

[114] Z. Li, L. Liu, Y. Deng, S. Yin, Y. Wang, and S. Wei, "Aggressive pipelining of irregular applications on reconfigurable hardware," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), p. 575–586, Association for Computing Machinery, 2017.

[115] A. C. Rajeev, S. Mohalik, M. G. Dixit, D. B. Chokshi, and S. Ramesh, "Schedulability and end-to-end latency in distributed ecu networks: Formal modeling and precise estimation," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '10, (New York, NY, USA), p. 129–138, Association for Computing Machinery, 2010.

[116] R. C. Gonzalez and R. E. Woods, *Digital image processing.* Upper Saddle River, N.J.: Prentice Hall, 2008.

[117] J. Serra, *Image Analysis and Mathematical Morphology.* USA: Academic Press, Inc., 1983.

[118] M. Couprie and G. Bertrand, "Topology preserving alternating sequential filter for smoothing two-dimensional and three-dimensional objects," *Journal of Electronic Imaging*, vol. 13, no. 4, pp. 720–731, 2004.

[119] L. Najman, "Using mathematical morphology for document skew estimation," in *SPIE Document Recognition and Retrieval IX*, pp. 182–191, 2004.

[120] S. Sinaei and O. Fatemi, "Novel heuristic mapping algorithms for design space exploration of multiprocessor embedded architectures," in *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 801–804, 2016.

[121] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp. 8 – 22, 1997.

[122] T. D. Braun, H. J. Siegel, and A. A. Maciejewski, "Heterogeneous computing: Goals, methods, and open problems," in *High Performance Computing — HiPC 2001* (B. Monien, V. K. Prasanna, and S. Vajapeyam, eds.), (Berlin, Heidelberg), pp. 307–318, Springer Berlin Heidelberg, 2001.

[123] A. Ghafoor and J. Yang, "A distributed heterogeneous supercomputing management system," *Computer*, vol. 26, no. 6, pp. 78–86, 1993.

[124] B. Tafesse, A. Raina, J. Suseela, and V. Muthukumar, "Efficient scheduling algorithms for mpsoc systems," in *2011 Eighth International Conference on Information Technology: New Generations*, pp. 683–688, 2011.

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

[126] Z. Houssam-Eddine, N. Capodieci, R. Cavicchioli, G. Lipari, and M. Bertogna, "The hpc-dag task model for heterogeneous real-time systems," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[127] S.-J. Chen, A.-Y. Wu, and J. Xu, "Networks-on-chip: Architectures, design methodologies, and case studies," *Journal of Electrical and Computer Engineering*, vol. 2012, 04 2012.

[128] L. Yang, W. Liu, N. Guan, and N. Dutt, "Optimal application mapping and scheduling for network-on-chips with computation in stt-ram based router," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1174–1189, 2019.

[129] A. Mehran, S. Saeidi, A. Khademzadeh, and A. Afzali-Kusha, "Spiral: A heuristic mapping algorithm for network on chip," *IEICE Electronics Express*, vol. 4, no. 15, pp. 478–484, 2007.

[130] A. Mehran, A. Khademzadeh, and S. Saeidi, "Dsm: A heuristic dynamic spiral mapping algorithm for network on chip," *IEICE Electronics Express*, vol. 5, no. 13, pp. 464–471, 2008.

[131] B. De Sutter, P. Raghavan, and A. Lambrechts, *Coarse-Grained Reconfigurable Array Architectures*, pp. 449–484. Boston, MA: Springer US, 2010.

[132] G. Gallo, G. Longo, S. Pallottino, and S. Nguyen, "Directed hypergraphs and applications," *Discrete Applied Mathematics*, vol. 42, no. 2, pp. 177–201, 1993.

[133] A. Isavudeen, N. Ngan, E. Dokladalova, and M. Akil, "Highly scalable monitoring system on chip for multi-stream auto-adaptable vision system," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, RACS '17, (New York, NY, USA), p. 249–254, Association for Computing Machinery, 2017.

[134] J. Sun and Y. Zhang, "An energy-aware mapping algorithm for mesh-based network-on-chip architectures," in *2017 International Conference on Progress in Informatics and Computing (PIC)*, pp. 357–361, 2017.

[135] D. Liu, S. Yin, Y. Peng, L. Liu, and S. Wei, "Optimizing spatial mapping of nested loop for coarse-grained reconfigurable architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 11, pp. 2581–2594, 2015.

[136] L. Chen and T. Mitra, "Graph minor approach for application mapping on cgras," in *2012 International Conference on Field-Programmable Technology*, pp. 285–292, 2012.

[137] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, May 2008.

[138] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," in *Leveraging Applications of Formal Methods, Verification and Validation* (T. Margaria and B. Steffen, eds.), (Berlin, Heidelberg), pp. 430–444, Springer Berlin Heidelberg, 2008.

[139] N. Frid, D. Ivošević, and V. Sruk, "Performance estimation in heterogeneous mpsoc based on elementary operation cost," in *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1202–1205, 2016.

[140] A. Hamann, D. Dasari, S. Kramer, M. Pressler, and F. Wurst, "Communication Centric Design in Complex Automotive Embedded Systems," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)* (M. Bertogna, ed.), vol. 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 10:1–10:20, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[141] Y. M. Lam, J. G. F. Coutinho, W. Luk, and P. H. W. Leong, "Mapping and scheduling with task clustering for heterogeneous computing systems," in *2008 International Conference on Field Programmable Logic and Applications*, pp. 275–280, 2008.

[142] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman and Co., 1990.

[143] H. Wang and O. Sinnen, "List-scheduling versus cluster-scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 8, pp. 1736–1749, 2018.

[144] A. I. Orhean, F. Pop, and I. Raicu, "New scheduling approach using reinforcement learning for heterogeneous distributed systems," *Journal of Parallel and Distributed Computing*, vol. 117, pp. 292 – 302, 2018.

[145] M. K. Bhatti, I. Oz, A. Muddukrishna, K. Popov, and M. Brorsson, "Noodle: A heuristic algorithm for task scheduling in mpsoc architectures," in *2014 17th Euromicro Conference on Digital System Design*, pp. 667–670, 2014.

[146] P. Black, "Dads: The on-line dictionary of algorithms and data structures," 2020-09-17 2020.

[147] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[148] J. Kaida, T. Hieda, I. Taniguchi, H. Tomiyama, Y. Hara-Azumi, and K. Inoue, "Task mapping techniques for embedded many-core socs," in *2012 International SoC Design Conference (ISOCC)*, pp. 204–207, 2012.

[149] W. H. Kohler, "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE Trans. Comput.*, vol. 24, p. 1235–1238, Dec. 1975.

[150] B. S. Landman and R. L. Russo, "On a pin versus block relationship for partitions of logic graphs," *IEEE Transactions on Computers*, vol. C-20, no. 12, pp. 1469–1479, 1971.

[151] D. Greenfield, A. Banerjee, J. Lee, and S. Moore, "Implications of rent's rule for noc design and its fault-tolerance," in *First International Symposium on Networks-on-Chip (NOCS'07)*, pp. 283–294, 2007.

[152] G. B. Bezerra, S. Forrest, M. Moses, A. Davis, and P. Zarkesh-Ha, "Modeling noc traffic locality and energy consumption with rent's communication probability distribution," in *Proceedings of the 12th ACM/IEEE International Workshop on System Level Interconnect Prediction*, SLIP '10, (New York, NY, USA), p. 3–8, Association for Computing Machinery, 2010.

[153] S. Jiang, Q. Wu, S. Chen, J. Wang, M. Ebrahimi, L. Huang, and Q. Li, "Optimizing dynamic mapping techniques for on-line noc test," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 227–232, 2018.

[154] L. Huang, H. Dong, J. Wang, M. Daneshtalab, and G. Li, "Wena: Deterministic run-time task mapping for performance improvement in many-core embedded systems," *IEEE Embedded Systems Letters*, vol. 7, no. 4, pp. 93–96, 2015.

[155] S. Craw, *Manhattan Distance*, pp. 639–639. Boston, MA: Springer US, 2010.

[156] G. Di Battista, M. Patrignani, and F. Vargiu, "A split&push approach to 3d orthogonal drawing," in *Graph Drawing* (S. H. Whitesides, ed.), (Berlin, Heidelberg), pp. 87–101, Springer Berlin Heidelberg, 1998.

[157] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[158] M. Naeem, S. T. H. Rizvi, and A. Coronato, "A gentle introduction to reinforcement learning and its application in different fields," *IEEE Access*, vol. 8, pp. 209320–209344, 2020.

[159] C. Shyalika, T. Silva, and A. Karunananda, "Reinforcement learning in dynamic task scheduling: A review," *SN Computer Science*, vol. 1, no. 6, pp. 1–17, 2020.

[160] N. Chauhan, N. Choudhary, and K. George, "A comparison of reinforcement learning based approaches to appliance scheduling," in *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, pp. 253–258, 2016.

[161] D. Zavyalova and V. Drozdova, "5g scheduling using reinforcement learning*," in *2020 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*, pp. 1–5, 2020.

[162] I. Comşa, S. Zhang, M. E. Aydin, P. Kuonen, Y. Lu, R. Trestian, and G. Ghinea, "Towards 5g: A reinforcement learning-based scheduling solution for data traffic management," *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1661–1675, 2018.

[163] X. Wei, J. Zhao, L. Zhou, and Y. Qian, "Broad reinforcement learning for supporting fast autonomous iot," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7010–7020, 2020.

[164] H. Zhao, J. Zhao, J. Qiu, G. Liang, and Z. Y. Dong, "Cooperative wind farm control with deep reinforcement learning and knowledge-assisted learning," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 11, pp. 6912–6921, 2020.

[165] T. Chu, J. Wang, L. Codecà, and Z. Li, "Multi-agent deep reinforcement learning for large-scale traffic signal control," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 3, pp. 1086–1095, 2020.

[166] D. Elavarasan and P. M. D. Vincent, "Crop yield prediction using deep reinforcement learning model for sustainable agrarian applications," *IEEE Access*, vol. 8, pp. 86886–86901, 2020.

[167] Z. Zhao, W. Sheng, Q. Wang, W. Yin, P. Ye, J. Li, and Z. Mao, "Towards higher performance and robust compilation for cgra modulo scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2201–2219, 2020.

[168] S. Huang, B. Lv, R. Wang, and K. Huang, "Scheduling for mobile edge computing with random user arrivals—an approximate mdp and reinforcement learning approach," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 7, pp. 7735–7750, 2020.

[169] A. Elgabli, H. Khan, M. Krouka, and M. Bennis, "Reinforcement learning based scheduling algorithm for optimizing age of information in ultra reliable low latency networks," in *2019 IEEE Symposium on Computers and Communications (ISCC)*, pp. 1–6, 2019.

[170] A. Nascimento, V. Olimpio, V. Silva, A. Paes, and D. de Oliveira, "A reinforcement learning scheduling strategy for parallel cloud-based workflows," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 817–824, 2019.

[171] D. Ding, X. Fan, Y. Zhao, K. Kang, Q. Yin, and J. Zeng, "Q-learning based dynamic task scheduling for energy-efficient cloud computing," *Future Generation Computer Systems*, vol. 108, pp. 361 – 371, 2020.

[172] J. Liu and H. Shen, "Dependency-aware and resource-efficient scheduling for heterogeneous jobs in clouds," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 110–117, 2016.

[173] L. Chen, I. Wu, and Y. Chang, "Reinforcement learning based fragment-aware scheduling for high utilization hpc platforms," in *2019 International Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pp. 1–7, 2019.

[174] A. Gocht, R. Schöne, and M. Bielert, "Q-learning inspired self-tuning for energy efficiency in hpc," in *2019 International Conference on High Performance Computing Simulation (HPCS)*, pp. 344–347, 2019.

[175] Q. Wu, Z. Wu, Y. Zhuang, and Y. Cheng, "Adaptive dag tasks scheduling with deep reinforcement learning," in *Algorithms and Architectures for Parallel Processing* (J. Vaidya and J. Li, eds.), (Cham), pp. 477–490, Springer International Publishing, 2018.

[176] K. Liu, Z. Wu, Q. Wu, and Y. Cheng, "Smart dag task scheduling with efficient pruning-based mcts method," in *2019 IEEE Intl Conf on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pp. 348–355, 2019.

[177] N. Grinsztajn, O. Beaumont, E. Jeannot, and P. Preux, "Geometric deep reinforcement learning for dynamic dag scheduling," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 258–265, 2020.

[178] H. Lee, J. Lee, I. Yeom, and H. Woo, "Panda: Reinforcement learning-based priority assignment for multi-processor real-time scheduling," *IEEE Access*, vol. 8, pp. 185570–185583, 2020.

[179] R. S. Luley and Q. Qiu, "A deep q-learning approach for gpu task scheduling," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2020.

[180] D. Liu, S. Yin, G. Luo, J. Shang, L. Liu, S. Wei, Y. Feng, and S. Zhou, "Data-flow graph mapping optimization for cgra with deep reinforcement learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 12, pp. 2271–2283, 2019.

[181] A. Honorat, K. Desnos, S. S. Bhattacharyya, and J.-F. Nezan, "Scheduling of synchronous dataflow graphs with partially periodic real-time constraints," in *Proceedings of the 28th International Conference on Real-Time Networks and Systems*, RTNS 2020, (New York, NY, USA), p. 22–33, Association for Computing Machinery, 2020.

[182] J. Yang, H. Xu, and P. Jia, "Task scheduling for heterogeneous computing based on bayesian optimization algorithm," in *2009 International Conference on Computational Intelligence and Security*, vol. 1, pp. 112–117, 2009.

[183] J. B. Mockus and L. J. Mockus, "Bayesian approach to global optimization and application to multiobjective and constrained problems," *J. Optim. Theory Appl.*, vol. 70, p. 157–172, July 1991.

[184] S. K. Biswas, A. Rauniyar, and P. K. Muhuri, "Multi-objective bayesian optimization algorithm for real-time task scheduling on heterogeneous multiprocessors," in *2016 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2844–2851, 2016.

[185] J. Liu and H. Shen, "Dependency-aware and resource-efficient scheduling for heterogeneous jobs in clouds," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 110–117, 2016.

[186] Y. Zhao, L. Chen, Y. Li, and W. Tian, "Efficient task scheduling for many task computing with resource attribute selection," *China Communications*, vol. 11, no. 12, pp. 125–140, 2014.

[187] C. Ferreira, *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence (Studies in Computational Intelligence)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[188] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, p. 720–748, Sept. 1999.

[189] T. Selvameena and R. A. Prasath, "Out-of-order execution on reconfigurable heterogeneous mpsoc using particle swarm optimization," in *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, pp. 1–6, 2017.

[190] A. Emeretlis, G. Theodoridis, P. Alefragis, and N. Voros, "Mapping dags on heterogeneous platforms using logic-based benders decompostion," in *2015 IEEE Computer Society Annual Symposium on VLSI*, pp. 119–124, 2015.

[191] J. N. Hooker and G. Ottosson, "Logic-based benders decomposition," *Mathematical Programming*, vol. 96, no. 1, pp. 33–60, 2003.

[192] R. Ferreira, J. G. Vendramini, L. Mucida, M. M. Pereira, and L. Carro, "An fpga-based heterogeneous coarse-grained dynamically reconfigurable architecture," in *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pp. 195–204, 2011.

[193] R. W. Hartenstein and R. Kress, "A datapath synthesis system for the reconfigurable datapath architecture," in *Proceedings of ASP-DAC'95/CHDL'95/VLSI'95 with EDA Technofair*, pp. 479–484, 1995.

[194] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, p. 558–562, Nov. 1962.

[195] L. Ma, W. Ge, and Z. Qi, "A graph-based spatial mapping algorithm for a coarse grained reconfigurable architecture template," in *Informatics in Control, Automation and Robotics* (D. Yang, ed.), (Berlin, Heidelberg), pp. 669–678, Springer Berlin Heidelberg, 2012.

[196] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, p. 345, June 1962.

[197] T. Bayes, "An essay towards solving a problem in the doctrine of chances," *Phil. Trans. of the Royal Soc. of London*, vol. 53, pp. 370–418, 1763.

[198] W. Buntine, *Bayesian Methods*, pp. 75–81. Boston, MA: Springer US, 2010.

[199] O. University, *Practical Modern Statistics: Bayesian statistics*. No. bk. 4 in M249 practical modern statistics, Open University, 2007.

[200] L. P. Santos and A. Proenca, "Scheduling under conditions of uncertainty: A bayesian approach," in *Euro-Par 2004 Parallel Processing* (M. Danelutto, M. Vanneschi, and D. Laforenza, eds.), (Berlin, Heidelberg), pp. 222–229, Springer Berlin Heidelberg, 2004.

[201] A. Y. Zomaya, M. Clements, and S. Olariu, "A framework for reinforcement-based scheduling in parallel processor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 3, pp. 249–260, 1998.

[202] M. A. M. Khan, M. R. J. Khan, A. Tooshil, N. Sikder, M. A. P. Mahmud, A. Z. Kouzani, and A. A. Nahid, "A systematic review on reinforcement learning-based robotics within the last decade," *IEEE Access*, vol. 8, pp. 176598–176623, 2020.

[203] Q. Huang, "Model-based or model-free, a review of approaches in reinforcement learning," in *2020 International Conference on Computing and Data Science (CDS)*, pp. 219–221, 2020.

[204] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[205] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," *CoRR*, vol. abs/1502.05477, 2015.

[206] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[207] C. J. C. H. Watkins and P. Dayan, "Technical note: Q -learning," *Mach. Learn.*, vol. 8, p. 279–292, May 1992.

[208] C. J. C. H. Watkins, *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford, 1989.

[209] G. A. Rummery and M. Niranjan, "On-line Q-learning using connectionist systems," Tech. Rep. TR 166, Cambridge University Engineering Department, Cambridge, England, 1994.

[210] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," , vol. 518, pp. 529–533, Feb. 2015.

[211] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.

[212] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, p. 2094–2100, AAAI Press, 2016.

[213] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning* (M. F. Balcan and K. Q. Weinberger, eds.), vol. 48 of *Proceedings of Machine Learning Research*, (New York, New York, USA), pp. 1995–2003, PMLR, 20–22 Jun 2016.

[214] A. Nascimento, V. Olimpio, V. Silva, A. Paes, and D. de Oliveira, "A reinforcement learning scheduling strategy for parallel cloud-based workflows," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 817–824, 2019.

[215] A. Lazaric, *Transfer in Reinforcement Learning: A Framework and a Survey*, pp. 143–173. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[216] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-learning algorithms: A comprehensive classification and applications," *IEEE Access*, vol. 7, pp. 133653–133667, 2019.

[217] Z. Wei, F. Liu, Y. Zhang, J. Xu, J. Ji, and Z. Lyu, "A q-learning algorithm for task scheduling based on improved svm in wireless sensor networks," *Computer Networks*, vol. 161, pp. 138 – 149, 2019.

[218] Q. Dang, D. Wu, and B. Boulet, "A q-learning based charging scheduling scheme for electric vehicles," in *2019 IEEE Transportation Electrification Conference and Expo (ITEC)*, pp. 1–5, 2019.

[219] Y. Zhao and J. Lee, "A reinforcement learning based low-delay scheduling with adaptive transmission," in *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 916–919, 2019.

[220] L. Ren, X. Ning, and J. Li, "Hierarchical reinforcement-learning for real-time scheduling of agile satellites," *IEEE Access*, vol. 8, pp. 220523–220532, 2020.

[221] Y. Li, E. Fadda, D. Manerba, R. Tadei, and O. Terzo, "Reinforcement learning algorithms for online single-machine scheduling," in *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*, pp. 277–283, 2020.

[222] G. Wang, F. Xu, and C. Zhao, "Multi-access edge computing based vehicular network: Joint task scheduling and resource allocation strategy," in *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–6, 2020.

[223] E. Even-Dar and Y. Mansour, "Learning rates for q-learning," *J. Mach. Learn. Res.*, vol. 5, p. 1–25, Dec. 2004.

[224] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008.

[225] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph – static and dynamic graph drawing tools," in *Graph Drawing Software*, pp. 127–148, Springer-Verlag, 2003.

[226] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[227] F. Chung, *Random Graphs, a Whirlwind Tour of*, pp. 2536–2548. New York, NY: Springer New York, 2012.

[228] P. Erdős and A. Rényi, "On the evolution of random graphs," in *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, pp. 17–61, 1960.

[229] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, pp. 440–442, June 1998.

[230] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999.

[231] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," 2019.

[232] *IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows*, 2010.

[233] S. Payvar, E. Pekkarinen, R. Stahl, D. Mueller-Gritschneder, and T. D. Hämäläinen, "Instruction extension of a risc-v processor modeled with ip-xact," in *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pp. 1–5, 2019.