FACULTÉ
DES SCIENCES

UNIVERSITÉ LIBRE DE BRUXELLES

Efficient scheduling for embedded hard real-time systems upon heterogeneous unrelated platform

Thesis presented by Xavier Poczekajlo

in fulfilment of the requirements of the PhD Degree in Computer Science ("Docteur en Informatique")

Année académique 2020–2021

Supervisor: Professor Joël Goossens

Thesis jury :
Yves Roggeman (Université libre de Bruxelles, Chair)
Joël Goossens (Université libre de Bruxelles, Supervisor)
Antoine Bertout (Université de Poitiers, France)
Sébastien Faucou (Université de Nantes, France)
Gilles Geeraerts (Université libre de Bruxelles)
Dragomir Milojevic (École polytechnique de Bruxelles)
Juan M. Rivas (Universidad de Cantabria, Spain)

Acknowledgements

It took exactly four years to write those words. Four years between my first contact with the Université libre de Bruxelles and the writing of this first (but also last) words. During this time, I could produce this document which will hopefully lead to a long-time goal. This elapsed time was obviously crucial, but I cannot stress enough the importance of the people I have met during this period, and before.

Professor Joël Goossens: you trusted in me four years ago. I thank you for that, but even more for the continued guidance you have given me during that time. You spent a lot of effort, patience and time into your role. Hopefully, you now feel that your dedication has paid off.

I also thank all the member of my jury for their careful reading of this thesis, the detailed comments and the improvement suggestions. It made a huge difference on the quality of this thesis. Thank you Yves Roggeman, Antoine Bertout, Sébastien Faucou, Gilles Geeraerts, Dragomir Milojevic and Juan M. Rivas.

The ULB hosts and hosted many other members that helped me through my Ph.D. Every member of the Sofist project, professor or student, helped me to understand the different aspects and importance of each domain related to my research. Jean-Michel Dricot, Olivier Markowich, Dragomir Milojevic and François Quitin: thank you for having initiated and led this project. Your work and ideas created four great projects. George Bousdras, Soultana Ellinidou and Gaurav Sharma: thank you for joining this project and its ambitious idea to merge four different research topics together. Thank you for the long conversations, during which we learned how to build bridges through our different fields.

Juan M. Rivas, Antonio Paolillo and Paul Rodriguez: you all helped me with my research, each one of you differently. Your experienced insights elevated my work. Thank you for your advice, for your trust in my work and for your company. The fact that I could count on you played a big role. Juan, you shared for a long time my office and my bad jokes. Your contribution to my enjoyment of both will not be forgotten.

Thank you Véronique Bastin, Pascaline Browaeys and Maryka Peetroons for your support in the administrative procedure and during the different organised events. Without your help, I would probably still be stuck in some irrelevant form or outdated

set of rules.

By construction, academic research could not be done within one institution. Antoine Bertout and Emmanuel Grolleau: the collaborative work conducted gave me confidence and proved to be fruitful. Thank you for all the time spent improving our articles, my knowledge and my skills.

Gurulingesh Raravi, you were very supportive at a very early stage of my research. You helped me kick-start my research. Thank you.

Of course, I have neither the space nor the memory to cite every person that had a positive impact on my work during those four years. Thanks to my interlocutors for your questions, during the various talks and poster sessions that I gave. Thanks also to you, you who shared an interesting conversation with me at this event. Thanks to who chatted at lunch during those conferences. Thanks to you all.

It would be foolish to restrict this section to the people I met at work for the past four years. Thanks to all the professors, who inspired my curiosity and pushed me further. Thanks to all my role models, who did the same.

Giving a few words here about those who haven't been mentioned yet is impossible. Fortunately, the value of words being not proportional to their quantity: this follows. Thank you for your help; thank you for the laughs; thank you for the colours.

Thank you for the sweetness; thank you for this day; thank you for the joy; merci pour le sourire.

List of publications

Juan Maria Rivas et al. “Implementation of Memory Centric Scheduling for COTS Multi-Core Real-Time Systems”. In: *31st Euromicro Conference on Real-Time Systems, ECRTS 2019, July 9-12, 2019, Stuttgart, Germany*. Ed. by Sophie Quinton. Vol. 133. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 7:1–7:23. DOI: 10.4230/LIPIcs.ECRTS.2019.7. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2019.7.

Joël Goossens et al. “ACCEPTOR: a model and a protocol for real-time multi-mode applications on reconfigurable heterogeneous platforms”. In: *Proceedings of the 27th International Conference on Real-Time Networks and Systems, RTNS 2019, Toulouse, France, November 06-08, 2019*. Ed. by Jérôme Ermont, Ye-Qiong Song, and Christopher Gill. ACM, 2019, pp. 209–219. DOI: 10.1145/3356401.3356420. URL: https://doi.org/10.1145/3356401.3356420.

Joël Goossens and Xavier Poczekajlo. “Multimode application on a reconfigurable platform: Introducing a new model and a first protocol”. In: *Proceeding JW*. 2017.

Antoine Bertout et al. “Workload assignment for global real-time scheduling on unrelated multicore platforms”. In: *28th International Conference on Real Time Networks and Systems, RTNS 2020, Paris, France, June 10, 2020*. Ed. by Liliana Cucu-Grosjean et al. Outstanding paper. ACM, 2020, pp. 139–148. DOI: 10.1145/3394810.3394823. URL: https://doi.org/10.1145/3394810.3394823.

Antoine Bertout et al. “Template schedule construction for global real-time scheduling on unrelated multiprocessor platforms”. In: *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*. IEEE, 2020, pp. 216–221. DOI: 10.23919/DATE48585.2020.9116409. URL: https://doi.org/10.23919/DATE48585.2020.9116409.

Antoine Bertout et al. “Invited paper in preparation: Workload assignment for global real-time scheduling on unrelated multicore platforms”. In: *Real-Time Systems Journal*.

Contents

List of Figures

List of Tables

List of notations and symbols

This thesis will use several mathematical notations and conventions, here is a list for disambiguation purpose.

- The symbol $\doteq$ denotes "equal by definition to";
- A vector v containing the element v_1 and v_2 is denoted as $v = < v_1, v_2 >$. It contains by definition $|v|$ elements, and is ordered; v_i denotes the i^{th} element of v.
- A set s containing the element v_1 and v_2 is denoted as $v = \{v_1, v_2\}$; It contains by definition $|s|$ elements.
- The notation $\max_{j=1}^{n}\{x_j\}$ represents the maximal value in the set $\{x_j \,|j = 1, \dots, n\}$;
- The notation $\min_{j=1}^{n}\{x_j\}$ represents the minimal value in the set $\{x_j \,|j = 1, \dots, n\}$;
- The notation $\max_s f(s)$ represents the maximal value in the set $\{f(x) \,|x \in s\}$;
- The notation $\min_s f(s)$ represents the minimal value in the set $\{f(x) \,|x \in s\}$;
- $\lceil x \rceil$ represents the ceiling function applied to x;
- $\lfloor x \rfloor$ represents the floor function applied to x;
- Upper-bound on the value *value* will be denoted as $\overline{\text{value}}$;
- Lower-bound on the value *value* will be denoted as $\underline{\text{value}}$.

Part I

Introduction

Table of Contents

Chapter 1

Introduction to hard real-time systems

1.1 Introduction

This thesis studies *software applications* running on *computing platforms*. The combination of the two forms a *system*. Among all the different kinds of systems, this thesis is focused on *embedded systems*.

Definition 1.1 (Embedded system)**.** An *embedded system* is a device that integrates both a computing platform and software but that is not a computer itself. In other words, the prime function of the device is not to be a computer. It may be communication device, (smart-)glasses, and many others.

Examples include phones, computer system of planes or cars. Most *embedded systems* will require their computing platform to perform operations (from the *software applications*) linked to their environment. Because of that, the system may need to wait for the operation results before performing a specific action. If this action is the activation of the breaking system on a car, it cannot be completed *too late*.

Definition 1.2 (Real-time applications)**.** A real-time application is an application which has time-based constraints.

The notion of time-based constraints is central in *real-time applications*. If an action is performed according to its time-based constraints, it is *on time*. It is late otherwise.

This lateness may also be referred to as *tardiness*. An *application* is a set of *tasks* defined in Section 2.2. A *platform* is a set of *hardwares*, defined in Section 2.3. Before defining rigorously what an *application* and a *platform* is, we will start by explaining what a real-time operating system is. Using this overview, we will define thoroughly what an *application* and what a *platform* is. With those two notions defined, we will define what a *scheduler* is. All the notations can be found in Table 11.2.

1.2 Real-time operating system

In this section, we will give an overview of what a system is. As said before, a system is the combination of an application layer and a hardware layer, the latter representing the platform. The *Real-time Operating System* (denoted as *RTOS*) is in charge of binding both layers as shown in Figure 1.1 After explaining the major responsibilities of the *RTOS*, we will explain the gain of using one.

An RTOS is an operating system (*OS*). An OS is responsible for the hardware abstraction. This hardware abstraction hides the complexity and details to the computer programmer, together with the compiler and libraries. It makes possible the use of high-level languages. This makes the development easier, but also portable. As the produced software is no longer hardware specific, it can be used from one hardware to another. Of course, this leads to an isolation of the different layers from one to each other. Isolation provides itself several benefits. Bugs or issues may be found more easily which eases the development but also the update of both part.

The OS is also a resource manager. It manages the different Input/Output ports in the system, such as the keyboard, screen or network devices. It also handles the processor(s). A processor needs to be given specific computation tasks. To do so, the OS chooses the processor upon which a task will be executed, at a given time. Choosing the executed task is a role dedicated to the *scheduler*. Section 2.4 will discuss this responsibility in details.

Another major responsibility of the *OS* is to handle the memory. As it will be shown in Section 2.6, the memory is composed in most modern systems of several components itself. Each component may have a different access speed and capacity. Generally speaking, for a similar quality, the biggest capacity a memory has, the slowest it is. We speak here of *memory hierarchy*. Each different memory type have different usages. A system may be pre-loaded with some data to process during its lifespan. It may also

record some data through the sensors, raw or processed, and store it for later processing or transmit it through its network communication device. Those first usages are quite obvious. However, one hidden use of the memory is the *working memory*. This memory is used by tasks while processing an object such as an image. To be usable in practice, it must be stored in very fast access memory. The use of the different memories must be optimised to take advantage of this different specifications of the components.

The specificities of an RTOS compared to other OS is that RTOS should be predictable. The computation of the tasks must both be *valid* and *on time*. To guarantee this, the scheduler will adopt various strategies that will be discussed in the next chapter.

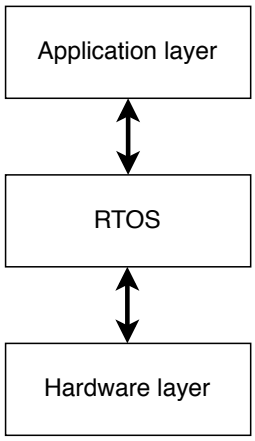


Figure 1.1: Layer hierarchy of an embedded system

1.3 Processing elements

A central element of the hardware layer is the processing elements, often referred to as *processors*. The meaning of the term *processor* depends on the context. It is used to refer to a single chip with only one processing element, such as a Central Processing Unit (CPU). Such a processor can only execute one instruction at a time[1]. However, the cost-performance ratio of such processors could not be increased indefinitely. Indeed, one way to increase the performance of those processors was to increase the clock frequency. This lead to a higher energy consumption and a lot of heat output. Nowadays, there may be several processing elements (or cores/CPUs) on a chip, permitting several

[1] Here, we omit the use of pipelining and superscalar design. A basic processor handles each instruction in several steps (or stages): fetching, decoding , executing, writing, etc. A given instruction is only at one step at a time. With pipelining, available steps are used in parallel for other instructions. This leads to a quasi parallel execution of instructions. Regarding superscalar processor, they can execute several identical instructions at a given time, by using several execution units. In addition, a superscalar processor can use pipelining to execute more instructions in parallel.

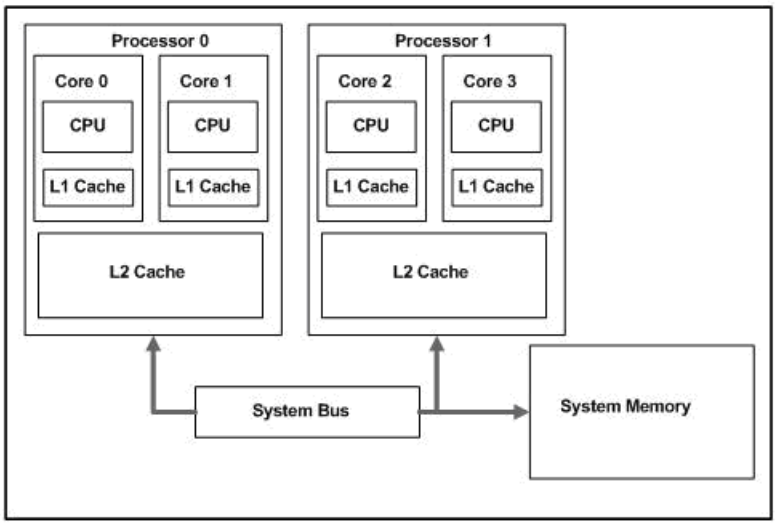


Figure 1.2: A dual-core, dual-processor system

instructions to occur at a time on a single *processor*. Modern architectures may have several multi-core processors, such as the Intel Core® Duo processor whose architecture is abstracted in Figure 1.2[2]. The Intel Core® Duo processor embeds two processors, each one having two CPUs. A system bus is used for the communication between the processors and the rest of the hardware. L1-caches, L2-caches and System memory are memory units, and will be presented in Section 2.6.

General purpose processors (GPPs) are single- or multi-core processing units. They offer a varied instruction set and are thus very flexible. As suggested by their name, they are not specialised in any specific tasks. Some specific processing unit exists, such as Graphics processing unit (GPUs). GPUs are specialised in the manipulation of images and may be used for other tasks, such as matrix computations for 3D imaging applications. Therefore, they offer a high level of parallelism to operate on larger memory buffers. Being specialised in those tasks makes GPUs faster and more energy-efficient than CPUs in those tasks. Application-specific integrated circuits (ASICs) are other *processors* that are designed for specific tasks. Unlike GPUs, they are custom-made which makes them really expensive and offer no flexibility once built. There exists a trade-off between the low cost but low performances of GPPs and the high performance but high cost of ASICs: application-specific instruction set processors (ASIPs). ASIPs are programmable with the use of custom instructions in order to be very specialised. Thanks to those custom instructions, the same ASIP model may be

[2] from https://software.intel.com/content/www/us/en/develop/articles/software-techniques-for-shared-cache-multi-core-systems.html?wapkw=smart+cache

used for very different applications such as cryptography, digital signal processing, mobile communication and many more. This makes ASIPs cheaper than ASICs to design for specific applications, while offering better performance than GPPs.

1.3.1 Field Programmable Gate Array

Field Programmable Gate Array (FPGA) is a circuit, that can perform operations alongside a processor. It can been seen as a *co-CPU*. An FPGA is mainly composed of Configurable Logic Blocks, which implement logic functions, and of Programmable Interconnects and Programmable I/O Blocks responsible respectively for internal communication and external communication.

The Configurable Logic Blocks are what makes FPGAs so useful. Each one can be configured at design-time and run-time and be used to perform complex tasks in a very efficient way. Design-time reconfiguration allows easy prototyping. Run-time reconfiguration permits versatility: the FPGA may be reconfigured based on the context of the application. Moreover, the Configurable Logic Blocks may be configured into one co-processor, as a whole, or into several co-processors. This way, the FPGA can execute several different tasks in parallel.

FPGAs can be found in new embedded processing platforms. For example, the Xilinx Zynq UltraScale+™ system on chip [1] is a single die integrating several ARM processing cores from different architectures together with other heterogeneous components (64 bit quad-core Cortex® A53 cores, dual-core Cortex® R5 real-time cores, a Mali™-400 MP2 GPU and an FPGA programmable logic). This processor's block diagram is illustrated by Figure 1.3.

The last generation of FPGA uses Dynamic Partial Reconfiguration (DPR) which permits to reconfigure only a part of the FPGA while the others are not interrupted. A new technology is rising: 3D face-to-face die stacking. For example, Foveros [3], designed by Intel, is built using 3D face-to-face die stacking. With this technology, the memory containing the possible configurations for the logic blocks are placed above the logic blocks. This way, the whole platform can be reconfigured simultaneously. And because the distance between the memory and the logic blocks is very small, the reconfiguration takes place very quickly.

[3] from https://newsroom.intel.com/news/up-close-lakefield-intels-chip-award-winning-foveros-3d-tech

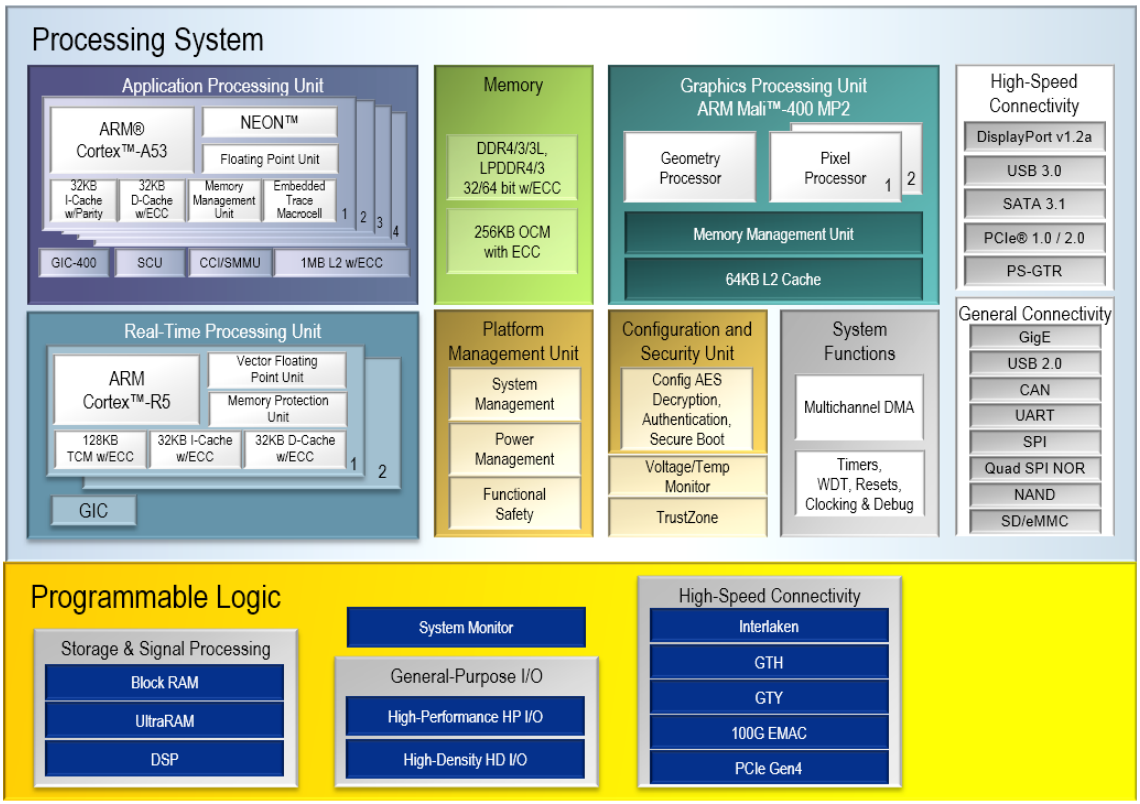


Figure 1.3: The Zynq UltraScale+™ EG processor block diagram. EG devices feature a quad-core ARM® Cortex-A53 platform running up to 1.5GHz, combined with dual-core Cortex-R5 real-time processors, a Mali-400 MP2 graphics processing unit, and a 16nm FinFET+ programmable logic [1].

1.4 Hardware considerations

The need for hardware variety comes from several perspectives. First of all, the various hardwares have different purposes. For example, FPGAs were dedicated for a long time to prototyping,whereas CPUs were designed as an end-product. The GPUs were also designed as an end-product, but for other applications as seen above.

Moore's law is a prediction made in 1965 by Gordon Moore stating that the number of transistors in integrated circuits would double every two years with no price increase. This prediction foresees a constant growth in performance of processors. However, in the past twenty years, the efficiency of single core processor has been limited due to physical constraints. The clock rate could not increase anymore in a cost-effective way, as both the power consumption (due to leakage) and heat output increase drastically with high frequencies. Also, the size of the transistors is a very important limiting factor. It has reduced through the years until reaching its physical limits at an atomic size, for

a cost-effective mass production. To overcome those limitations, manufacturers design multi-core processors since year 2000. With the rise of embedded systems and new computing paradigms, the market now requires versatile hardware systems. Those systems are massively multi-core, and permit parallel computing. Yet, the performance improvement granted by parallel computing is limited as well, according to Amdhal's law. Amdhal's law states that the speed-up factor [4] obtained by parallelisation is limited by the following formula:

$$\frac{1}{1-p}$$

where p is the proportion of the program that cannot be executed in parallel. For example, if a 10-hour program have 9 hours that can be ran in parallel, Amdhal's law states that the maximal speed-up factor is $\frac{1}{1-0.9} = 10$. Therefore, parallelisation is not a solution to an infinite performance increase of the systems. Specialisation and versatility of the systems are thus a way to improve the performance. FPGA reconfiguration can be used so that the tasks are executed on specialised hardware, without having a specialised processor per task. With 3D technologies becoming more and more important in the future, leading to quicker reconfigurations, we believe that is a suitable options for the future of computing.

[4] The speedup factor is a number that measures the relative performance of two solutions processing the exact same problem, here one with parallelisation and one without.

Chapter 2

Common notions and models

This chapter defines the concepts used through this thesis. All the notations introduced are referenced in Table 11.2.

2.1 Time model

There are two major time models. Time may be viewed as discrete or continuous. The difference is explained below.

Definition 2.1 (Discrete time)**.** Assuming *discrete time* means that the time is represented as a natural number. The smallest atomic time interval is arbitrarily defined. It may be measured in millisecond or more likely in processor cycles for example. If this representation of time may seem to be unrealistic from the real-life application, it actually fits the behaviour of processors. One major component of any computing machine is its clock which synchronises its components.

Definition 2.2 (Continuous time)**.** Assuming *continuous time* means that the time is represented as a real number. This may be seen accurate to the *real world time*. However, it is hard to directly use this model in practice as a processor cannot be interrupted in between cycles. This representation of time fits theoretical results which may be then adapted to discrete time models.

2.2 Application model

The application is dedicated to fulfil the functionalities of the embedded system. Each functionality may have different characteristics, which will be explained later in this

section. In the application model, a functionality may be composed of one or several task(s). A task contains all the required information on how to execute the portion or the whole functionality it represents. To do so, it will release jobs at run-time that will perform the basic operations, and therefore handle the functionalities of the system. All those tasks form the task set of the application.

After defining the hierarchy of an application, we will define the different characteristics of the tasks and jobs.

Application hierarchy

Definition 2.3 (Job)**.** A system executes *jobs* on its platform. A *job* J_i is defined by three components: an arrival time a_i, a *WCET* (Worst-Case Execution Time) c_i and a deadline d_i. To complete a job, a set of specific operations must be done. The *absolute deadline* d_i represents the time by which the job should ideally be completed. In this work, we consider that a job cannot be executed on several processors at once.

Bounding the WCET of a job is not trivial. It requires a complete analysis of both the hardware and the software. Some existing literature covers this aspect, which is out of scope of this work. We thus assume the WCET as known in this thesis.

Definition 2.4 (Recurring task)**.** A *recurring task* is defined by two components (C_i, D_i). Its WCET is represented by C_i. Its relative deadline is D_i. A *recurring task* τ_i releases a potential infinite sequence of jobs. When a *recurring task* releases a job at a_i, its absolute deadline is $d_i \doteq a_i + D_i$ and its WCET is $c_i \doteq C_i$.

In this work, a job always belongs to a task.

Definition 2.5 (Task set)**.** The application will now be referred to as the *task set*. The task set $\Gamma \doteq \{\tau_1, \tau_2, \dots, \tau_n\}$ is composed of n recurring tasks. The task set contains all the different functionalities of the software, each one being represented as a task with its own characteristics and constraints.

For example, the software of a plane has at least two functionalities: it must know its position at any instant and controls the pressure inside the plane. In this example, we abstract any other functionalities. It may be modelled by two tasks: *GPS*, denoted as τ_g and *pressure control* denoted as τ_p. The plane software task set is thus $\Gamma = \{\tau_g, \tau_p\}$.

Computing the position is very quick so $C_g = 1$. It must be done quite fast: $D_g = 5$. Because it is not required to compute the position when the plane is on the ground, the first job of τ_g is released at 10, 0 being the start of the system, before the take-off of the plane. The first job arrival time of J_g is thus $a_g = 10$. Therefore, $d_g = 15$. Another job J_g is released at 20, and its job arrival time is $a_g = 20$, and its absolute deadline is $d_g = 25$.

Real-time criticality

In the following, the term *application* refers to *task set*. A task set may contain different kind of tasks, regarding the time constraints and the criticality of each task. The criticality refers to the level of risks encountered by the system in case of a deadline miss.

Definition 2.6 (Non real-time tasks)**.** A *non real-time task* is a task that has no deadline. It may be handled with best effort: when and where possible.

Definition 2.7 (Soft real-time tasks)**.** A *soft real-time task* is a task that is not critical. It may miss a deadline without leading to serious issue for the embedded system. Moreover, the output of the task may stay valid even after deadline misses. In case of a job deadline miss, the job may be aborted or completed depending on the task of this job.

Definition 2.8 (Firm real-time tasks)**.** A *firm real-time task* is a task that is not critical. It is close to a soft real-time task as few failures would not affect the functionality but only the quality of the service. The amount of deadline misses for an acceptable quality of service depends on the application. An acceptable quality of service must here be guaranteed.

Such tasks can be found in planes, for example the on-board entertaining systems. The task responsible for video streaming in planes is a firm real-time task. This task is a real-time task, as it must stream the frame in the correct order, otherwise the displayed video would be false. If too much frames are switched: the service becomes non-usable. However, failure of the streaming of all the frames in the right order would not produce any issue regarding the plane safety.

Definition 2.9 (Hard real-time)**.** A *hard real-time task* is a critical task. If it misses a deadline, the security of its embedded system may be compromised. As opposed to

soft real-time deadline misses, we never consider hard real-time deadline misses to be acceptable.

In cars, such a task may be the braking system, which would be probably composed of more than one task. If the brakes are activated later than expected, it may cause a serious issue to the car.

Deadline model

The relation between the task and the deadline is used in the task classification. We denote three kinds of deadlines.

Definition 2.10 (Constrained deadline)**.** A deadline is said to be *constrained* if and only if $\forall i,\ D_i \leq T_i$. Informally, an *constrained* deadline job must be completed strictly before the next job arrival of this task.

Definition 2.11 (Implicit deadline)**.** A deadline is said to be *implicit* if and only if $\forall i,\ T_i = D_i$. Informally, an implicit deadline job must be completed no later than the next job arrival of this task.

Definition 2.12 (Arbitrary deadline)**.** If there are no relation between T_i and D_i for a given task τ_i, a deadline is said to be *arbitrary*.

Periodicity model

Another taxonomy is the release rate of a task. Based on this criteria, we distinguish between two kind of tasks. We first introduce the notion of inter-arrival time.

Definition 2.13 (Inter-arrival time)**.** When considering two successive jobs J_i and $J_{i'}$ released by the same recurring task with $a_i < a_{i'}$, the *inter-arrival time* between them is the delay between the arrival time of J_i and the one of $J_{i'}$: $a_{i'} - a_i$.

Definition 2.14 (Sporadic task)**.** A *sporadic task* is a recurring task defined by three components (C_i, D_i, T_i). C_i and D_i are used with respect to Definition 2.4. The parameter T_i is the minimum inter-arrival time. Formally, for any two successive jobs J_i and $J_{i'}$ released by τ_i, with $a_i < a_{i'}$, the inter-arrival time is equal to or greater than T_i: $a_{i'} - a_i \geq T_i$.

Definition 2.15 (Periodic task)**.** A *periodic task* is a recurring task defined by three components (C_i, D_i, T_i). C_i and D_i are used with respect to Definition 2.4. The parameter

T_i is the period. Formally, for any two successive jobs released by τ_i, the inter-arrival time is always equal to T_i. Its k^{th} job J_i will thus be released at time $a_i = k \times T_i$, assuming that the first job was released at time $t = 0$.

Run-time property

Aside all those classifications, a job may be active or inactive at run-time.

Definition 2.16 (Active job)**.** Informally, a job is *active* if it may be executed. Formally, a job J_i is active at t if both following conditions are met:

1. If $a_i \leq t < d_i$;
2. The job is not completed yet.

Definition 2.17 ((In-)Active task)**.** An *active task* τ_i is a task that may release new job J_i. It is said to be inactive otherwise. By default, all the tasks from a task set are active at system start. A task may be deactivated and re-activated later.

Definition 2.18 (Rem-job)**.** A *rem-job* is a job whose task has been deactivated before completion. It must be completed, as a regular job would.

2.3 Hardware model

The platform is responsible for the execution of the application. It is based on the hardware layer of the system. In this thesis, we will consider the hardware layer as being only composed of one or several processors.

After giving a short definition of a processor, we will present different possible forms of a platform.

2.3.1 Processor model

Definition 2.19 (Processor)**.** A *processor* π_j is a unit which may perform operations to execute jobs, at a given *processing rate*, depending on its type.

The type of a processor depends on both its instruction set and its clock speed. If a task may be executed by a given processor, its *processing rate* is strictly positive and depends on the speed of the processor. The *processing rate* is null otherwise.

2.3.2 Platform model

A platform contains several components such as *memory elements*. In this work, we will abstract most of the times the other platform components to focus only on the processors. The platform is thus denoted as $\Pi \doteq \{\pi_1, \pi_2, \dots, \pi_m\}$, and is composed of m processors. Processor π_j has a type π^k, with $k \in [1, \dots, \phi]$, where ϕ is the number of different *processor types* on Π. This type will define the characteristics of the processor. The vector Ψ of size m contains the type of each processor. The j^{th} element of Ψ_j corresponds to the type of the processor π_j. For example, if $\Pi \doteq \{\pi_1, \pi_2, \pi_3\}$ with $m = 3$ and $\Psi \doteq \{\pi^1, \pi^2, \pi^1\}$ with $\phi = 2$: we know that π_1 and π_3 share the same type π^1, as $\Psi_1 = \Psi_3 = \pi^1$.

Definition 2.20 (Uniprocessor platform)**.** A *uniprocessor platform* is a platform containing a single processor, *network devices* and often many others components. *Uniprocessor platforms* may execute one job at a time. However, there are still a lot of open problems regarding such platforms.

Uniprocessor platforms are integrated nowadays in very conservative systems but also in systems that do not require a lot of processing capabilities. More complex platforms contain more than one processor. Please note that in the abstraction of this model, we do not take into account processor pipelining nor superscalar processors (as presented in Section 1.3) in the limitation of one job execution at a time.

Definition 2.21 (Multi-processor platform)**.** A *multi-processor platform* is a platform containing m processors. As for uniprocessor platforms, such a platform also contains other components as *memory elements*, *network devices* and many others depending on the system. In a *multi-processor platform*, there may be (at most) one job per processor executed at anytime. In this thesis, a uniprocessor platform is considered as a particular case of a *multi-processor platform*, where $m = 1$.

In our model, we do not consider parallelisation for tasks: a task may be executed only on one processor at a time. In other words, we forbid intra-task parallelism. We will distinguish between four different kinds of multi-processors platforms:

Definition 2.22 (Identical platform)**.** On *identical platforms*, all the processors are identical. Hence, processor types are abstracted and all processors execute the tasks at the same processing rate of 1. To complete a job of task τ_i having WCET of c_i, it takes c_i units of time on any processor.

Definition 2.23 (Heterogeneous uniform platform)**.** A *heterogeneous uniform platform* is a multi-processor platform. However, all the processors share the same instruction set, and thus only the clock speed differs from one processor to another. Therefore, in such a platform, the processing rate depends only on the processor type. Formally, the processing rate of π_j is denoted as R_j. To complete a job of task τ_i with a WCET of C_i, it takes $\frac{C_i}{R_j}$ units of time on the processor π_j. This processing rate is relative to a processing rate of a fictional processor.

For example, in an heterogeneous uniform platform, a processor of processing rate of 2 executes any task at twice the rate of a processor of processing rate 1.

Definition 2.24 (Heterogeneous unrelated platform)**.** A *heterogeneous unrelated platform* is a multi-processor platform. Unlike heterogeneous uniform platform, the processors may be completely different: having different instruction sets and/or different clock speeds. Thus, the processing rate depends on both the executed task τ_i and the processor π_j. Formally, the processing rate of task τ_i on π_j is denoted as $R_{i,j}$. On the processor π_j, it takes $\frac{C_i}{R_{i,j}}$ units of time to complete a job of task τ_i with a WCET of C_i, This processing rate is relative to a processing rate of a fictional processor.

The following formalises the notion of *heterogeneous consistent platforms*. First to be *consistent* the platform must have a *relative order* on the processors.

Definition 2.25 (Faster processor)**.** A processor π_j is *faster* than processor π_ℓ ($\pi_j \geq \pi_\ell$) if

$$\forall 1 \leq i \leq n, \quad R_{i,j} \geq R_{i,\ell}$$

Now we introduce a tie-breaker to have the notion of the *fastest* processor:

Definition 2.26 (Fastest processor)**.** π_j is defined to be *the fastest* processor if j is the smallest index such that $\forall 1 \leq \ell \leq m,\ \pi_j \geq \pi_\ell$.

Definition 2.27 (Heterogeneous consistent platform)**.** A *heterogeneous consistent platform* is a particular case of *unrelated* platforms where the heterogeneity is *consistent*. On an *heterogeneous consistent platform*, we have a total order on the processors. Without loss of generality (by reordering the processors) we can assume that π_1 is the fastest processor. By repeating the same definition on the remaining processors, we have $\pi_1 > \pi_2 > \cdots > \pi_m$.

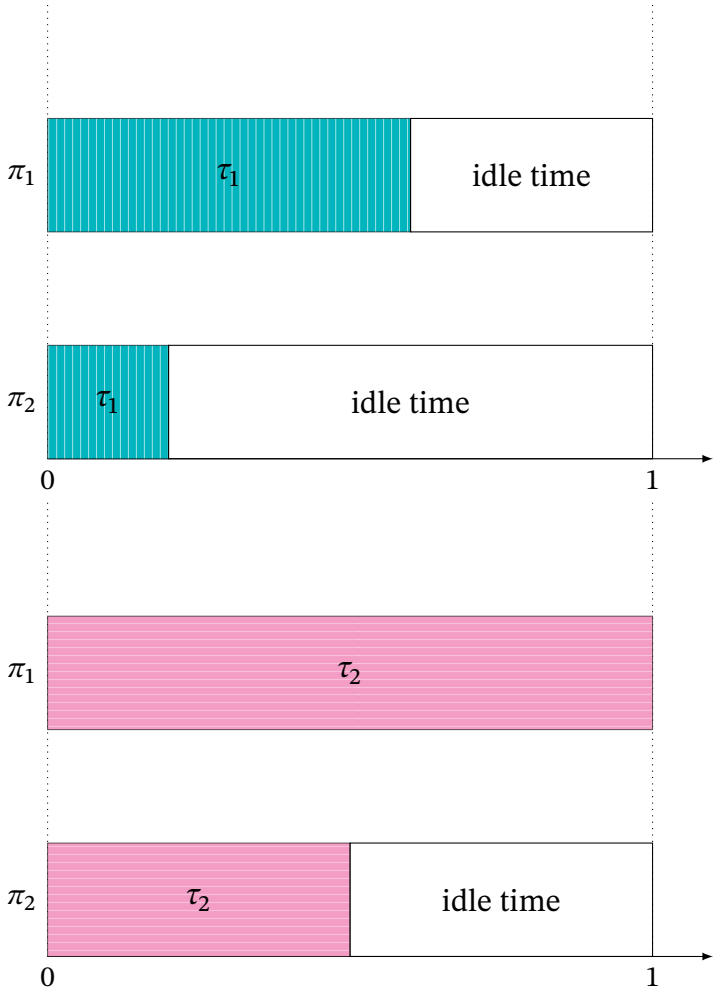


Figure 2.1: Visual representation of task utilisation

We now define the notions of idle and processor utilisation.

Definition 2.28 (Idle)**.** If a processor doesn't perform operations, it is said to be in *idle state* or simply *idle*.

Definition 2.29 (Processor utilisation)**.** The *processor utilisation* of a task τ_i on a given processor represents the portion of time it requires to be executed over its deadline to be completed.

Formally, the utilisation of a task τ_i on a processor π_j is $U_{i,j} \doteq \frac{C_i}{T_i \times R_{i,j}}$, where $R_{i,j}$ is the processing rate of τ_i on π_j. In the case of uniprocessor and identical platforms, we consider $R_{i,j} = 1$, $\forall i, j$. In the case of homogeneous platforms, we consider $R_{i,j} = R_j$, $\forall i$. Thus, it may be simplified for uniprocessor or identical platforms as $U_i = \frac{C_i}{T_i}$. This notion is illustrated in Figure 2.1.

In the following example, we illustrate the notion of utilisation on an heterogeneous unrelated platform with 2 processors. The task set specifications are the following:

	C_i	T_i	$R_{i,1}$	$R_{i,2}$
τ_1	6	10	1	3
τ_2	10	10	1	2

This task set contains two different tasks. The first task has a processing rate of 1 on processor π_1 and of 3 on processor π_2. The second task has a processing rate of 1 on processor π_1 and of 2 on processor π_2. Therefore, $U_{1,1} = \frac{C_1}{T_1 \times R_{1,1}} = \frac{6}{10}$. Similarly, $U_{1,2} = \frac{2}{10}$, $U_{2,1} = 1$, $U_{2,2} = \frac{5}{10}$. This is represented on Figure 2.1. It can clearly be seen that processor π_2 could execute up to 5 tasks with similar utilisation to τ_1, when π_1 could execute no more than 1 task similar to τ_2.

2.3.3 Reconfigurable processors

Beyond heterogeneity, modern platforms include reconfigurable processors, such as the ones presented in Section 1.3. Such reconfigurable processor of type π^k is configured at any time in a *configuration* $\theta_c \in \Theta_k$ from its set of configurations Θ_k. The configuration of a processor defines several parameters like the instruction set of the processor or its processing rate. The set of sets $\Theta \doteq \{\Theta_1, \Theta_2, \dots, \Theta_\phi\}$ contains the set of allowed configurations for each type. A configuration belongs to at most one set of configurations: i.e. $\forall k, k', k \neq k' \implies \Theta_k \cap \Theta_{k'} = \emptyset$. There are o different configurations, with $o \doteq \sum_{k=1}^{\phi} |\Theta_k|$. Reconfiguring a processor is not instantaneous. It takes δ_c time-units (denoted as the *reconfiguration delay*) to reconfigure a processor of type π^k to θ_c if $\theta_c \in \Theta_k$, otherwise it takes $+\infty$. We consider here parallel reconfigurations: the whole platform may be reconfigured simultaneously.

While our model aims at targeting platforms formed by reconfigurable processors, those often contain one or several non-reconfigurable processors.

Definition 2.30 ((Non-)Reconfigurable)**.** A processor π_j of type π^k may be *reconfigurable* or *non-reconfigurable*. It is said to be *reconfigurable* if and only if $|\Theta_k| > 1$.

A reconfigurable processor may be turned on and off dynamically. This may be used to manage power consumption. To handle this aspect, we now define the notion of active processor.

Definition 2.31 ((In-)Active reconfigurable processor)**.** A processor π_j is *inactive* at time t_1 if and only if it is configured in an idle-configuration. An idle-configuration is a configuration in which a processor cannot execute any job, i.e. its processing rate is null for any job. It is said *active* otherwise. By definition, a non-reconfigurable processor has no idle-configuration, as it cannot be turned off. Thus, a non-reconfigurable is always active.

We refine here the notion of *idle processor* for reconfigurable processor.

Definition 2.32 (Idle reconfigurable processor)**.** A processor is *idle* if it is neither executing tasks nor being reconfigured.

Definition 2.33 (Processing rate on reconfigurable processors)**.** On reconfigurable processors, the *processing rate* depends on both the task and the processor. Specifically, the job processing rate $R_{i,c}$ on the processor π_j depends on both the task τ_i and the current configuration θ_c of π_j. Formally, a processor executes $t \times R_{i,c}$ computing units when configured in θ_c and executing a job J_i for t time-units. This amount is null if the task cannot be executed on this configuration.

2.3.4 Clustered platforms

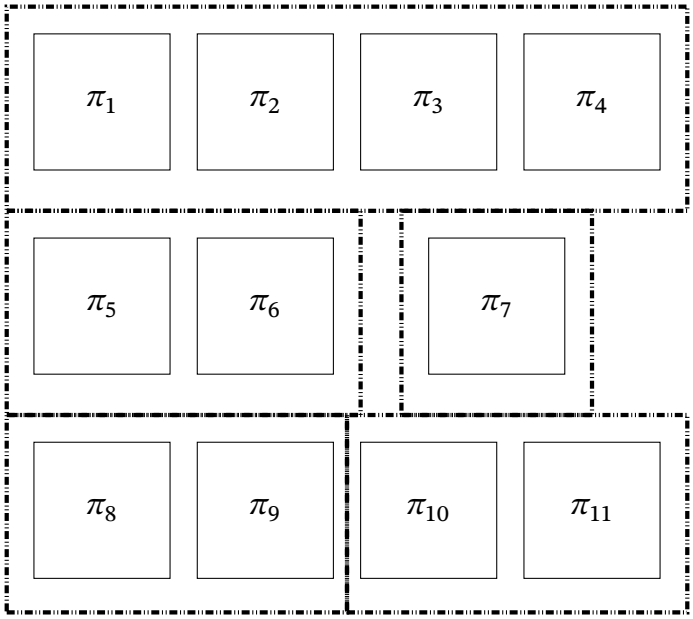
A platform may be divided into several groups of processors. By default, there is only one group containing all the processors. If there are more than one group, the platform is said to be clustered. Inside each group, the processors are referred to as *cores*. The groups of cores are then called *clusters*.

Formally, an unrelated multi-processor platform is modelled by a set of $\dot{m}$ clusters $\dot{\Pi} \doteq \{\dot{\pi}_h \mid h = 1, \dots, \dot{m}\}$. Each cluster $\dot{\pi}_h$ is formed by $\dot{m}_h$ *cores*: $\dot{\pi}_h \doteq \{\pi_{h_1}, \dots, \pi_{h_{\dot{m}_h}}\}$. By construction, $\sum_{h=1}^{\dot{m}} \dot{m}_h = m$. Clusters are formed arbitrary: no constraint holds on whether a core can or cannot belong to a specific cluster.

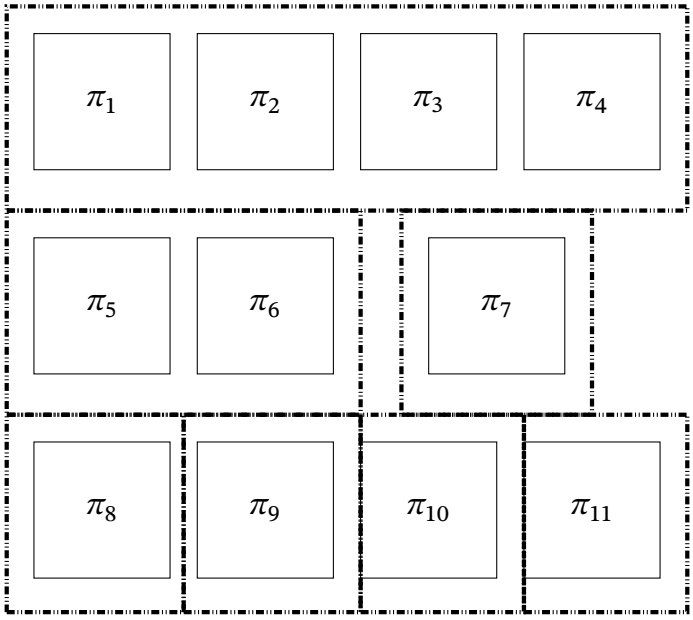
Figure 2.2 represents the same platform with two different clustering. For example, π_8 is on a cluster with π_9 in the first platform division (see (a)) but is then alone in the second platform division (see (b)).

We now introduce the following concepts on clusters.

Definition 2.34 (Idle cluster)**.** A cluster is said to be *idle* if and only if all its cores are idle.



(a) Plausible platform division



(b) Implausible platform division

Figure 2.2: Two possible divisions of the example platform (based on the Zynq UltraScale+™) into several clusters.

Definition 2.35 (π^k-cluster)**.** A π^k-cluster is composed of cores of type π^k, configured identically.

2.4 Scheduler

A system is composed of an application, a hardware and a middleware. Among other tasks, the middleware handles the communication between the application and the hardware. In this work, the middleware is the real-time operating system (RTOS). To work as expected, the task set representing the application must be executed on the hardware according to its specifications. One responsibility of the real-time operating system is to choose a processor to execute a job at a chosen instant. As every task in the task set has its own requirements, with possibly different *affinities* on different processors, making the *right* choice is not trivial. The part of the RTOS responsible of that choice is called the *scheduler*. It may be performed before the system lifespan and/or during the system lifespan. This section presents the types of scheduler considered in this thesis, and their characteristics.

Definition 2.36 (Offline)**.** A scheduler is said to be *offline* if all the scheduling decisions are made before the start of the system. At run-time, after the start of the system, the scheduler simply applies the decisions made offline. The most common form of offline schedule decisions is the production of a pattern that will be repeated over time online. In the case of a repeated pattern schedule (or *pattern scheduling*), the scheduler constructs a pattern offline of a specific size. In Figure 2.3, a pattern of size 1 is displayed. This pattern will be repeated every unit of time: formally, $\forall t \in \mathbb{N}$, at $t + 0.5$, if τ_1 has an active job then it will be scheduled on processor π_2 for 0.5 unit of time.

One of the major advantages of an offline scheduler is its low run-time complexity. Because it has no or very low computation to perform at run-time, it is very easy to implement in the system and takes a very low computing capacity. Using a low computing capacity makes the remaining computing capacity as large as possible for the actual work: the execution of the application. Of course, its main drawback is that it lacks of flexibility to any event occurring at run-time. For example, in Figure 2.3, if only τ_1 is active then it is probably possible to schedule τ_1 on only one processor instead of scheduling it on both processors π_1 and π_2.

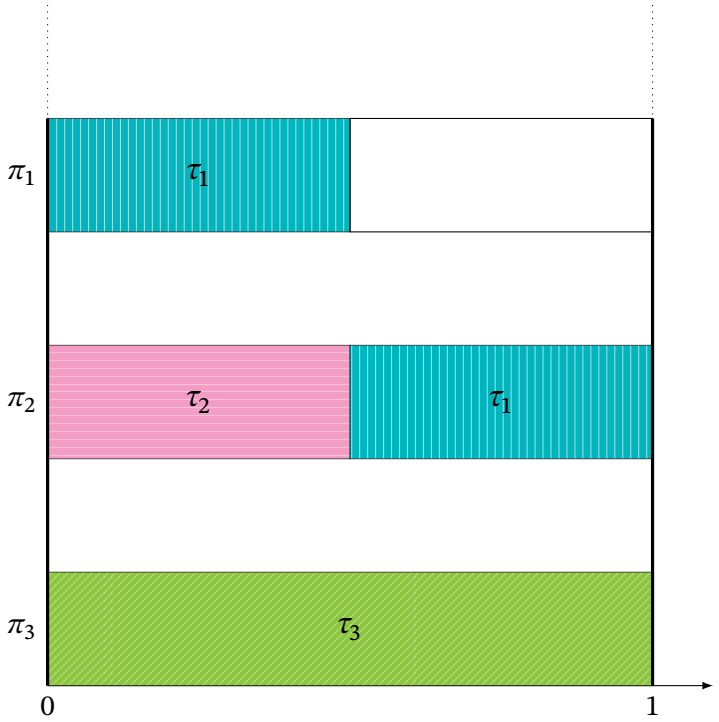


Figure 2.3: Example of a scheduling pattern

Definition 2.37 (Online)**.** A scheduler is said to be *online* if some scheduling decisions are made online. Those decisions could be to make up a priority order for the tasks, jobs or any other decisions. In the first case, the schedule is called *fixed task priority scheduler*. Each job is given a priority according to the priority of its tasks, when released. This results in a priority order. In such a schedule, if both τ_1 and τ_2 have an active job at time t, J_1 is preferred over J_2 at time t if and only if τ_1 has a higher priority than τ_2. This is denoted as $\tau_1 \succ \tau_2$.

In a fixed job priority scheduler, the priority is given at job release. Thus, the jobs of two tasks may not always be ordered in the same way. Formally, if $J_{i_1}, J_{i'_1}$ are jobs from τ_{i_1} and $J_{i_2}, J_{i'_2}$ are jobs from τ_{i_2}, it is possible that $J_{i_1} \succ J_{i_2}$ when $J_{i'_1} \prec J_{i'_2}$.

With a dynamic scheduler, the priorities may change at any instant. Formally, if there are two jobs J_{i_1}, J_{i_2} in the system, and $J_{i_1} \succ J_{i_2}$ at instant t_1, it is possible that $J_{i_1} \prec J_{i_2}$ at instant t_2, if $t_2 \neq t_1$. A dynamic scheduler may thus adjust its behaviour to the state of the system. However, they have a highest run-time complexity and tend to be more complex to implement.

An important characteristic of the scheduler is defined next.

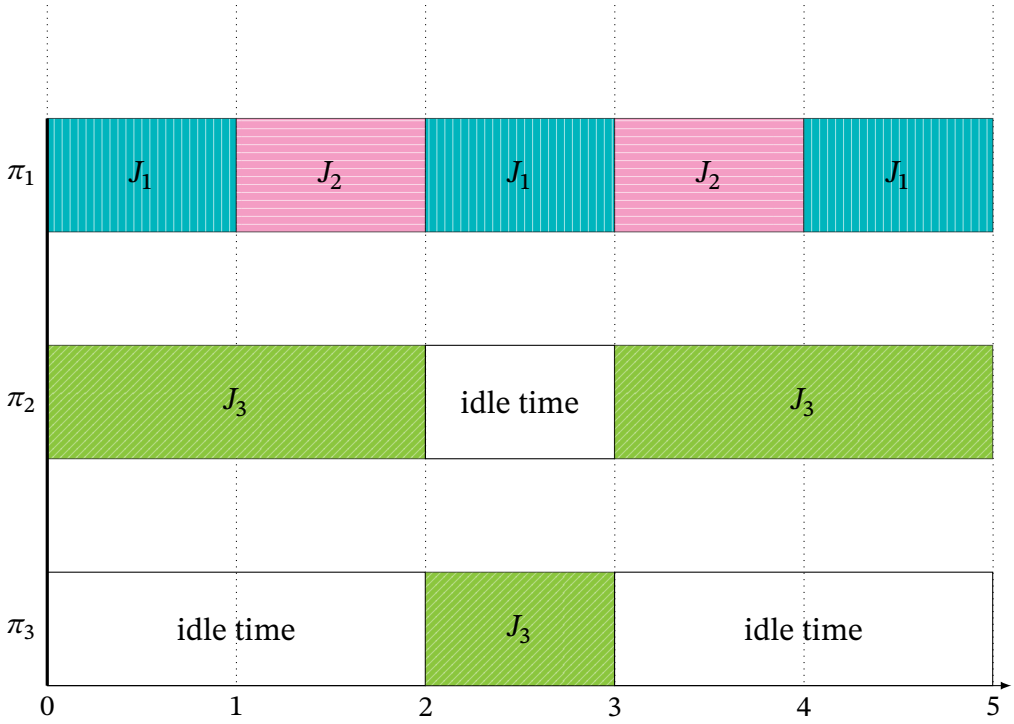


Figure 2.4: Visual representation of job preemptions and migrations

Definition 2.38 (Work-conserving scheduler)**.** A scheduler is *work-conserving* if and only if a processor is idle when there are no job waiting to be executed.

Scheduling events

When a job is executed, it becomes completed after a certain amount of time. However, it is not always mandatory to execute a job from start to finish with no interruption, and the two following definitions illustrate common events on real-time scheduling.

Definition 2.39 (Job preemption)**.** A *job preemption* occurs when an executing job is stopped before completion, in the favour of another job being executed on the same processor. It may be executed again later, on the same processor or another one. The previous computation time is taken into account. Formally, if J_i must be executed for c_i and is preempted $t_1 < c_i$ units of time after the start of its execution, it will need to be executed for $c_i - t_1$ units of time to be completed.

As it will be explained in Section 2.6, a job may use local memory during its execution. In the case of a preemption, a part or all the content of this local memory may be transferred to another memory to be stored until the resumption of the execution of the job. This is called a *context switch*. We observe that a *context switch* takes a delay

which lessens the performance of the system.

Preemptions may be prohibited by certain systems or tasks. The latter are then called *non-preemptive tasks*. In the theoretical frame of this thesis, we consider that no delay occurs when performing a preemption.

This concept is depicted in Figure 2.4.

Definition 2.40 (Job migration)**.** A *job migration* occurs when a job is stopped before completion and is resumed on another processor.

In the case of a job migration, a part or all the content of this local memory may be required to be transferred to the new processor, which forms a *context switch*. This is why migrations must be avoided when possible. Job migrations are even unauthorised by certain systems or tasks. In the theoretical frame of this thesis, we consider that no delay occurs when performing a migration.

This concept is depicted in Figure 2.4.

Definition 2.41 (Task migration)**.** A *task migration* occurs when two successive jobs of a given task are not executed on the same processor. They are thus less costly than job migrations, but may still be forbidden on some systems. In addition, to migrate a task from a processor to another with a different instruction set, the system must have different compiled versions (binary codes) of the task. This is costly in terms of memory usage.

In Figure 2.4, we can observe several job preemptions and job migrations. For example, J_3 migrates from π_2 to π_3 at 2, and then back to π_2 at 3. On π_1, J_1 is preempted several times by J_2.

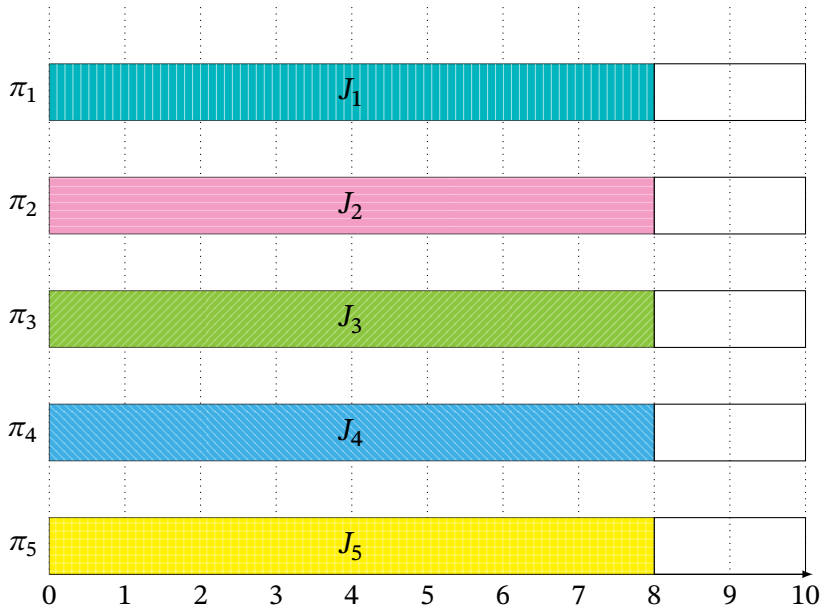
Multi-processor scheduling paradigms

Several approaches exist to schedule a task set on multi-processor systems. The following definitions describe them: the first and second one are the extreme cases, and the two others are in between.

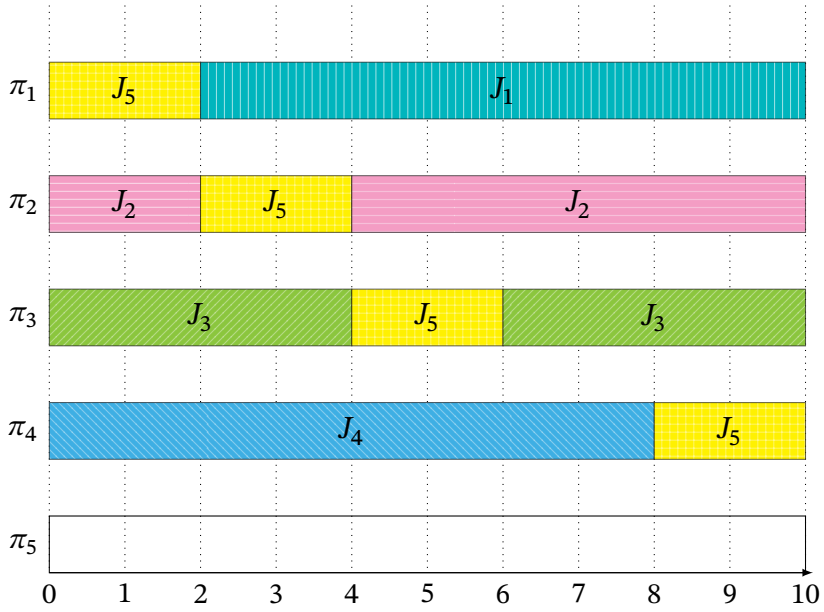
Definition 2.42 (Partitioned)**.** In *partitioned scheduling*, no migrations are allowed. Each task is assigned offline a processor, and all the jobs will be executed on this very same processor. Each task subset may then be scheduled by a uniprocessor scheduler.

	C_i	T_i
τ_1	8	10
τ_2	8	10
τ_3	8	10
τ_4	8	10
τ_5	8	10

(a) Task set specifications



(b) Example of a partitioned schedule



(c) Example of a global schedule

Figure 2.5: Visual representation of partitioned and global schedules

This paradigm permits to re-use existing work on uniprocessor scheduling. It is a simple approach to multi-processor system scheduling and avoid any migration cost by construction. However, it does not benefit from most of the advantages of multi-processor systems. It is illustrated in Figure 2.5(b).

The limitation of partitioned scheduling may be seen in the following example. Here, we are considering a 5-tasks task set with periodic and preemptive tasks. $C_1 = C_2 = ... = C_5 = 8$ and $T_1 = T_2 = ... = T_5 = 10$. In this example, the utilisation of each task is 0.8. Such a task set cannot be executed on less than 5 identical processors with a partitioned scheduler as any couple of tasks would have a utilisation greater than 1. This leads to *losing* the equivalent of $0.2 \times 5 = 1$ processor, that is idle.

Definition 2.43 (Global)**.** In *global scheduling*, there is no restriction regarding migrations. Any job may migrate from any processor to another. This paradigm allows the use of the whole platform. It is illustrated in Figure 2.5(c).

Let's re-use our previous example, with a 5-tasks task set with periodic and preemptive tasks, where: $C_1 = C_2 = ... = C_5 = 8$ and $T_1 = T_2 = ... = T_5 = 10$. A global scheduler may schedule this task set with only 4 processors, resulting in no idle time at all *if migrations are instantaneous*.

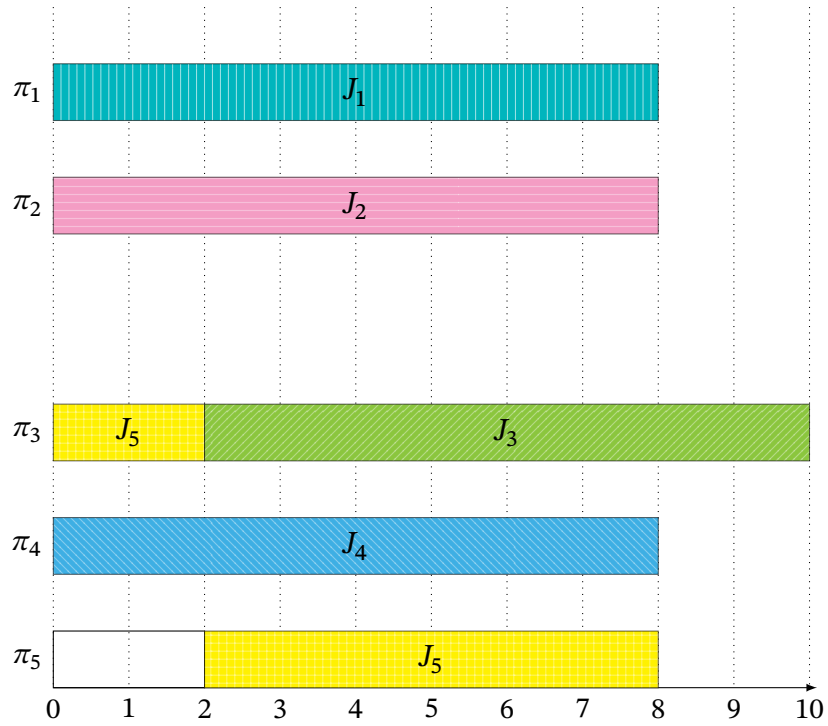
Definition 2.44 (Semi-partitioned)**.** In *semi-partitioned scheduling*, some tasks are assigned to a processor, when others may migrate. In Figure 2.5(c), most tasks are partitioned, when τ_5 migrates.

Definition 2.45 (Clustered)**.** In *clustered scheduling*, tasks are allowed to migrate to certain processors. Formally, the platform is divided into sets of processors. Each set is called a *cluster*, as introduced in Section 2.3.4. The task set is also divided in subsets, and each subset is assigned to a specific cluster. Each task subset may be scheduled *globally* on this cluster, but may never migrate to a core of another cluster.

Each cluster may be formed of identical processors to simplify the scheduling problem, or use heterogeneous clusters for more flexibility. It is illustrated in Figure 2.6(b), where π_1 and π_2 form the first cluster, and the others the second cluster.

	C_i	T_i
τ_1	8	10
τ_2	8	10
τ_3	8	10
τ_4	8	10
τ_5	8	10

(a) Task set specifications



(b) Example of a clustered schedule

Figure 2.6: Visual representation of a clustered schedule

Task set feasibility and schedulability

Two important notions regarding scheduling are the notions of feasible and schedulable task set.

Definition 2.46 (Feasible task set)**.** Some task sets are impossible to schedule on a given platform due to their constraints. A task set is said to be *feasible* on a given platform if and only if there exists a schedule with no missed deadline. It is said to be *unfeasible* otherwise.

For example, it is impossible to schedule a task set composed of two tasks τ_1, τ_2, where $C_1 = C_2 = 1$ and $T_1 = T_2 = 1$ on a uniprocessor platform.

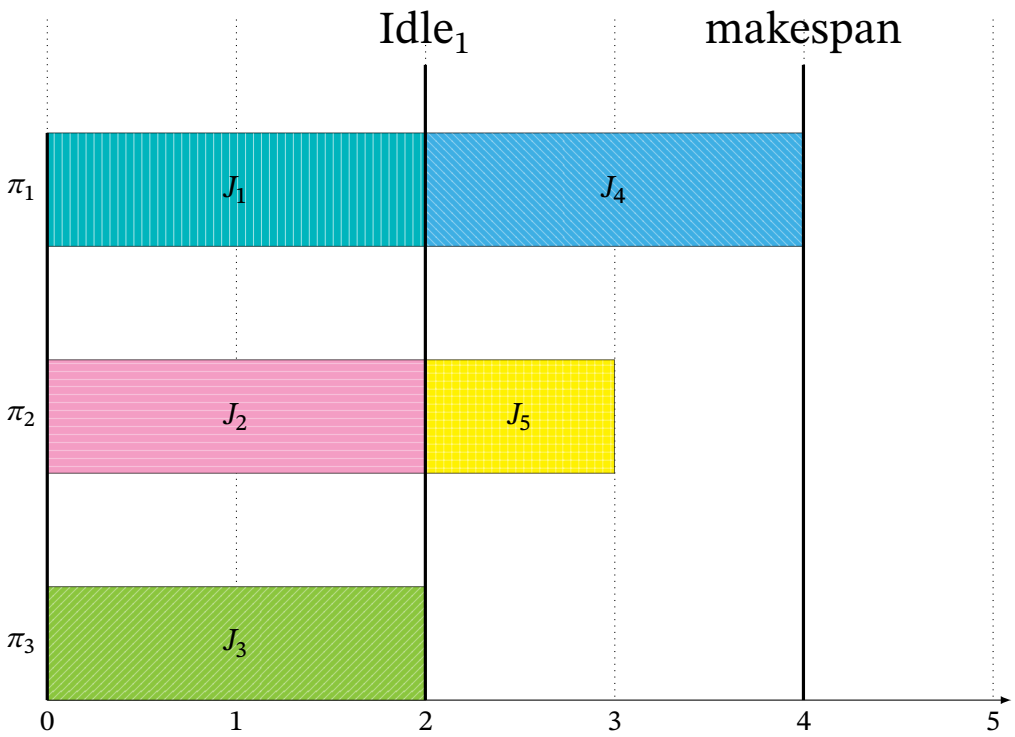


Figure 2.7: Visual representation of the idle instant and makespan

Definition 2.47 (Schedulable task set)**.** A task set is said to be *schedulable* on a given platform by a given scheduler if and only if this scheduler schedules it with no missed deadline.

Definition 2.48 (Optimal scheduler)**.** A scheduler S is said to be *optimal* if and only if any feasible task set is schedulable with S.

In this thesis, a task parameter is its WCET. However, this worst-case is not always reached.

Definition 2.49 (C-sustainable scheduler (from [2]))**.** A scheduler is *C-sustainable* if and only if a system schedulable when tasks are using their WCET remains schedulable even if some tasks do not use up to their WCET.

The same notion applies to feasibility and schedulability tests. In this thesis, we will prove that all schedulers and schedulability tests introduced in this thesis are C-sustainable.

Definition 2.50 (Idle$_j$ instant (from [3]))**.** Let J be any finite set of n jobs. Let $\dot{\pi}_h$ be a multi-processor cluster formed by $\dot{m}$ cores, upon which jobs of J are scheduled. If

s denotes that schedule, then the $Idle_j(J, \dot{m})$ instant (with $j = 1, \dots, \dot{m}$) is the earliest instant in s such as at least j processors are idle, assuming that no more jobs will be released. An upper-bound on the $\text{Idle}_j(J, \dot{m})$ instant is denoted as $\overline{\text{Idle}}_j(J, \dot{m})$. We will use the shortened notations Idle_j and $\overline{\text{Idle}}_j$ to enhance the readability.

This notion is illustrated in Figure 2.7

Definition 2.51 (Cluster makespan)**.** Let J be any finite set of n jobs. Let $\dot{\pi}_h$ be a multi-processor cluster formed by $\dot{m}$ cores, upon which J are scheduled. If s denotes that schedule, then the *cluster's makespan* is the required time to complete all the jobs J, assuming that no more job will be released. [4] proposed an upper-bound, on the worst-case makespan $\overline{\text{makespan}}(J, \dot{m})$, based on this assumption. $\overline{\text{makespan}}(J, \dot{m})$ will be denoted as $\overline{\text{makespan}}$ to enhance the readability.

This notion is illustrated in Figure 2.7

2.5 Multi-mode application

In [5], the authors model an application running with a single set of functionalities, on a uniprocessor platform. Nowadays, some applications are much more complex. The hardware tends to offer more and more possibilities to increase its efficiency, and therefore becomes also more complex. Those two combined make the seminal model from [5] unfit for some applications. For example, a plane operates in different contexts, and its application is in different states with different functionalities. But some functionalities are never active at the same time, as shown in Figure 2.8. This figure shows a plane application in three different states: on the ground, on cruise mode and on the ground after the landing. If the *pressure regulation* is performed through the different states, the others functionalities are used in only one state.

In the seminal model, the application always runs the same set of functionalities, represented by the task set. It is always in the same state or *mode*. In the previous example, as a task (for example: ensuring the correct refuelling after the landing) will not be active all the time, the use of sporadic tasks will match the reality of the application. However, because no information about the real state in which the plane is in, the analysis is not aware that the refuelling may never occur alongside the GPS task. Therefore, it will be highly pessimistic. To tackle this issue, the different *modes* of the application can be explicitly modelled. Each mode may execute a different task

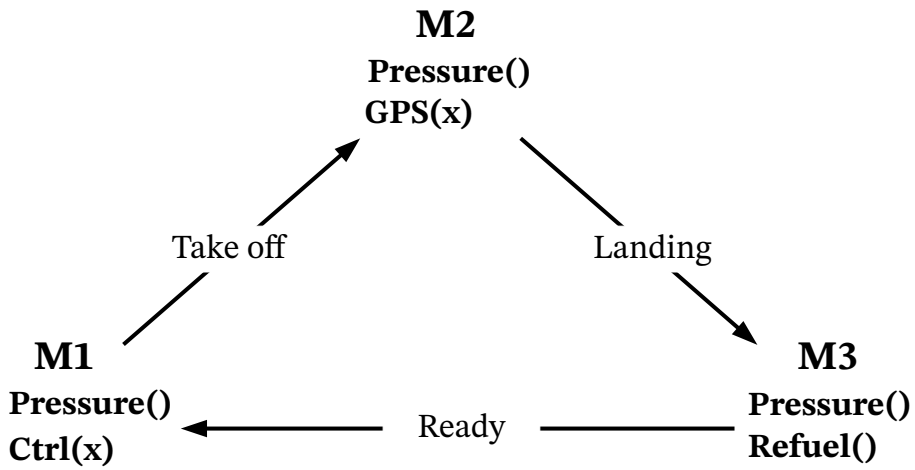


Figure 2.8: Multi-mode illustration: different functionalities of a plane

Task	Utilisation
Pressure regulation	0.2
Various control	0.8
GPS	0.7
Refuel	0.8

Figure 2.9: Multi-mode illustration: processor utilisation of the different functionalities

subset and thus the analysis accuracy is increased.

The functionality utilisations of the example are displayed in Table 2.9. In this example, if the worst-case analysis supposes that all the tasks may be active at the same time, the total utilisation is $0.2 + 0.8 + 0.7 + 0.8 = 2.5$. This application would require at least 3 processors to be successfully scheduled. However, if the different *modes* are explicitly defined, the worst-case utilisation will equal to $\max\{0.2 + 0.8, 0.2 + 0.7, 0.2 + 0.8\} = 1$. This new worst-case utilisation is ~60% smaller than the previous one. Because the maximal utilisation equals 1, the application may be ran with only one processor. This drops the requirements of the application in term of hardware, directly reducing the cost, energy consumption, space and weight of the system.

The system specifications must define for each *mode* the task subset out of the task set. Each mode task subset is independent or partially independent, as tasks may be present in different task subsets. In the example, the task *Pressure regulation* is present in all the different *modes*. More than simply the task subsets, the transition from one mode to another must be defined. A transition occurs when a *mode change request* occurs. This request can be based on an event, or occur at a specific time. For example, mode change requests on a plane are event based: the pilot manually triggers the landing. However, mode change requests in a thermal sensor may occur at fixed time, to operate only during working hours. During a transition or *mode change* phase, the current mode is deactivated while the new mode is activated. As we deal with hard real-time applications, the transitions must be done in a given delay. This delay is here based on the future mode. Of course, not all transitions are allowed. Only a subset of all the transitions are allowed, which is specified at design time. To handle those *mode change phase*, a *protocol* is used. It needs to respect the real-time constraints specified. Section 2.5.1 and Section 2.5.2 define precisely those notions.

2.5.1 Mode model

Formally, a multi-mode application is composed of μ *modes* $M \doteq \{M^1, M^2, \dots, M^\mu\}$. A mode $M^q =< T^q, \Delta^q >$ contains a task subset T^q. For each mode, Δ^q is the real-time constraint. This real-time constraint ensures that every mode will be enabled on time. It constraints the maximal duration between the mode change request and the completion of the mode change phase. It is chosen at design time.

Definition 2.52 (Mode task subset)**.** The mode M^q contains a task subset T^q. It is formed by n^q tasks. It is allowed that two modes share the same task subset.

Definition 2.53 (Mode real-time constraint)**.** Switching from one mode to another is not instantaneous. Rem-jobs cannot be aborted before completion (see Definition 2.18). However, this delay must be bounded to take into account the real-time constraints of the application. Δ^q represents the maximum allowed delay for reconfiguring the system after a mode change request to M^q. The set $\Delta \doteq \{\Delta^1, \Delta^2, \dots, \Delta^\mu\}$ contains the real-time constraint of each mode.

Mode transitions

The application executes at any instant one and only one mode M^q. This mode M^q is the *active mode*. The active mode may only change during a *mode change phase*. A

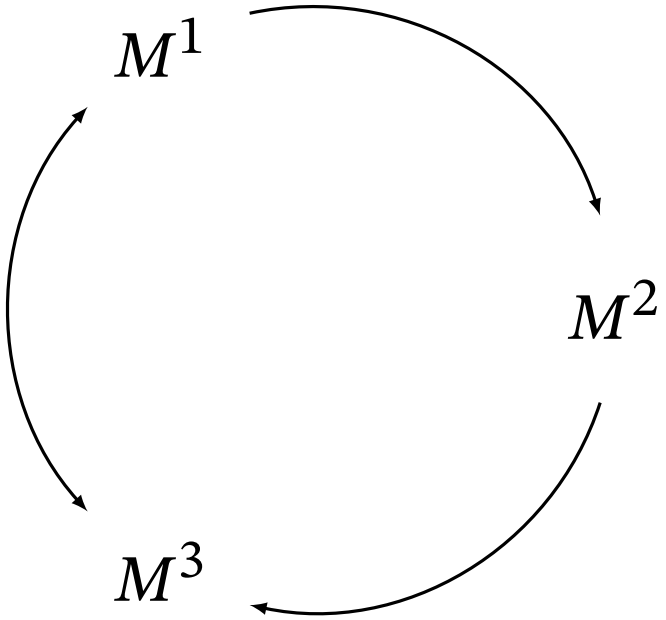


Figure 2.10: Graph transition example

mode change phase is triggered when the system receives a *mode change request*. When a mode change request MCR(M^{dst}) occurs at t_{MCR}, the current mode is immediately deactivated , and new mode M^{dst} must be activated by $t_{\text{MCR}} + \Delta^{\text{dst}}$. Δ^{dst} is a hard real-time constraint specified at design time for each mode M^{dst}. The mode change phase ends when the new mode M^{dst} is activated. The protocol is in charge of enabling and disabling the task subsets. Please note that this constraint depends only on the destination mode M^{dst}, independently from M^{src}.

Mode change graph

In an application, some transitions will never occur. The allowed transitions are represented in the *mode change graph* $\mathcal{G} \doteq \{\text{V}, E \subseteq \text{V}^2\}$. The *mode change graph* is a directed graph, where V contains one and only one node for each mode $M^q \in M$, and E represents all the allowed transitions from one mode to another. A mode change phase from a mode M^{src} to a mode M^{dst} is allowed if and only if $(M^{\text{src}}, M^{\text{dst}}) \in E$. A graph example is given in Figure 2.10. In this example, the mode following M^1 may be M^2 or M^3. However, the mode following M^2 must be M^3 and may not be M^1: the directed edge from M^1 to M^2 is unidirectional.

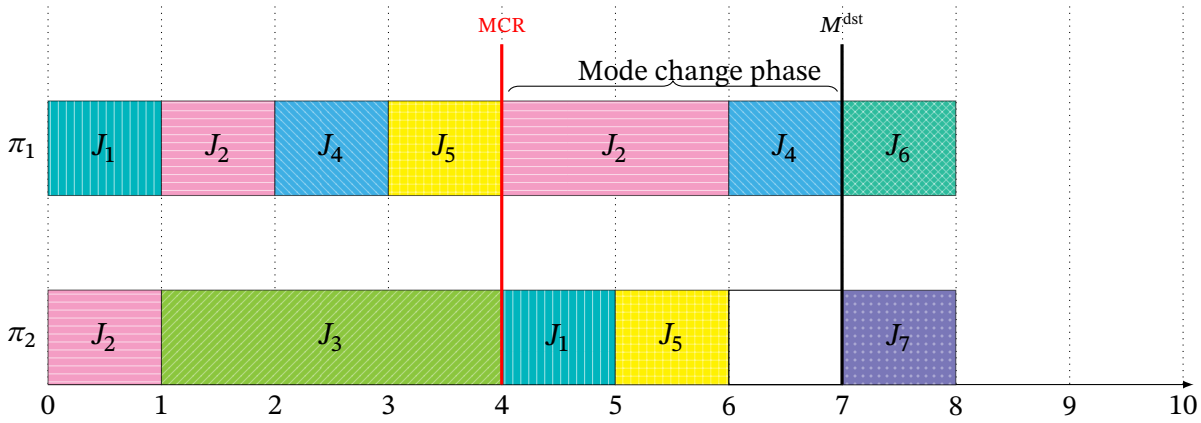


Figure 2.11: Mode transition illustration

More on tasks

Definition 2.54 (Mode-dependent task)**.** A *mode-dependent* task is a task that is present in at one and only one mode task subset. Formally, τ_i is mode-dependent if and only if $\exists q, \tau_i \notin T^q$. If an application contains only mode-dependent tasks, the intersection of the task subsets is empty. Formally, $\forall q, q', q \neq q' \implies T^q \cap T^{q'} = \varnothing$.

Definition 2.55 (Mode-independent task)**.** A *mode-independent* task appears in all the modes. Formally, if τ_i is mode-independent, then $\forall q, q', \tau_i \in T^q \cap T^{q'}$. Moreover, such tasks must not be affected by a mode change phase. It must not be disabled when a mode change request occurs. It will therefore continue to release new jobs and will produce no rem-job.

In Figure 2.11, a mode change phase is illustrated. A mode change request occurs at 4. At this instant, 4 jobs are active: J_1, J_2, J_4, J_5. All tasks are deactivated by the protocol, and the active jobs are converted to rem-jobs. Once completed at 7, the new mode M^{dst} may be enabled. Its task set is enabled as well: both tasks τ_6 and τ_7 are activated and both release a job.

The following definitions characterise the tasks depending on their presences in both the old and the new task subset, during a mode transition.

Definition 2.56 (Unchanged task (from [6]))**.** In a given mode transition from M^{src} to M^{dst}, a task is an *unchanged task* if it is present in both T^{src} and T^{dst}.

Definition 2.57 (Wholly new task (from [6]))**.** In a given mode transition from M^{src} to M^{dst}, a task is a *wholly new task* if it is not present in T^{src} but present in T^{dst}.

2.5.2 Protocol model

A protocol is an algorithm. It must handle the *mode change phases*, during the lifespan of the multi-mode application. Thus, it must handle the software reconfigurations. This includes the activation and deactivation of the modes, and the enabling and disabling of the mode task subsets. The protocol operates above the schedulers. A scheduler is only responsible for the scheduling of the task subsets.

As we consider real-time multi-mode applications, we care about the respect of the real-time constraints. A protocol *handles successfully* an application if no real-time constraints are violated during its lifespan. To be usable, it needs to have a validity test associated. This test checks whether a given application will always be handled successfully by a given protocol, or if a real-time constraint may be violated in one or several scenarios.

Two definitions regarding protocols follow.

Definition 2.58 (Protocol with periodicity (from [6]))**.** A protocol is *with periodicity* if and only if unchanged tasks are executed independently from the mode change in progress, preserving their activation pace. It is *without periodicity* otherwise.

Definition 2.59 (Synchronous protocol (from [6]))**.** A protocol is *synchronous* if and only if mode-dependent tasks of two different modes can never be active simultaneously. It is said to be *asynchronous* otherwise.

2.6 Memory considerations

In this section, we will not give a detailed model of the memory components. It is out of the scope of this thesis. We propose here instead a short introduction to the memory behaviour of a platform.

Each *processor* has its own memory component, denoted as a *local cache* or the *L1-memory*. It is a very fast read/write memory, however it has a very limited capacity. As the name suggest, it can be accessed by only one processor. A group of *processors* share the *L2-memory* via their own buses. If required, those processors may

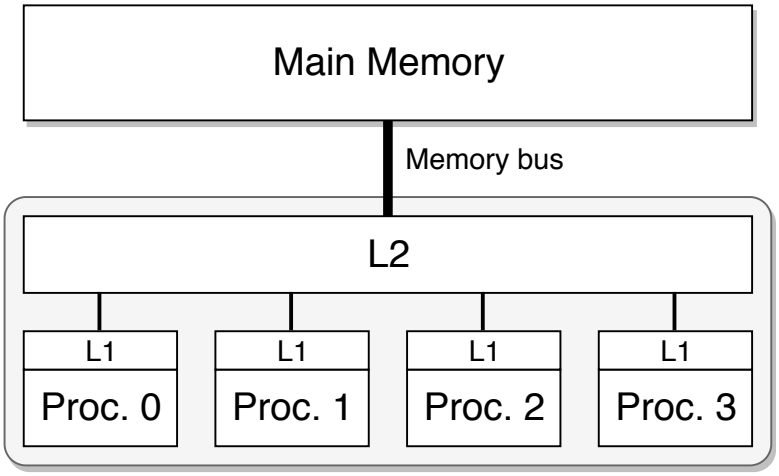


Figure 2.12: Example of a memory scheme

access the *main memory* through the *memory bus*. As there is only one *memory bus* for those processors, the access to the *main memory* is very limited. Figure 2.12 illustrates this categorisation.

The handling of both *L2-memory* and the access to the *memory bus* is out of scope of this thesis. Because some data are stored in the *local cache*, accessible from only one processor, if those data have to be accessed from another processor, they have to be transferred to another memory block. This may be the case if a job starts its execution on a given processor, and continues it on another one. The fact that the state of the job has to be restored on another processor is called a *context switch*. It is very costly in terms of time, and must therefore be limited.

A job	J_i
Job J_i WCET	c_i
Job J_i deadline	d_i
Job J_i arrival time	a_i
Task τ_i WCET	C_i
Task τ_i relative deadline	D_i
Minimum inter-arrival time of τ_i	T_i
The task set	Γ
Number of tasks in a task set	n

(a) Task notations

A processor	π_j
The platform	Π
Number of proc. in Π	m
Type of a proc.	π^k
Number of types	ϕ
Set of proc. types	Ψ
Rate of proc. π_j for a task τ_i	$R_{i,j}$
Utilisation of τ_i on π_j	$U_{i,j}$
Set of configuration sets	Θ
A π^k configuration	θ_c
Number of configurations	o
Reconfiguration delay of θ_c	δ_c
Progression rate for J_i on θ_c	$R_{i,c}$

(b) Platform notations

Set of clusters	$\dot{\Pi}$
A cluster	$\dot{\pi}_h$
A core of cluster h	π_{h_1}
Number of clusters	$\dot{m}$
Number of cores in cluster h	$\dot{m}_h$

(c) Cluster notations

Set of modes	M
A mode	M^q
Number of modes	μ
Task set of a mode M^q	T^q
Number of tasks in T^q	n^q
Sets of r.t. constraints	Δ
Mode M^q's r.t. constraints	Δ^q

(d) Multi-mode notations

is equal by definition to	$\doteq$
A given instant	t
A scheduler	S

(e) Misc. notations

Figure 2.13: Notation summary

Chapter 3

Motivation and organisation

3.1 Motivation

Until recently the efficiency of a single core processor had increased in a predictable way, following Moore's law. However, due to physical limits, uniprocessor performance cannot increase in a cost-effective way. The raise of the multi-processor paradigm follows. According to Amdhal's law, this paradigm is limited as well. To make the most out of their processors, the versatile hardware systems contain different types of processors, with some of them being reconfigurable — at both design-time and run-time. In the future, we expect hardware to become more and more reconfigurable, with the raise of 3D integration. 3D integration could allow implementation of memory on FPGA for extremely fast system reconfiguration. This new paradigm allows faster reconfigurations, and with a higher parallelism. Hard real-time scheduling on those modern architectures raises new challenges. The challenges come from the complexity of the platforms combined with the need for predictability for hard real-time scheduling.

What are the existing scheduling solutions for heterogeneous architectures? Are those techniques effective when used with modern and even future architectures? Do they take advantages of every aspect of them, including the run-time reconfiguration? In this thesis, we will focus on the scheduling aspects on the processing elements only. The objective is to find new solutions that can later be derived with new considerations, such as memory or energy consumption. Those two are thus out of the scope of this thesis. To reach new solutions, we will divide the whole into two sub-problems. First of all, we will explore the question of *global scheduling* on such architecture. We will

then examine the options to take advantage of the run-time reconfigurations.

3.2 Related work

Research concerning reconfigurable platforms combining CPUs and FPGA elements used for real-time systems is relatively new, as the platforms themselves are new to the market [7]. Cornil et al. [8] assess the research challenges to face with this kind of platforms. Ahmad et al. [9] provide tools to optimise the design of real-time applications running on reconfigurable devices (with regards to different metrics such as performance and energy consumption). Pagani et al. [10] propose the integration of Dynamic Partial Reconfiguration (DPR, a technique to reconfigure an FPGA at run-time) as part of a provided service of operating systems. Biondi et al. [11, 12] provide several timing analyses and run-time framework works that make use of DPR, enabling reconfigurable heterogeneous platforms as target candidates for real-time systems. They also provide an extensive state of the art as part of their research paper [11]. Their approach is based on modelling the dependency between heterogeneous components with self-suspending tasks, waiting for resources to free remote processing units. Later, Pagani et al. [13] provide an implementation of their DPR framework for the Linux operating system. Bini [14] presents the Adaptive Fair Scheduler technique, that considers resource allocation and provides guarantees to the application. His approach is general enough to be applied to heterogeneous and reconfigurable computing.

Heterogeneous platforms may be used for hard real-time scheduling, with different paradigms. In operational research, a pioneer work on the scheduling of jobs on unrelated multiprocessor platforms was proposed in [15]. Partitioned scheduling on heterogeneous platforms is a NP-hard problem and has been studied in several works [16, 17, 18]. Global scheduling on heterogeneous platforms, also known as unrelated multi-processor platforms, was initiated by the seminal paper [19]. Since then, the global scheduling on unrelated platforms has received less attention. This may be due to the fact that hardware platforms generally do not support inter-cluster migration of tasks, that may require a full software support. However, global scheduling allows theoretically a full utilisation of the platform. Moreover, optimal global scheduling (regarding schedulability) can be solved in polynomial time. In the literature, e.g. in [19, 20], the global scheduling of unrelated platforms is performed in two phases. First, a workload assignment matrix is computed. The workload assignment decides what fraction of processing capacity of a core has to be assigned to each task. Secondly,

given this workload assignment, a schedule is built. MPSoCs with unrelated clusters sharing the same ISA, like the ARM big.LITTLE® architecture, have motivated some work [20] on the optimal global scheduling. Indeed, sharing the same ISA makes the inter-cluster migrations more realistic. In the latter work, authors adopt a novel strategy, taking into account the hierarchical nature of the set of clusters. They first focus on the assignment of tasks to clusters, and then on cores, which limits the number of inter-cluster migrations. Nevertheless, this method, called Hetero-Split, is limited to a platform with only *two types* of clusters. These two-types platforms also motivates clustered approach with intra-migration like in [21]. New platforms, integrating more than two types of clusters like the Mediatek Helio X20® are developed. This MPSoC includes three clusters (two fast Cortex-A72® cores, four middle speed Cortex-A53® cores and four slow Cortex-A53® cores) sharing the same ISA with a hardware support for inter-cluster migration. More recently, a heuristic to schedule periodic tasks on unrelated multiprocessor platforms has been proposed in [22], but it is considering each job in a hyperperiod (the least common multiple of the task periods), which has in general an exponential complexity. This revives interest in the global scheduling of unrelated clusters.

Reconfigurable platforms may also be used for multi-mode applications. Multi-mode applications have been widely studied in the literature, for uni- and multi-processor systems. A survey [6] proposes various solutions and (re-)defines the main vocabulary and concepts for multi-mode applications on *uni*-processor systems. Concerning *multi*-processors, the literature reports several multi-mode protocols which handle the transition from one mode to another. As for multi-processors scheduling, protocol may be separated whether the schedule use for each task subset is partitioned, semi-partitioned, global or clustered. Cluster-based scheduling —where tasks are assigned to a given set of processors called *cluster* and cannot migrate to a different one, has been well-studied for heterogeneous systems. Raravi et al. [23] propose, to the best of our knowledge, the most efficient approach to this problem. Regarding multi-mode application with *global* scheduling, V. Nelis' works include several protocols for *homogeneous* or *heterogeneous uniform* multi-processors. This is the case in [3, 4]. Those articles introduce two protocols without periodicity, one being synchronous and the other one asynchronous. [24] proposes an analysis for mode changes using Global EDF to schedule a set of mode independent tasks. More recently, Shih et al. [25] provide a schedulability analysis for global scheduling of mode change for the imprecise computation model upon identical multi-processors, a paradigm where a task

is divided into a mandatory subtask and one or several optional subtasks. Concerning *partitioned* scheduling a short contribution by Marinho et al. [26] formalises the scheduling problem and shows two counter-intuitive phenomena. Emberson et al. [27] propose heuristics to handle the mode change. Lastly, Goossens et al. [28] consider the partitioned scheduling problem of multi-mode real-time systems upon *identical* multi-processors. The authors propose two methods for handling mode changes in partitioned scheduling.

3.3 Outline of the thesis

In this thesis, we are going to answer the questions raised as follows.

Chapters 4–9 propose solutions to use the whole platform, by using a global scheduling approach. This approach was already used for heterogeneous unrelated platform, and we here improve the state of the art. In those chapters, we divide the algorithm into steps and then improve each one of them. Our improvements are two fold: first of all, we propose a change of paradigm to produce more realistic schedules. We also correct some flaws from the literature. Our solutions contain refinement, optimisations, but also variations that lead to future works.

Chapters 10–14 propose a new paradigm that exploits the reconfigurability of the modern platforms. This new paradigm combines the hardware reconfiguration with the multi-mode software literature. This new paradigm lessens the overall hardware requirements and adjusts it overtime, which directly reduces weight, cost and energy consumption. We first propose a new model for this new paradigm. We then propose a first protocol to use that model, and study it thoroughly. The study of the protocol aim at setting a *competitor* in terms of performance, pessimism and time-complexity. We then propose a more advanced protocol, and compare its characteristics to the *competitor*. We show that this new protocol outperforms the first one, and we propose several improvement directions for further research.

Part IV concludes the thesis. It sums up the results presented in this thesis, and presents the different ways of improvement.

Part II

Global scheduling

Table of Contents

Chapter 4

Introduction to Global scheduling on heterogeneous unrelated platform

4.1 Motivation

In the two last decades, the chips market has been very active in developing heterogeneous multi-processor system-on-chip platforms (MPSoCs). These heterogeneous MPSoCs are widely used in everyday embedded systems, from the smartphones to infotainment processors in the automotive domain.

In this part, we propose new techniques to take advantages of those modern platforms, with a global scheduling approach. As shown in Section 3.2, the literature offers very few techniques to tackle those platforms. The existing techniques offer very few applicability due to heavy online overheads. Moreover, we will show that some of them are flawed. We will correct those flaws and propose a new paradigm. This new paradigm aims at reducing the online overheads by capturing more accurately the nature of the modern architecture. Doing so improves the applicability of the scheduling techniques.

4.2 Seminal model

This section summarises the basic model used through this part. It is decomposed of two parts: the *Task model* and the *Platform model*. This model is illustrated with a *guideline example* that will be also used in the next section as well.

Task model The workload is modelled by a set of *n periodic tasks* $\Gamma \doteq \{\tau_1, \tau_2, \dots, \tau_n\}$. Each *task* is defined by two parameters (C_i, T_i) where C_i is the *worst-case execution time* —on the same *fictional processor* for every task—, and T_i is the release *period*: tasks are said to have implicit deadlines. Formally, $\forall i,\ T_i = D_i$. As the tasks are periodic, each task releases a *job* every period T_i. The first job of a task is released at $t = 0$, the k^{th} at $k \times T_i$ and has to complete at or before $(k+1) \times T_i$. A job may *be preempted* or *migrated* during its lifespan from a processor to another, with no time penalty.

Guideline example. In this example, we have a task set composed of two tasks τ_1 and τ_2 with following parameters:

τ_i	C_i	$T_i = D_i$	U_i
τ_1	4	2	2
τ_2	3	1	3

Platform model In this part, we consider *unrelated multi-processor platforms*. An *unrelated platform* $\Pi \doteq \{\pi_1, \pi_2, \dots, \pi_m\}$ consists of m processors. In this paradigm, the *processing rate* of a task τ_i depends on the processor π_j where it is being executed and is denoted as $R_{i,j}$. A processor π_j executing a job of τ_i for t time units will process $R_{i,j} \times t$ units of its execution time. If $R_{i,j} = 0$, then τ_i cannot be executed on π_j. This task/processor couple is said to be incompatible.

Guideline example. We have three processors and define the following rates $R_{i,j}$ for the task set:

τ_i	$R_{i,1}$	$R_{i,2}$	$R_{i,3}$
τ_1	1	3	0
τ_2	0	5	1

We observe that τ_1 cannot be executed on π_3 and is executed three times faster on π_2 than on π_1.

4.3 Related works

In this section, we first briefly review the seminal work of Baruah et al. [19, 29] and its context. While the operation research domain considered firstly unrelated multi-

processor scheduling problems (see for instance [15]) the work of Baruah is considered seminal for the scheduling of recurrent real-time tasks. Baruah proposed a two-step method to build an offline schedule pattern. That pattern, called *template schedule*, may be repeated over time to constitute a feasible real-time schedule.

Each step of the algorithm is depicted in Figure 4.1. Starting from the task set specification, the first step called *Workload assignment* is performed. It solves (if possible) a Linear Programming (LP) problem from the input tasks and platform models. Solving the LP problem computes optimally the processor ratios assigned to tasks in polynomial time, forming an assignment matrix. Note that the success of this step ensures the system feasibility. The second step, called *Matching*, is based on the workload assignment matrix to build a template schedule. The main goal is to avoid intra-task parallelism. It is called *matching* because it corresponds to iteratively solving matching problems in a bipartite graph. Finally, this template schedule may be used with usual deadline partitioning and stretching techniques.

After detailing the *Workload assignment* step, we will explain the *Matching* step. The model guideline example will be re-used to illustrate both steps.

We now describe the workload assignment technique of [19]. The technique is divided into several steps. The first step is to split and distribute the execution time of each task over the processors. In Figure 4.1, this step is referenced by *Workload assignment*. It is made offline. The solution is based upon Linear Programming (LP in short) technique. The LP *LP-Feas* is defined which splits the utilisation of each task into one or several portions and assigns them to the processors, with respect to their capacities. Working on the utilisations rather than on WCETs abstracts time constraints such as the deadline of each task.

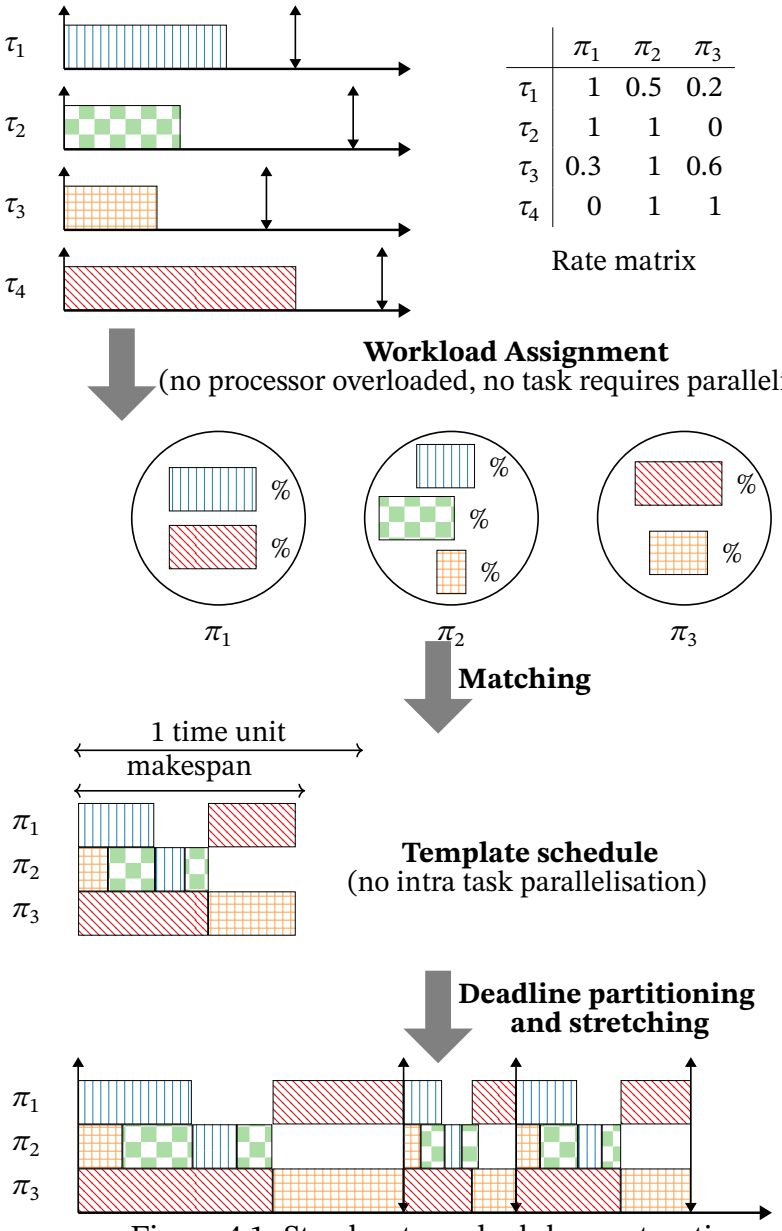
Formally, the LP *LP-Feas*(Γ, Π) is the following:

LP 1 (LP-Feas [19])**.** The workload assignment is solution of the following LP:

$$\sum_{j=1}^{m} x_{i,j} \times R_{i,j} = U_i \qquad i = 1, 2, \dots, n \tag{4.1}$$

$$\sum_{j=1}^{m} x_{i,j} \leq \ell \qquad i = 1, 2, \dots, n \tag{4.2}$$

$$\sum_{i=1}^{n} x_{i,j} \leq \ell \qquad j = 1, 2, \dots, m \tag{4.3}$$



	π_1	π_2	π_3
τ_1	1	0.5	0.2
τ_2	1	1	0
τ_3	0.3	1	0.6
τ_4	0	1	1

Figure 4.1: Step by step schedule construction

Minimise makespan objective: Minimise ℓ, the system is feasible if and only if $\ell \leq 1$.

LP 1 must ensure that each task is fully executed. The value $x_{i,j}$ denotes the ratio of π_j to be assigned to τ_i. In other words, it represents the required amount of execution time of τ_i on processor π_j. Therefore, the sum of every utilisation portion of a given task times the speed of the assigned processors must be equal to the task utilisation (Equation (4.1)). The sum of the $x_{i,j}$ represents the total portion of each time unit

where τ_i will be executed per time unit. ℓ represents the portion of a time unit where processors are busy and then Equation (4.2) avoids the parallel execution of tasks. It is trivial that ℓ must be lower or equal to 1 to avoid deadline misses. The last constraint of Equation (4.3) ensures that processors cannot execute more task workload than their capacity.

Theorem 1 from [19] introduces a major result on the feasibility of a task set on an heterogeneous unrelated platform.

Theorem 4.1 ([19]). The unrelated multi-processor platforms Π feasibly schedules the task system Γ if and only if LP 1 has a solution s.t. $\ell \leq 1$.

The value ℓ is called the *makespan* and the objective function of LP-Feas aims at minimising it. All the processors will be idle in $[\ell, 1)$.

Guideline example. Using our example task set and platform, the following given assignment respects *LP-Feas* constraints:

- $\ell = 1$;
- $x_{1,1} = x_{1,2} = 0.5, x_{1,3} = 0$;
- $x_{2,2} = x_{2,3} = 0.5, x_{2,1} = 0$.

Therefore, the processor π_2 will be equally shared on both tasks. The remaining workload will be performed by the other processors.

Based on the Workload assignment step, the *Matching* step may be performed. It is shown in [19] that a feasible solution of *LP-Feas* (with $\ell \leq 1$) ensures the possible construction of a template schedule on one unit of time.

Please note that the template schedule is built iteratively *in reverse*. In the following, we use the notion of *full processors* at time t to denote the processors that will be busy in the interval $[0, t)$, with $0 \leq t \leq \ell$. Symmetrically, the *urgent tasks* at time t are the tasks that must be executed continuously in $[0, t)$. Formally, a processor π_j is *full* at t if and only if $\sum_{i=1}^{n} x_{i,j} = t$. A task τ_i is *urgent* at t if and only if $\sum_{j=1}^{m} x_{i,j} = t$. It is important to keep in mind that t is the time relative to the template schedule which starts at $t = \ell$ and ends at $t = 0$. Also, a task (resp. processor) may become urgent

(resp. full) at a given time. By definition, there is at least one urgent task and/or one full processor at $t = \ell$, but there is none for $t > \ell$. At each iteration made at time t, the algorithm assigns a subset of tasks (including *all* the urgent tasks) to a subset of processor (including *all* the full processors) for a duration δ in the newly formed interval $[t - \delta, t)$ in the template schedule. It is important to note that each task is assigned to at most one processor during each iteration. Consequently no intra-parallelism can be created. The duration δ is chosen such that no unassigned task (resp. unassigned processor) can become urgent (resp. full) within this interval, otherwise the schedule would not be feasible. The set of assignments is called a *matching*. The resulting matching is composed of pairs (τ_i, π_j), representing the assignment of the task τ_i on the processor π_j in the interval $[t - \delta, t)$. The computation of the matching proposed in [19, 29] is detailed in Section 6.1. Once the matching is found and δ is computed, time t is decreased by δ and the workload assignment matrix is updated: for every (τ_i, π_j) in the matching, the corresponding $x_{i,j}$ is decreased by δ. Intuitively, δ is the largest value that respects the following constraints: *a)* no task τ_i must be assigned on π_j for more than $x_{i,j}$; *b)* no unassigned task becomes urgent in the interval $[t - \delta, t)$; *c)* no unassigned processor becomes full in the interval $[t - \delta, t)$; *d)* and obviously, by construction $t - \delta \geq 0$.

Iterations are performed starting from $t = \ell$ until $t = 0$, with a new matching and a new δ computed at each iteration. By construction, at time $t = 0$, all the tasks have been fully assigned to processors, with no intra-parallelism.

Guideline example. The following example illustrates the overall template schedule construction. No detail are given here on the matching algorithm on the big picture.

The initial workload assignment $n \times m$ matrix X, based on the previously computed x_{ij} values, is used to start the construction of the template schedule at time $t = \ell = 1$:
$X_{t=\ell=1} = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \end{bmatrix}$.

At time $t = \ell = 1$, $\sum_{j=1}^{m} x_{1,j} = 0.5 + 0.5 = 1$. Therefore, τ_1 is urgent. The same applies to τ_2. Also, we can see that π_2 is full. In order to respect the constraints, τ_1, τ_2 must be assigned to a processor and π_2 must have a task assigned. The matching algorithm matches τ_1 on π_1 and τ_2 on π_2. Once the matching has been computed, we determine δ. Since $x_{1,1} = x_{2,2} = 0.5$, it may be at most 0.5. Also, π_3 becomes full at $t = 0.5$. For those reasons, $\delta = 0.5$. Knowing δ, we now update the workload assignment matrix by

subtracting 0.5 from both $x_{1,1}$ and $x_{2,2}$. The updated matrix is $X_{t=0.5} = \begin{bmatrix} 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix}$.

At $t = 0.5$, both tasks remain urgent and the processor π_2 is still full. The processor π_3 becomes full. Here, the only possibility is to match τ_1 on π_2 and τ_2 on π_3, for a duration $\delta = 0.5$. The updated workload assignment matrix is null, therefore the construction is finished. The resulting template schedule is given on Figure 4.2.

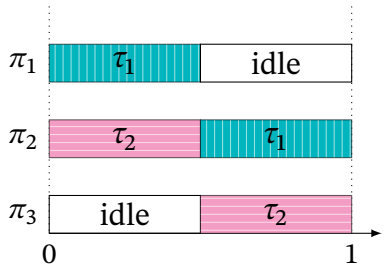


Figure 4.2: Resulting template schedule for the guideline example

Chwa et al. [20] propose a global scheduler for a very limited case of *2*-type unrelated platforms, for task sets having periodic tasks with implicit deadlines. Since there are only *two* types of processors, tasks are classified into two categories: either with a better rate on type 1 or type 2. The method exploits this dual property and thus cannot be extended to more than two types of clusters (at least this is neither straightforward nor easy). Nevertheless, it allows the use of the McNaughton wrap-around rule [30] to efficiently create a schedule conforming to the workload assignment. That technique provides good overall metrics, and we will give more details in Chapter 7. It will be used as a competitor to our contributions.

4.4 Organisation and contributions

In this part, we tackle the problem of global scheduling for heterogeneous unrelated platforms. As it has been dealt with in the literature, our scheduler will be composed of a workload assignment, followed by a matching algorithm to produce a pattern used at run-time. From the model to the use of the pattern at run-time, we will propose a contribution to every step.

We first emphasise the issue with the seminal model and its limitation for modelling modern architecture. We then propose new workload assignment methods with this new model and compare them to the existing one. After that, we work on the matching algorithm. We show that the seminal matching algorithm is flawed, and propose a cor-

rection. We then propose alternative methods to the workload assignment, alongside with several optimisations to the pattern construction. At last, we consider the issue of the scheduling of sporadic tasks on heterogeneous unrelated platforms. We show that an existing scheduler, targeting such task sets, is flawed.

This part is organised as follows:

- Chapter 5 proposes solutions optimised for modern platforms:
  - Section 5.1 determines a new platform model;
  - Section 5.2 shows that this new model is compatible with the workload assignment methods;
  - Section 5.3 presents an adaption of the seminal linear program, compatible with the new model;
  - Section 5.4 presents two new workload assignment methods, compatible with the new model;
  - Section 5.5 presents an optimal workload assignment method, with an exponential complexity;
  - Section 5.6 evaluates and compares the different methods;
- Chapter 6 explores the existing construction step of the algorithm:
  - Section 6.1 details the seminal algorithm from [19, 29];
  - Section 6.2 shows that this seminal construction algorithm is flawed;
  - Section 6.3 proposes a new algorithm, correcting the seminal construction algorithm;
  - Section 6.4 proves the correctness of this new construction algorithm;
- Chapter 7 proposes an alternative construction step and an optimisation:
  - Section 7.1 introduces a new construction method;
  - Section 7.2 empirically evaluates the new construction method;
  - Section 7.3 proposes a post-construction optimisation method,
  - Section 7.4 empirically evaluates the efficiency of the optimisation:
- Chapter 8 explores another algorithm, from [29]:

- Section 8.1 presents the seminal scheduler;
- Section 8.2 proposes a counter example for the seminal scheduler.

Chapter 5

Workload assignment

We propose in this chapter a *new* platform model which emphases the heterogeneity of the platforms.

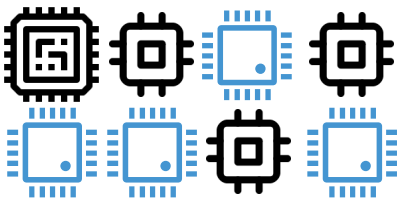
We then formulate the cluster workload assignment as an LP and show that it provides an *exact* feasibility test. Using the new model, we propose several LPs to improve the performance of the scheduler in terms of online overhead. Finally, we present the experimentation conducted with the different proposed solutions.

5.1 New model

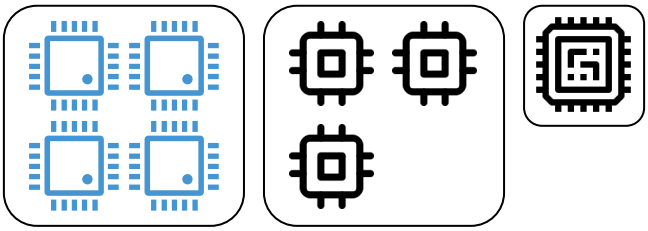
We introduce here a new platform model that takes the hierarchy of the platform into account. This model is based on the clustered approach introduced in Section 2.3.4.

5.1.1 Motivation

In the literature, a platform is often viewed as "flat", as represented in Figure 5.1(a). This lack of hierarchy between cores prevents any modelling of the potential migration costs. This is an abstraction since most modern platforms are composed of one or several clusters of cores, as represented in Figure 5.1(b). The cores of a cluster are identical, but may differ from one cluster to another in the case of *unrelated clusters*. In this part, the jobs are executed on a computing platform of *unrelated clusters*. Each cluster is characterised by its number of *identical* cores, and each task has a specific processing rate on each cluster.



(a) Flat platform model



(b) Clustered platform model

Figure 5.1: Illustration of flat versus clustered platform model

The scheduler on a multi-processor platform can be *global* or *partitioned*. As said before, in *global* scheduling, any job may be executed on any core, i.e. migrate without restriction. By contrast, in *partitioned* scheduling each task is assigned to a single core and neither task nor job migration are allowed. The multi-core cluster model allows for an intermediary category: in *clustered* scheduling [31] each task is assigned to a single cluster and jobs can only migrate between cores within the cluster. In this part, we assume a *global* scheduling of heterogeneous *unrelated* multi-core platforms. The migrations between cores of the *same* cluster are defined as *intra*-cluster migrations while *inter*-cluster migrations correspond to migrations between cores of *different* clusters. On most platforms, inter-cluster migrations require software support and a specific development effort which is very costly. Today, popular scheduler implementations support symmetrical multiprocessing (SMP) that allows intra-cluster migrations (e.g. the Completely Fair Scheduler (CFS) of the Linux kernel). They are therefore transparent to the application developer, and the online overhead generated is smaller than the *inter-cluster migration* one.

5.1.2 Empirical measurements

Disclaimer: These experiments have been performed by Roy Jamil, Ph.D. student at ENSMA and engineer at Ac6.

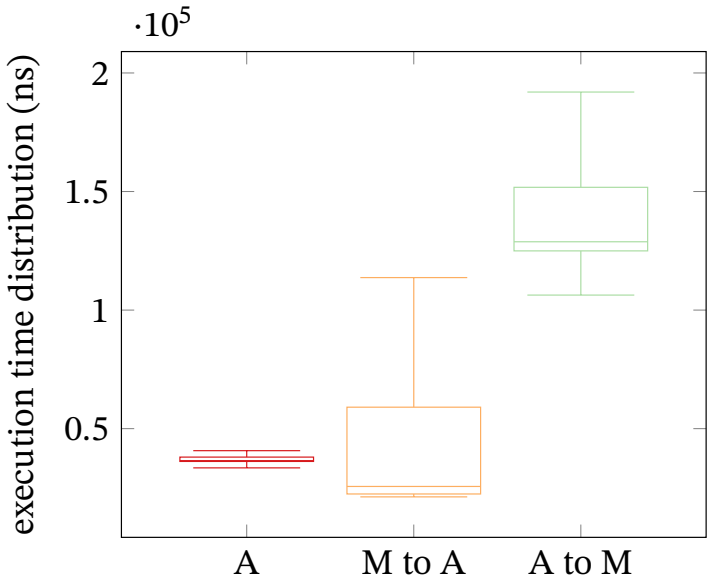


Figure 5.2: Effect of migrations on execution time distribution

In this section, we investigate the costs of inter-cluster migration on a real platform. In theory, assuming a zero-cost migration is however mandatory to find a polynomial time feasibility test. Indeed, feasibility is NP-hard in the strong sense for a single-core processor platform as soon as preemption delay is taken into account [32, 33].

We experimentally measured several program execution times on a STM32MP157C-DK2® platform. On the dual ARM Cortex-A® 650 MHz core cluster, we deployed a Linux stm32mp1 4.19.10-rt8 Operating System (OS) with PREEMPT-RT patch, while the 209 MHz Cortex-M core is used without any OS. We measured fifty thousands times how long a migration transferring 512 bytes of data took *(a)* inside the Cortex-A, *(b)* from the Cortex-M to a Cortex-A , and *(c)* from a Cortex-A to the Cortex-M.

The results are depicted in Figure 5.2. We observe that inter-cluster migration, especially from the A cluster to the M cluster, takes significantly longer than intra-cluster migration within the A cluster. This is due to the communication cost between the clusters. This shows why inter-cluster migrations must be avoided.

Depending on the platforms, they may also be unpredictable and produce a high latency.

5.1.3 Model

An unrelated multi-core platform is modelled by a set $\dot{\Pi}$ of $\dot{m}$ clusters $\dot{\Pi} \doteq \{\dot{\pi}_h \mid h = 1, \dots, \dot{m}\}$. Each cluster $\dot{\pi}_h$ contains $\dot{m}_h$ identical cores $\dot{\pi}_h \doteq \{\pi_{h_1}, \dots, \pi_{h_{\dot{m}_h}}\}$. A job of τ_i that is executed on a core π_{h_k} for t time units will progress by $\dot{r}_{i,h} \times t$ units of its execution time. Within the cluster $\dot{\pi}_h$, every core has the same processing rate $\dot{r}_{i,h}$ for each task τ_i. If $\dot{r}_{i,h} = 0$, then τ_i cannot be executed on the cluster $\dot{\pi}_h$, this couple task/cluster is said to be incompatible. A job of τ_i is completed when its progress reaches its WCET C_i. The jobs are preemptible, i.e. a job being executed may be interrupted at anytime and resumed later. The tasks are sequential so they cannot be executed in parallel. A task set is *feasible* on a given platform if and only if there exists a schedule where every job of every task can be completed by its deadline.

Please note that the notion of processor has been replaced by the notion of cluster of cores. In this new model, a processor would correspond to a single core.

5.2 Designing new LPs

In this chapter we revisit the workload assignment phase dedicated to our clustered platform model. In particular we aim to minimise the *inter*-cluster migrations. As far as we know, every optimal scheduling method of the literature [17] for unrelated multi-processor platforms (from real-time [19] or operation research [15] areas), starts with a workload assignment phase. From an input made of tasks parameters and platform rates, this phase decides the fraction of processing capacity of each core assigned to tasks. The tasks have to be completed within their period thanks to this assignment, without overloading the cores. With the exception of [20], introduced briefly in Section 4.3, most of the existing works have expressed the workload assignment phase as an LP (which can be solved in polynomial time [34]).

The solution of the LP is a *cluster workload assignment* matrix $X = [x_{i,h}]_{i=1,\dots,n}^{h=1,\dots,\dot{m}}$ where $x_{i,h}$ is the fraction of a core in the cluster $\dot{\pi}_h$ used by a task τ_i.

It is a well-known fact that inter-cluster migrations are more costly in terms of time overhead and task programming effort. We quantify the impact of such migrations by the definition of the *presence* of a task on a cluster, introduced in [19].

Definition 5.1 (Presence)**.** Formally, a task τ_i has a *presence* on a cluster $\dot{\pi}_h$ if and only if $x_{i,h} > 0$. The *number of presences* $\dot{\Pr}_i$ corresponds to the number of different

clusters on which task τ_i is assigned: $\dot{\Pr}_i \doteq |\{x_{i,h} > 0 \mid h = 1, \ldots, \dot{m}\}|$

A task τ_i will have to migrate between clusters if and only if $\dot{\Pr}_i > 1$. Therefore, any presence greater than one is a *presence in excess* that will generate at least one inter-cluster migration per pattern repetition.

Assigning the workload of tasks on clusters can be expressed using three sets of constraints, defined in CS-Cluster:

Constraint Set 1 (CS-Cluster)**.**

$$\sum_{h=1}^{\dot{m}} x_{i,h} \times \dot{r}_{i,h} = u_i \qquad i = 1, 2, \ldots, n \tag{5.1}$$

$$\sum_{h=1}^{\dot{m}} x_{i,h} \leq 1 \qquad i = 1, 2, \ldots, n \tag{5.2}$$

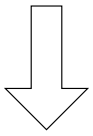
$$\sum_{i=1}^{n} x_{i,h} \leq \dot{m}_h \qquad h = 1, 2, \ldots, \dot{m} \tag{5.3}$$

Equation 5.1 ensures that enough processing capacity is allocated to each task by reserving a processing capacity fraction on each cluster. Equation 5.2 constrains the total capacity fraction allocated to a task to be less than one. This ensures that the task can be scheduled without being executed on two cores at the same time (see Theorem 5.1). Equation 5.3 states that the used capacity of a cluster $\dot{\pi}_h$ is less than or equal to its total capacity, which is the sum of the capacities of its $\dot{m}_h$ cores. Please note that CS 1 is not an LP, but a set of constraints (we will present an actual LP for our new model later in this chapter). Also, regarding the constraints themselves there is a slight difference compared to the seminal LP (LP 1), in particular regarding Equation 4.2 and Equation 5.2. CS 1 guarantees *feasibility* only while LP 1 guarantees feasibility *and makespan minimisation*.

If the CS 1 is feasible then $x_{i,h}$ represent a feasible cluster workload assignment (or assignment of tasks on clusters), as stated in Theorem 5.1. To construct a template schedule, our method requires a core workload assignment. To obtain it, we derive the cluster workload assignment by applying a First-Fit strategy. The result is a successful core workload assignment that we will be able to use to construct a template schedule.

Theorem 5.1. The unrelated multi-core platform $\dot{\Pi}$ feasibly schedules the task system Γ if and only if CS 1 has a solution.

$$
X = \begin{array}{c} \\ \tau_1 \\ \vdots \\ \vdots \\ \tau_n \\ \\ \end{array}
\begin{array}{c}
\begin{array}{ccc} \dot{\pi}_1 & \cdots & \dot{\pi}_{\dot{m}} \end{array} \\
\begin{pmatrix} x_{1,1} & \dots & x_{1,\dot{m}} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ x_{n,1} & \dots & x_{n,\dot{m}} \end{pmatrix}
\begin{array}{c} \leq 1 \\ \vdots \\ \vdots \\ \leq 1 \end{array} \\
\begin{array}{ccc} \leq \dot{m}_1 & \cdots & \leq \dot{m}_{\dot{m}} \end{array}
\end{array}
$$



Cluster to cores extension

$$
X_c = \begin{array}{c} \\ \tau_1 \\ \vdots \\ \vdots \\ \tau_n \\ \\ \end{array}
\begin{array}{c}
\begin{array}{cccccccccc} \pi_{1_1} & \dots & \pi_{1_{\dot{m}_1}} & \pi_{2_1} & \dots & \pi_{2_{\dot{m}_2}} & \cdots & \cdots & \pi_{\dot{m}_{\dot{m}_{\dot{m}}}} \end{array} \\
\begin{pmatrix} x_{1,1}/\dot{m}_1 & \dots & x_{1,1}/\dot{m}_1 & x_{1,2}/\dot{m}_2 & \dots & x_{1,2}/\dot{m}_2 & \cdots & \cdots & x_{1,\dot{m}}/\dot{m}_{\dot{m}} \\ \vdots & & \vdots & \vdots & & \vdots & & & \vdots \\ \vdots & & \vdots & \vdots & & \vdots & & & \vdots \\ x_{n,1}/\dot{m}_1 & \dots & x_{n,1}/\dot{m}_1 & x_{n,2}/\dot{m}_2 & \dots & x_{n,2}/\dot{m}_2 & \cdots & \cdots & x_{n,\dot{m}}/\dot{m}_{\dot{m}} \end{pmatrix}
\begin{array}{c} \leq 1 \\ \vdots \\ \vdots \\ \leq 1 \end{array} \\
\begin{array}{cccccccccc} \leq 1 & \dots & \cdots & \cdots & \dots & \cdots & \cdots & \cdots & \leq 1 \end{array}
\end{array}
$$

Figure 5.3: Proof sketch for Theorem 5.1

Proof. First we prove that (i) if there is no solution to the CS 1, then the system is not feasible. This will occur if Equation 5.2 or Equation 5.3 are not satisfied. In the first case, there would be at least one task τ_i such that $\sum_{h=1}^{\dot{m}} x_{i,h} > 1$. It means that τ_i must be executed in parallel which is forbidden in our model of sequential tasks. In the second case, a cluster $\dot{\pi}_h$ would need a processing capacity higher than its total capacity $\dot{m}_h$.

Now we prove that (ii) finding a solution to this CS 1 problem guarantees that the system is feasible. The proof sketch is depicted in Figure 5.3. The cluster workload assignment matrix X is of dimension $n \times \dot{m}$. Indeed, by construction of the LP problem, it has n rows with a sum of coefficients less than one, and $\dot{m}$ columns with a sum of coefficients less than $\dot{m}_h, h = 1, \dots, \dot{m}$. First, we replace each column h, corresponding to the task assignment to cluster $\dot{\pi}_h$ by $\dot{m}_h$ columns, one for each core, such that the sum of the coefficients on each of the columns is not greater than one. For the sake of the proof, we simply consider, on each row i of the new columns $k = 1, \dots, \dot{m}_h$, $x'_{i,h_k} \doteq \frac{x_{i,h}}{\dot{m}_h}$,

such that the total capacity fraction allocated to each task on each cluster is evenly distributed on each of its cores. In this manner, we obtain a workload assignment matrix on the cores X_c of dimension $n \times M$, where $M \doteq \sum_{h=1}^{\dot{m}} \dot{m}_h$ is the total number of cores. On each column, $x'_{i,h_k} \doteq \frac{x_{i,h}}{\dot{m}_h}$ represents the capacity fraction of core π_{h_k} allocated to task τ_i. By construction, since originally the used capacity of cluster $\dot{\pi}_h$ to tasks was $\sum_{i=1}^{n} x_{i,h} \leq \dot{m}_h$, we have on each column for π_{j_k}, $\sum_{i=1}^{n} x'_{i,h_k} \leq 1$. From this *core workload assignment matrix*, we can easily create a bistochastic matrix B of size $(n+M)\times(n+M)$, as done in [15]. A *bistochastic* (or doubly stochastic) matrix is a square matrix of non-negative real numbers, having each of its rows and columns summing to 1. Formally, $\forall i = 1, \dots, n+M : \sum_{h=1}^{n+M} B_{i,h} = 1$ and $\forall h = 1, \dots, n+M : \sum_{i=1}^{n+M} B_{i,h} = 1$. B is constructed as follows:

$$B \doteq \left(\begin{array}{c|c} X_c & B_n \\ \hline B_M & X_c^t \end{array}\right)$$

B_n is a $n \times n$ diagonal matrix, such that $B_n(i,i) \doteq 1 - \sum_{h=1}^{\dot{m}} \sum_{k=1}^{\dot{m}_h} x'_{i,h_k} \; \forall i$. The diagonal coefficients of B_n correspond to the *laxity* of the task τ_i, i.e. the fraction of time during which τ_i is left idle. B_M is a $M \times M$ diagonal matrix, such that $B_M(h_k, h_k) \doteq 1 - \sum_{i=1}^{n} x'_{i,h_k} \forall h_k$. The diagonal coefficients of B_M correspond to the *slack* of the core π_{h_k}, i.e. the fraction of time during which π_{h_k} is left idle. X_c^t is the transpose of the *core workload assignment matrix* X_c, and has a dimension $M \times n$. By construction, we obtain a square bistochastic matrix B of dimension $(n+M) \times (n+M)$ expressing the fraction of each core that has to be allocated to each task, as well as the slack of the cores and the laxity of the tasks. Following the Birkhoff-von Neumann (BvN) theorem, such a matrix can be decomposed into a convex combination of permutation matrices $A \doteq \delta_1 P_1 + \delta_2 P_2 + \cdots + \delta_k P_k$ [15], where δ_i is a real coefficient $\in (0,1]$, $\sum_{i=1}^{k} \delta_i = 1$, and P_i is a permutation matrix. A *permutation matrix* is a binary square matrix where there is exactly one 1 on each row and each column. This can be seen as a matching between tasks (rows) and cores (columns). Indeed, one and only one coefficient $P_i(h,k) = 1$ means that task on column k will be assigned to the core of the row h for a duration δ_i. The assignment matrix X_c states that assigning a ratio of x'_{i,h_k} of core π_{h_k} to task τ_i during each of its periods ensures that its jobs will be completed. However, we need to ensure that a job is never executed on two different clusters at the same time.

For each time window $[t_1, t_k)$, between two successive releases at times t_1 and t_k (or

deadlines since tasks have implicit deadlines), we can use the BvN decomposition to create such a schedule. We use the matching P_1 on the time window $[t_1, \delta_1 \times (t_k - t_1))$, by definition of a permutation matrix, this matching ensures that a task is assigned to at most one core in this time windows. Similarly, we can use the following permutation matrices obtained in the BvN decomposition, each permutation matrix P_i covering a sub-interval of duration $\delta_i \times (t_k - t_1)$. Since by the BvN theorem, $\sum_{i=1}^{k} \delta_i = 1$, we can completely schedule every task on the interval $[t_1, t_k)$, ensuring that a task is never executed on more than one core at the same time. This one time unit schedule can then be *stretched* to fit into intervals of time delimited by successive task release dates. This technique is also referred to as deadline partitioning. □

Theorem 5.1 shows that CS 1 feasibility is necessary and sufficient to prove the feasibility of the system. Moreover, the proof of Theorem 5.1 shows that building a schedule from a workload assignment matrix is exactly equivalent to finding a Birkhoff-von Neumann (BvN) decomposition of this matrix. This result indicates that linear algebra results could be used to improve the schedule construction.

One may note that minimising the number of permutation matrices in a BvN decomposition is similar as minimising the number of scheduling decisions. Indeed, each different permutation matrix corresponds to a different schedule decision (i.e. which jobs are executed at a given instant, and on which cores). Taking schedule decisions leads to preemptions and/or migrations (both inter- or intra-cluster). Therefore, minimising the number of scheduling points may be a solution to reduce the number of preemptions and migrations. This is an example of optimisation of the template schedule construction [19, 20, 35].

From [36], we know the complexity of such a problem through Theorem 5.2:

Theorem 5.2 (Dufossé 2016 [36])**.** The problem of deciding whether there is a BvN decomposition of a given doubly stochastic matrix with k permutation matrices is NP-complete in the strong sense.

Since the decision problem is NP-complete in the strong sense, the optimisation problem of minimising the number of permutation matrices in a BvN decomposition is NP-hard in the strong sense. Thus, optimising the number of scheduling decisions cannot be done efficiently.

In the remainder, we focus only on modifying the workload assignment to reduce the number of preemptions and migrations. However, using linear algebra techniques to sub-optimally reduce the number of scheduling decisions can be explored.

5.3 LP-Feas and LP-CFeas

In [19], the author presents LP-Feas, an LP model for assigning the workload on an unrelated real-time multi-processor platform. This work was primarily focused on feasibility, and does not aim at minimising the number of presences. It is very close to the LP formulation of the makespan minimisation in job shop scheduling on unrelated single-core processors given in [15]. In that work, the model is using a *flat* hardware representation. To fit our model notations, we consider a hierarchical hardware with one core per cluster, i.e. $\forall h, \dot{m}_h = 1$.

LP 2 (LP-Feas [19])**.** The workload assignment is solution of the following LP:

$$\sum_{h=1}^{\dot{m}} x_{i,h} \times \dot{r}_{i,h} = u_i \qquad i = 1, 2, \dots, n \tag{5.4}$$

$$\sum_{h=1}^{\dot{m}} x_{i,h} \leq \ell \qquad i = 1, 2, \dots, n \tag{5.5}$$

$$\sum_{i=1}^{n} x_{i,h} \leq \ell \qquad h = 1, 2, \dots, \dot{m} \tag{5.6}$$

Minimise makespan objective: Minimise ℓ, the system is feasible if and only if $\ell \leq 1$.

The immediate extension of LP-Feas to clusters is the following:

LP 3 (LP-CFeas)**.** The workload assignment is solution of the following LP:

$$\sum_{h=1}^{\dot{m}} x_{i,h} \times \dot{r}_{i,h} = u_i \qquad i = 1, 2, \dots, n \tag{5.7}$$

$$\sum_{h=1}^{\dot{m}} x_{i,h} \leq \ell \qquad i = 1, 2, \dots, n \tag{5.8}$$

$$\sum_{i=1}^{n} x_{i,h} \leq \dot{m}_h \times \ell \qquad h = 1, 2, \dots, \dot{m} \tag{5.9}$$

(a) LP-CFeas (b) LP-CLoad

Figure 5.4: Rectangle schedule computed from LP-CFeas versus schedule favouring fast cores utilisation computed from LP-CLoad

Minimise makespan objective: Minimise ℓ, the system is feasible if and only if $\ell \leq 1$.

LP-Feas and LP-CFeas differ in Equations 5.6 and 5.9: since on an unrelated multi-core platform, a cluster $\dot{\pi}_h$ has $\dot{m}_h$ cores, a total capacity of $\dot{m}_h$ can be allocated to tasks. It is straightforward that the condition $\ell \leq 1$ constrains solutions of LP-CFeas to be solutions of CS-Cluster. Therefore by Theorem 5.1, a solution of LP-CFeas with $\ell \leq 1$ can be used to build a feasible schedule.

5.4 LP-Load and LP-CLoad

LP-Feas and LP-CFeas tend to reduce the makespan of the schedule that will be stretched between successive releases. As an example, consider two tasks scheduled on two very different cores: one being ten times faster than the other one for all the tasks. Consider the system of two tasks $\Gamma = \{\tau_1, \tau_2\}$, with both WCET given by $C_1 = C_2 = 5$ and both periods given by $T_1 = T_2 = 10$. The platform is composed of two clusters of one core each, with $\Pi = \{\dot{\pi}_1, \dot{\pi}_2\}$, both clusters having only one core $\dot{m}_1 = \dot{m}_2 = 1$, and having respective rates $\dot{r}_{1,1} = \dot{r}_{2,1} = 10$ for $\dot{\pi}_1$, and $\dot{r}_{1,2} = \dot{r}_{2,2} = 1$ for $\dot{\pi}_2$. The workload assignment matrix computed by LP-CFeas (or equivalently LP-Feas since clusters have one core) is given by $X_{\text{LP-CFeas}} = \begin{bmatrix} 5/11 & 5/11 \\ 5/11 & 5/11 \end{bmatrix}$ ($5/11 \approx 0.4545$). This would lead to a schedule repeated between every successive release (which is every ten time units in our simple example since both tasks have a period of ten), as shown in Figure 5.4(a).

When considering the number of presences of tasks on clusters, a more interesting workload assignment would favour a high utilisation, or load, on faster cores: $X_{\text{LP-CLoad}} = \begin{bmatrix} 1/2 & 1/2 \\ 0 & 0 \end{bmatrix}$. Such workload assignment could lead to a schedule such as Figure 5.4(b), which does not produce any inter-cluster migration. LP-CLoad is an LP

formulation with the same constraints as CS-Cluster, with the objective of minimising the used capacity of the system. On the unrelated multi-core platforms problem, it is defined for CS-Cluster as: ***LP-CLoad:*** Minimise $\sum_{i=1}^{n} \sum_{h=1}^{\dot{m}} x_{i,h}$.
LP-CLoad can be used in the context of a flat platform model. To do so, one simply has to assume that each core is a cluster of size one, i.e. $\dot{m}_h = 1$ for every cluster $\dot{\pi}_h$. LP-Load is an LP formulation with the same constraints as LP-Feas, with the same objective as LPCLoad but for processors instead of clusters: ***LP-Load:*** Minimise $\sum_{i=1}^{n} \sum_{h=1}^{m} x_{i,h}$.

5.5 Minimal number of presences: ILP-CMig

Even if non polynomial, an optimal method minimising the number of presences of tasks on clusters can be useful. Indeed, a system designer may prefer spending a couple of hours waiting for the assignment to be computed rather than spending development time and facing the complexity to implement an inter-cluster migration. Since we are working at the cluster level, the size of the problem, at least in the number of clusters, can be relatively small in practice. We propose a Mixed Integer Linear Programming (*MILP*) formulation called *ILP-CMig*, based on the CS-Cluster. In addition, we introduce a boolean variable $b_{i,h}$. Variable $b_{i,h}$ is 1 if task τ_i is present on cluster $\dot{\pi}_h$, and 0 otherwise. The objective is to minimise the total number of presences.

LP 4 (ILP-CMig)**.** The workload assignment is solution of CS-Cluster (Equations 5.1, 5.2, 5.3) with the following additional constraints:

$$b_{i,h} \in \{0,1\} \qquad i = 1,\dots,n; h = 1,\dots,\dot{m} \tag{5.10}$$

$$x_{i,h} \leq b_{i,h} \qquad i = 1,\dots,n; h = 1,\dots,\dot{m} \tag{5.11}$$

$$b_{i,h} < 1 + x_{i,h} \qquad i = 1,\dots,n; h = 1,\dots,\dot{m} \tag{5.12}$$

Minimise makespan objective: Minimise $\sum_{i=1}^{n} \sum_{h=1}^{\dot{m}} b_{i,h}$.

The non-clustered version ILP-Mig has the same set of constraints than CS-Cluster where each cluster is considered as a single core with the computing capacity of m cores.

5.6 Workload assignment evaluation

When neglecting the migration cost, every workload assignment method presented in this chapter is optimal regarding the feasibility. Since we know that this hypothesis is unrealistic, we compare the number of presences in excess $\dot{\text{Pr}}_i - 1$ for the six presented methods. The number of presences in excess is a lower-bound on the number of inter-cluster migrations. The LP based methods, as well as Hetero-Split, are polynomial time methods, while the MILP based method has an exponential time complexity regarding the number of clusters. In this section, we compare the following methods:

- LP-Feas is the method minimising the makespan proposed in [19] considering the "flat" core model, while its clustered version LP-CFeas presented in Section 5.3 considers the hierarchical clustered model;
- LP-Load (see Section 5.4), whose objective is to minimise the total core utilisation, its clustered version is LP-CLoad;
- Hetero-Split ([20]) a linear algorithm limited to two types of clusters;
- ILP-Mig is the "flat" core-based version of ILP-CMig, a MILP problem minimising the number of presences on clusters. In practice, ILP-CMig uses significantly fewer variables than ILP-Mig.

5.6.1 Experimental setup

In our opinion, the notion of consistent clusters fits more precisely to certain realistic platforms where cores have different micro-architectures but identical ISA, as the big.LITTLE® or the Helio X20®.

For the number of presences and simulation experiments, we have generated the systems as follows. The number of types of clusters $\dot{m}$ is either 2 or 5. The former in order to compare Hetero-Split to the other methods, and the latter because five different types of clusters is considered a large size for a heterogeneous MPSoC nowadays. Then, the number of cores per type of cluster is set in $[2, 5]$. The number of tasks n is arbitrary bounded as follows: $\dot{m} \leq n \leq 10 \times \dot{m}$. We then generate every task such that its period T_i is determined using [37]. The parameter C_i is based on T_i: $\frac{T_i}{2} \leq C_i \leq T_i$. We then generate the rates randomly and adjust them so that the tasks fit the given utilisation. For experimentation purposes, the clusters (the rates in particular) may be set to consistent.

Using this generator, we generate 1 000 systems per total utilisation range $u \in [p - 0.1, p)$, increasing p from 0.4 to 1, for both $\dot{m} = 2$ and $\dot{m} = 5$. The ratio $p = 1$ corresponds to a full utilisation of the platform by the tasks. Here, p is equal to the value of the LP-CFeas objective function result, which is the minimal platform utilisation. The experimentation compares the different scheduling methods over 28 000 randomly generated test systems. As ILP-Mig and ILP-CMig have an exponential time complexity, they are tested using only a subset of the generated systems.

5.6.2 Inter-cluster number of presences in excess

The workload assignment methods are compared in terms of inter-cluster presences in Figure 5.5. First note that the scale is 10^{-2}, meaning that in average, very few tasks are assigned to different clusters, for both two and five types of clusters.

On the top graph, with $\dot{m} = 2$, we observe that Hetero-Split performs close to LP-CFeas for low platform utilisation. At higher platform utilisation, Hetero-Split dominates the other polynomial time assignment methods. We can see that both the *Feas*-based LP solutions perform poorly at low platform utilisation compared to the *Load*-based LP solutions for both two-types and five-types of cores. This is due to *Feas* objective that tends to create "rectangular" (i.e. all processors tend to be idle at the same instant) schedules by balancing the tasks workload on different cores or clusters, as illustrated in Figure 5.4. While the platform utilisation increases, the slack left at the right-hand side of this rectangle reduces, and the solutions provided by both objective functions tend to be similar. At high platform utilisation, we thus see that both clustered versions of the LP outperform both non-clustered versions. When combining the two advantages —both the clustered version and the Load objective function—, we observe two to four times fewer inter-cluster migrations compared to the seminal non-clustered Feas objective function. On bottom graph, we see the proportion of generated systems for which the assignment produce no presence in excess. This means that the here tasks are completely clustered for two types of clusters. It is close to 100% for the ILP-CMig, while the clustered LP-CLoad dominates all the other methods in terms of ratio of completely clustered workload assignments. When comparing both graphs, we can observe that LP-CLoad performs better than Hetero-Split regarding tasks clustering on consistent two-types systems. However, Hetero-split performs better in average on arbitrary clusters.

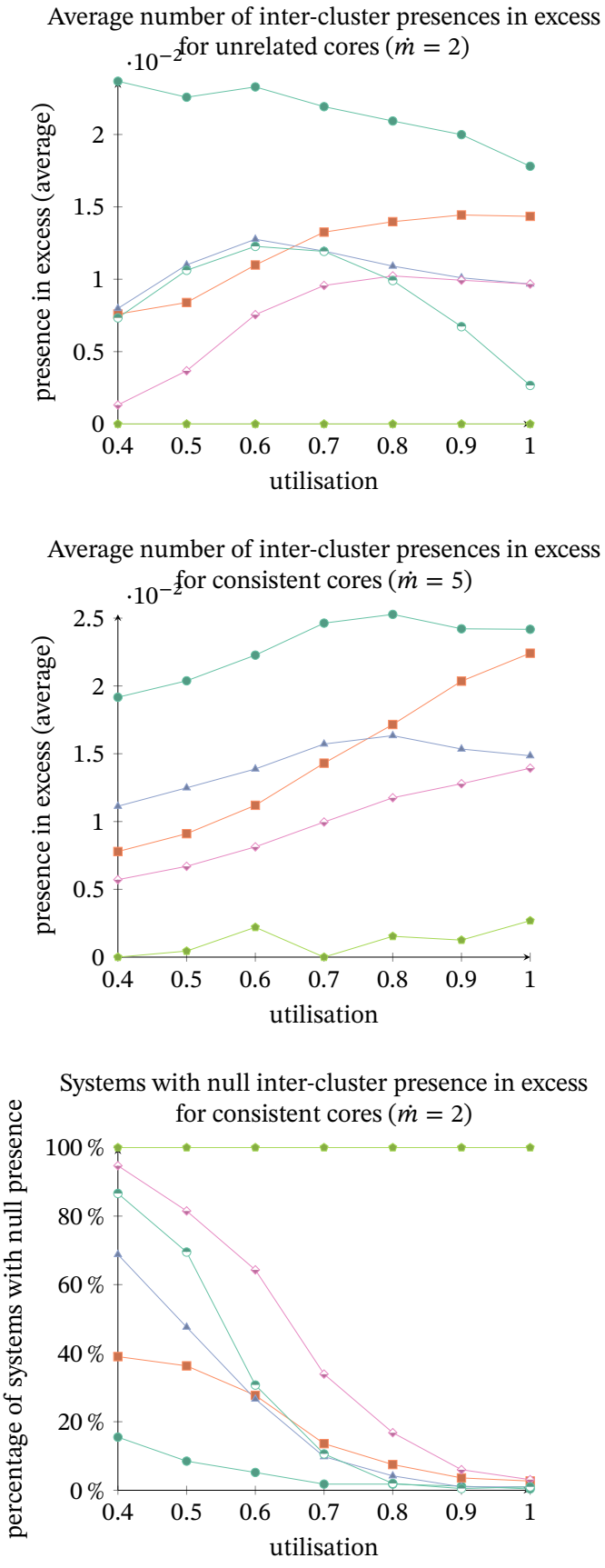


Figure 5.5: Number of presences by workload assignment method for unrelated and consistent clusters

	$\dot{m} = 2$	$\dot{m} = 5$
LP-Feas	0.013	0.464
LP-Load	0.012	0.562
LP-CFeas	0.002	0.027
LP-CLoad	0.002	0.029
Hetero-Split	0.007	NA
ILP-Mig	0.061	NA
ILP-CMig	0.018	0.068

ILP-Mig, $\dot{m} = 2$	
n	time (s)
10	0.811
11	1.736
12	3.562
13	8.271
14	16.665
15	28.492
16	69.782
17	130.582

Table 5.1: Average execution time of the workload assignment methods in seconds

5.6.3 Run-time measurement

The performance of the LP/ILP based solution in terms of execution time is depicted in Table 5.1. The experiment has been conducted on a Intel I7500® multi-core processor. The left table gives the performance of the LP/ILP based solution with the same test systems. In this experiment, the system utilisations are uniformly distributed in the range [0.3, 1.0]. The rest of the system parameters are generated as in Section Section 5.6.1. The table on the right gives the average performance with test systems ordered by number of tasks. Thus, both tables are not comparable because they do not have the same test systems. The left table gives the average computation time, per LP or ILP for both $\dot{m} = 2$ and $\dot{m} = 5$ on unrelated clusters. For example, ILP-Mig took an average of 0.061 seconds to compute the workload assignment with $\dot{m} = 2$. We observe that the clustered version of a LP or an ILP is always faster than the non-clustered version, which can be explained by the fact that there are fewer variables. Also, the execution time from $\dot{m} = 2$ to $\dot{m} = 5$ increases drastically and this affects less the clustered versions, since there are fewer additional cluster variables than core variables. The table on the right gives the performance of ILP-Mig with $\dot{m} = 2$ for n tasks. It clearly shows that the execution time grows exponentially with the number of tasks, making it intractable for large systems.

Chapter 6

Flaw & correction in the schedule construction of [19, 29]

In this chapter, we discuss some of the existing work presented in Chapter 4.

The presented algorithm is based on graph theory. It takes place at each iteration of the template construction, described previously in Section 4.3. The goal is to find an assignment at a given time between the tasks and processors. By definition, each full processor must be assigned to a task, and each urgent task must be assigned to a processor. Also, non-urgent tasks (resp. non-full processors) can be assigned as well, to full processors (resp. to urgent tasks). To compute this assignment, the problem is represented as a bipartite graph $G \doteq (\Gamma \cup \Pi, E \subseteq \Gamma \times \Pi)$, where the partition of vertices Γ (resp. Π) corresponds to the tasks (resp. processors). In our graph representation, there exists an edge between vertices τ_i and π_j if and only if τ_i may be assigned to π_j. Formally, $(\tau_i, \pi_j) \in E$ if and only if $x_{i,j} > 0$. Such bipartite graph for guideline example (introduced in 4.2) at time $t = 1$ is depicted on Figure 6.1. In this figure, edges are represented by dashed lines, urgent tasks and full processors symbolised by square nodes and, non-urgent and non-full processors symbolised by circle nodes.

We first present in details the matching algorithm from [19] and [29]. We then show that it is flawed, and provide a new algorithm correcting it.

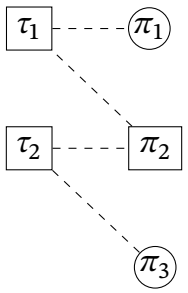


Figure 6.1: Bipartite graph corresponding to the workload assignment matrix at time $t = 1$.

6.1 Seminal algorithm from [19, 29]

In the following, the authors construct an assignment through a matching in the bipartite graph. In this respect, we first recall usual definitions of graph theory that will be used in the remainder of this part.

Definition 6.1 (Degree)**.** The *degree* of a vertex v in a graph is the number of edges that are connected to v.

Definition 6.2 (Walk)**.** A finite *walk* is a sequence of edges $(e_1, e_2, \dots, e_{n-1})$ which connects a sequence of vertices $(v_1, v_2, \dots, v_n)$ such that $e_i = (v_i, v_{i+1})$ for $i = 1, 2, \dots, n-1$.

Definition 6.3 (Trail)**.** A *trail* is a walk where no edge is repeated. A *path* is a trail without repeated vertices. A graph *traversal* is the process of visiting each vertex of a graph. A *connected component* is a sub-graph in which any two vertices are connected to each other by paths.

Definition 6.4 (Matching)**.** A *matching* in a bipartite graph G is a subset of its edges $M \subseteq E$ without common vertices. A vertex v is said to be *saturated* (or matched) by M if and only if an edge in M connects v.

The technique of [19] states that a matching saturating every urgent task and every full processor can always be found. We summarised this result in Theorem 6.1.

Theorem 6.1 (Fact 2 combined with Theorem 2 from [19])**.** Given a bipartite graph built from a workload assignment matrix such that the makespan is not greater than one, i) it is always possible to find a matching that saturates all the urgent tasks and ii) it is always possible to find a matching that saturates all the full processors.

Using Theorem 6.1, the idea is to combine both matchings in order to obtain a matching that saturates both the urgent tasks and the full processors. $\Gamma_u \subseteq \Gamma$ denotes the set of urgent tasks and $\Pi_f \subseteq \Pi$ the set of full processors. The whole matching algorithm is divided in four steps. The first three steps are from [19] and the last step has been added in [29].

1. Determine a matching M_τ from all vertices in Γ_u to a subset of the vertices in Π — by Theorem 6.1, this can always be done.
2. Determine a matching M_π from all vertices in Π_f to a subset of the vertices in Γ — by Theorem 6.1, this can always be done.
3. If an urgent task-vertex (i.e. , one in Γ_u) appears in this second matching as well, then discard the edge that it was matched to in the initial Γ_u-to-Π matching.
4. (From [29]) If a full processor remains matched with two tasks with these remaining edges, then discard the edge matching it with a non-urgent task, and retain only the edge that matches it with an urgent task

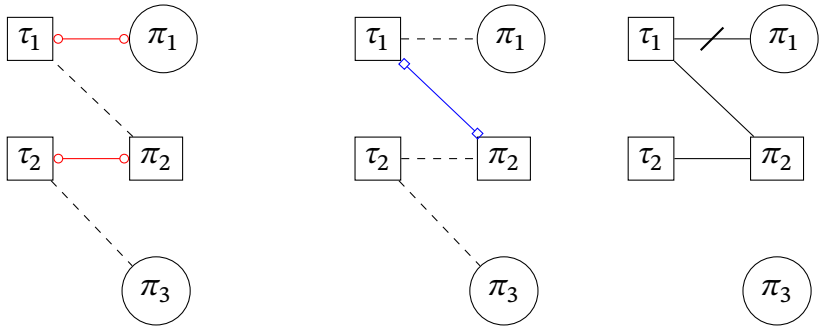
The authors propose the following claim on that algorithm:

Claim 1 (Fact 3 from [19] corresponding to Fact 2 from [29])**.** What remains after application of the algorithm is a matching that satisfies the following properties: each full processor is matched, and each urgent task is matched. (There may be additional matched vertices, corresponding to non-full processors and non-urgent tasks, as well.)

6.2 Counter-example of the seminal algorithm

We now show that the algorithm presented in Section 6.1 is flawed.

This matching algorithm is applied on the guideline example at time $t = 1$. Its result is represented on Figure 6.2. We observe that the first two steps of the algorithm are successfully achieved. The matching M_τ (edges with circle tips in Figure 6.2(a)) saturates all the urgent tasks, while the matching M_π (edges with square tips in Figure 6.2(b)) saturates all the full processors. We now apply step (3) of the algorithm because τ_1 is an urgent task appearing both in M_τ and M_π. Consequently, the edge (τ_1, π_1) is discarded from M_τ (the edge is stroke out in Figure 6.2(c)). The step (4) proposed in [29] is not



(a) Matching M_τ (b) Matching M_π (c) $M_\tau \cup M_\pi$

Figure 6.2: Illustration of the application of the cleaning algorithm in [19, 29]

applicable since we have a full processor matched with two urgent tasks. As a result, π_2 is both paired with τ_1 and τ_2, meaning that it is supposed to be allocated both to τ_1 and τ_2. Consequently, we do not obtain a matching, which contradicts Claim 1.

We have shown that both cleaning phases (steps (3) and (4)) are incomplete and may lead to unfeasible schedules. Therefore, we propose in the following section a new algorithm to perform a correct matching.

6.3 Correction of the matching algorithm

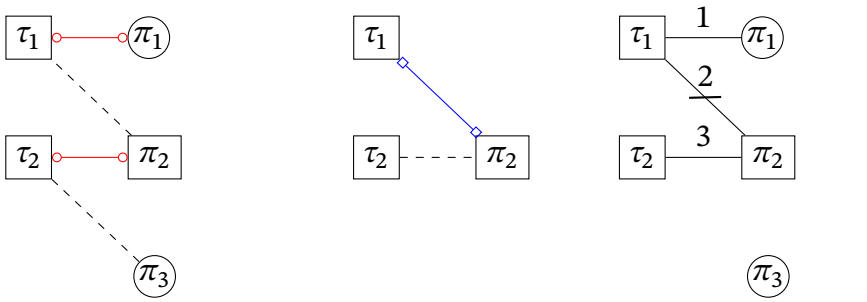
To simplify and to make our solution more generic, we rely on the following definition.

Definition 6.5 (Important vertex)**.** An *important* vertex is either a vertex corresponding to an urgent task or to a full processor.

Thus, the problem can be formulated as:

Given a bipartite graph having in each partition a subset of ***important*** *vertices. Determine a matching saturating all important vertices. (There may be additional non-important vertices in the matching, as well.)*

As far as we know, this problem is not addressed in the literature of graph theory [38, 39]. The closest problem we found is the assignment problem with seniority and job priority constraints [40, 41] but these works are not straightforwardly applicable to the problem raised here. Therefore, we propose in the following a correction of the procedure introduced in [19].



(a) Matching M_τ (b) Matching M_π (c) $M_\tau \cup M_\pi$

Figure 6.3: Illustration of the proposed matching algorithm

In our solution, we keep the two first steps proposed in [19, 29], but add as a third step a novel and efficient cleaning phase of the graph resulting from the union of the two matchings. Starting from a graph $G = (\Gamma \cup \Pi, E \subseteq \Gamma \times \Pi)$ obtained from the workload assignment matrix as illustrated in Figure 6.1, we apply the three following steps:

1. Determine a matching M_τ saturating the urgent tasks — by Theorem 6.1, this can always be done.

2. Determine a matching M_π saturating the full processors — by Theorem 6.1, this can always be done.

3. Let $G' = (\Gamma \cup \Pi, \{M_\tau \cup M_\pi\} \subseteq E)$. For each connected component g of G': let v_1 in g be an important 1-degree vertex if it exists or any 2-degree vertex otherwise. Traverse g from v_1 and discard every visited edge $e_i = (v_i, v_{i+1})$ having an even edge index. The first visited edge is $e_1 = (v_1, v_2)$, the second one is $e_2 = (v_2, v_3)$, etc.

Steps (1) and (2) are polynomially solvable using a *maximum cardinality matching* algorithm, as the one in [42]. It is applied to both sub-graph of G: $G_\tau = (\Gamma_u \cup \Pi, E_\tau = E \cap (\Gamma_u \times \Pi))$ and $G_\pi = (\Gamma \cup \Pi_f, E_\pi = E \cap (\Gamma \times \Pi_f))$. G_τ contains only the urgent task vertices (and all the processors vertices), when G_π contains only the full processor vertices (and all the task vertices). The Figure 6.4(a) (resp 6.4(b)) illustrates the matching M_τ on G_τ (resp. M_π on G_π).

Theorem 6.1 ensures that a maximum cardinality matching algorithm will find a matching saturating every urgent task vertex and every full processor vertex, respectively.

Step (3) is illustrated in Figure 6.4(c): in the guideline example, the edge (τ_1, π_2) with the even label number 2 is discarded, giving a correct matching saturating all the full processors and all the urgent tasks. This step simply ensures that i) every important vertex, which was paired in one of the two matchings in step (1) or (2), remains paired, and ii) the maximum degree of a vertex in G' is 1.

6.4 Proof of correctness of the algorithm

For the sake of the proof and without loss of generality, we consider G' introduced in Step (3) as a directed bipartite graph where edges in M_τ are oriented from an urgent task to a paired processor and edges in M_π are oriented from a full processor to a task. Please note that the directed version of G' only differ from G' in that the union of M_τ and M_π retains edges with the same vertices in opposite direction. For example, if the urgent task τ_i was matched to full processor π_j at Step 1, and that π_j was matched to τ_i, both edges (τ_i, π_j) and (π_j, τ_i) will be in the oriented graph. Only one of them would be kept in the set of edges of the non-oriented graph, as they would be seen as identical.

Property 6.1. In the directed version of G' built from the union of matchings $M_\tau \cup M_\pi$, every important vertex has exactly one outgoing edge, while every non-important vertex has no outgoing edge.

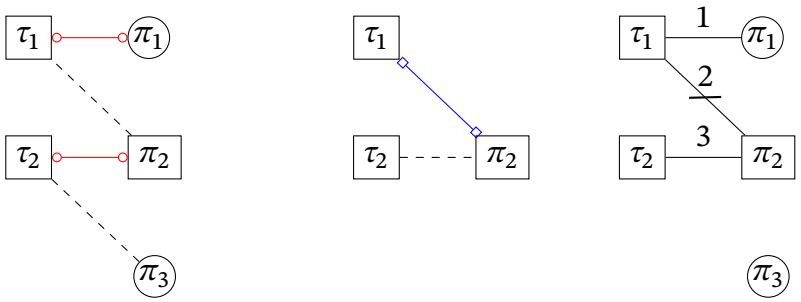
Proof. By Definition 6.5, an important vertex corresponds to either an urgent task or a full processor. By construction of both matching, the outgoing edges are only created from important vertices. The property follows. □

Property 6.2. The maximum degree of the directed version of graph G' built from the union of matchings $M_\tau \cup M_\pi$ is 2.

Proof. Since this graph is obtained by the union of two matchings where each vertex is present once, this graph has vertices with a degree of at most 2. □

Those two properties will be used to prove Theorem 6.2 in the following.

Theorem 6.2. Applying the step (3) of our algorithm to G' ensures a *correct matching* in the resulting graph, i.e. all important vertices have a degree of 1 (therefore every urgent task or full processor is saturated), and the maximum degree of the resulting graph is 1.



(a) Matching M_τ (b) Matching M_π (c) $M_\tau \cup M_\pi$

Figure 6.4: Illustration of the proposed matching algorithm

(a) Cycle (b) Path

Figure 6.5: Illustration of two possible types of connected components

Proof. Without loss of generality, we consider only one connected component of G' in this proof. Step 3 of the algorithm will process every connected component individually. 0-degree vertices are not in the matching and therefore not considered.

By property 6.2, G' has a maximum degree of 2 and a minimum degree of 1 (as it is a connected component). As G' is connected, there exists a trail connecting all vertices (namely from v_1 to v_n) in G'. Property 6.1 ensures that this trail vertices $v_1, \ldots, v_{n-1}$ are necessarily important. For the ending vertex v_n, there are only two possibles cases:

Case 1 (cycle) v_n is important. Thus, it has an outgoing edge and this trail is necessarily a *cycle*, i.e. a trail where no other vertices are repeated but the starting vertex v_1.

Case 2 (path) v_n is non-important. Thus, it has no outgoing edge and this trail is a path.

Both cases are illustrated in Figure 6.5 and addressed separately hereafter.

Case 1 $v_1 - v_n$ **cycle** It is known that a bipartite graph contains no odd cycles. Consequently, there is a unique cycle in G', $(e_1 = (v_1, v_2), e_2 = (v_2, v_3), \dots, e_{n-1} = (v_{n-1}, v_n), e_n = (v_n, v_1))$ containing an even number of n edges. It is now clear that discarding every edge with an even index lets every vertex with a degree of 1. It results that all vertices are degree 1 vertices, so Case 1 is proved.

Case 2 $v_1 - v_n$ **path** Starting from the important 1-degree vertex v_1, we discard every even indexed edge. The result depends on the path length parity:

- The number of edges is even, with the following sequence:

  $$e_1 = (v_1, v_2), e_2 = (v_2, v_3), \dots, e_{2k-1} = (v_{2k-1}, v_{2k}), e_{2k} = (v_{2k}, v_{2k+1})$$

  At the end of Step 3, the last edge is also discarded, then only the last vertex becomes 0-degree, which does not matter because it is a non-important vertex. Every other vertex is of degree 1.

- The number of edges is odd, with the following sequence:

  $$(e_1 = (v_1, v_2), e_2 = (v_2, v_3), \dots, e_{2k-2} = (v_{2k-2}, v_{2k-1}), e_{2k-1} = (v_{2k-1}, v_{2k}))$$

  At the end of Step 3, the last edge of the sequence is preserved and every vertex is of degree 1.

All important vertices are of degree 1 and the non-important vertex has a maximum degree of 1, so Case 2 is also proved and Theorem 6.2 is demonstrated.

□

Chapter 7

Schedule construction optimisation

In this chapter, we review two intuitive approaches that may be applied to optimise the template schedule in terms of preemptions and migrations. The first is a pre-optimisation because it aims at constructing an optimised template schedule from the workload assignment matrix. Differently, the second operates on a given template schedule and can thus be considered as a post-optimisation. Interestingly, those approaches can be used separately or combined.

7.1 Pre-optimisation: minimising the number of schedule points

In the Chapter 6, we showed how to build a template schedule from the workload assignment matrix using a BvN decomposition of a bistochastic matrix. One may note that minimising the number of permutation matrices in a BvN decomposition is similar as minimising the number of scheduling points. Indeed, each different permutation matrix corresponds to a different schedule decision (i.e. which jobs are executed at a given instant, and on which cores). Taking schedule decisions leads to preemptions and/or migrations (both inter- or intra-cluster). Therefore, minimising the number of scheduling points may be a solution to reduce the number of preemptions and migrations. The problem of deciding whether there is a BvN decomposition of a given doubly stochastic matrix with k permutation matrices is NP-complete in the strong sense[36].

Since the decision problem is NP-complete in the strong sense, the optimisation prob-

lem of minimising the number of permutation matrices in a BvN decomposition is NP-hard in the strong sense. Thus, in the following, we apply a greedy heuristic [36] from linear algebra and evaluate its efficiency in reducing the number of preemptions and migration when applied to a workload assignment matrix. This heuristic aims at successively maximising the size of the current scheduling window by solving the *bottleneck* matching problem at each step.

The bottleneck matching problem is also referred to as the Linear Bottleneck Assignment Problem (LBAP). The LBAP consists in finding in a square matrix of costs a matching (i.e. , permutation matrix) maximising the minimum value of the costs that are matched (or reversely minimising the maximum value). Formally, given a matrix A of dimension $a \times a$ with coefficients (costs) $[A(i,j)]_{i=1,\dots,a}^{j=1,\dots,a}$, find a permutation matrix P of the same dimension, with binary coefficients $P(i,j) \in \{0,1\}$ such that $\forall j = 1,\dots,a, \sum_{i=1}^{a} P(i,j) = 1$ and $\forall i = 1,\dots,a, \sum_{j=1}^{a} P(i,j) = 1$, which is maximising $\min_{i,j}(P(i,j)A(i,j))$.

Solving the LBAP problem on the bistochastic matrix B will give a matching P_1 which is maximising the value of δ_1, therefore maximising the duration of the first template schedule interval. Following the greedy heuristic from Dufossé et al. [36], after choosing P_1 and δ_1, we apply the assignment, by replacing B with $B - \delta_1 P_1$. We repeat the same operation until we replace B with a null matrix. Each permutation matrix P_i gives the assignment of tasks to cores until the next scheduling point. There are several efficient polynomial algorithms that we may use to solve the LBAP [43, 44].

7.2 LBAP experiments

We use here the same experiment setup as in Section 5.6.1. We have generated the systems as follows. The number of types of clusters $\dot{m}$ is either 2 or 5. The former in order to compare Hetero-Split to the other methods, and the latter because five different types of clusters is considered a large size for a heterogeneous MPSoC nowadays. Then, the number of cores per type of cluster is set in $[2, 5]$. The number of tasks n is arbitrary bounded as follows: $\dot{m} \leq n \leq 10 \times \dot{m}$. We then generate every task such that its period T_i is determined using [37]. The parameter C_i is based on T_i: $\frac{T_i}{2} \leq C_i \leq T_i$. We then generate the rates randomly and adjust them so that the tasks fit the given utilisation. For experimentation purposes, the clusters (the rates in particular) may be set to consistent.

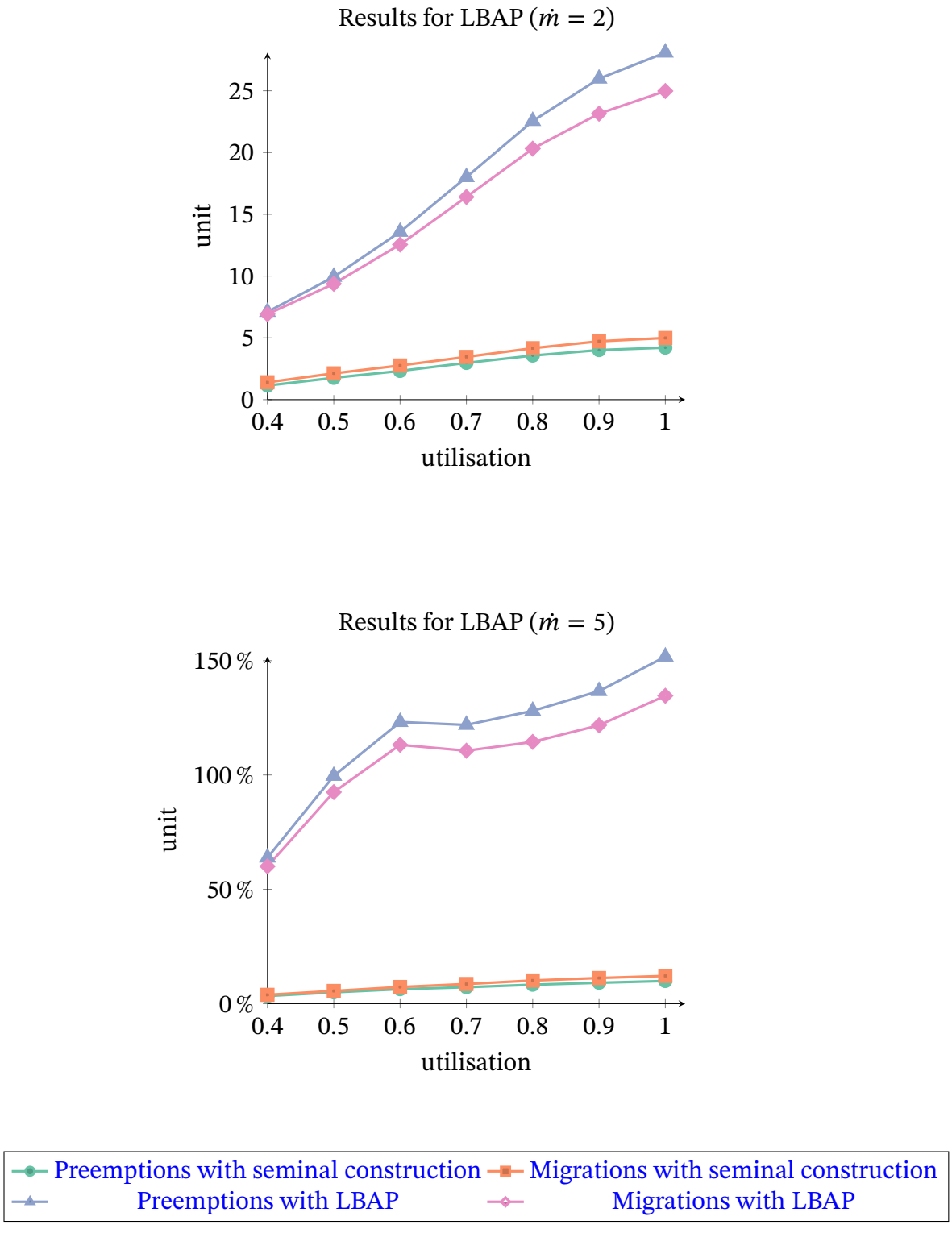


Figure 7.1: Results for LBAP ($\dot{m} = 2, 5$)

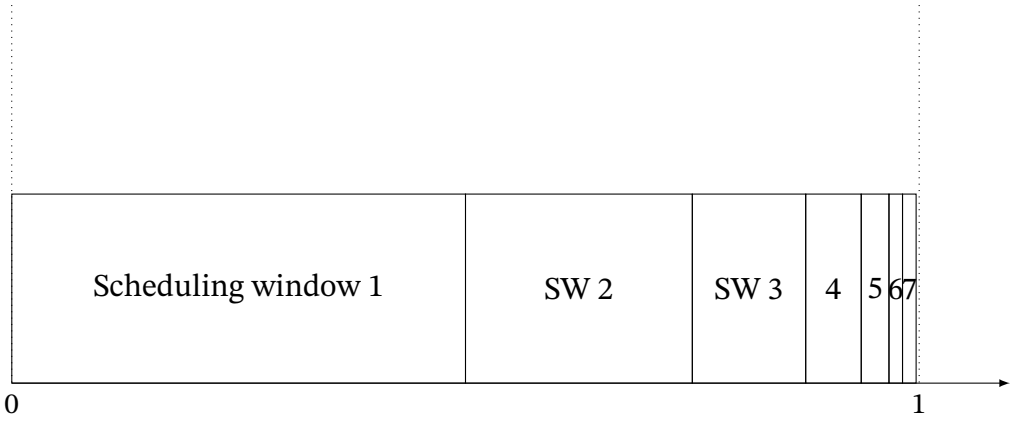


Figure 7.2: Example of LBAP pitfall

Using this generator, we generate 1 000 systems per total utilisation range $u \in [p - 0.1, p)$, increasing p from 0.4 to 1, for both $\dot{m} = 2$ and $\dot{m} = 5$. The ratio $p = 1$ corresponds to a full utilisation of the platform by the tasks. Here, p is equal to the value of the LP-CFeas objective function result, which is the minimal platform utilisation. The experimentation compares the different scheduling optimisations over 28 000 randomly generated test systems.

We can see very clearly in Figure 7.1 that LBAP performs very badly in term of both preemptions and migrations, when compared to the seminal construction from 4.2. For both $\dot{m} = 2$ and $\dot{m} = 5$, the number of preemptions and migrations is higher when using the LBAP construction instead of the seminal construction. Maximising the length of the intervals in a greedy way empirically seems to be a very bad way of minimising the number of scheduling point. Indeed, we observe that the last scheduling windows are getting smaller and smaller, as shown in Figure 7.2. Thus, the number of scheduling windows increases and so does the number of preemptions and migrations.

This schedule construction is not giving the expected results. However, it shows that the number of scheduling windows can be manipulated. We need to find a better method to achieve the goal of reducing the number of preemptions and migrations.

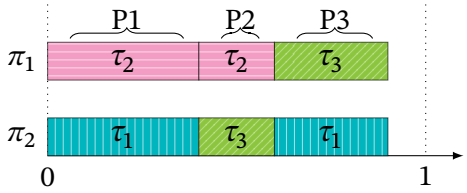


Figure 7.3: Non-optimised template schedule

7.3 Post-optimisation: Reordering the template schedule

The possibility of reordering a template schedule is mentioned in[29] through the following example. Given, the workload assignment $x_{1,2} = 0.7, x_{2,1} = 0.6, x_{3,2} = 0.2$ and $x_{3,1} = 0.3$, a valid template schedule is illustrated in Figure 7.3. We observe that τ_3 preempts τ_1 on π_2 at 0.4 and that τ_3 preempts τ_2 on π_1 at 0.6. In the following, the permutation $P_k = (\tau_i, \tau_j)$ corresponds to a scheduling window with τ_i on π_1 and τ_j on π_2. It is clear that a simple reordering, by switching the first (τ_1, τ_2) and second (τ_3, τ_2) permutations (or windows), as shown in Figure 7.4, saves one preemption. In the following, we present a method to systematise this intuition. We can model as a graph the permutations (as vertices) and the possible transition between (as edges), as shown in Figure 7.5. Edges are weighted by a distance metric evaluating how different two permutations are, i.e. the cost of the transition from a window to another. A template schedule corresponds to a path in the graph. With a distance metric based, for example, on the number of preemptions, the template schedule of Figure 7.3 corresponds to path (P_1, P_2, P_3) with a total weight of 3 with $P1 = (\tau_1, \tau_2), P2 = (\tau_3, \tau_2)$ and $P3 = (\tau_1, \tau_3)$. Also, the template schedule of Figure 7.4 corresponds to path (P_2, P_1, P_3) with a total weight of 2. The latter path is trivially the shortest possible path, i.e. the template schedule with the least number of preemptions. Finding the shortest path corresponds to the Traveller Salesman Problem (TSP). In the following, we evaluate this post-optimisation with two distance metrics in order to reduce the number of preemptions and migrations. On a given template schedule, the latter scheduling decisions are counted by summing the number of task preemptions on every processor plus the number of presences in excess. For example, the template of Figure 7.3 and Figure 7.4 have (1+2)+1 and (1+1)+1 scheduling decisions, respectively.

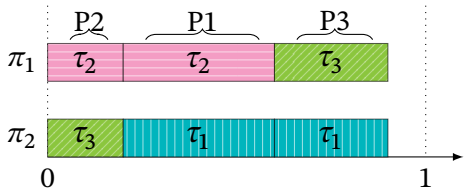


Figure 7.4: Template schedule optimised with TSP

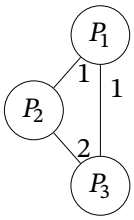


Figure 7.5: Scheduling windows through a TSP

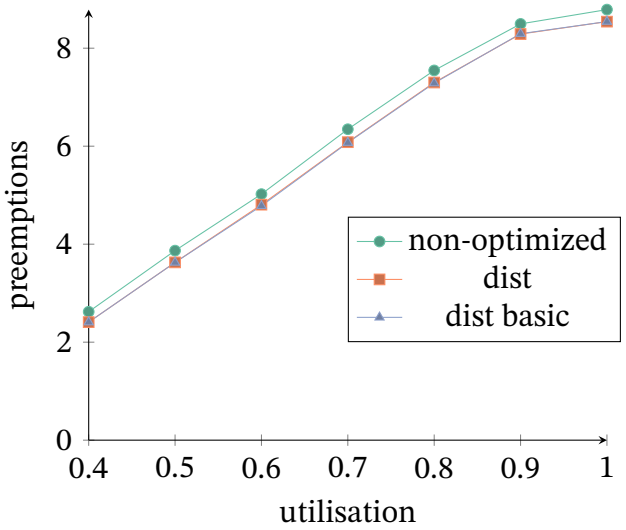


Figure 7.6: TSP ($\dot{m} = 2$)

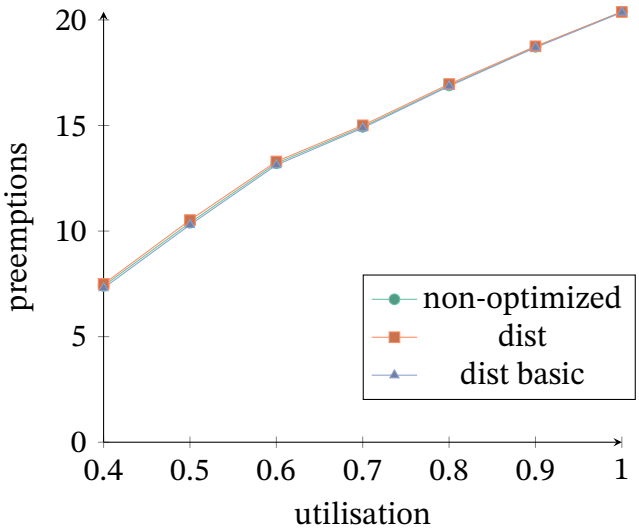


Figure 7.7: TSP ($\dot{m} = 5$)

7.4 TSP experiments

To empirically evaluate this approach, we use here the same experiment setup as in Section 5.6.1 and Section 7.2. We used two distance functions to evaluate the distance from one scheduling windows to another. Function *dist basic* counts every difference between two scheduling windows (in the same way as a hamming distance) while *dist* metric does not take into account passing to or from an idle time. The main limitation of this techniques is that it can only observe the change from one scheduling window to another. Therefore, a migration will not be detected if it doesn't happen in two subsequent scheduling windows. We cannot detect all the migrations. This is why we here count the preemptions. We make no difference whether the preemptions are followed by a migration or not.

The results are depicted in Figure 7.6 and Figure 7.7. We can see that the optimisation slightly improves the number of preemption. For example, the average number of preemptions is 2.4 with the function *dist* for a task set with a utilisation of 0.4. Without any optimisation, the average number of preemptions is 2.6 with the same task sets. However, this is a very small improvement. Moreover, we see that for $\dot{m} = 5$, there is no improvement at all, regardless of the task set utilisation. This may be due to the fact that the more tasks and processors there are, smaller is the average cost difference between the different scheduling windows.

Chapter 8

Flaw in the sporadic scheduler of [29]

In this chapter, we review another part from [29]. We prove that it contains another flaw. In this paper, the authors propose a scheduler for sporadic task sets running on a specific architecture as explained in Section 8.1.1. In Section 8.1, we quote the relevant part of the paper. We then show in Section 8.2 that this algorithm is flawed.

8.1 Seminal algorithm

In this section, we describe the seminal model and algorithm from [29].

8.1.1 Seminal model

Unlike the rest of the part, we consider sporadic tasks in this chapter. For such a task, the parameters T_i represents the minimum inter-arrival time. As defined in Definition 2.14, for any two successive jobs J_i and $J_{i'}$ released by τ_i, with $a_i < a_{i'}$, the inter-arrival time is equal to or greater than T_i: $a_{i'} - a_i \geq T_i$.

Also, the processing rates are restricted: a processing rate $R_{i,j}$ may be equal to 0 or 1 only.

8.1.2 Seminal algorithm offline phase

The offline phase of this algorithm has been described in Section 4.3. It produces a template schedule as shown in Figure 8.2 on page 93. This template schedule will be used during the run-time phase, as described in Section 8.1.3.

8.1.3 Seminal algorithm run-time phase

In this section, we give more details about Section IV.C from [29]. It describes the proposed run-time scheduler. This run-time scheduler uses the template schedule computed offline. It will stretch the pattern based on the active jobs and the potential job arrivals. To start our study, we will first replicate exactly the content of the procedure of [29], in the following italic text.

Suppose that a job of task τ_i, that executes upon more than one processor in the template schedule, arrives at some time-instant t_0. This job has a deadline at time-instant $t_0 + T_i$; hence,we will need to make reservations over the interval $[t_0, t_0 + T_i)$. Let $d_1, d_2, ..., d_k$ denote the (absolute) deadlines, indexed in increasing order (i.e., $d_j < d_{j+1}$ for all j) of jobs that had arrived prior to t_0, are still active at time t_0, and have deadlines within $[t_0, t_0 + T_i)$. Let Δ denote (an upper-bound on) the smallest relative deadline of any job that may arrive over this interval, and that may "interact" with the scheduling of τ_i's job, by, e.g., executing upon one of the processors on which τ_i executes. (A safe value for Δ is the minimum period of any task in the instance, although larger values may be obtained by more careful analysis — we postpone consideration of an optimal choice for Δ to future work.) To determine the reservations that must be made, we will

1. *First, let $d_0 \doteq t_0$. For each value of $j, 0 \leq j < k$, scale the template schedule by a factor of $(d_{j+1} - d_j)$, and invoke the reservations of this scaled template schedule over the interval $[d_j, d_{j+1})$.*

2. *Next, scale the template schedule by a factor Δ, and invoke the reservations of this scaled template schedule $\lfloor (T_i - d_k)/\Delta \rfloor$ times contiguously beginning at time-instant d_k.*

3. *Finally, scale the template schedule by a factor $((T_i - d_k) \bmod \Delta)$, and invoke the reservations of this scaled template schedule once, over the interval $[t_0 + T_i - ((T_i - d_k) \bmod \Delta), t_0 + T_i)$.*

8.2 Counter-example

In this section, we present a counter-example to the algorithm presented in Section 8.1.3. The counter-example is based on the task set presented in Figure 8.1.

First of all, we don't know what should happen if two absolute deadlines d_i and d_{i+1} are equal. We assume that equal absolute deadlines may be treated in any order.

	C_i	D_i	$r_{i,1}$	$r_{i,2}$	$r_{i,3}$	$r_{i,4}$
τ_1	$\frac{20}{3}$	10	1	1	0	0
τ_2	$\frac{20}{3}$	10	1	0	0	1
τ_3	$\frac{20}{3}$	10	0	1	1	0
τ_4	$\frac{20}{3}$	10	0	0	1	1
τ_5	$\frac{20}{3}$	10	1	0	1	0
τ_6	$\frac{20}{3}$	10	0	1	0	1

Figure 8.1: Counter-example task set

Section 8.2.1 presents the behaviour of the offline phase of the algorithm. Section 8.2.2 simulates the run-time phase in a given scenario. Finally, Section 8.2.2 proves that the produced schedule is flawed.

8.2.1 Seminal offline phase

As said above, the offline phase is out of the scope of this chapter. In this new algorithm, we use directly the corrected version of the seminal algorithm from Section 6.3

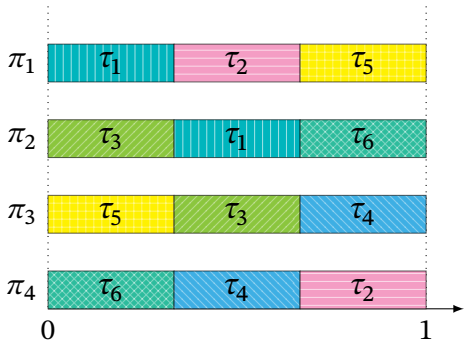


Figure 8.2: Counter-example template schedule

8.2.2 Seminal run-time phase

In this section, we detail the run-time phase. The run-time phase algorithm depends on the job arrivals. In the scenario of this counter-example, we look up two job arrivals:

one at 0 and one at 1.

At $t = 0$, tasks $\tau_2, \tau_3, \tau_4, \tau_5, \tau_6$, all release a job. By following the 3-steps algorithm, we first have $i = 6, t_0 = 0, T_i = 10, k = 0, \Delta = 10$. Then,

1. Let $d_0 \doteq t_0 = 0$. Because $k = 0, \forall j, 0 \leq j < k = 0$, is empty and there is nothing to do.

2. Next, scale the template schedule by a factor $\Delta = 10$, and invoke the reservations of this scaled template schedule $\lfloor (T_i - d_k)/\Delta \rfloor = \left\lfloor \frac{10-0}{10} \right\rfloor = 1$ times contiguously beginning at time-instant $d_k = 0$.

3. Because $((T_i - d_k) \bmod \Delta) = ((10 - 0) \bmod 10) = 0$, there is nothing to do here. The template scaled by a factor 0 is empty.

This steps give us the required reservations. The reservations made are referenced in Figure 8.3.

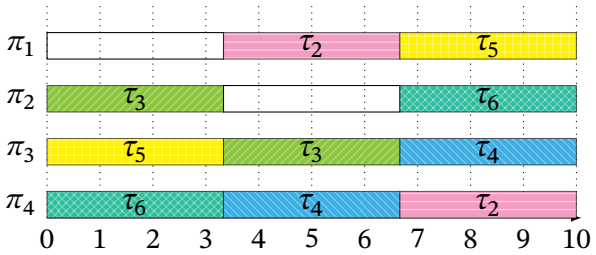


Figure 8.3: Counter-example: reservation at $t = 0$

Then, task τ_1 releases a job at $t = 1$. Following the 3-steps algorithm: we first have $i = 1, t_0 = 1, T_i = 10, k = 6, d_1 = 10, \Delta = 10$. Then,

1. First, let $d_0 \doteq t_0 = 1$. For each value of $j, 0 \leq j < k = 6$, scale the template schedule by a factor of $(d_{j+1} - d_j) = (10 - 1) = 9$, and invoke the reservations of this scaled template schedule over the interval $[d_j, d_{j+1}) = [1, 10)$. Because $\forall j < k, d_{j+1} = 10$, we do that only once.

2. Next, because $\lfloor (T_i - d_k)/\Delta \rfloor = \lfloor (10 - 10)/10 \rfloor = 0$, there is nothing to do here.

3. Finally, because $((T_i - d_k) \bmod \Delta) = 0$, there is nothing to do here.

Previous reservations are discarded. The new reservations made are referenced in Figure 8.4. This schedule shows the executed task between 0 and 1 and the reservations between 1 and 10.

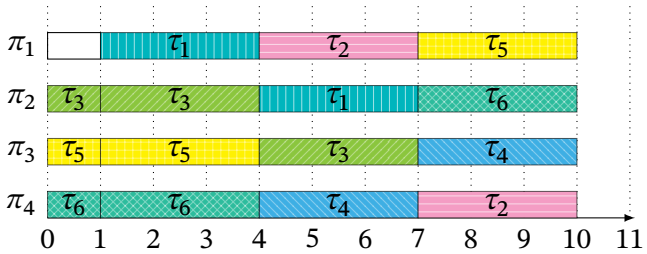


Figure 8.4: Counter-example: reservation at $t = 1$

Schedule incorrectness

According to the schedule presented in Figure 8.4, we have the following facts:

- Tasks τ_3, τ_5, τ_6 will be executed 7 units of time in window $[0, 10]$ (instead of $\frac{20}{3} \approx 6.66$).
- Tasks τ_2, τ_4 will be executed 6 units of time in window $[0, 10]$ (instead of $\frac{20}{3} \approx 6.66$).

We therefore have three tasks that are over-executed, which is incorrect. Worse, we have two tasks that are not completed before their deadlines in this scenario. This example is thus a counter-example showing that this algorithm fails to schedule this task set with those job arrivals.

Chapter 9

Conclusion

In this part, we have addressed the question of global scheduling for heterogeneous unrelated platforms. We followed the scheduler scheme commonly used in the literature for this problem. This scheme decomposes the problem into several steps, some being offline and some being online. It first assigns the workload among the different processors, using an LP. It then constructs a template schedule offline by using a matching algorithm. At last, it stretches the template schedule at run-time. We reviewed every step, starting from the model, and improved each one of them. Most improvements were variations or alternative algorithms, and one of them was a correction of a flawed algorithm. We also proposed post-optimisations to the construction step. By improving every step, this part builds a brand new global optimal scheduler for heterogeneous unrelated platforms for periodic tasks. We have shown that our clustered approach was performing better than the non-clustered from [19, 29] by reducing the number of presences and thus the run-time overheads. At last, we showed that another scheduler present in the literature was flawed. This scheduler was designed on the same scheme but for sporadic tasks instead of periodic tasks. We showed that it was flawed by exhibiting a counter example.

Future research could lead to a correction of a second flawed algorithm. However, the scheduler scheme used through this part is based on fairness. The unpredictability of sporadic tasks seems to be incompatible with the required fairness of the used scheme. Also, some of the presented variations and optimisations did not perform as well as expected. However, we do believe that the approaches taken are promising. Pursuing the work for those parts may lead to great improvement in terms of migrations.

Part III

Multi-mode applications

Table of Contents

Chapter 10

Introduction to multi-mode applications

As presented in Chapter 2, multi-mode applications are used today in order to fit the software to different internal and/or external contexts. Some modern hardwares offer run-time reconfiguration, to adapt the hardware to the current system workload. In this part, we propose a new approach: combining both software changes and hardware reconfigurations. In this approach, each mode is defined by a set of functionalities and a specific hardware configuration adapted to those functionalities. The hardware reconfigurations are handled during the transition from one mode to another. To the best of our knowledge, combining both multi-mode changes and hardware reconfiguration has never been studied before. However, fitting the hardware to the current functionalities of the application leads to a more efficient execution. Efficiency may be in terms of computation speeds, energy consumption and/or hardware requirements. Indeed, an FPGA accelerator is way faster for some tasks than a general purpose processor, as shown in the general introduction. It needs to be configured in a specific configuration for each different software task in order to be efficient. Similarly, some tasks never run in a given context. In such cases, the energy consumption may be reduced by reducing the number of active processors and turning some processors off for a given mode.

In practice, we propose to use the software transition phase to reconfigure the hardware and change the software. Several challenges rise from this new paradigm. First of all, a new model is necessary in order to precisely define the problem. Designing a protocol requires to specify the required hardware reconfiguration, and to perform those

reconfigurations alongside with the existing real-time software constraints. At last, a protocol should provide a validity test in order to be used in practice. Its performance must be acceptable, in order to be usable in practice.

10.1 Contributions and organisation

In this part, we first provide the first model combining both software changes and hardware reconfigurations. This model combines a model of the software, a model of the hardware, and defines also the multi-mode aspect of the application and platform. We then introduce a first ACCEPTOR protocol that handles the mode change phase of a multi-mode application, where the hardware may be reconfigured and the software may be changed. This protocol is designed to be used with a FJP work-conserving scheduler, in a clustered based approach. We first describe precisely the protocol. We then present a validity test suitable for this protocol, so that it can be used in practice. We also perform a complete analysis of the protocol, both theoretical and empirical, tackling time complexity, pessimism and competitivity. This first protocol sets of a competitor for future protocols. We introduce in the last chapter a second protocol SQUARER. Similarly to ACCEPTOR, SQUARER is designed to be used with a FJP work-conserving scheduler, in a clustered based approach. This second protocol has a different approach of the mode change phases. After a complete description of this new protocol, we proposed its validity test. We then evaluate this protocol and compare it to the previous one. At last, we propose some directions for improvement in future works.

This part is organised as follows:

- Chapter 11 introduces a new model for multi-mode applications:
  - Section 11.1 describes the hardware model used trough this part;
  - Section 11.2 describes the software model used trough this part;
  - Section 11.3 introduces the first multi-mode model combining both hardware reconfigurations and software changes;
- Chapter 12 proposes the ACCEPTOR protocol for multi-mode applications:
  - Section 12.1 defines precisely the scheduling problem encountered during a mode change phase;

  - Section 12.2 proposes an in-depth description of the protocol ACCEPTOR;
  - Section 12.3 introduces an upper-bound on the duration of a mode change phase when handled by ACCEPTOR;
  - Section 12.4 proposes a validity test for ACCEPTOR;
  - Sections 12.5–12.7 analyse the performance of ACCEPTOR, both theoretically and empirically,
  - Section 12.8 discusses the limitation of ACCEPTOR;
  - Section 12.9 explores a variation of ACCEPTOR without changing the approach used;
- Chapter 13 proposes the alternative protocol SQUARER:
  - Section 13.1 describes the second protocol SQUARER;
  - Section 13.2 introduces a validity test for SQUARER;
  - Sections 13.3–13.4 evaluate empirically SQUARER and compare it to ACCEPTOR.

Chapter 11

Introducing a new multi-mode application model

In this chapter, we present the first contribution of this part: a new model to multi-mode applications. To the best of our knowledge, this model is the first to combine both the multi-mode application model aspect and the reconfigurable platform model aspect.

We first present the hardware model, and then the software model. Those models are heavily based on the ones presented in Chapter 2. We then present the new multi-mode and cluster models, which are modified versions, adapted from the ones presented in Chapter 2. In order to be self-sufficient, this chapter repeats some information. A table of the notations is at the end of the chapter.

11.1 Hardware model

We consider here an heterogeneous unrelated platform, composed of m reconfigurable processors. It is denoted as $\Pi \doteq \{\pi_1, \pi_2, \dots, \pi_m\}$. Processor π_j has a type π^k, with $k \in [1, \dots, \phi]$, where ϕ is the number of *processor types* on Π.

A reconfigurable processor of type π^k is configured at any time in a *configuration* $\theta_c \in \Theta_k$ from its set of configurations Θ_k. The configuration of a processor defines several parameters like the instruction set of the processor or its processing rate. The set of sets $\Theta \doteq \{\Theta_1, \Theta_2, \dots, \Theta_\phi\}$ contains the set of allowed configurations for each type. A configuration belongs to at most one type: i.e. $\forall k, k', k \neq k' \implies \Theta_k \cap \Theta_{k'} = \varnothing$.

There are o different configurations, with $o \doteq \sum_{k=1}^{\phi} |\Theta_k|$. Reconfiguring a processor is not instantaneous. It takes δ_c time-units (denoted as the *reconfiguration delay*) to reconfigure a processor of type π^k to θ_c if $\theta_c \in \Theta_k$, otherwise it takes $+\infty$. We consider here parallel reconfigurations: all the platform may be reconfigured simultaneously.

11.1.1 Clustered platforms

The platform will be divided at run-time into $\dot{m}$ clusters: $\dot{\Pi} \doteq \{\dot{\pi}_h \mid h = 1, \dots, \dot{m}\}$. The processors forming a cluster are referred to as *cores*. Each cluster $\dot{\pi}_h$ is formed by $\dot{m}_h$ *cores*: $\dot{\pi}_h \doteq \{\pi_{h_1}, \dots, \pi_{h_{\dot{m}_h}}\}$. By construction, $\sum_{h=1}^{\dot{m}} \dot{m}_h = m$.

In this first multi-mode model with reconfigurable hardware, clusters are formed with cores of the same type, configured identically. There may be only one cluster of cores configured in a given configuration θ_c. Because a cluster depends on the configuration of the cores, if the cores are reconfigured at run-time, the clusters will change. Also, we consider here only clustered scheduling, with a task subset for each cluster $\dot{\pi}_h$. As a processor always belongs to a cluster, we will now use exclusively the term of *core* instead of *processor*.

The following definition will be used in our protocol.

Definition 11.1 (θ_c-cluster)**.** The θ_c-cluster is the cluster composed of the cores configured in θ_c.

11.2 Software model

We consider here sporadic tasks, with implicit deadlines. A task τ_i is defined by two components (C_i, T_i), where C_i is the WCET and T_i is the minimal arrival time. A *task* τ_i releases a potential infinite sequence of jobs. When a task releases a job J_i at a_i, its absolute deadline $d_i \doteq a_i + D_i$ and its WCET $c_i \doteq C_i$.

The *processing rate* depends on both the job and the core. Specifically, the job processing rate $R_{i,c}$ on the core $\dot{\pi}_h$ depends on both the task τ_i and the current configuration θ_c of $\dot{\pi}_h$. Formally, a core executes $t \times R_{i,c}$ computing units when configured in θ_c and executing a job J_i for t time-units. This amount may be null if the task cannot be executed on this configuration.

11.3 Multi-mode model

There are two majors changes from the model presented in Section 2.5.1. In this new model, a mode contains a hardware specification. It represents the required configurations for the platform, for this mode. The second major difference is that the platform is clustered, with cluster composed of cores configured identically, and therefore of the same type. At last, no mode-independent tasks are allowed.

Formally, a multi-mode application is composed of μ *modes* $M \doteq \{M^1, M^2, \dots, M^\mu\}$. A mode $M^q = < \tilde{\Theta}^q, T^q, \Delta^q >$ contains a task subset T^q. $\tilde{\Theta}^q$ represents the required platform configurations. For each mode, Δ^q is the real-time constraint. It constraints the maximal duration between the mode change request and the completion of the mode change phase. This real-time constraint ensures that every mode will be activated on time. It is chosen at design time.

Definition 11.2 (Hardware)**.** The platform must be configured in a specific way to respect the application specifications. This specifications ensure the correct execution of the task set. *The configurations vector* $\tilde{\Theta}^q \doteq< m_1^q, m_2^q, \dots, m_o^q >$ contains for a specific mode M^q the required number m_c^q of cores configured in θ_c. The required configurations for all the modes are specified as a vector of configuration vectors: $\tilde{\Theta} \doteq< \tilde{\Theta}^1, \tilde{\Theta}^2, \dots, \tilde{\Theta}^\mu >$.

Definition 11.3 (Mode task subset)**.** The mode M^q contains a specific task subset T^q. It is formed by n^q tasks. To partition the task set among several clusters, T^q itself is divided into task subsets: one per different configuration present in the mode M^q. The cores configured in θ_c must run the task subset $T^{q,c}$. The tasks are said to be *mode-dependent*: each task may appear in at most one mode, i.e. $\forall q, q', q \neq q' \implies (\bigcup_c T^{q,c}) \cap (\bigcup_c T^{q',c}) = \varnothing$. This limitation will be discussed in Section 12.8. Task subset $T^{q,c}$ contains $n^{q,c}$ tasks.

Definition 11.4 (Mode real-time constraint)**.** Switching from one mode to another is not instantaneous. Rem-jobs cannot be aborted before completion. However, this delay must be bounded to take into account the real-time constraints of the application. Δ^q represents the maximum allowed delay for reconfiguring the system after a mode change request to M^q. The set $\Delta \doteq \{\Delta^1, \Delta^2, \dots, \Delta^\mu\}$ contains the real-time constraint of each mode.

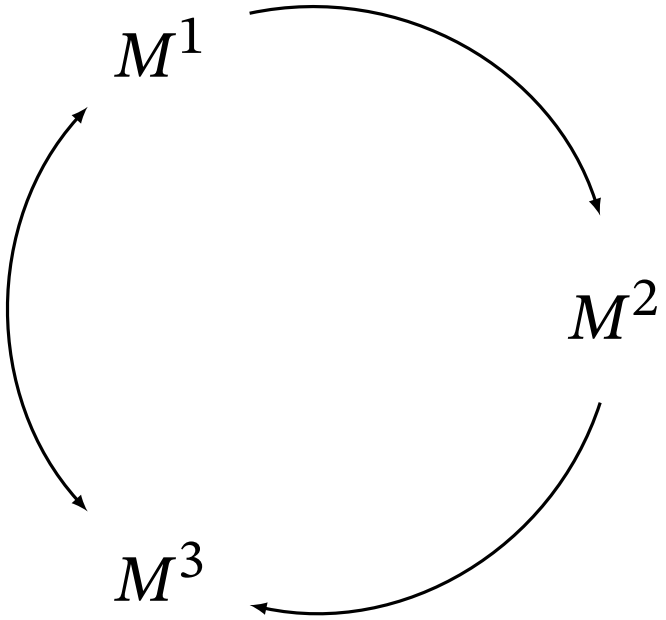


Figure 11.1: Graph transition example

Mode transitions

The application executes at any instant one and only one mode M^q. This mode M^q is the *active mode*. The active mode may only change during a *mode change phase*. A *mode change phase* is triggered when the system receives a *mode change request*. When a mode change request MCR(M^{dst}) occurs at t_{MCR}, the current mode is immediately deactivated (and its task subset is disabled), and new mode M^{dst} must be activated (and its task subset enabled) by $t_{\text{MCR}} + \Delta^{\text{dst}}$. The mode change phase ends when the new mode M^{dst} is activated. Please note that this constraint depends only from the destination mode M^{dst}, independently from M^{src}.

Mode change graph

In an application, some transitions will never occur. The allowed transitions are represented in the *mode change graph* $\mathcal{G} \doteq \{\text{V}, E \subseteq \text{V}^2\}$. The *mode change graph* is a directed graph, where V contains one and only one node for each mode $M^q \in M$, and E represents all the allowed transitions from one mode to another. A mode change phase from a mode M^{src} to a mode M^{dst} is allowed if and only if $(M^{\text{src}}, M^{\text{dst}}) \in E$. A graph example is given in Figure 11.1. In this example, the mode following M^1 may be M^2 or M^3. However, the mode following M^2 must be M^3 and may not be M^1: the directed edge from M^1 to M^2 is unidirectional.

11.4 Model example

In this section, we provide instances of our model to match existing real-world platforms.

Example 1.

The platform is organised in several unrelated clusters of identical cores. When using DPR (Dynamic Partial Reconfiguration), the Zynq UltraScale+™ may be configured such that it is composed of the following clusters:

- a four cores identical platform meant to execute general purpose application software (the 4 Cortex-A53 cores);
- a dual core identical platform that may host highly critical and predictable software (the dual Cortex-R5 cores);
- a GPU dedicated to display information (the Mali™-400 GPU). We model it as a cluster with a single core;
- 4 independent hardware accelerators that are each dedicated to one specific task. These accelerators are hosted in the Programmable Logic (FPGA) which may arbitrarily be divided in 4 independent slots by the system designer (and managed by the DPR engine). Each one of these accelerators/slots corresponds to a single core and the whole Programmable Logic is then modelled as a cluster with 4 cores. The ability of the others cores to change some features—such as their voltage/frequency— will be ignored, to keep this example short.

The Zynq UltraScale+™hardware is illustrated in Figure 11.3. In this example, we assumed that the system designer used Dynamic Partial Reconfiguration (DPR) capabilities offered by the Zynq chip family [45]. Please note that earlier research allow the use of DPR in real-time systems as described in Section 1.3. We assume that the designer divided the FPGA into 4 cores that are reconfigurable as defined in Section 2.3

In this example, the platform P is composed of the following $m = 11$ cores:

- $\pi_1, \pi_2, \pi_3, \pi_4$: the application cores, forming the Application Processing Unit (APU). All these cores have the same architecture (A53) and belong to type π^1:

$\Pi_1 = \Pi_2 = \Pi_3 = \Pi_4 = \pi^1$. As the APU is a static CPU, it has only one configuration, meaning $\Theta_1 = \{\theta_1\}$, where θ_1 denotes the only possible configuration of the APU.

- π_5, π_6: the real-time cores, forming the Real-Time Processing Unit (RPU). These cores have the same architecture (R5) and then belong to type π^2. Similarly to the APU, we have $\Theta_2 = \{\theta_2\}$, where θ_2 denotes the only possible configuration of the RPU.

- π_7: it represents the GPU, which has its own type π^3 and is also a non-reconfigurable cores: $\Theta_3 = \{\theta_3\}$ where θ_3 denotes the only possible configuration of the GPU.

- $\pi_8, \pi_9, \pi_{10}, \pi_{11}$: the different dynamically partially reconfigurable cores of the chip Programmable Logic (from the FPGA). These 4 cores are independent. We assume that the cores are specialised for some processing: we have two types of cores π^4 and π^5. We assume that in this example, π_8 and π_9 are image processing kernels (allowing to process an input image and apply a filter on it), and that π_{10}, π_{11} implement cryptographic accelerators in order to implement several block cipher algorithms. The image processing kernels can be in three configurations: sepia, sobel and grayscale. The cryptographic accelerators can be configured either to run AES or 3DES block ciphers. Therefore:

$$\begin{cases} \Theta_4 = \{\theta_{\text{sepia}}, \theta_{\text{sobel}}, \theta_{\text{gray}}\} \\ \Theta_5 = \{\theta_{\text{aes}}, \theta_{\text{des}}\} \end{cases}$$

The rates of task τ_{aes} depend on the configuration of its cores. This task can be performed only on RPU, GPU or a specialised FPGA. Therefore, we could have : $R_{\text{aes,aes}} > R_{\text{aes,3}} > R_{\text{aes,2}} > R_{\text{aes,1}} > R_{\text{aes,des}} = 0$. Obviously, this task is completed (way) faster on a specialised FPGA, and cannot be executed on an FPGA wrongly configured (hence $R_{\text{aes,des}} = 0$). This platform example is depicted in Figure 2.2(a). The platform is represented with several divisions into *clusters*. Typical reconfiguration times of the Programmable Logic are of the order of a few milliseconds [10]. These timings could be used to set the values of $\delta_{\text{sepia}}, \delta_{\text{sobel}}, \delta_{\text{gray}}, \delta_{\text{aes}}$ and δ_{des}.

Example 2. The ARM big.LITTLE architecture is a technique allowing to switch between high-performance cores and low-power cores. A typical design of this architecture is composed of 4 A57 cores (the high-performance cluster) and 4 A53 cores (the low-power cluster) used alternately. These kind of platforms suits our model as the

multi-mode reconfiguration is implemented in the hardware. In this case, we model the platform P as a set of $m = 8$ cores:

- $\pi_1, \pi_2, \pi_3, \pi_4$: the high-performance cluster. All cores are of type π^1 and $\Theta_1 = \{\theta_{A57}, \theta_{\text{off}\,57}\}$, these configurations meaning that the cores are either active or inactive.
- $\pi_5, \pi_6, \pi_7, \pi_8$: the low-power cluster. They are of type π^2 and, similarly to the other type, we have $\Theta_2 = \{\theta_{A53}, \theta_{\text{off}\,53}\}$

In some platform designs such as Samsung Exynos 5 Octa, only one kind of core can run simultaneously — the operating system must explicitly switch the whole platform from one kind of core to the other. This constraint can be enforced by defining the clusters and modes accordingly and by using the inactive configurations $\theta_{\text{off}\,53}$ and $\theta_{\text{off}\,57}$. For example, if we define two modes M^{53} and M^{57}, $\tilde{\Theta}^{53} =< m^{53}_{A53}, m^{53}_{\text{off}\,53}, m^{53}_{A57}, m^{53}_{\text{off}\,57} >=< 4, 0, 0, 4 >$ and $\tilde{\Theta}^{57} =< m^{53}_{A53}, m^{53}_{\text{off}\,53}, m^{53}_{A57}, m^{53}_{\text{off}\,57} >=< 0, 4, 4, 0 >$. This means that during mode M^{53}, only the cores of type π^2 will be active.

These two examples illustrate how generic our model is and how it is able to capture the essence of real-world platforms.

A job	J_i
Job J_i WCET	c_i
Job J_i deadline	d_i
Job J_i arrival time	a_i
Task τ_i WCET	C_i
Task τ_i relative deadline	D_i
Minimum inter-arrival time of τ_i	T_i
The task set	Γ
Number of tasks in a task set	n

(a) Task notations

A processor	π_j
The platform	Π
Number of proc. in Π	m
Type of a proc.	π^k
Number of types	ϕ
Set of proc. types	Ψ
Rate of proc. π_j for a task τ_i	$R_{i,j}$
Utilisation of τ_i on π_j	$U_{i,j}$
Set of configuration sets	Θ
A π^k configuration	θ_c
Number of configurations	o
Reconfiguration delay of θ_c	δ_c
Progression rate for J_i on θ_c	$R_{i,c}$

(b) Platform notations

Set of clusters	$\dot{\Pi}$
A cluster	$\dot{\pi}_h$
A core of cluster h	π_{h_1}
Number of clusters	$\dot{m}$
Number of cores in cluster h	$\dot{m}_h$

(c) Cluster notations

Set of modes	M
A mode	M^q
Number of modes	μ
Task set of a mode M^q	T^q
Number of tasks in T^q	n^q
Sets of r.t. constraints	Δ
Mode M^q's r.t. constraints	Δ^q

(d) Multi-mode notations

is equal by definition to	$\doteq$
A given instant	t
A scheduler	S

(e) Misc. notations

Figure 11.2: Notation summary

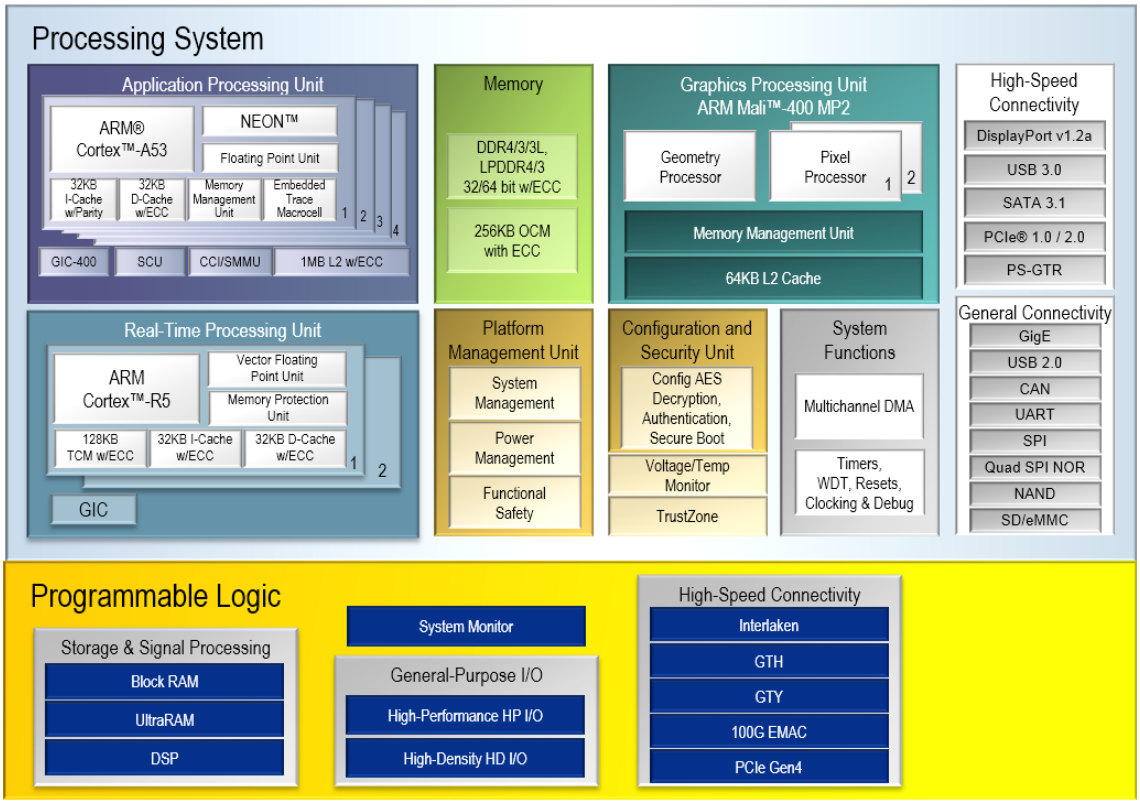


Figure 11.3: The Zynq UltraScale+™ EG processor block diagram. EG devices feature a quad-core ARM® Cortex-A53 platform running up to 1.5GHz, combined with dual-core Cortex-R5 real-time processors, a Mali-400 MP2 graphics processing unit, and a 16nm FinFET+ programmable logic [1].

Chapter 12

A first protocol for multi-mode applications: ACCEPTOR

In this chapter, we introduce the second contribution of this part: the first protocol that handles software and hardware mode transitions. It is called ACCEPTOR, for: AsynChronous ClustEr-based ProTOcol for multi-mode applications on Reconfigurable platforms. It is designed to handle multi-mode applications modelled by the model presented in Chapter 11. The ACCEPTOR protocol conducts the mode change phase: it schedules the rem-jobs, performs the requested reconfigurations and enables the new mode tasks. It also respects (if possible) the real-time constraints such as the delay for new mode's activation and the rem-jobs hard deadlines.

The chapter is organised as follows. Section 12.1 describes the problem treated by ACCEPTOR. Section 12.2 presents the mode change protocol proposed as a first solution to handle the transition between the different modes of a multi-mode application. Section 12.3 proposes an upper-bound of the duration of the transitions, when using the introduced protocol. Section 12.4 provides a validity test and proves its correctness. Sections 12.5–12.7 evaluate the protocol through a complete evaluation using simulations for the upper-bound efficiency, and both a complexity and competitive analysis of the protocol. Section 12.8 discusses some of the limitations of the model. Finally, Section 12.9 explores an improvement path for this first protocol.

12.1 Scheduling problem

At a given instant, the mode M^q is activated and the platform is composed of $\dot{m}$ clusters. Each cluster $\dot{\pi}_h$ has $\dot{m}_c^h$ cores, configured in a given configuration θ_c. The task set $T^{q,c}$ is scheduled upon the $\dot{m}_c^h$ cores configured in θ_c. When a mode change request is received, the application must be changed by the protocol and the hardware reconfigured. The task sets of the new mode M^{dst} must be enabled and its clusters must be formed as well, by reconfiguring some cores if needed. Active jobs must still be completed before their deadlines. Because core reconfiguration delays depend on the new configurations, reconfiguration times may differ from one core to another inside a cluster, if the target configurations differ.

The protocol must schedule the active jobs, find the necessary reconfigurations and perform it. Because clusters depend on core configuration, a cluster may be split into two or more clusters when its cores are reconfigured. For example, let us consider $\dot{\pi}_1$ composed of two cores configured in θ_1, and $\dot{\pi}_2$ composed of two cores configured in θ_2. During the mode change phase, one core from $\dot{\pi}_1$ is reconfigured into θ_3 and so is one core from $\dot{\pi}_2$. After the mode change, there will not be two but three clusters: one composed of one core in θ_1, one composed of one core in θ_2 and one composed of two cores in θ_3. The cores from both clusters that have been reconfigured in θ_3 are now merged in a single cluster.

Here, we only consider the optimisation of a mode change phase from one mode to another. The mode are given as an input of the problem. Defining the optimal hardware requirements for a given task set is out of the scope of this chapter. Also, no reconfiguration are made to optimise potential future mode change. Such optimisations could be done independently from the presented work. For example, potential future mode change could be anticipated by reconfiguring unused processors during the execution of the current mode.

12.2 ACCEPTOR

The ACCEPTOR protocol is an aperiodic asynchronous protocol, with respect to Definition 2.58 and Definition 2.59. It can be used with sporadic task sets, scheduled by a clustered work-conserving scheduler.

The global approach is the following: after a *mode change request*, it first schedules the

rem-jobs of all clusters and then reconfigures each core (if necessary) once they become idle. This protocol is composed of an offline phase and a run-time phase. The offline phase determines, for each cluster $\dot{\pi}_h$, the required reconfigurations for each possible mode change transition from M^{src} to M^{dst} (see Section 12.2.1). The run-time phase is dedicated to schedule the rem-jobs and reconfigure the cores (see Section 12.2.2). This run-time phase considers each cluster $\dot{\pi}_h$ individually, and is applied independently on each cluster. Finally, an example of the protocol is given in Section 12.2.4.

12.2.1 Offline phase: computing the required reconfigurations

The offline phase is the first step of the protocol. It is applied independently for each possible mode change phase from M^{src} to M^{dst}. It simply computes which reconfigurations must be done during a *Mode Change Phase* from M^{src} to M^{dst}. This computation is performed independently for each different type π^k, independently of their configurations in M^{src}. This is possible because no configuration are shared between the different types. It uses the following upper-bound on the makespan instant.

Lemma 12.1 (Lemma 5 in [4])**.** Suppose that J is ordered by non-decreasing job processing times, i.e. , $c_1 \leq c_2 \leq \cdots \leq c_n$, all starting at $t = 0$. Suppose that no reconfigurations are performed on none of the m' cores. Then, whatever the job priority assignment made by a FJP work-conserving global scheduler, an upper-bound $\overline{\text{makespan}}$ on the makespan instant is given by:

$$\overline{\text{makespan}} \doteq \begin{cases} c_n & \text{if } \leq m' \\ \\ \frac{\sum_{i=1}^{n-1} c_i}{m'} + c_n & \text{otherwise} \end{cases} \tag{12.1}$$

To perform this step, the protocol first computes the differences between the required reconfigurations of both modes. Then, it computes the makespan of each π^k-cluster of M^{src} using the bound provided by [4]. It then assigns the longest (resp. shortest) required reconfigurations to the π^k-clusters having the shortest (resp. longest) makespans, based on the reconfiguration delays. Inside each cluster, the shortest (resp. longest) reconfigurations are assigned to the cores being idle the first (resp. last), based on the idle-instant upper-bound from [3]. The reconfiguration assignment is denoted

$\{\theta_c, \theta_{c'}\}$ where θ_c is the configuration of a core in M^{src} and $\theta_{c'}$ is the configuration of a core in M^{dst}. At run-time, all the reconfiguration assignments $\{\theta_c, \theta_{c'}\}$ must be handled, by reconfiguring a core currently in θ_c to $\theta_{c'}$. Obviously, a core may be reconfigured only once during a *Mode Change Phase*.

Formally,

- For any θ_c-cluster in each mode M^{src}, compute the makespan upper-bound $\overline{\text{makespan}}$ for the task subset $T^{\text{src},c}$ on $\dot{m}_c^h$ cores;

Then, $\forall (M^{\text{src}}, M^{\text{dst}}) \in \mathcal{G}, k \in [1, \dots, \phi]$:

1. The missing configurations for π^k-clusters are all the configurations $\theta_c \in \Theta_k$ such that $\dot{m}_c^- > 0$, where $\dot{m}_c^- \doteq \max(0, \dot{m}_c^{\text{dst}} - \dot{m}_c^{\text{src}})$. Symmetrically, the excess configurations are all the $\theta_c \in \Theta_k$ such that $\dot{m}_c^+ > 0$, where the value $\dot{m}_c^+$ is defined by $\dot{m}_c^+ \doteq \max(0, \dot{m}_c^{\text{src}} - \dot{m}_c^{\text{dst}})$.
2. To perform the cluster reconfiguration, we use a vector containing missing configurations. This vector will be used at run-time to track the required configurations. Store $\tilde{\theta}^- \doteq < \theta_c, \theta_{c'}, \dots >$ a vector of missing configurations. $\tilde{\theta}^-$ contains $\dot{m}_c^-$ times the configuration θ_c. $\tilde{\theta}^-$ is ordered by reconfiguration delay, in decreasing order.
3. Symmetrically, store $\tilde{\theta}^+ \doteq < \theta_c, \theta_{c'}, \dots >$ a vector of excess configurations. $\tilde{\theta}^+$ contains $\dot{m}_c^+$ times the configuration θ_c. $\tilde{\theta}^+$ is ordered by the cluster's makespan of the cores configured in $\theta_c \in \tilde{\theta}^+$, in increasing order.
4. Bind the j^{th} longest reconfiguration to the j^{th} idle core. The j^{th} longest reconfiguration θ_c is determined using its reconfiguration delay δ_c. The j^{th} idle core is the j^{th} core to become idle on this cluster during this mode change phase. To do so, define the assignment $\{\theta_c, \theta_{c'}\}$, where θ_c is the j^{th} element of $\tilde{\theta}^+$ and $\theta_{c'}$ is the j^{th} of $\tilde{\theta}^-$.
5. Store the assignment $\{\theta_c, \theta_{c'}\}$ in the multi-set $\text{RT}^{\text{src,dst}}$.

The results will be used at run-time. Because a cluster is formed by cores of the same type and configured identically the required reconfigurations for the θ_c-cluster are contained in the multi-set $\{\theta_c, \theta_{c'}\} \in \text{RT}^{\text{src,dst}}$.

12.2.2 Run-time: scheduling and reconfiguring

At run-time, the protocol must ensure for all the clusters the successful execution of the rem-jobs and core reconfigurations as computed in the offline phase. It must at last enable the new mode.

To do so, it will first ensure the execution of the rem-jobs using the same scheduler as before. During the execution of the rem-jobs, when a core becomes idle, it may be reconfigured. For each θ_c-cluster it takes as input the list of remaining rem-jobs J and the required reconfigurations to make $\{\theta_c, \theta_{c'}\} \in \mathrm{RT}^{\mathrm{src,dst}}$.

Formally,

- Schedule J using the same scheduler as before;
- When a core becomes idle, reconfigure it to the longest required reconfiguration $\theta_{c'}$ that hasn't been performed yet, where $\{\theta_c, \theta_{c'}\} \in \mathrm{RT}^{\mathrm{src,dst}}$.

Whenever $\dot{m}^{\mathrm{dst}}_{c'}$ cores are reconfigured to $\theta_{c'}$, the $\theta_{c'}$-cluster of the new mode M^{dst} is formed and its task set $T^{c',\mathrm{dst}}$ is enabled. Several clusters may have cores reconfigured in $\theta_{c'}$ and merged to form this new $\theta_{c'}$-cluster. This *Mode Change Phase* ends once all the rem-jobs have been completed and the required reconfigured performed. The old mode M^{src} is then disabled, and the new mode M^{dst} is enabled. A more formal version is presented in Appendix A.1.

12.2.3 Note on the offline phase

The offline phase could be lighter. Indeed, the matching made in Step 4 is not strictly necessary and could be easily done at run-time, with very few extra cost. However, it makes the worst-case of every mode change phase very predictable, and the results will be used in the computation of the upper-bound on the mode change phase duration, presented in Section 12.3.

12.2.4 Mode change phase example

Figure 12.1 depicts an example of a *Mode Change Phase* handled by ACCEPTOR. The platform is composed of 4 processors. Cores π_1 and π_2 have the same type π^1 and cores π_3 and π_4 have the same type π^2. Two cores must be configured in θ_1 for mode M^{src}

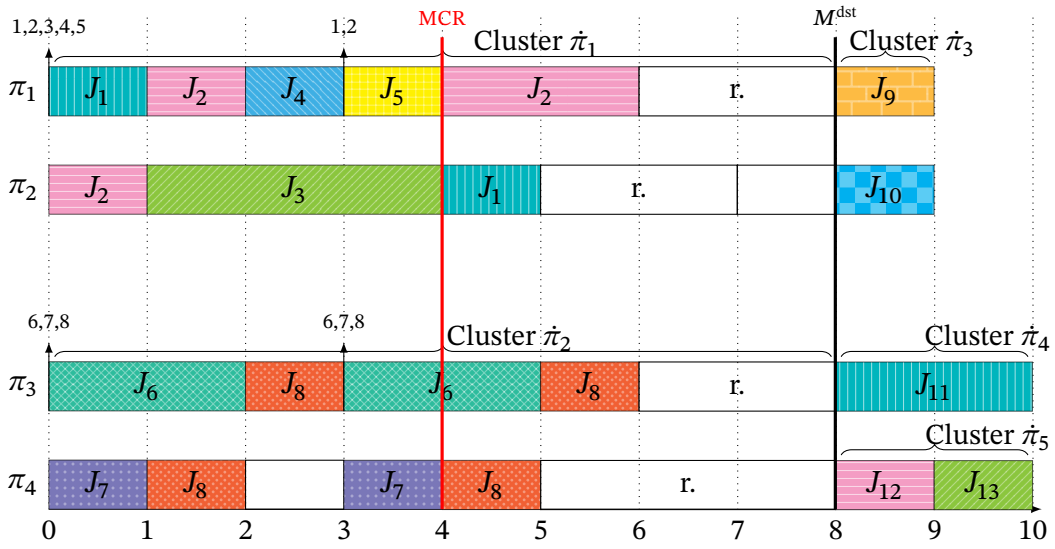


Figure 12.1: ACCEPTOR protocol illustration

and two cores must be configured in θ_2 for mode M^{src} : $\tilde{\Theta}^{src} =< 2, 2, 0, 0, 0 >$. At $t = 0$, mode M^{src} is active. Thus, there are two clusters: cluster $\dot{\pi}_1$ and cluster $\dot{\pi}_2$. Cluster $\dot{\pi}_1$ has two cores configured in θ_1, so $\dot{m}_1^{src} = 2$. Cluster $\dot{\pi}_2$ has two cores configured in θ_2, so $\dot{m}_2^{src} = 2$. For mode M^{dst}: $\tilde{\Theta}^{dst} =< 0, 0, 2, 1, 1 >$. This means that there will be three clusters in this mode, because there are three different required configurations. Here are some of the different reconfigurations delay: $\delta_3 = 2$, $\delta_4 = 3$, $\delta_5 = 2$. The real-time constraint for mode M^{dst} is $\Delta^{dst} = 10$.

The protocol computes the required configuration per core type. For example, for the π^2-cores, $\tilde{\theta}^+ =< \theta_2, \theta_2 >$ and $\tilde{\theta}^- =< \theta_5, \theta_4 >$. This means that the protocol needs to reconfigure the cores configured in θ_2, one in θ_4 and one in θ_5 The similar computation for π^1-cores indicates that the π^1-cores must be reconfigured into θ_3.

Cluster $\dot{\pi}_1$ and cluster $\dot{\pi}_2$ both have a specific task set, respectively $T^{src,1}$ and $T^{src,2}$. $\tau_1 \in T^{src,1}$ and $\tau_2 \in T^{src,1}$ release a job at $t = 3$, which become rem-jobs at $t = 4$ when the mode change request is received. The same goes for τ_6, τ_7 and $\tau_8 \in T^{src,2}$.

Core π_4 becomes idle at $t = 5$. There are no other rem-job to execute in this cluster, which leads the protocol to reconfigure it. The cluster $\dot{\pi}_2$ has the following reconfigurations to make: $(\theta_2, \theta_4), (\theta_2, \theta_5)$. Because $\delta_5 > \delta_4$, π_4 is reconfigured to θ_5. This reconfiguration ends at $t = 8$. Because $\dot{m}_5^{dst} = 1$, a cluster of the new mode is formed and its task set $T^{dst,5} = \{\tau_{12}, \tau_{13}\}$ is activated. On the other hand, even if π_2 has been reconfigured at $t = 7$, because $\dot{m}_3^{dst} = 2$: the cluster is not formed yet. At $t = 8$, all the

new mode's clusters have been formed. The mode M^{dst} can then be enabled. The mode change phase lasts for $8 - 4 \leq \Delta^{\text{dst}}$ units of time, and all the rem-jobs were successfully scheduled so the mode change phase succeeds.

12.3 The upper-bound $\overline{\text{reconfigured}}$

To assert the validity of a given system using the ACCEPTOR protocol, we now introduce an upper-bound of the instant in which the rem-jobs of a cluster are completed, and the required reconfigurations are done. This instant is denoted as $\overline{\text{reconfigured}}$, and can be immediately derived to obtain an upper-bound on the duration of any mode change phase. It will be used to demonstrate the validity of the protocol in Section 12.4. We first use an existing upper-bound on the Idle_j instant, defined in Definition 2.50.

Lemma 12.2 (Lemma 2.11 in [3])**.** Suppose that J is ordered by non-decreasing job processing times, i.e. , $c_1 \leq c_2 \leq \cdots \leq c_n$, all starting at $t = 0$. Suppose that no reconfigurations are performed on none of the m' cores. Then, whatever the job priority assignment made by a FJP work-conserving global scheduler, an upper-bound $\overline{\text{Idle}}_j$ on the Idle_j instant, where $1 \leq j \leq m'$, is given by:

$$\overline{\text{Idle}}_j \doteq \begin{cases} c_j & \text{if } n = m' \\ \\ \frac{\sum_{i=1}^{n} c_i + (j-1) \times c_{n-m'+j}}{m'} & \text{otherwise} \end{cases} \tag{12.2}$$

The new upper-bound $\overline{\text{reconfigured}}$ is based on $\overline{\text{Idle}}_j$. It is the maximal sum of idle time plus reconfiguration delay, for any core of the cluster.

Corollary 12.1 (Upper-bound $\overline{\text{reconfigured}}$)**.** Suppose that J is a list of jobs released by $T^{\text{src},c}$, and that J is ordered by non-decreasing job processing times, i.e. , $c_1 \leq c_2 \leq \cdots \leq c_n$, all starting at $t = 0$. Suppose that the required reconfiguration delays are in the vector δ, ordered by reconfiguration time in decreasing order. Suppose that the cluster is composed of m' cores. Then, whatever the job priority assignment made by a FJP work-conserving global scheduler, an upper-bound of the instant in which the

cluster's rem-jobs are completed, and the required reconfigurations are done is given by:

$$\overline{\mathsf{reconfigured}}(T^{\mathrm{src},c}, m', \delta) \doteq \max_{j=1}^{m'}\{\overline{\mathsf{Idle}}_j + \delta_j\} \tag{12.3}$$

Proof. The proof is trivially obtained by construction. □

From now on, $\overline{\mathsf{reconfigured}}(T^{\mathrm{src},c}, m', \delta)$ will be denoted as $\overline{\mathsf{reconfigured}}$ for readability.

12.4 Validity test

A validity test is used to determine whether the mode change phases of an application will be successfully managed by a multi-mode protocol, on a given platform. To ensure this, we need to verify that every deadline will be met during the transition phase, and that the new mode will always be timely enabled.

The notion of C-sustainability from Definition 2.49 will be useful to prove that if a task set can be correctly scheduled by S, S can also correctly schedule rem-jobs generated by this task set. Lemma 12.3 (Corollary 8 from [46]) will be used to show that any scheduler used in the protocol will be C-sustainable, according to the hypotheses. This lemma applies to any uniform multi-processor platform, and thus to identical multi-processor platform. Therefore, we consider in the following that the job set J is composed of one job of each task, as a worst-case.

Lemma 12.3 (Corollary 8 from [46])**.** Any work-conserving and Fixed Job Priority (FJP) algorithm is C-sustainable on uniform multi-processor platforms.

The schedulability of the rem-jobs during mode change phase is ensured by the Lemma 12.4.

Lemma 12.4 (Rem-job's schedulability)**.** Every rem-job's deadline of every cluster will be respected during every possible mode change phase, if and only if (i) the scheduler is a clustered, FJP, preemptive, work-conserving scheduler and (ii) $T^{\mathrm{src},c}$ is schedulable by the scheduler upon $\dot{m}_c^{\mathrm{src}}$ cores configured in θ_c.

Proof. Because of (i) and Lemma 12.3, we know that the scheduler is C-sustainable. Hence, the scheduler is C-sustainable and (ii): the rem-jobs of the task set $T^{\text{src},c}$ are schedulable on their cluster. By construction, for any rem-job J_i, its remaining execution time is lesser than C_i. Thus, the schedulability is ensured. □

Lemma 12.5 ensures that the real-time constraint of each mode will be respected during every possible transition.

Lemma 12.5 (Transition time constraint)**.** The protocol will respect the time constraint Δ^{dst} if and only if $\forall$ src, dst, $(M^{\text{src}}, M^{\text{dst}}) \in \mathcal{G}$, $\forall \theta_c$-cluster, with $\delta_c^{\text{src,dst}}$ being the required reconfiguration delays ordered in decreasing order:

$$\Delta^{\text{dst}} \geq \max_{\theta_c\text{-cluster}} \overline{\text{reconfigured}}$$

Proof. The right part of the inequation computes the maximal upper-bound $\overline{\text{reconfigured}}$. By construction, if the hypothesis is respected, every cluster of each possible transition will be reconfigured when required. □

Theorem 12.2 proves the validity of our protocol, under its hypotheses.

Theorem 12.2 (Validity of ACCEPTOR)**.** The ACCEPTOR protocol is valid if and only if (i) the scheduler is a clustered, FJP, preemptive, work-conserving scheduler , (ii) $T^{\text{src},c}$ is schedulable by the scheduler upon $\dot{m}_c^{\text{src}}$ cores configured in θ_c and (iii) $\forall$ src, dst, $(M^{\text{src}}, M^{\text{dst}}) \in \mathcal{G}$, $\forall \theta_c$-cluster, with $\delta_c^{\text{src,dst}}$ being the required reconfiguration delays ordered in decreasing order:

$$\Delta^{\text{dst}} \geq \max_{\theta_c\text{-cluster}} \overline{\text{reconfigured}}$$

Proof. To prove the validity of the protocol, we must prove that every rem-job's deadline will be respected during the mode change phase and that the real-time constraint Δ^{dst} will be respected during each possible transition from every mode M^{src} to mode M^{dst}.

- Because of the hypotheses (i–ii), Lemma 12.4 can be applied and thus prove that every job's deadline will be respected during the mode change phase.

- Because of the hypotheses (i–iii), Lemma 12.5 can be applied. Every real-time constraint will be therefore respected.

Because the job's deadlines and the real-time constraint will be respected, the protocol is valid when the hypotheses (i–iii) are respected. □

12.5 Evaluation: Time complexity

We start the evaluation of the protocol with a theoretical worst-case time complexity analysis. We evaluate the theoretical worst-case time complexity of both the offline and run-time phases of the protocol.

Concerning the offline phase: the computation of $\overline{\mathsf{Idle}}_j$ has a worst-case time complexity of $O(n)$, as $\overline{\mathsf{makespan}}$. First, the protocol computes m' makespan upper-bounds, where $m' \leq m$ is the number of clusters and $n' \leq n$ the largest number of tasks that a cluster has to schedule, hence a complexity of $O(m' \times n') = O(m \times n)$. Steps 2 and 3 performed on clusters contain a sort on at most m elements, and Step 3 also computes $\overline{\mathsf{Idle}}_j$ for at most m cores. Thus, complexity of Step 2 is $O(m \times \log(m))$ and complexity of Step 3 is $O(m \times \log(m) + m) = O(m \times \log(m))$. The total complexity of the offline phase is thus $O(m \times \log(m) + m \times n)$.

Concerning the run-time phase, it has the same complexity as the scheduler. Choosing the reconfiguration to perform can be made in $O(1)$ with an efficient choice of data structure.

The complexity of the upper-bound $\overline{\mathsf{reconfigured}}$ is straightforward. Computing once the first sum of C_i has a worst-case time complexity of $O(n)$. Because it is constant for any value of j, with a given task set, it may be computed once. Thus, once computed, the worst-case complexity of the computation can be obtained in $O(1)$. $\overline{\mathsf{reconfigured}}$ computes m $\overline{\mathsf{Idle}}_j$, and the total worst-case time complexity is thus $O(m \times 1 + n) = O(m + n)$.

Complexities for both offline and run-time phases are very low, and make the protocol easily scalable.

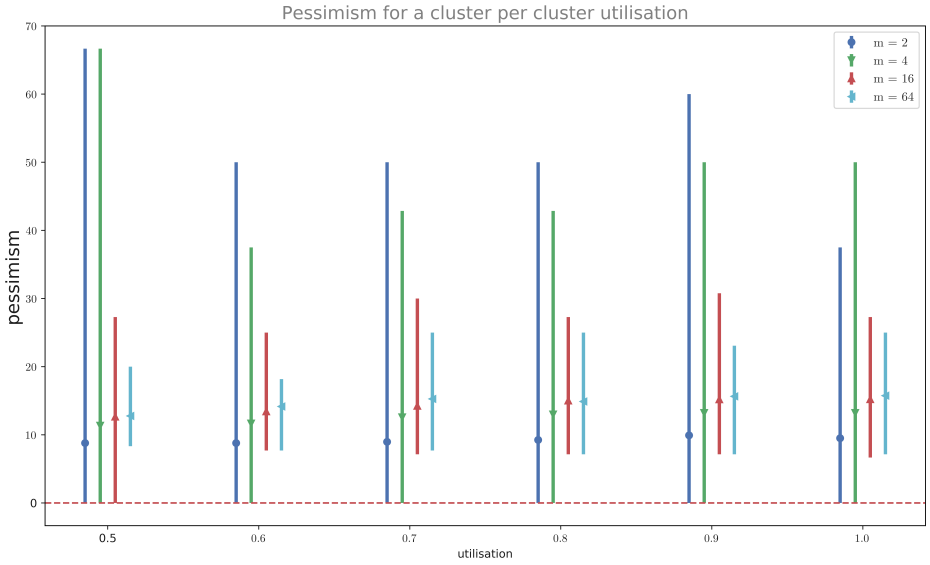


Figure 12.2: Pessimism of $\overline{\text{reconfigured}}$

12.6 Evaluation: empirical pessimism of $\overline{\text{reconfigured}}$

We continue the evaluation with the use of simulations to evaluate the pessimism of the bound introduced in Section 12.3.

To the best of our knowledge, the upper-bound $\overline{\text{idle}}_j$ pessimism has not been evaluated, neither theoretically nor empirically. We provided here an empirical evaluation of $\overline{\text{reconfigured}}$, which is based on $\overline{\text{idle}}_j$. We measure the pessimism ratio $\frac{\text{upper-bound}}{\text{mode change time}}$, where the *mode change time* is a measurement of the simulated mode change phase, and where the *upper-bound* corresponds to the computation of $\overline{\text{reconfigured}}$ for a given cluster.

The experiment has been conducted for clusters with $m' = 2, 4, 16$ and 64 cores. The reconfiguration times are uniformly chosen in the range $[0, 10]$, where 0 indicates that no reconfiguration is required. We generate approximately 1 000 feasible task sets with utilisations $\in (p - 0.1, p]$, where we increase p from 0.5 to 1.0 in steps of 0.1. The deadlines are uniformly chosen in the range $[2, 10]$. As a scheduler, we use Global-RM and remove any non-schedulable task set from our experiments. For those experiments, time is discrete.

The results are displayed in Figure 12.2 and can be read this way: the average pessimism

is 14% for a cluster with $m' = 16$ and a task set utilisation of 0.7. This means that the upper-bound is (on average) 14% larger than the actual duration in this configuration. The maximal observed pessimism is 30% and the minimal is 8%. The pessimism is correlated to both the utilisation and the number of cores. A higher utilisation leads to a higher (but bounded) pessimism, and so does a higher number of core in a platform. Also, the minimal and maximal pessimism tend to be closer and closer to the average with a higher number of cores. You may note that the number of cores per cluster is quadrupled between each simulations. However, the pessimism is only slightly increased every time the number of cores is quadrupled once $m' > 1$. In our experiment, the average pessimism is empirically bounded.

These experimental results show that the upper-bound $\overline{\text{reconfigured}}$ is accurate and can be used for clusters having a large amount of cores or a high utilisation.

12.7 Evaluation: Competitive analysis of ACCEPTOR

After the empirical evaluation of the pessimism of the upper-bound $\overline{\text{reconfigured}}$, we now evaluate the competitive analysis of the protocol itself.

Definition 12.1 (λ-competitive protocol)**.** A protocol is said to be λ-competitive if it takes at most $\lambda \times O$ time to complete a mode change phase, where O is the optimal mode change phase duration.

To perform the competitive analysis of the protocol, we search for a worst-case scenario, i.e. a system for which the relative difference between our protocol's performance and an optimal one is the highest. We first define what a *squeezable* system is, and then prove several properties on those systems, which leads to prove that they correspond to the worst-case. Once we have this worst-case, we compute the competitiveness of the protocol. An example of a *squeezable* system is depicted in Figure 12.3. The top graph shows a specific mode change phase handled by ACCEPTOR, and below is the same mode change phase handled by an optimal one.

After the introduction of several definitions and notations, we perform the competitive analysis.

12.7.1 Preliminary definitions and notations

We now introduce several definitions. The first ones concern different aspect of the idle time. As defined in Definition 2.50 on page 31, an Idle_j instant is the earliest instant such that at least j cores are idle. A formal definition is given below.

Definition 12.2 ($\mathsf{Idle}_j(J, m')$)**.** $\mathsf{Idle}_j(J, m')$ is the earliest instant such as at least j cores may be idle when the job set J is scheduled by any work-conserving scheduler upon a cluster composed of m' cores,

Unlike $\overline{\mathsf{Idle}}_j$, this value is the exact instant. We also introduce the following new notations:

Definition 12.3 ($\mathsf{Idle}_{\max}(S)$)**.** $\mathsf{Idle}_{\max}(S)$ is the length of the longest period during which a core is idle when a mode change phase of a system S is handled by ACCEPTOR.

We also introduce the notion of *excess idle*, denoted Idle_+.

Definition 12.4 (Idle_+)**.** Idle_+ is the extra duration where cores are idle during a mode change with our protocol, in comparison to an optimal protocol.

For example, in Figure 12.3 $\mathsf{Idle}_+ = 2 \times 6 = 12$.

To increase the readability, we now introduce two notations.

Definition 12.5 ($|\mathrm{US}(S)|$)**.** The required time to handle the mode change phase of the system S, in the worst-case were all the jobs are released at t_{MCR}, when using ACCEPTOR is denoted as $|\mathrm{US}(S)|$.

Definition 12.6 ($|\mathrm{OPT}(S)|$)**.** The required time to handle the mode change phase of the system S, in the worst-case were all the jobs are released at t_{MCR}, when using an optimal protocol for this given system S is denoted as $|\mathrm{OPT}(S)|$.

Please note that an optimal protocol doesn't have the same constraints as ACCEPTOR. For example, it doesn't have to schedule the rem-jobs using a work-conserving scheduler.

Finally, we define the notion of *squeezable* system, in the context of a mode change transition.

Definition 12.7 (*Squeezable*)**.** A system S is said to be *squeezable* if and only if S has only one required reconfiguration θ_c —with a reconfiguration delay of δ_c— to perform during the mode change transition and a set of rem-jobs J on a cluster having m' cores such that:

i $\sum C_i = \delta_c \times (m' - 1)$,

ii $\mathsf{Idle}_1(J, m') = \mathsf{Idle}_2(J, m') = \cdots = \mathsf{Idle}_{m'}(J, m') = \delta_c \times \frac{m'-1}{m'}$,

iii $\mathsf{Idle}_1(J, m' - 1) = \mathsf{Idle}_2(J, m' - 1) = \cdots = \mathsf{Idle}_{m'-1}(J, m' - 1) = \delta_c$,

iv $\forall i, d_i \geq \delta_c$.

This notions will be heavily used in the following lemmas, corollaries and theorems, see Figure 12.3 for an illustration.

12.7.2 Competitive analysis

Here is an overview of the competitive analysis. Lemma 12.7 and Lemmas 12.8–12.10 introduce several properties that stand for any system. Lemma 12.6, Corollary 12.3 and Lemma 12.11 use both properties on *squeezable* systems and the global properties to prove that squeezable systems are a worst-case in Corollary 12.4. Using the worst-case, Theorem 12.5 states that ACCEPTOR is $\frac{2m'-1}{m'}$-competitive on a cluster composed of m' cores. Then, Corollary 12.6 proves that ACCEPTOR is 2-competitive.

In the following, we assume that any system S has the required reconfigurations $\tilde{\Theta}$ to perform, and a set of rem-jobs J to execute on a cluster having m' cores during any mode change. We also need to know the duration of the longest element, whether it is a job execution or a reconfiguration. Formally, we denote for such a system S the longest atomic duration —either the WCET of a job or a reconfiguration delay— the value $C \doteq \max\{\max_{i=1}^{n}\{C_i\}, \max_{\theta_c \in \tilde{\Theta}}\{\delta_c\}\}$. This value is by definition strictly positive.

Lemma 12.6. Any optimal mode change of any *squeezable* system S may be handled in δ_c time-units. Formally, $|\mathrm{OPT}(S)| = \delta_c = C$.

Proof. The rem-jobs can all be scheduled on $m' - 1$ cores, and one core can be reconfigured at $t = 0$. The rem-jobs will be complete at $t = \mathsf{Idle}_{m'-1}(J, m' - 1) = \delta_c$ and the reconfiguration will be done at $t = \delta_c$. Because of iv), the deadlines will be

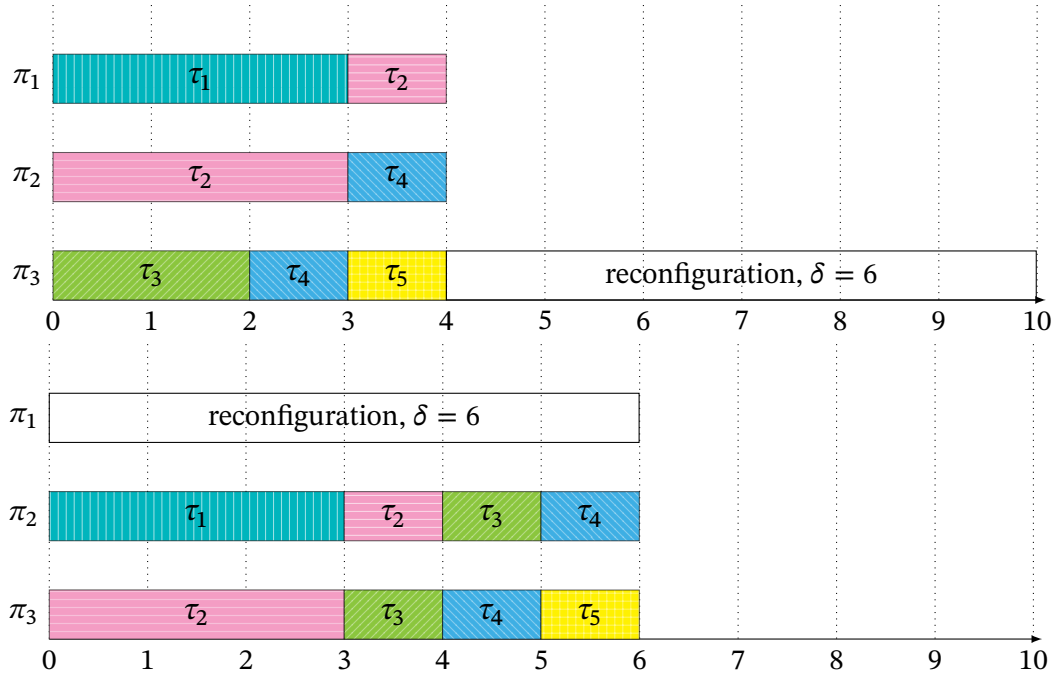


Figure 12.3: A *squeezable* system with $m' = 3$ and $n = 5$

respected. By construction, $\delta_c = C$. The lemma follows. This lemma is illustrated in Figure 12.3. □

Lemma 12.7. Any optimal mode change of any system S takes more than C time-units. Formally, $|\text{OPT}(S)| \geq C$.

Proof. No intra-parallelism is allowed, so the system S may not last less than the longest element to schedule, whether it is a job or a reconfiguration. □

Corollary 12.3. For any mode change of any system S, the ratio $\frac{C}{|\text{OPT}(S)|}$ is maximised when $|\text{OPT}(S)| = C$.

Proof. This is a direct implication of Lemma 12.7. □

Lemma 12.8. During any mode change handled by ACCEPTOR of any system S, a core cannot be idle longer than C time-units. Formally, $\text{Idle}_{\max}(S) \leq C$.

Proof. ACCEPTOR uses a work-conserving scheduler. Thus, a core may be idle only if the number of jobs and reconfigurations waiting to be executed or performed is lower than the number of cores. It immediately follows that all the reconfigurations and jobs will be completed at most C time-units later. The lemma follows. □

Lemma 12.9. For any mode change of any system S, $|\text{US}(S)| = |\text{OPT}(S)| + \frac{\text{Idle}_+}{m'}$.

Proof. We introduce here the notion of Idle_{opt}. The value Idle_{opt} is the amount of idle time with an optimal protocol. It stands that:

$$m' \times |\text{OPT}(S)| = \sum_{i=1}^{n} C_i + \sum_{\theta_c \in \tilde{\Theta}} \delta_c + \text{Idle}_{\text{opt}}$$

Informally, this equation represents the sum of the work to perform plus the idle time. Because Idle_{opt} is the amount of idle time with an *optimal* protocol it is incompressible. By definition of the excess idle, the following stands as well:

$$m' \times |\text{US}(S)| = \sum_{i=1}^{n} C_i + \sum_{\theta_c \in \tilde{\Theta}} \delta_c + \text{Idle}_{\text{opt}} + \text{Idle}_+$$

With the two previous equations, we trivially have that:

$$|\text{US}(S)| = \frac{\sum_{i=1}^{n} C_i + \sum_{\theta_c \in \tilde{\Theta}} \delta_c + \text{Idle}_{\text{opt}} + \text{Idle}_+}{m'} = |\text{OPT}(S)| + \frac{\text{Idle}_+}{m'}$$

□

Lemma 12.10. For any mode change of any system S handled by ACCEPTOR,

$$\text{Idle}_+ \leq (m' - 1) \times C$$

Proof. By definition, the excess idle is smaller than the sum of the length of all the periods where a core is idle. At most $m' - 1$ cores will be idle, because at least one core will never be idle. Therefore:

$$\begin{aligned} \text{Idle}_+ &\leq (m' - 1) \times \text{Idle}_{\max}(S) \\ &\leq (m' - 1) \times C \qquad \text{(Lemma 12.8)} \end{aligned}$$

The lemma follows. □

Lemma 12.11. For any mode change of any system S handled by ACCEPTOR, when $\frac{C}{|\text{OPT}(S)|}$ is maximised, $\frac{|\text{US}(S)|}{|\text{OPT}(S)|}$ is maximised as well.

Proof.

$$
\begin{aligned}
&|\mathrm{US}(S)| = |\mathrm{OPT}(S)| + \frac{\mathsf{Idle}_+}{m'} && \text{(Lemma 12.9)}\\
\Rightarrow &\frac{|\mathrm{US}(S)|}{|\mathrm{OPT}(S)|} = \frac{|\mathrm{OPT}(S)| + \frac{\mathsf{Idle}_+}{m'}}{|\mathrm{OPT}(S)|}\\
\Rightarrow &\frac{|\mathrm{US}(S)|}{|\mathrm{OPT}(S)|} \leq \frac{|\mathrm{OPT}(S)| + C \times \frac{m'-1}{m'}}{|\mathrm{OPT}(S)|} && \text{(Lemma 12.10)}\\
\Rightarrow &\frac{|\mathrm{US}(S)|}{|\mathrm{OPT}(S)|} \leq 1 + \frac{C}{|\mathrm{OPT}(S)|} \times \frac{m'-1}{m'}
\end{aligned}
$$

Given a system S, the only variable here is the ratio $\frac{C}{|\mathrm{OPT}(S)|}$. The lemma follows. □

Corollary 12.4. When handling any mode change of any *squeezable* system S with ACCEPTOR, the ratio $\frac{|\mathrm{US}(S)|}{|\mathrm{OPT}(S)|}$ is maximised. Thus, any *squeezable* system is a worst-case of ACCEPTOR.

Proof. Lemma 12.6 and Corollary 12.3 states that this corollary's hypotheses leads to a maximal ratio $\frac{C}{|\mathrm{OPT}(S)|}$ and Lemma 12.11 shows that maximising the ratio $\frac{C}{|\mathrm{OPT}(S)|}$ is equivalent to maximising the ratio $\frac{|\mathrm{US}(S)|}{|\mathrm{OPT}(S)|}$. Thus, this corollary follows. □

Theorem 12.5. For any system composed of m' cores, the ACCEPTOR protocol is $\frac{2m'-1}{m'}$-competitive.

Proof. To prove that ACCEPTOR is $\frac{2m'-1}{m'}$-competitive, we have to prove that for any system S, the ratio $\frac{|\mathrm{US}(S)|}{|\mathrm{OPT}(S)|} \leq \frac{2m'-1}{m'}$. The Corollary 12.4 states that the maximal ratio will be obtained on any *squeezable* system S. On such a system, we know that:

$$
\begin{aligned}
\frac{|\mathrm{US}(S)|}{|\mathrm{OPT}(S)|} &= \frac{\mathsf{Idle}_1(J, m') + \delta_c}{\delta_c}\\
&= \frac{\delta_c \times \frac{m'-1}{m'} + \delta_c}{\delta_c}\\
&= \frac{m'-1}{m'} + 1\\
&= \frac{2m'-1}{m'}
\end{aligned}
$$

Thus, ACCEPTOR takes at most $\frac{2m'-1}{m'}$ of the required time than an optimal protocol. The theorem follows. □

Figure 12.3 shows an example of such a system with $m' = 3$, $n = 5$ and $\delta_c = 6$.

Corollary 12.6. For any system composed of m' cores, the ACCEPTOR protocol is 2-competitive. This upper-bound is tight.

Proof. Theorem 12.5 states that ACCEPTOR is $\frac{2m'-1}{m'}$-competitive. It stands that

$$\forall m' > 0, \frac{2m' - 1}{m'} < 2$$

Thus, ACCEPTOR is 2-competitive. This upper-bound is tight, because of the following limit:

$$\lim_{m' \to \infty} \frac{2m' - 1}{m'} = 2$$

□

12.8 Handling mode independent tasks

In our model, all tasks are mode dependent. Real-time applications may have mode independent tasks: e.g. tasks that run during the whole lifespan of the application such as the OS. Our model 'as is' does not permit such tasks to run. However, a trivial extension with no effect on the validity nor the complexity would be to restrain those tasks to a specific cluster with no reconfiguration allowed. Still, mode dependent tasks could run on such a cluster and because of Lemma 12.3, their rem-jobs would be correctly scheduled.

Doing so removes the limitation in a very easy way and makes the model usable for real-world applications.

12.9 Improving the upper-bound $\overline{\text{reconfigured}}$

A way to improve this protocol usability is to improve the upper-bound $\overline{\text{reconfigured}}$. By reducing the pessimism of this bound, it can improve the acceptance rate of the protocol. In this section, we will explore one possible approach, and evaluate its efficiency.

12.9.1 Changing the idle upper-bound

The upper-bound $\overline{\text{reconfigured}}$ is heavily based on the chosen idle upper-bound. As a reminder:

$$\overline{\text{reconfigured}} \doteq \max_{j=1}^{m'}\{\overline{\text{Idle}}_j + \delta_j\} \tag{12.4}$$

A way to reduce the upper-bound is to use different and better idle upper-bound. [3] presents several upper-bounds, and especially this one:

Lemma 12.12 (Lemma 2.15 in [3])**.** Suppose that J is ordered by non-decreasing job processing times, i.e. , $c_1 \leq c_2 \leq \cdots \leq c_n$, all starting at $t = 0$. Suppose that no reconfigurations are performed on none of the m' cores. Then, whatever the job priority assignment made by a FJP work-conserving global scheduler, an upper-bound $\overline{\text{Idle}'}_j$ on the Idle_j instant, where $1 \leq j \leq m'$, is given by:

$$\overline{\text{Idle}'}_j \doteq \frac{\sum_{i=1}^{n} c_i + \sum_{i=1}^{j-1} \underline{\text{idle}}_i}{m} \tag{12.5}$$

where

$$\underline{\text{Idle}}_j \doteq \frac{\sum_{i=1}^{n-m+j} c_i}{m} \tag{12.6}$$

This other upper-bound is theoretically incomparable with the previous one. Therefore, we decided to try it empirically by using it in the alternative upper-bound $\overline{\text{reconfigured}'}$:

$$\overline{\text{reconfigured}'} \doteq \max_{j=1}^{m}\{\overline{\text{Idle}'}_j + \delta_j\} \tag{12.7}$$

12.9.2 Time complexity

The theoretical complexity of this bound is slightly higher. As stated in Section 12.5, the computation of $\overline{\text{Idle}}_j$ has a worst-case time complexity of $O(n)$, but it may be reduced for several computations. This drops the total worst-case time complexity of $\overline{\text{reconfigured}}$ to $O(m+n)$.

$\underline{\text{Idle}}_j$ has a complexity of $O(n)$. $\overline{\text{Idle}'}_j$ computes up to m $\underline{\text{Idle}}_j$, and the sum of C_i may be computed once per task set, with a complexity of $O(n)$. Thus, the complexity of $\overline{\text{Idle}'}_j$

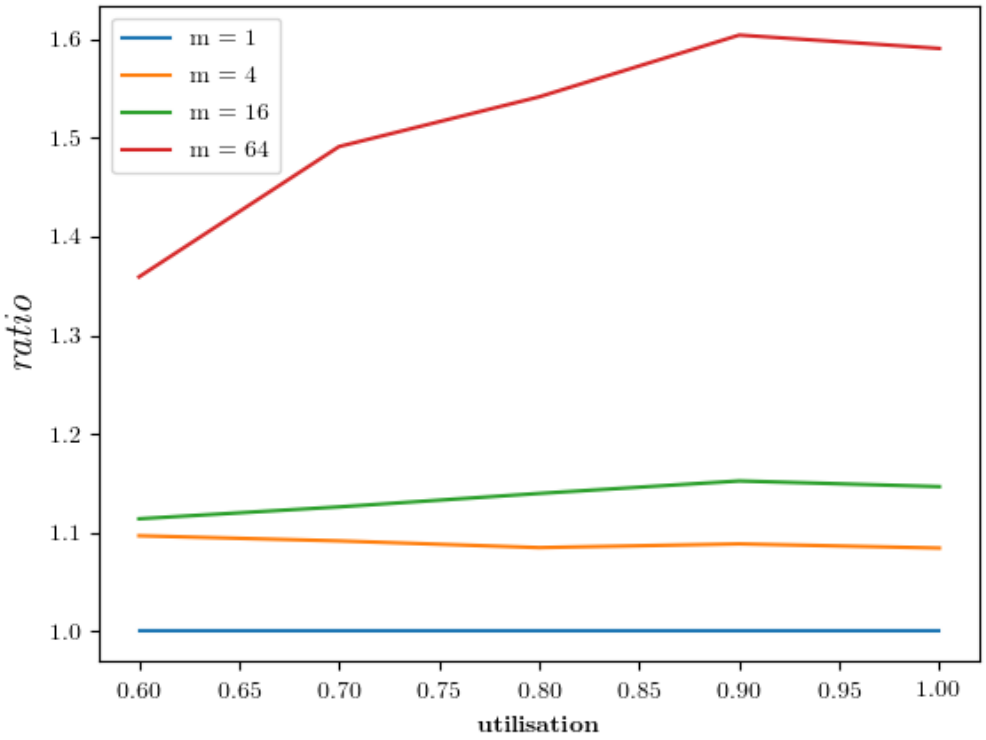


Figure 12.4: Average ratio $\frac{\overline{\text{reconfigured}'}}{\overline{\text{reconfigured}}}$

is $O(m \times n + n) = O(m \times n)$. Computing m times this complexity gives a worst-case time complexity of $O(m^2 \times n)$ for $\overline{\text{reconfigured}'}$.

The time worst-case complexity of $\overline{\text{reconfigured}'}$ is higher than the one of $\overline{\text{reconfigured}}$, it remains however low.

12.9.3 Evaluation

As it is theoretically incomparable, we have conducted simulations to observe how this new bound was performing compared to the previous one, in the same simulation environment as in Section 12.6.

We use here the same evaluations parameters as in Section 12.6. The experiment has been conducted for clusters with $m' = 1, 4, 16$ and 64 cores. The reconfiguration times are uniformly chosen in the range $[0, 10]$, where 0 indicates that no reconfiguration is required. We generate approximately 1 000 feasible task sets with utilisations $\in (p - 0.1, p]$, where we increase p from 0.6 to 1.0 in steps of 0.1. The deadlines are

uniformly chosen in the range $[2, 10]$. As a scheduler, we use Global-RM and remove any non-schedulable task set from our experiments. For those experiments, time is discrete. We measured the ratio $\frac{\overline{\text{reconfigured}'}}{\overline{\text{reconfigured}}}$. The results are displayed in Figure 12.4 and can be read this way: the average ratio $\frac{\overline{\text{reconfigured}'}}{\overline{\text{reconfigured}}}$ is 1.1 for a cluster with $m' = 4$ and a task set utilisation of 0.9. This means that the upper-bound $\overline{\text{reconfigured}'}$ is (on average) 10% larger than the upper-bound $\overline{\text{reconfigured}}$.

For the specific case of clusters with only one processor, we see that both upper-bound are equivalent. This is because both $\overline{\text{Idle}}$ and $\overline{\text{Idle}'}$ are exact in this case. For any other value, the upper-bound $\overline{\text{reconfigured}'}$ is greater than $\overline{\text{reconfigured}}$, and thus less precise. Moreover, this looseness increases the more cores there is in a cluster, and the higher the utilisation is.

12.9.4 Last words about $\overline{\text{reconfigured}'}$

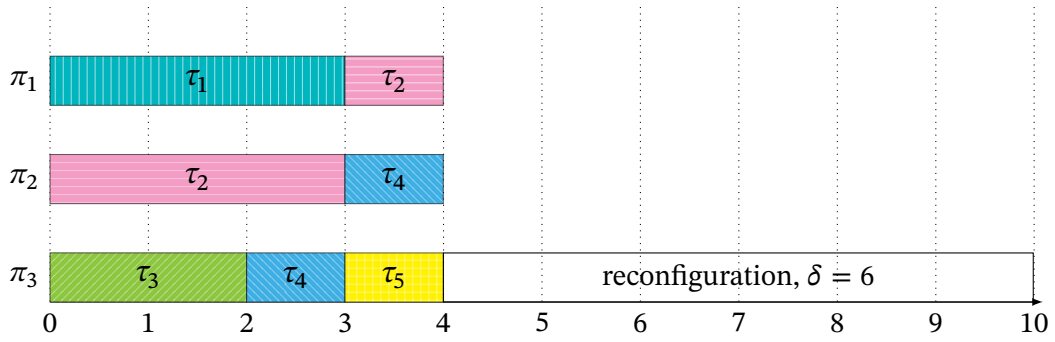
This new bound was proved to be less precise than the other one. Therefore, it has no use in practice. However, it shows the available possibilities to improve the whole protocol by changing only a part of it. Further researches could be led this way, with a positive results.

Chapter 13

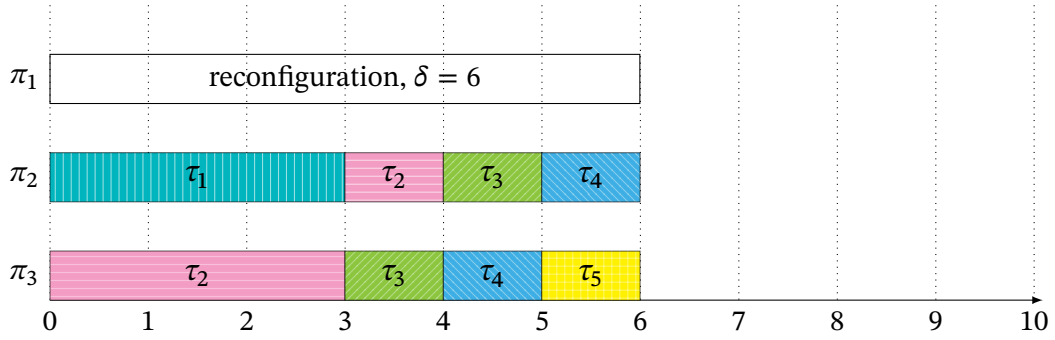
SQUARER

In this chapter, we propose another protocol for multi-mode applications, where both the hardware may be reconfigured and the software may be changed. This protocol is called SQUARER, which stands for SQUaring AsynchRonous clustEr-based pRotocol. SQUARER aims at reducing the duration of mode change phases and to propose an upper-bound of the mode change phases smaller than $\overline{\text{reconfigured}}$ in order to be able to handle more multi-mode applications. The idea is to balance the work done through the mode change phase, by reducing the delay between the instant where the first core has no more reconfiguration to do nor job to execute and the last one. Informally, it *squares* the shape of the mode change phase. Balancing the work during the mode change phase leads to a reduced mode change phase duration, and thus potentially accepting more multi-mode application than ACCEPTOR. To do so, SQUARER may reconfigure a processor that still has jobs to execute. In the worst-case, SQUARER will behave exactly like ACCEPTOR. SQUARER has the same characteristics as ACCEPTOR: an aperiodic asynchronous protocol, designed to handle multi-mode applications modelled by the model presented in Chapter 11. It may be used with any FJP work-conserving C-sustainable scheduler.

We first present the protocol, and then its upper-bound. At last, we perform an evaluation in terms of performance and time complexity and compare this new protocol with ACCEPTOR introduced in Chapter 12.



(a) With ACCEPTOR



(b) *Square* mode change phase

Figure 13.1: Same system with two different protocols

13.1 Protocol squarer description

SQUARER aims at balancing the work done through the mode change phase, by reducing the delay between the instant where the first core has no more reconfiguration to do nor job to execute and the last one. The following definition specifies this instant.

Definition 13.1 (Reconfigured)**.** A core that has no reconfiguration to do nor job to execute is said to be *reconfigured*. Specifically:

- If a core was reconfigured once during the mode change phase, it is said to be *reconfigured*;
- If a core has no reconfiguration to perform, it is said to be *reconfigured* when it becomes idle.

$reconfigured_j$ refers to the instant where j^{th} cores are *reconfigured* in the platform.

Thus, SQUARER aims at reducing the delay between the $reconfigured_1$ and $reconfigured_{m'}$, where m' is the number of core in the cluster. Reducing this delay leads to reducing the amount of time where cores are idle, also referred to as idle time, as shown in Figure 13.1. To do so, SQUARER may constraint some cores to be reconfigured, even if they still have some jobs to execute. Indeed, the protocol places the reconfigurations as soon as possible. Figure 13.1(b) shows a perfect *square* mode change phase, with 0 idle time and thus no delay between $reconfigured_1$ and $reconfigured_3$. Of course, it is not always possible to reach 0 idle time, depending on the task set and/or the required reconfigurations. However, it is often possible to reduce the idle time drastically.

To operate, the protocol will use the following lower-bound of the instant in which the rem-jobs of the cluster are completed, and the required reconfigurations are done.

Corollary 13.1 (Lower-bound $\underline{\text{reconfigured}}$)**.** Suppose that J is a list of jobs released by $T^{\text{src},c}$, and that J is ordered by non-decreasing job processing times, i.e. , $c_1 \leq c_2 \leq \cdots \leq c_n$, all starting at $t = 0$. Suppose that the required reconfiguration delays are in the vector δ, ordered by reconfiguration time in decreasing order. Suppose that the cluster is composed of m' cores. Then, whatever the job priority assignment made by any global scheduler, a lower-bound of $reconfigured_{m'}$ is given by:

$$\underline{\text{reconfigured}}(T^{\text{src},c}, m', \delta) \doteq \frac{\sum_{i=1}^{n} c_i + \sum_{x=1}^{m'} \delta_x}{m'} \tag{13.1}$$

Proof. The first sum $\sum_{i=1}^{n} c_i$ represents the required computing time while the second sum $\sum_{x=1}^{m'} \delta_x$ represents the time spent by the cores being reconfigured. Obviously, it is not possible to perform all this work faster than by equally balancing it on all the cores, if possible. It will thus take at most $\frac{\sum_{i=1}^{n^{\text{src},c}} c_i + \sum_{x=1}^{m'} \delta_x}{m'}$ units of time to schedule all the rem-jobs and have all the cores reconfigured. □

In the following, we first present the offline phase in Section 13.1.1, and then the run-time phase in Section 13.1.2. In the following, $\underline{\text{reconfigured}}(T^{\text{src},c}, m', \delta)$ will be denoted as $\underline{\text{reconfigured}}$ for readability concerns.

13.1.1 Offline phase presentation

The offline phase of this protocol starts with the offline phase of ACCEPTOR, defined in Section 12.2.1. As a reminder: this first offline phase computes for every cluster of every possible mode change transition the required reconfigurations to perform. It is followed by a second offline phase, performed independently for every cluster of every possible mode change transition. This second offline phase computes the required constraints on the instant of reconfigurations in order to square each mode change phase, while respecting the real-time constraints. To do so, the second offline phase computes the maximal reconfiguration instant for the reconfigurations of each core, in each cluster, for each mode change transitions. The maximal reconfiguration instant for a given core is the instant at which a given reconfiguration should have started, according to SQUARER. This computation is divided in two steps, done independently for every cluster M^{src} for every possible mode change transition from M^{src} to M^{dst}. Here is an overview of those steps:

- Once the offline phase from ACCEPTOR has been done, determine for every required reconfiguration the initial instant of reconfigurations, using the lower-bound reconfigured;

- Check the rem-job deadlines or intra-task parallelism and delay everything accordingly, potentially several times.

Figure 13.1 shows the same mode change phase performed by two different protocols. Figure 13.1(a) shows the mode change phase performed by ACCEPTOR. The rem-jobs are completed first and then, when processor π_3 has no more job to execute it is reconfigured. Because all the processors become idle at the same instant, and that only one must be reconfigured: it creates a lot of idle time. In Figure 13.1(b), the mode change phase is performed by SQUARER. During the second offline phase, the maximal reconfiguration instant for π_3 has been computed to be 0, to limit the idle time. As seen in the figure, this leads to a mode change phase with 0 idle time, which is ideal. In this example, we see that SQUARER significantly reduces the idle time when compared to ACCEPTOR by constraining the reconfiguration instants.

Offline phase 1

The first offline phase is exactly the same as the one presented in Section 12.2.1. It determines for each cluster the required reconfigurations for every cluster M^{src} for every possible mode change transition from M^{src} to M^{dst}. Once done, the following offline phase can be performed for every cluster of every possible mode change transition.

Offline phase 2

This phase is divided in two steps. It is applied independently for every cluster of every possible mode change transition. Without loss of generality, we assume that all the cores must be reconfigured. If a core must not be reconfigured, we consider that it performs an instantaneous reconfiguration. This reconfiguration has thus a reconfiguration delay of 0 units of time.

The first step of this second offline phase is to compute an optimistic instant of reconfiguration for every core of the cluster. To do so, the j^{th} longest reconfiguration having a reconfiguration delay of δ_j is assigned on core j at $r_j \doteq \underline{\text{reconfigured}} - \delta_j$. This is very optimistic, as no idle time can occur in order to respect those reconfiguration instants while successfully scheduling the rem-jobs. It is important to note that by construction, all the cores would end up reconfigured at the same instant $\underline{\text{reconfigured}}$ if reconfigured at this instant of reconfiguration.

The second step aims at computing the idle time introduced by the scheduler, in order to compute an upper-bound. To do so, the rem-job deadlines are checked, and the reconfiguration instants are shifted if required such that the rem-jobs may be successfully scheduled. All the reconfigurations instants are shifted alongside, and so all the cores would still end up reconfigured at the same time. There is no need for a core to be reconfigured sooner as the real-time constraint considers only the last processor being reconfigured. This second step may be done several times to have a successful schedule. The deadline checking technique is explained in Section 13.1.3. The final instants of reconfiguration are stored, and will be used at run-time.

13.1.2 Run-time phase

At run-time, the protocol is quite similar to ACCEPTOR. It will first ensure the correct execution of the rem-jobs using the same scheduler as before, with a slight variation. We assume that the reconfiguration instants of core $\dot{\pi}_j$ is $r_j \geq 0, \forall j$. During the execution of

the rem-jobs, when a core becomes idle, it may be reconfigured. The major difference with ACCEPTOR is that core $\dot{\pi}_j$ is reconfigured at most at r_j, even if it is not idle yet. For each θ_c-cluster, the protocol takes as input the list of the remaining rem-jobs J and the required reconfigurations to make $\{\theta_c, \theta_{c'}\} \in \mathrm{RT}^{\mathrm{src,dst}}$.

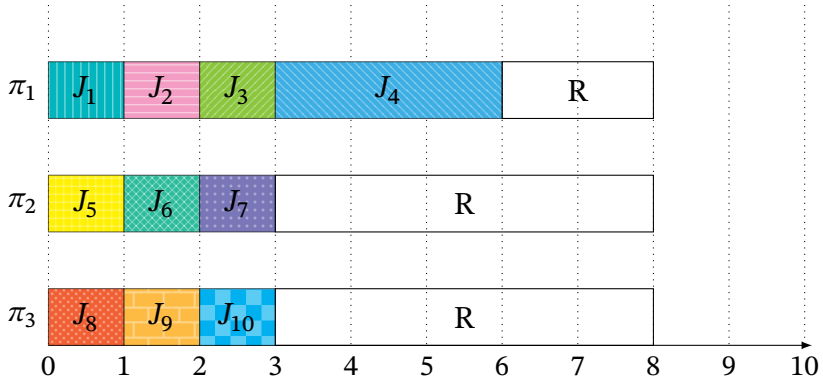
Formally,

- Schedule J using the same scheduler as before, and apply *Rule 1* and *Rule 2* when possible;
- *Rule 1*: When a core that hasn't been reconfigured becomes idle, reconfigure it to the longest required reconfiguration $\theta_{c'}$ that hasn't been performed yet, where $\{\theta_c, \theta_{c'}\} \in \mathrm{RT}^{\mathrm{src,dst}}$;
- *Rule 2*: If core $\dot{\pi}_j$ is not being reconfigured nor reconfigured at r_j, reconfigure it to the longest required reconfiguration $\theta_{c'}$ that hasn't been performed yet, where $\{\theta_c, \theta_{c'}\} \in \mathrm{RT}^{\mathrm{src,dst}}$.

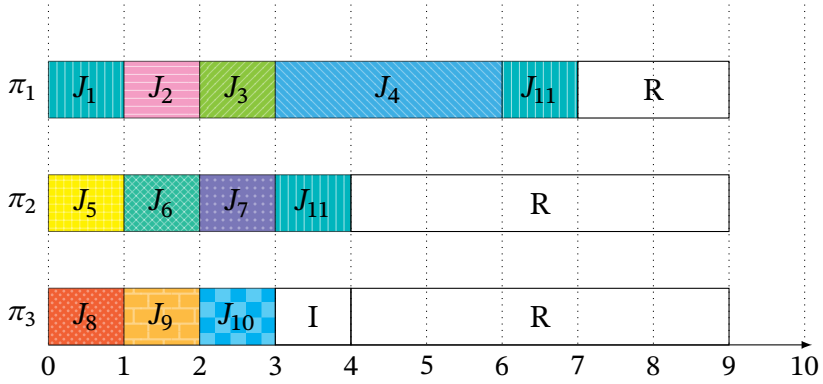
Whenever $\dot{m}_{c'}^{\mathrm{dst}}$ cores are reconfigured to $\theta_{c'}$, the $\theta_{c'}$-cluster of the new mode M^{dst} is formed and its task set $T^{c',\mathrm{dst}}$ is enabled. Several clusters may have cores reconfigured in $\theta_{c'}$ and merged together to form a new $\theta_{c'}$-cluster. This *mode change phase* ends once all the rem-jobs have been completed and the required reconfigurations performed. The old mode M^{src} is then disabled, and the new mode M^{dst} is enabled. A more formal version is presented in Appendix A.2.

13.1.3 Preventing deadline misses

We use simulation to check that no rem-job deadline are missed on a given cluster, with given reconfiguration instants. The simulation is ran in the worst-case scenario, with each task releasing one job at the mode change request instant. Without loss of generality, we assume that the processing rate of every couple task/configuration is 1. As a job will be executed on one and only one configuration, we may simply see its WCET as $\frac{\mathrm{WCET}}{R_{i,c}}$, where c is the configuration of the cluster upon which the task τ_i is partitioned. With the simulation, we compute the maximal deadline miss, denoted as $\max^{\mathsf{DM}}$. The maximal deadline miss is the maximal difference between the WCET of a rem-job and the time it has been executed , for any rem-job of the cluster. For example, if a rem-job J_i has a WCET $c_i = 3$ and was aborted after being executed for only 1 unit



(a) Seminal schedule, J_{11} misses a deadline



(b) Shifted schedule, J_{11} is successfully scheduled

Figure 13.2: Illustration of non-parallel idle time

of time: its requires 2 more units of execution time. If another aborted rem-job require 1 more unit of execution time, and all the other rem-jobs have been completed, we have $\max^{\mathsf{DM}} = \max\{1, 2\} = 2$.

If $\max^{\mathsf{DM}} = 0$, the current reconfiguration instants will lead to a successful execution of the rem-jobs, by construction. If $\max^{\mathsf{DM}} > 0$, it is required to shift the reconfiguration instants in order to schedule successfully the rem-jobs. To do so, we now introduce the following lemma.

Lemma 13.1. In a given mode change transition on a given cluster, where r is the vector containing the reconfiguration instants for every core of the cluster. If there are x different unique reconfiguration instants, shifting all the reconfiguration instants by 1 would create x units of non-parallel execution slots. Therefore, a lower-bound on the required shifting of the reconfiguration instants is given by:

$$\left\lceil \frac{\max^{\mathsf{DM}}}{|\{r_j | r_j \in r\}|} \right\rceil,$$

where the expression $|\{r_j | r_j \in r\}|$ denotes the number of unique different reconfiguration instants.

Proof. The proof is trivially obtained by construction. □

For example in Figure 13.2(a), we see a first schedule with no idle time. However, there is a deadline miss because J_{11} is not executed, where $c_{11} = 2$. In this case, $\max^{\mathsf{DM}} = 2$. We have three different reconfiguration instants: 3, 3 and 6. This means that there are two unique different reconfiguration instants: 3 and 6. The required shifting to schedule c_{11} is given by:

$$\left\lceil \frac{\max^{\mathsf{DM}}}{|\{r_j | r_j \in r\}|} \right\rceil = \left\lceil \frac{2}{2} \right\rceil = 1$$

The result of the shifted schedule is depicted in Figure 13.2(b). It can be seen that shifting the reconfigurations of one creates two non-parallel execution slots, that gets occupied by J_{11}.

Shifting the reconfiguration instants may be done recursively, as several rem-jobs may be aborted. However, one may note that several aborted rem-jobs does not necessary require several shifting. In Figure 13.2(b), there is still room for an extra job with a WCET $c_i = 1$, with only one shifting.

13.2 Upper-bound and validity test

Unlike ACCEPTOR, we do not provide a mathematical upper-bound formula for SQUARER. To compute an upper-bound on the duration required to schedule all the rem-jobs and to perform the required cores, it is required to execute the two offline phases. This upper-bound is denoted as $\overline{\text{reconfigured}''}$, and is given by the following theorem.

Theorem 13.2 ($\overline{\text{reconfigured}''}$)**.** When using SQUARER protocol for a given mode change transition on a given cluster, where r is the vector containing the reconfiguration instants for every core of the cluster. Assuming that δ is a vector containing the reconfiguration delays in decreasing order, and thus that the j^{th} longest reconfiguration having a reconfiguration delay of δ_j is assigned on the j^{th} reconfigured core. The upper-bound $\overline{\text{reconfigured}''}$ is given by:

$$\overline{\text{reconfigured}''}(r, \delta) \doteq r_0 + \delta_0$$

Proof. The proof is trivially obtained by construction. □

The following property is trivially true:

$$\forall j \text{ s.t. } 0 \leq j \leq m,\ r_0 + \delta_0 = r_j + \delta_j.$$

In other words, every core will be reconfigured at the same time, if a core is reconfigured at its instant of reconfiguration.

Similarly from ACCEPTOR, we can derive the upper-bound $\overline{\text{reconfigured}''}$ to obtain a validity test. To ensure the validity of the protocol for a given application, we need to verify that every deadline will be met during the transition phase, and that the new mode will always be timely enabled.

The schedulability of the rem-jobs during mode change phase is ensured by the Lemma 13.2.

Lemma 13.2 (Rem-job's schedulability)**.** Every rem-job's deadline of every cluster will be respected during every possible mode change phase, if and only if (i) the scheduler is C-sustainable and (ii) simulation proved that the rem-jobs generated by $T^{\text{src},c}$ were schedulable with the computed reconfiguration instants r.

Proof. Because the scheduler is C-sustainable, we know that any sets of rem-jobs generated by $T^{\text{src},c}$ will be schedulable with the given reconfiguration instants r. Thus, the schedulability is ensured. □

Lemma 13.3 ensures that the real-time constraint of each mode will be respected during every possible transition.

Lemma 13.3 (Transition time constraint)**.** Assuming that r^c is the vector containing the reconfiguration instants for every core of the θ_c-cluster. Assuming that δ is a vector containing the reconfiguration delays in decreasing order, and thus that the j^{th} longest reconfiguration having a reconfiguration delay of δ_j is assigned on the j^{th} reconfigured core. The protocol will respect the time constraint Δ^{dst} if and only if

$$\Delta^{\text{dst}} \geq \max_{\theta_c\text{-cluster}} \overline{\text{reconfigured}''}(r^c, \delta)$$

m'	$\overline{\text{reconfigured}}$	$\overline{\text{reconfigured''}}$
1	1.4	4.7
4	1.5	5.0
16	7.0	21.0
32	36.7	69.8
64	68,3	112.3

Table 13.1: Execution time comparison (seconds)

Proof. The right part of the inequation computes the maximal upper-bound $\overline{\text{reconfigured''}}$ for any θ_c-cluster. By construction, if the hypothesis is respected, every cluster of each possible transition will be reconfigured when required according to Δ^{dst}. □

Theorem 13.3 proves the validity of our protocol, under its hypotheses.

Theorem 13.3 (Validity of SQUARER)**.** The ACCEPTOR protocol is valid if and only if (i) the scheduler is a clustered, C-sustainable scheduler, (ii) simulation proved that the rem-jobs generated by $T^{\text{src},c}$ were schedulable with the computed reconfiguration instants r^c, (iii) $\forall$ src, dst, $(M^{\text{src}}, M^{\text{dst}}) \in \mathcal{G}, \forall\, \theta_c$-cluster where r^c is the vector containing the reconfiguration instants for every core of the θ_c-cluster:

$$\Delta^{\text{dst}} \geq \max_{\theta_c\text{-cluster}} \overline{\text{reconfigured''}}(r^c, \delta)$$

Proof. To prove the validity of the protocol, we must prove that every rem-job's deadline will be respected during the mode change phase and that the real-time constraint Δ^{dst} will be respected during each possible transition from every mode M^{src} to mode M^{dst}.

- Because of the hypotheses (i–ii), Lemma 13.2 can be applied and thus prove that every job's deadline will be respected during the mode change phase.
- Because of the hypotheses (iii), Lemma 13.3 can be applied. Every real-time constraint will be therefore respected.

Because the job's deadlines and the real-time constraint will be respected, the protocol is valid when the hypotheses (i–iii) are respected. □

13.3 Execution time

The theoretical worst-case time complexity is hard to evaluate, as the protocol includes simulation. We provide here the average execution time of both the computation of $\overline{\text{reconfigured}}$ and $\overline{\text{reconfigured}''}$. To compute this average execution time, we have executed 6000 task sets per value of m' with utilisation varying from 0.5 to 1, with $m' = 1, 4, 16, 32, 64$. The experiments have been run on an Intel® I7-7500U. In order to provide a fair comparison, we have executed the exact same task sets for both upper-bound. The results are shown in Table 13.1.

We can see that the time taken to compute the upper-bound $\overline{\text{reconfigured}''}$ is 2 to 3 times larger than the time taken to compute the upper-bound $\overline{\text{reconfigured}}$. This ratio actually reduces with a higher number of core per cluster. Being 2 to 3 times longer only makes the approach comparable in time complexity, as it is a constant factor of the offline phase duration. Therefore, despite having a non-predictable time complexity due to the use of simulation, the actual time complexity remains very low and comparable to the one of $\overline{\text{reconfigured}}$.

13.4 Empirical performances evaluation of $\overline{\text{reconfigured}''}$

By construction, $\overline{\text{reconfigured}''}$ has no pessimism as it uses simulation to place reconfigurations, and is thus an exact upper-bound. Of course, this is based on the assumption that all the tasks would release a job just before the mode change request and take up to their WCET after the mode change request to be completed. In other scenarii, some jobs will not be active or not need to be executed up to their WCET to be completed. Thus, some mode change phase would be faster than $\overline{\text{reconfigured}''}$.

Despite the fact that $\overline{\text{reconfigured}''}$ has no pessimism, we wanted to compare it to the competitor introduced in the previous chapter: $\overline{\text{reconfigured}}$. We measured the ratio $1 + \frac{\overline{\text{reconfigured}''} - \overline{\text{reconfigured}}}{\overline{\text{reconfigured}}}$. We used here same experiment parameters as in Section 12.6, described in the next paragraph.

The experiment has been conducted for clusters with $m' = 1, 4, 16$ and 64 cores. The reconfiguration times are uniformly chosen in the range $[0, 10]$, where 0 indicates that no reconfiguration is required. We generate approximately 1 000 feasible task

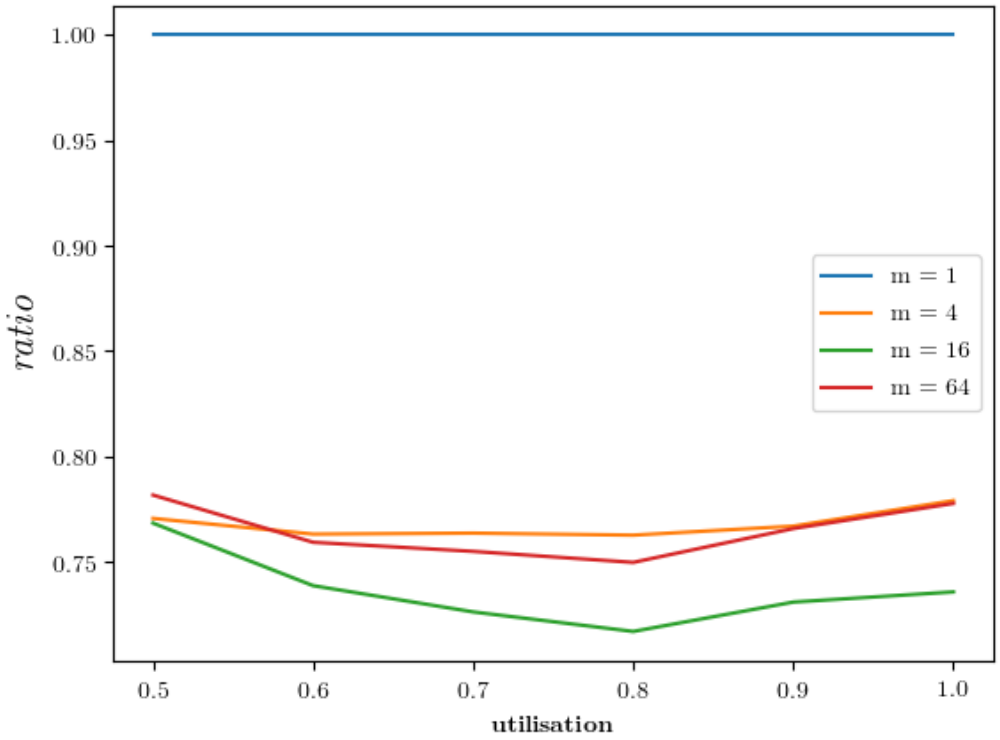


Figure 13.3: Pessimism comparison between $\overline{\text{reconfigured}}$ and $\overline{\text{reconfigured}''}$

sets with utilisations $\in (p - 0.1, p]$, where we increase p from 0.5 to 1.0 in steps of 0.1. The deadlines are uniformly chosen in the range $[2, 10]$. As a scheduler, we use Global-RM and remove any non-schedulable task set from our experiments. For those experiments, time is discrete.

The results are displayed in Figure 13.3 and can be read this way: the average ratio is 0.76 for a cluster with $m' = 64$ and a task set utilisation of 0.8. This means that the upper-bound $\overline{\text{reconfigured}''}$ is (on average) 24% smaller than the upper-bound $\overline{\text{reconfigured}}$. It is shown that both upper-bounds are equal (and exact) when $m' = 1$. There seems to be no correlation here between the utilisation and the ratio neither between the number of cores and the ratio, for any number strictly above 1. This is probably due to the fact that the shape of the task set and how *squarable* they are matters the most. However, you may note that for any number of core above 1, $\overline{\text{reconfigured}''}$ is at least shorter than $\overline{\text{reconfigured}}$ by 20%. The minimal and maximal ratio here are irrelevant. Indeed, as $\overline{\text{reconfigured}}$ is never better than $\overline{\text{reconfigured}''}$, the maximal ratio is 1. The minimal ratio is close to 0.5 for most cases, as the worst-case of ACCEPTOR previously

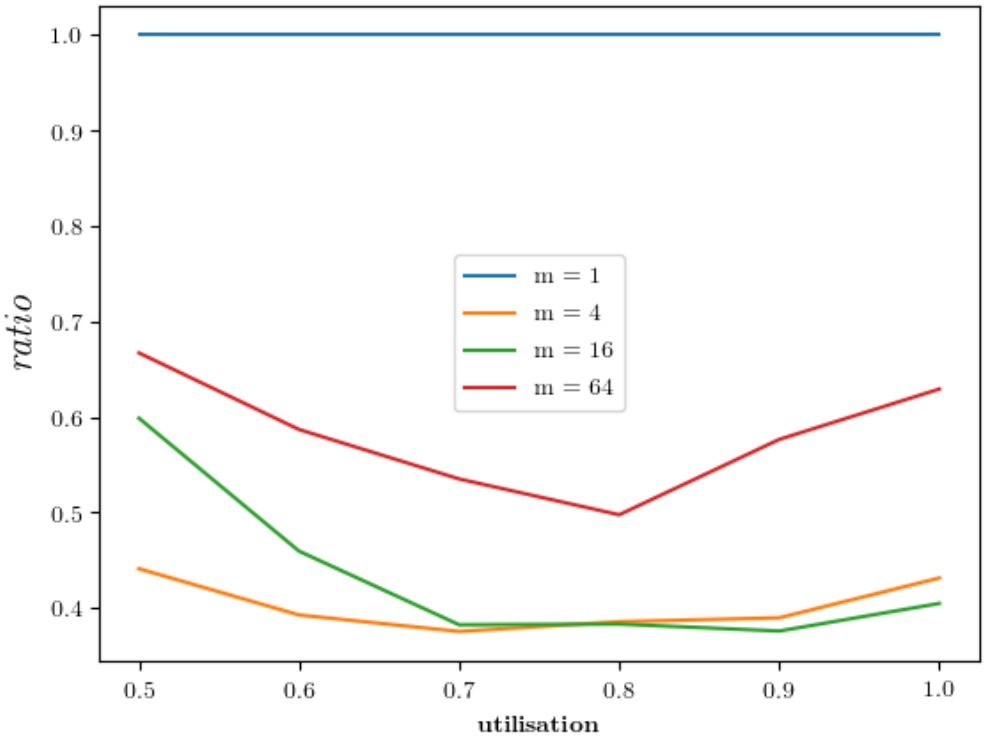


Figure 13.4: Total idle comparison between $\overline{\text{reconfigured}}$ and $\overline{\text{reconfigured}}''$

shown is often met.

Next, we have measured the same ratio with the same experiment parameters, but with the performances of the protocols ACCEPTOR and SQUARER in Figure 13.4. We have measured the average total idle time during a mode change phase with both of the protocols. Of course, as there is a fixed amount of work to perform during a mode change phase, there is a correlation between the total idle time and the mode change phase duration. We can see that SQUARER outperforms ACCEPTOR for any value of $m' > 1$. SQUARER has up to 60% less idle time than ACCEPTOR, in the case of $m' = 4$, with a task set having a utilisation of 0.7.

Chapter 14

Conclusion

In this part, we have addressed the problem of multi-mode applications on reconfigurable platforms. We proposed the first model modelling this new paradigm, i.e. where *both* the hardware configuration and the software workload are potentially different before and after the reconfiguration. We then proposed two protocols to handle such applications. Both ACCEPTOR and SQUARER can be used with multi-mode application on reconfigurable platforms, alongside with an FJP work-conserving clustered scheduler. We proposed a validity test associated with both protocols, so that it can be used in practice. Through a complete evaluation, we showed that both protocols were usable in practice and could scale for large systems. Their offline and run-time complexity are indeed low enough. We can also deduce from the result that SQUARER was outperforming ACCEPTOR in term of acceptance rates. The trade-off in terms of time complexity versus performance is in favour of SQUARER which happens to be slightly slower than ACCEPTOR for a better performance and acceptance rate. Using both run-time configuration and multi-mode software help the designer to make the most out of the platform, by adjusting the hardware to the current need of the application. We believe that our model captures the reality of those applications, and that our protocols provide the required performance and complexity to be used by application designers.

In the future, we would like to propose a closed-form solution for SQUARER to remove the need of simulation for this protocol. We would also like to include mode independent tasks in the multi-mode application with fewer constraints. Those tasks are released even during a mode change phase, and thus the schedulability problem becomes much more complex. Also, we would like to explore the possibility of having

a custom scheduler designed only for the scheduling of rem-jobs during a mode change phases. Indeed, the scheduler used is designed to handle task periodicity. Thus, a new scheduler specifically designed for mode change phases could reduce the makespan and therefore improve the efficiency of the protocol.

Part IV

General conclusion and perspectives

Real-time systems are everywhere, and their importance is growing. They are not just limited to specific customers but distributed widely to the public, as cars or drones. The hardware embedded in those systems is getting more and more powerful. As for any hard real-time systems, the major constraint is the absence of errors that guarantees the security of the system and their users. To provide more powerful systems running complex application, the hardware systems need to evolve and remain cost-effective. It has been long time since following Moore's law was no longer possible with strictly uniprocessor platforms. The cost of producing high clock frequency uniprocessor platform would be very high, as well as the energy consumption and thermal aspects.

Since year 2000, embedded systems are now composed of multi-processor platforms. Multi-processor platforms lead to the use of the parallel programming paradigm. However, according to Amdhal's law, parallelisation cannot lead to infinite speed up. To pursue the efficiency improvement, a lot of modern platforms are now heterogeneous to provide versatile and specialised hardware. Some platforms have different processors, of different types, with some of them being reconfigurable. Therefore, those new platforms are way more complex than the uniprocessor platforms. Also, others considerations are taken into account. The energy consumption needs to remain as low as possible, the hardware weight must be low. Temperature may be constrained as well.

The shift to parallel computing was already challenging for hard real-time scheduling, as it brings up scheduling anomalies and because avoiding intra-task parallelism is not trivial. The new layers of complexity bring even more challenges to hard real-time scheduling. The literature contains very few usages of such platforms, especially when it comes to using the whole potential of it.

In this thesis, we have proposed several approaches to tackle those issues. First of all, we have explored the global scheduling paradigm for heterogeneous unrelated platforms. The literature proposes very few solutions for such platforms. In fact, only one algorithm was tackling the complete problem while being optimal. Of course, this optimality stands only by assuming certain hypothesis. Moreover, the only optimal solution presented in [19] was flawed, as shown in Chapter 6. One of our contributions was to exhibit the flaw, to correct it and to prove the correctness of our new solution. We also have shown that all the existing approaches were neglecting a major cost in global scheduling: the migrations from one cluster to another.

To take into account the costs of inter-cluster migrations encountered in practice, we proposed a new model. This new model captures more accurately the reality of the platform, which permits to reduce the costs of inter-cluster migrations. We took advantages of this new model in Chapters 4–9. In those chapters, we decomposed the existing scheme for global scheduler for heterogeneous unrelated platforms into different steps. We then improved every step and proposed variations and optimisations to it. This way, we have built a completely new optimal algorithm and shown that it was performing better than the existing ones. In particular, we have measured the migrations from one cluster to another, and showed a significant reduction. This is a significant contribution to the literature, as it is the second optimal algorithm for heterogeneous unrelated platform. We also have proved its correctness, in order to ensure the validity of our work.

The optimisations and variations proposed also offer directions to future research. We first introduced the use of linear algebra for the template construction. We also proposed to use optimisation after the template schedule construction, by making use of the Travel Salesman Problem (TSP). Other similar optimisations could be derived from those optimisations. These optimisations and variations open a new field of research for scheduling with original approaches to the problem, and constitute thus a contribution.

The last contribution from Chapters 4–9 is the proof that the only global optimal algorithm for heterogeneous unrelated platforms for sporadic tasks presented in [29] was flawed as well. We do not provide any correction for this algorithm.

In this thesis, we have covered a significant part of the problem of global scheduling for heterogeneous unrelated platforms, by proposing a new model and a new optimal algorithm, with better performance compared to the literature. We have also pointed out two flaws in the literature, and introduced new directions for further research.

This domain of the real-time literature still offers a lot of subjects to work on. Despite the fact that our solutions improve the state of the art, there are still too much migrations to be used in practice ‘as is’. Further works would focus on reducing this number of migrations to improve the usability of the schedulers. Also, the new directions could be explored to optimise the schedule performance as well. At last, the use of sporadic tasks is worth the investigation. It may be done by trying to correct the flawed algorithm, or with a brand new algorithm.

Exploiting every processing capacities available is one important aspect of this thesis. In order to get the full potential from the capabilities of modern heterogeneous unrelated platforms, we have also explored the use of processor reconfigurations. We chose to merge processor reconfigurations into the existing multi-mode application paradigm. It was a natural fit to reconfigure the processors to the new requirements, while simultaneously changing the software. Multi-mode applications have several advantages. A lot of applications will execute a workload based on a given context or internal states. This is the case of a security camera, that will probably not operate in the same way during day and night, or when there is nothing in its field of vision. One way to model this is to use sporadic tasks. However, it is pessimistic as some tasks will never be ran alongside others. Multi-mode applications thus help to analyse the real hardware needs of an application. It leads to a gain in hardware requirements, which directly leads to gains in terms of cost, space and energy consumption: three key factors for embedded systems.

Our first contribution to the use of processor reconfigurations is the creation of the first model that merges those two aspects. Our model is simple enough to be used in practice, while having the potential to deliver complex and useful results.

With this new model, we proposed our second contribution: a first protocol to handle application with hardware reconfigurations and software modes. This protocol is dedicated to the handling of mode change phases. This first protocol adopts a simple approach, and we led a full study to demonstrate theoretically and empirically its performance and limitations. We have shown that it was 2-competitive, and with a time-complexity that is low enough to be used with large applications.

This protocol is divided into two offline steps and one run-time step. As it is the first protocol for the new model, it sets of a standard for the following protocols. One or several of those three steps may be improved to produce better performing protocols. Our thorough study of this first protocol also makes it comparable with future protocols.

The third contribution of this thesis regarding multi-mode protocols is another protocol. This new protocol is based on the same first offline step as the first one, but significantly improves the second and third steps. It proposes a more complex approach. It leads to better performance for a comparable time-complexity, as shown by the different comparison ran. However, it proposes no closed-form when it comes to acceptance of new applications. We believe that it is still a significant improvement compared to

the first protocol, as its performance is way better, and the empirical acceptance rate is significantly improved.

By proposing the two first protocols to our new model combining for the first time hardware reconfiguration and multi-mode applications, we have made in this thesis a complete set of contributions to the state of the art. We have filled a gap in the literature, and we believe that this new paradigm will become more and more important in the future, with the raise of reconfigurable platforms. Some of the very last platforms offer a 3D architecture with very fast reconfiguration time. Such platforms are very promising, and our results are ready to be applied to them.

Our work does not directly address energy consumption nor thermal aspects. Although it may be derived from our solutions, these aspects are very important for embedded systems. Also, our solutions do not propose a complete integration of mode-independent tasks.

Future works could directly target those last points. Mode-independent tasks could have a better integration through new protocols. As our protocols require the designer to provide the hardware requirement and the software for each mode, it would be easy to integrate energy and/or thermal aspects. Indeed, taking into account those aspects while designing the mode definition would lead to better energy and/or thermal aspect handling. At last, providing a closed-form for the last protocol would improve its comparability. It could thus be used as a new competitor for future protocols.

In conclusion, this thesis provides a lot of different contributions, covering different aspects of hard real-time scheduling for modern heterogeneous unrelated platforms. We have provided several contributions to the literature of global scheduling for those platforms, as well as exhibiting flaws and a correction to it. The importance of hardware reconfigurations in the last generation of platforms and even more in the future generations made us merge two existing paradigms to a new one. We have here set the ground for future research, and created the very first results. We strongly believe that this will become more and more important as those hardware reconfigurations will become more and more powerful. By contributing in those two domains, we have covered a lot of open questions. The next step would be to improve our results, in order to enhance the usability of our solutions.

"Run, rabbit, run
Dig that hole, forget the sun
And when at last the work is done
Don't sit down, it's time to dig
another one"

Pink Floyd - Breathe

Appendix A

Multi-mode protocol algorithms

This chapter proposes formal algorithms for both protocol run-time phases, presented in Part III.

A.1 ACCEPTOR algorithm

This section proposes the run-time protocol algorithm presented in Section 12.2.2, in Figure A.1. This algorithm takes as input the current set of rem-jobs, the set of processors, and the used scheduler. It also takes as input in vector R the required reconfigurations $\mathrm{RT}^{\mathrm{src,dst}}$, computed offline. Those must be ordered by reconfiguration time, decreasing.

A.2 SQUARER algorithm

This section proposes the run-time protocol algorithm presented in Section 13.1.2, in Figure A.2. This algorithm takes as input the current set of rem-jobs, the set of processors, and the used scheduler. It also takes as input in vector R the required reconfigurations $\mathrm{RT}^{\mathrm{src,dst}}$ and the instant of reconfigurations in vector T, computed offline. R must be ordered by reconfiguration time, decreasing. T must be sorted in increasing order.

Require: J rem-job set; P processor set; S scheduler; R required reconfigurations
Ensure: R is ordered by reconfiguration time (decreasing)
  while $|J| > 0$ **do**
    try:
      schedule(S, J, R)
    stop if p $\in$ P is idle:
      reconfigure(p, $R[0]$)
      $R \leftarrow R[1{:}|\mathrm{R}|]$
      $P \leftarrow P \backslash p$
  end while
  while $|R| > 0$ **do**
      $p \leftarrow P[0]$
      reconfigure(p, $R[0]$)
      $R \leftarrow R[1{:}|\mathrm{R}|]$
      $P \leftarrow P \backslash p$
  end while

Figure A.1: ACCEPTOR run-time algorithm

Require: J rem-job set; P processor set; S scheduler; R required reconfigurations
Require: T reconfiguration instant set
Ensure: R is ordered by reconfiguration time (decreasing)
Ensure: T is sorted in increasing order
  while $|J| > 0$ **do**
    try:
      schedule(S, J, R, T)
    stop if p $\in$ P is idle:
      reconfigure(p, $R[0]$)
      $R \leftarrow R[1{:}|\mathrm{R}|]$
      $P \leftarrow P \backslash p$
    or stop if $t = T[0]$:
      $p \leftarrow P[0]$
      reconfigure(p, $R[0]$)
      $R \leftarrow R[1{:}|\mathrm{R}|]$
      $T \leftarrow T[1{:}|\mathrm{T}|]$
      $P \leftarrow P \backslash p$
  end while
  while $|R| > 0$ **do**
      $p \leftarrow P[0]$
      reconfigure(p, $R[0]$)
      $R \leftarrow R[1{:}|\mathrm{R}|]$
      $P \leftarrow P \backslash p$
  end while

Figure A.2: SQUARER run-time algorithm

Index

Bibliography

[1] Xilinx. *Zynq UltraScale+ MPSoC*. 2018. URL: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html (visited on 01/28/2018).

[2] Alan Burns and Sanjoy K. Baruah. "Sustainability in Real-time Scheduling". In: *JCSE* 2.1 (2008), pp. 74–97. URL: http://jcse.kiise.org/PublishedPaper/year_abstract.asp?idx=15.

[3] Vincent Nélis. "Energy-Aware Real-Time Scheduling in Embedded Multiprocessor Systems". PhD thesis. Université libre de Bruxelles, 2010.

[4] Vincent Nélis, Joël Goossens, and Björn Andersson. "Two Protocols for Scheduling Multi-mode Real-Time Systems upon Identical Multiprocessor Platforms". In: *Euromicro Conference on Real-Time Systems*. 2009, pp. 151–160.

[5] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20.1 (1973), pp. 46–61. DOI: 10.1145/321738.321743. URL: http://doi.acm.org/10.1145/321738.321743.

[6] Jorge Real and Alfons Crespo. "Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal". In: *Real-Time Systems* 26.2 (2004), pp. 161–197. DOI: 10.1023/B:TIME.0000016129.97430.c6. URL: https://doi.org/10.1023/B:TIME.0000016129.97430.c6.

[7] Laura Hopperton. *embedded world: Xilinx introduces 'industry's first' extensible processing platform*. 2011. URL: https://goo.gl/EuNhir (visited on 01/31/2018).

[8] Martin Cornil et al. “Research and implementation challenges of RTOS support for heterogeneous computing platforms”. In: *Heterogeneous Architectures and Real-Time Systems Seminar, Brussels*. 2017.

[9] Ahmad Sadek et al. “Supporting Utilities for Heterogeneous Embedded Image Processing Platforms (STHEM): An Overview”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by Nikolaos Voros et al. Cham: Springer International Publishing, 2018, pp. 737–749. ISBN: 978-3-319-78890-6.

[10] Marco Pagani et al. “Towards real-time operating systems for heterogeneous reconfigurable platforms”. In: *OSPERT 2016* (2016), p. 49.

[11] A. Biondi et al. “A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs”. In: *2016 IEEE Real-Time Systems Symposium (RTSS)*. 2016, pp. 1–12. DOI: 10.1109/RTSS.2016.010.

[12] A. Biondi and G. Buttazzo. “Timing-aware FPGA partitioning for real-time applications under dynamic partial reconfiguration”. In: *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. 2017, pp. 172–179. DOI: 10.1109/AHS.2017.8046375.

[13] M. Pagani et al. “A Linux-based support for developing real-time applications on heterogeneous platforms with dynamic FPGA reconfiguration”. In: *2017 30th IEEE International System-on-Chip Conference (SOCC)*. 2017, pp. 96–101. DOI: 10.1109/SOCC.2017.8226015.

[14] Enrico Bini. “Adaptive Fair Scheduler: Fairness in Presence of Disturbances”. In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS ’16. Brest, France: ACM, 2016, pp. 129–138. ISBN: 978-1-4503-4787-7. DOI: 10.1145/2997465.2997468. URL: http://doi.acm.org/10.1145/2997465.2997468.

[15] Eugene L Lawler and Jacques Labetoulle. “On preemptive scheduling of unrelated parallel processors by linear programming”. In: *Journal of the ACM (JACM)* 25.4 (1978), pp. 612–619.

[16] Sanjoy K. Baruah. “Partitioning Real-Time Tasks among Heterogeneous Multiprocessors”. In: *33rd International Conference on Parallel Processing (ICPP)*. IEEE, 2004, pp. 467–474.

[17] Jagpreet Singh and Nitin Auluck. “Real time scheduling on heterogeneous multiprocessor systems—A survey”. In: *Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*. IEEE. 2016, pp. 73–78.

[18] Sanjoy K. Baruah et al. “ILP models for the allocation of recurrent workloads upon heterogeneous multiprocessors”. In: *J. Scheduling* 22.2 (2019), pp. 195–209.

[19] Sanjoy Baruah. “Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms”. In: *Real-Time Systems Symposium*. IEEE. 2004, pp. 37–46.

[20] Hoon Sung Chwa et al. “Optimal real-time scheduling on two-type heterogeneous multicore platforms”. In: *Real-Time Systems Symposium*. IEEE. 2015, pp. 119–129.

[21] Gurulingesh Raravi et al. “Task assignment algorithms for two-type heterogeneous multiprocessors”. In: *Real-Time Systems* 50.1 (2014), pp. 87–141.

[22] Sanjay Moulik, Rajesh Devaraj, and Arnab Sarkar. “Hetero-sched: A low-overhead heterogeneous multi-core scheduler for real-time periodic tasks”. In: *20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE. 2018, pp. 659–666.

[23] Gurulingesh Raravi et al. “Task assignment algorithms for two-type heterogeneous multiprocessors”. In: *Real-Time Systems* 50.1 (2014), pp. 87–141. DOI: 10.1007/s11241-013-9191-3. URL: https://doi.org/10.1007/s11241-013-9191-3.

[24] Vincent Nélis et al. “Global-EDF Scheduling of Multimode Real-Time Systems Considering Mode Independent Tasks”. In: *23rd Euromicro Conference on Real-Time Systems, ECRTS 2011, Porto, Portugal, 5-8 July, 2011*. 2011, pp. 205–214. DOI: 10.1109/ECRTS.2011.27. URL: https://doi.org/10.1109/ECRTS.2011.27.

[25] Chi-Sheng Shih and Chang-Min Yang. “Schedulability Analysis of Mode Change for Imprecise Computation on Multi-Core Platforms”. In: *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*. ACM. 2017, pp. 261–268.

[26] José Marinho et al. “Partitioned Scheduling of Multimode Systems on Multiprocessor Platforms: when to do the Mode Transition?” In: *RTSOPS* (2011).

[27] Paul Emberson and Iain Bate. “Minimising task migration and priority changes in mode transitions”. In: *13th IEEE Real Time and Embedded Technology and Applications Symposium*. IEEE. 2007, pp. 158–167.

[28] Joël Goossens and Pascal Richard. "Partitioned scheduling of multimode multiprocessor real-time systems with temporal isolation". In: *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. ACM. 2013, pp. 297–305.

[29] Sanjoy Baruah and Björn Brandenburg. "Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities". In: *Real-Time Systems Symposium*. IEEE. 2013, pp. 160–169.

[30] Robert McNaughton. "Scheduling with deadlines and loss functions". In: *Management Science* 6.1 (1959), pp. 1–12.

[31] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

[32] Guillaume Phavorin, Pascal Richard, and Claire Maiza. "Complexity of scheduling real-time tasks subjected to cache-related preemption delays". In: *20th Conference on Emerging Technologies & Factory Automation*. IEEE, 2015, pp. 1–8.

[33] Guillaume Phavorin et al. "Online and offline scheduling with cache-related preemption delays". In: *Real-Time Systems* 54.3 (2018), pp. 662–699.

[34] Narendra Karmarkar. "A new polynomial-time algorithm for linear programming". In: *Combinatorica* 4.4 (1984), pp. 373–396. DOI: 10.1007/BF02579150. URL: https://doi.org/10.1007/BF02579150.

[35] Antoine Bertout et al. "Template schedule construction for global real-time scheduling on unrelated multiprocessor platforms". In: *Design, Automation and Test in Europe Conference (Grenoble, France, March 2020)*. To appear. 2020. Forthcoming for DATE.

[36] Fanny Dufossé and Bora Uçar. "Notes on Birkhoff–von Neumann decomposition of doubly stochastic matrices". In: *Linear Algebra and its Applications* 497 (2016), pp. 108–115.

[37] Joël Goossens and Christophe Macq. "Limitation of the hyper-period in real-time periodic task set generation". In: *In Proceedings of the RTS Embedded System (RTS'01)*. 2001, pp. 133–148.

[38] Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems: Revised Reprint*. SIAM, 2012.

[39] László Lovász and Michael D Plummer. *Matching theory*. Vol. 367. American Mathematical Soc., 2009.

[40] A. Volgenant. “A note on the assignment problem with seniority and job priority constraints”. In: *European journal of operational research* 154.1 (2004), pp. 330–335.

[41] Gaetan Caron, Pierri Hansen, and Brigitte Jaumard. “The assignment problem with seniority and job priority constraints”. In: *Operations Research* 47.3 (1999), pp. 449–453.

[42] John E Hopcroft and Richard M Karp. “An n^5/2 algorithm for maximum matchings in bipartite graphs”. In: *SIAM Journal on computing* 2.4 (1973), pp. 225–231.

[43] Iain S Duff and Jacko Koster. “On algorithms for permuting large entries to the diagonal of a sparse matrix”. In: *SIAM Journal on Matrix Analysis and Applications* 22.4 (2001), pp. 973–996.

[44] Rainer E Burkard and Eranda Cela. *Linear assignment problems and extensions*. Chap. Unknown.

[45] Christian Kohn (Xilinx). *Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices (XAPP1159)*. 2013.

[46] Liliana Cucu-Grosjean and Joël Goossens. “Predictability of Fixed-Job Priority schedulers on heterogeneous multiprocessor real-time systems”. In: *Inf. Process. Lett.* 110.10 (2010), pp. 399–402. DOI: 10.1016/j.ipl.2010.03.009. URL: https://doi.org/10.1016/j.ipl.2010.03.009.