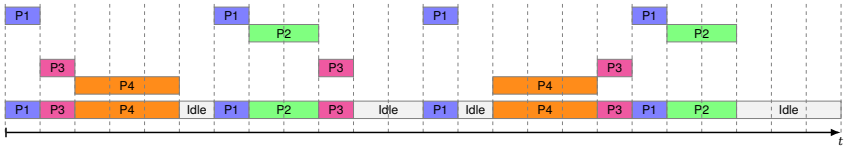
TÉCNICO LISBOA



Partition Scheduling in Distributed Integrated Modular Avionics

João Miguel Fonseca Gonçalves

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisor: Prof. Luís Manuel Marques Custódio
Eng. João Miguel Cintra de Jesus Silva

Examination Committee

Chairperson: Prof. José Fernando Alves da Silva
Supervisor: Prof. Luís Manuel Marques Custódio
Members of the Committee: Prof. Tânia Rute Xavier de Matos Pinto Varela

October 2019

Resumo

A arquitetura aviónica modular integrada substituiu as arquiteturas federadas no domínio aviónico, o que permite obter reduções significativas em peso e consumo energético, além de permitir desenvolvimento mais competitivo de aplicações.

A natureza de partilha de recursos desta arquitetura requer segregação temporal e espacial robusta entre aplicações, o que é alcançado através de um escalonamento temporal estático em hardware aviónico partilhado. Isto levanta um problema complexo de escalonamento em múltiplos processadores, cujo progresso na indústria é limitado, mas que representa um desafio significativo para a integração de sistemas aviónicos.

Esta dissertação propõe um modelo matemático para o problema de escalonamento temporal de partições complementado com um critério de otimização baseado na escalabilidade e flexibilidade do sistema, e propõe métodos exatos e heurísticos para a sua resolução baseados na literatura existente.

Palavras-Chave: Arinc 653, Aviónica Modular Integrada, Escalonamento, Otimização

Abstract

The Integrated Modular Avionics architecture has replaced federated architectures in the avionic domain, allowing significant weight and power savings and enabling more competitive application development.

The resource-sharing nature of this architecture requires robust temporal and spatial segregation between applications, which is achieved by statically scheduling applications on shared avionic hardware. This raises a multiprocessor scheduling problem, automation of which has seen limited progress in industry, but representing a significant challenge for system integration.

This dissertation proposes a mathematical model for the partition scheduling problem associated with an optimization criterion based on system scalability and flexibility, and provides both heuristic and exact methods for its solution based on existing literature.

Keywords: Arinc specification 653, Integrated Modular Avionics, Scheduling, Optimization

Acknowledgements

First and foremost, I would like to thank my advisor Professor Luís Custódio for all the guidance and counseling throughout the semester.

I would also like to extend my thanks to everyone I met at GMV, for making my work there a pleasant moment. A word of appreciation goes to João Cintra and Miguel Barros, who provided me with this topic and advised me all along the way.

Lastly, my deepest thanks to my family, who supported me all throughout my studies.

Contents

List of Figures

List of Tables

Glossary

ACO	Ant Colony Optimization 17
AFDX	Avionics Full-Duplex Switched Ethernet™ 10, 11
APEX	Application Executive Interface 6, 8–10
Arinc 429	Aeronautical Radio, Incorporated Specification 651 – Digital Information Transfer System 10
Arinc 651	Aeronautical Radio, Incorporated Specification 651 – Design Guidance for Integrated Modular Avionics 6
Arinc 653	Aeronautical Radio, Incorporated Specification 653 – Avionics Application Standard Software Interface iii, v, 1, 3, 6–8, 11, 16, 17, 69
Arinc 664	Aeronautical Radio, Incorporated Specification 664 – Avionics Full-Duplex Switched Ethernet 10
COP	Combinatorial Optimization Problem 12, 14
COTS	Commercial off-the-Shelf 7, 10
CPIOM	Core Processing Input-Output Module 6
CPM	Core Processing Module 6, 8, 11–13, 15–18, 59
CSP	Constraint Satisfaction Problem 3, 28, 34–36, 41, 48, 55, 56
DIMA	Distributed Integrated Modular Avionics 6, 13
ETE	End-to-End 10, 22, 27
FIFO	First-In First-Out 10
GCD	Greatest Common Divisor 25
I/O	Input/Output 5, 6, 8, 9, 12, 13
IMA	Integrated Modular Avionics v, 1–3, 5–7, 10, 11, 14, 16, 59, 69
IMA-SP	Integrated Modular Avionics for Space 7
IP	Integer Programming 14, 15, 17, 18, 36
LCM	Least Common Multiple 11
LIFO	Last-In First-Out 54, 75
LRU	Line Replaceable Unit 1, 5
MILP	Mixed Integer Linear Programming 3, 4, 14–19, 30, 34, 36, 57, 59–64, 66, 68, 69

MTBF	Mean Time Between Failures 6
MTF	Major Time Frame 11, 12, 20
RDC	Remote Data Concentrator 6
RTOS	Real-Time Operating System 1, 3, 6–9
SA	Simulated Annealing 17, 48, 53, 54, 64, 68, 74, 75
SMT	Satisfiability Modulo Theories 14, 16, 18
TTEthernet	Time-Triggered Ethernet 11, 16
UAV	Unmanned Aerial Vehicle 7
WCET	Worst-Case Execution Time 11–13, 19, 27

Nomenclature

Latin Symbols

$a_{i,m}$ boolean assignment of partition with index i to module with index m 30–33

$\mathcal{B}_i$ partition with index i set of possible breakpoints 24, 51

$\mathcal{C}$ set of modules 19, 21, 26, 28, 31–33, 37, 50, 51

c_m module with index m 19, 21–23, 26, 28, 30–33, 35, 46, 51

$Cr()$ crossover operator 49, 52

D_i partition with index i domain 21, 31, 32, 35

d_i partition with index i maximum response time (relative deadline) 24, 44

$E_{i,j}$ processing time of chain $i \rightarrow j$ 21–23, 27, 37

e_i partition with index i execution requirement 19, 20, 23–29, 31–33, 37, 39–43, 58

$e_{i,k,u}$ partition with index i, k-th job, u-th window duration 24, 29, 44

$E_{i,j}^{max}$ maximum processing time of chain $i \rightarrow j$ 21, 27, 32, 33, 37, 42, 43

$ev(S)$ evaluation function 48, 49, 54, 62, 74, 75

f_i module assignment of partition with index i 19–21, 31, 33–35, 50, 52, 55, 66

$g_{i,j}$ greatest common divisor of periods T_i, T_j 25–27, 29, 31, 33, 37, 39–43

H_m module with index m major time frame 20, 29

IQR interquartile range 63

$\mathcal{J}_i$ set of utility intersection points 40, 41, 43

K_i job count of partition with index i 20, 44

k job index 20, 23, 24

$l_{i,j}$ latency delay separating partitions indexed i and j 25, 27–29, 31, 32, 37, 43

$Lop()$ local optimization operator 49, 51–53, 55–57, 74, 75, 77

$M_{i,k}$ partition with index i number of windows at job k 23, 24, 29, 44

$\mathcal{M}$ subgroup of modules 49, 51

$Mov()$ move operator 49, 50, 52, 53, 55, 56, 74–76

N_c number of modules 19, 21, 26, 30, 34, 48, 62

N_p number of partitions xii, 19, 30, 34, 41, 48, 58–62

$\mathcal{P}$ set of partitions 19, 21, 31–35, 37, 50, 52, 55

p_i partition with index i 19–21, 23–35, 37, 39, 41, 42, 50–52, 55

P_a acceptance probability 53, 54, 74

$\mathcal{P}_m$ set of partitions scheduled in module with index m 19–21, 23, 25–28, 30, 32, 37, 39, 41, 51

$q_{i,j}$ quotient associated to $l_{i,j}$ 31, 33, 42, 43

r_i partition with index i response time 24

s_i partition with index i memory requirement 19, 21, 31, 33, 35

S_m module with index m memory capacity 19, 21, 31, 33, 35, 52, 56

S state 48–52, 54–56, 62

$Sh()$ shuffle operator 49, 51, 52, 55, 56, 74–76

$Sl()$ slice operator 49, 52, 55, 56, 74–76

$Sw()$ swap operator 49, 50, 52, 55, 56, 74–76

T_i partition with index i period 19, 20, 22–29, 31–33, 37, 38, 40–42, 44, 58, 59

t_i partition with index i starting time offset 20, 23, 25, 28, 31–33, 37–43, 52, 55, 66

$t_{i,k,u}$ partition with index i, k-th, u-th window offset 24, 29, 30, 44, 51

$\mathcal{T}_i$ partition with index i set of feasible offsets 39–42

U processor usage fraction 26, 36, 46

$x_{i,j}$ boolean stalling of a chain $i \rightarrow j$ for one period 32, 33

$\mathcal{X}$ subgroup of partitions 49, 50, 52

$y_{i,j,m,n}$ boolean assignment of partition with index i to module with index m and j to n 32, 33

Greek Symbols

α schedule evaluation parameter 28–33, 38, 39, 41, 43, 45, 48–52, 54, 59–63, 66, 67

α_i partition with index i utility 28–30, 39–41, 66

$\lambda_{i,k,u}$ partition with index i, k-th job, u-th window 24, 30, 44, 51

Λ_i set of windows composing partition with index i 24, 30, 45, 51

δ Kronecker delta 26, 35, 37

ε_m module with index m context switch time 19, 24, 44, 62

T worst-case communication delay matrix 22, 27

$\tau_{m,n}$ worst-case communication delay between modules indexed m and n 22, 23, 27, 32, 33

$\hat{\tau}_{i,j}$ worst-case communication delay between partitions indexed i and j 32, 42, 43

Chapter 1

Introduction

1.1 Motivation

In the past decades, the avionic industry adopted the Integrated Modular Avionics (IMA) architecture, replacing the older federated architectures.

In federated architectures, each avionic functionality is deployed as an independent black-box component called a Line Replaceable Unit (LRU), with its own dedicated hardware and software. Despite providing excellent safety and fault containment, each further addition of a function to the system requires adding a new LRU, and this escalates the mass, weight, and power requirements of the avionic system, not to mention it being an inefficient usage of resources. With the industry demanding more and more functionality from avionic systems, the federated concept became unsustainable [38].

The IMA paradigm is the aviation industry's response to these problems, whose architecture principle relies on resource sharing between generally unrelated components, including computing resources, power, and communication media, as roughly demonstrated in figure 1.1. Functions which were previously implemented in isolated LRUs now coexist on shared hardware, and in order to maintain isolation between components, IMA adopts robust time and space partitioning between applications. Space partitioning protects the application's data from corruption by unrelated applications that share the same hardware, and time partitioning ensures the required access to computing resources and communication channels [32]. Due to this, loosely coupled avionic applications are designated as partitions in the context of IMA.

Time and space partitioning is ensured through compliance with the Arinc 653 standard, which defines a standardized interface between partitions and an underlying Real-Time Operating System (RTOS), satisfying all safety-critical requirements of the avionic applications [17]. For this purpose, Arinc 653 requires that partitions execute in a static, periodic manner. Unlike regular operating systems, where processes are scheduled at runtime, the time slots allocated to partitions are predefined in a static schedule, and repetition is deterministic because this schedule is periodic. A special characteristic of these schedules, imposed by Arinc 653, is that partitions should execute non-preemptively in strictly periodic intervals, or in other words, there must be no jitter in the time between different executions of the same partition, providing it unique access to resources in its given time budget. This, coupled with

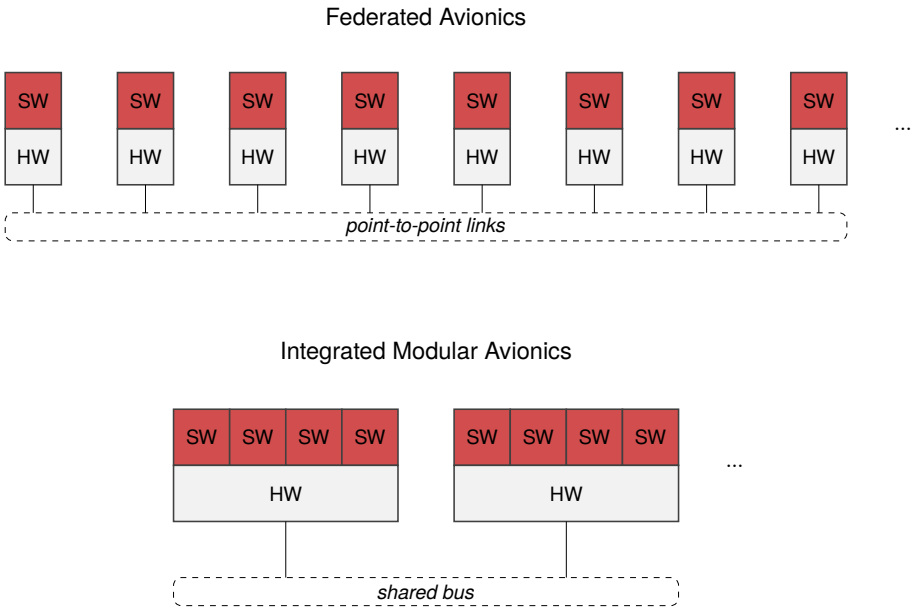


Figure 1.1: Simple overview of Federated and Integrated Modular Avionics.

distributing partitions among the available hardware, all while verifying the functional requirements of the avionic system, consists in a complex multiprocessor scheduling problem, which for long has been known to be NP-complete [7].

The schedule is determined by the system integrator at the system integration phase. It consists in gathering all system characteristics and requirements from suppliers, distributing partitions over the available hardware, and scheduling them in a strictly periodic manner, such that, on one hand, there is no temporal overlap between two partitions in the same module, and also other requirements related to inter-partition communications and an appropriate redundancy configuration are met. The resulting schedule is then encoded in a specific XML format and integrated in the system configuration files at build time, with any modifications requiring the complete reintegration of the system. This process is insufficiently automated in industry, from the modelling of the problem requirements, to the scheduling itself, and its certification. Overall, system integration is often a long, manual process, consequently error-prone and inefficient, with modern systems requiring over 40 000 configuration entries.

Furthermore, as the IMA architecture matures, there is more demand to optimize this process, shorten development cycles, and provide more flexible systems. For this, partition scheduling is a key component, and modern approaches aim not only to satisfy all system requirements, but also to optimize them in a relevant way.

Partition scheduling, in its simpler form, consists in assigning partitions to the available processors, and defining a relative starting time offset, which implicitly defines all execution windows given the strict periodicity of the setting. This must be done in a way that delivers an appropriate redundancy configuration; hardware resources are not exceeded; time and space partitioning are maintained, meaning that execution windows in the same module cannot overlap; and other functional requirements are assured by guaranteeing timely communication between partitions.

1.2 Objectives

The purpose of this dissertation is to provide a methodology for solving the partition scheduling problem in IMA, that can cope with the increased complexity that modern avionic systems are experiencing. Furthermore, we are interested not only in providing valid solutions, but also in finding one that provides the system with more flexibility, thus an optimization criterion is considered which aims to increase system flexibility and scalability by allowing the expansion of partition execution windows without having to completely reschedule the whole system.

This thesis was produced in collaboration with GMV, inserted in its Aerospace and Defence department. GMV is the supplier of an Arinc 653-compliant RTOS named XKY, and the goal is also to develop a partition scheduler to be integrated in the configuration tool suite for this product.

1.3 Contributions

The contributions of this dissertation are as follows:

- A comprehensive mathematical model of the system is provided, containing distribution constraints, which restrict the assignment of partitions to modules, communication constraints, via limiting the delay in a chain of partition executions, and also a multiple window model is introduced, which allows partition execution to be divided in multiple windows, where only some of them must be scheduled strictly periodically.
- A Mixed Integer Linear Programming (MILP) formulation describes a subset of the overall problem.
- A sequential assignment algorithm and a Constraint Satisfaction Problem (CSP) formulation are developed to produce an initial assignment of partitions to modules.
- A local optimization algorithm based on Game Theory [28] is used to improve the schedule for a single model, and it is extended to accommodate inter-partition communications and multiple windows.
- Stochastic optimization algorithms are added to complement local search and explore larger portions of the search space.

1.4 Thesis outline

The present chapter introduces the motivation and objectives for this dissertation. The relevant concepts already mentioned in the Introdduction, like federated and Integrated Modular Avionics, time and space partitioning, and Arinc 653 are described in detail in chapter 2. Also in this chapter, the partition scheduling problem is described, and the chapter concludes with a review of literature on the subject.

Chapter 3 is dedicated to the mathematical representation of the problem, detailing the free variables, problem variables, constraints and optimization criteria. A MILP formulation describes part of the overall problem.

Chapter 4 describes the algorithms and strategies developed to heuristically solve generic problem instances, with considerations for computational performance.

In chapter 5, the developed algorithms are evaluated, test cases of varying dimension are defined, and the performance of the scheduling tool is analysed, by comparing it with exact approaches and related work.

Chapter 6 is the conclusion to this dissertation, and includes a discussion about future work. Also included in annex are pseudo-code for optimization algorithms (appendix A), and the complete dataset of our test cases (appendix B).

Chapter 2

Background

This chapter describes the state of the art regarding avionic system architectures and relevant concepts to the partition scheduling problem. In section 2.5, an overview of literature centred around the partition scheduling problem is given.

2.1 Evolution of avionic architectures

2.1.1 Federated avionics

Avionic systems have traditionally followed a federated architecture, with each component or subsystem having its dedicated hardware and software, in what is defined as LRUs. Suppliers were responsible for developing both the hardware and software, and supply it as its own self-contained black-box component. This 'one function – one computer' concept coupled with redundancy provided high safety and reliability. Applications have guaranteed, deterministic access to processor resources, and Input/Output (I/O) with bounded latency and jitter. Maintenance is straightforward and inexpensive, as LRUs can be readily replaced by equivalent ones. Most importantly, with loosely coupled LRUs, critical functions cannot be impaired by low-criticality functions, and the modularity increases fault containment.

However, the disadvantages of the federated architecture are evident. With a federated architecture, a function being added to the avionic system requires the addition of one of these LRUs, and this quickly escalates the mass, volume, cost and power consumption of the entire avionic system to infeasible amounts. Regarding costs, functions sharing a processor must be certified to the highest criticality level of those functions, and this encourages the usage of many processors with decreased utilization. On the other hand, modern processors have far more capability than a single critical function requires, and this constitutes an inefficient usage of resources [38].

2.1.2 Integrated Modular Avionics

The aviation industry aimed to adopt a new paradigm that keeps the benefits of the federated concept and solves the problems mentioned above. The industry has adopted the IMA concept starting with

the F-22 project, and reaching passenger aircraft in the 1990's, with the two best examples being the Airbus A380 and the Boeing 787 [37]. Currently, all new passenger aircraft models employ a form of this architecture. The main architectural principle is the introduction of shared computation resources, which contain functions from multiple applications – a Core Processing Input-Output Module (CPIOM). It rests on the concept of robust partitioning, specified in Arinc 651, which prevents a failure in a function or in hardware unique to a function to cause another function to fail, thus inducing fault containment. With this, each piece of software can be certified independently to a proper criticality level, and risk is reduced since critical software is isolated.

Early iterations of the IMA concept failed to achieve this desired feature, and combined industry effort pushed the establishment of the Arinc 653 standard, which specifies a standard interface between partitions and the underlying RTOS, called the Application Executive Interface (APEX). Arinc 653 and APEX are described in the section 2.2.

The standardized interface allows for the development, testing and certification of avionic hardware and software to be made independently, which means smaller companies can supply just a specific part of software, being it a RTOS or functionality packaged in a partition, which in turn increases market competitiveness and leads to reduced costs. With respect to hardware, weight and power requirements are substantially reduced, as well as the different types of hardware which are closer standardized, and this leads to decreased integration and maintenance effort.

Mairaj [38] compares federated and IMA architectures having surveyed 35 projects that underwent the transition. The results show weight reductions of around 50 % for all cases, volume reductions of 30 % to 40 %, power savings of 25 % to 30 %, and Mean Time Between Failures (MTBF) increasing by a factor of 1.4 to 3.8.

Adoption of IMA has been gradual, with developers adding higher criticality applications as confidence in the architecture was gained, and today, the overwhelming majority of avionic systems in passenger aircraft follows this architecture. However, one should note that even on modern aircraft, flight control systems are still implemented on a federated architecture, due to the need of tight synchronization and minimal jitter, as well as to concerns about redundancy management [37].

2.1.3 Improvements to Integrated Modular Avionics

Current developments on the IMA concept aim to further abstract the applications from the hardware. In classic IMA, peripherals such as sensors and actuators are connected directly to the CPIOMs, that often must contain specific hardware to interact with these. The introduction of Remote Data Concentrators (RDCs) allows to connect the peripherals directly to the avionic network. These are hardware devices that carry the necessary drivers for these peripherals and perform their I/O to the avionic network, hence, the CPIOMs, now simply called Core Processing Modules (CPMs), do not require specific hardware and are further standardized. Another advantage of this is the reduced cabling needed, since CPMs are usually confined to the avionics bay, potentially far from the peripherals. These systems are often called Distributed Integrated Modular Avionics (DIMA) since they distribute I/O across the aircraft [48].

This finally accomplishes another goal for IMA, increased flexibility and reconfigurability. Addition or removal of functionality does not necessarily require recertification of the entire avionic system, and with efficient usage of the computational resources, addition of functionality does not require more hardware.

A particular segment that is often overlooked is that of multi-mission Unmanned Aerial Vehicles (UAVs). These are especially constrained in terms of weight and power, and benefit from expedite reconfigurability. There is an industry effort to enable automatic reconfiguration, via implementing multiple partition schedules that are switched when required. Albeit the schedules are static in the way that they are determined and validated in the system integration phase, the goal is that they can be loaded automatically at runtime when required.

Also in the space segment, where weight minimization is even more critical, IMA is known by the designation 'Integrated Modular Avionics for Space (IMA-SP)'. In this domain there are substantial differences, mainly because space systems do not have the opportunity for human intervention, and their mass, volume and power are tightly constrained. In the case of satellites, this leads to there being typically one or two main computing modules connected to the payloads with robust data buses [31]. As a result, applications are integrated into a single binary, increasing the risk of fault propagation and forcing all software components to be certified to the highest criticality level. The adoption of IMA-SP would provide a hardware abstraction layer which in turn would allow software development for these applications to be divided among several teams, as well as reducing the integration effort and decreasing the overall system complexity due to its partitioning system [27].

Another active field in IMA is the inclusion of multicore processors. Multicore processors are replacing uniprocessor analogues in many application domains, with clear advantages in performance, weight, energy consumption, volume and cooling requirements [46]. However, safety-critical systems and in particular avionics have delayed its adoption, with the current cases where multicore processors are used disabling all but one core on each processor. It is clear that the technical advantages of multicore hardware and the manufacturing market moving away from uniprocessor technology are incentives for the aviation industry to adopt this technology. The main technical challenges to multicore technology in IMA stem from interference between applications, and loss of determinism. Multicore platforms allow true parallelism with multiple threads existing simultaneously on separate cores, but with shared access to physical resources still being serialized [35]. Possible solutions to these include new processor designs that focus on predictability rather than optimizing for the average-case, but the ideal is still to use Commercial off-the-Shelf (COTS) processors. Other non-technical difficulties include the burden of migrating legacy software and complicated certification [46]. Scheduling tasks for multi-core IMA systems is a complex problem which is not addressed in this dissertation.

2.2 Arinc 653

Arinc specification 653 [36] standardizes the interface between the RTOS and avionic application software. It is the product of joint effort by many major parties in the air transport industry, including airframe manufacturers, avionics and RTOS suppliers, governmental entities, and academia [17]. Figure 2.1

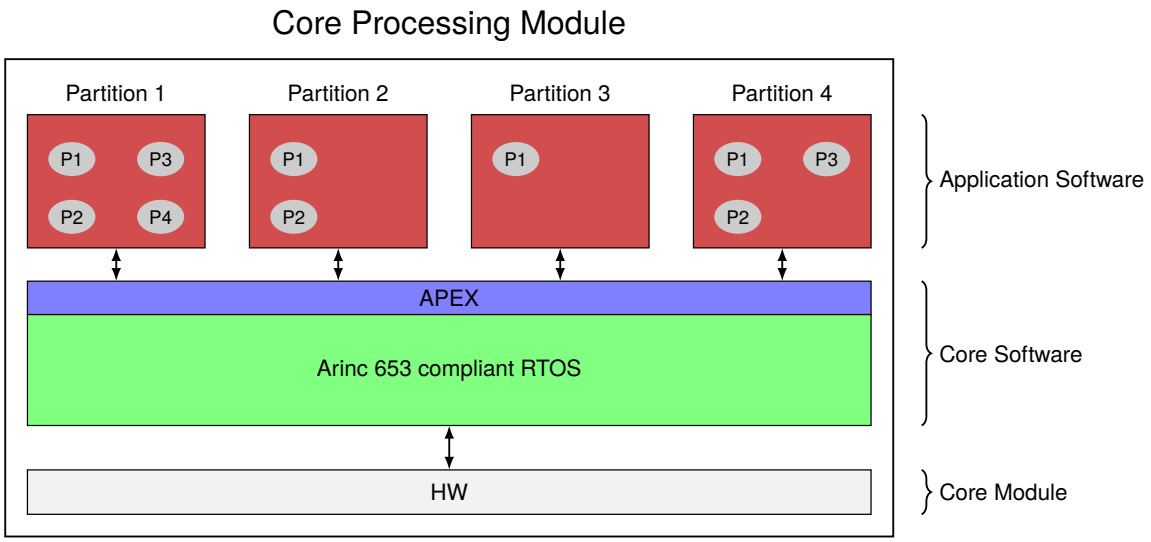


Figure 2.1: Arinc 653 architecture.

sketches the Arinc 653 architecture in one CPM, where the RTOS handles the partition management and exposes services to these partitions via the APEX. Internally, each partition consists an arbitrary number of periodic and aperiodic processes.

At the time of writing, the specification is composed of 6 parts, with part 1 being of special interest to this dissertation.

Part 0 is a global overview of the standard.

Part 1 specifies the set of required services the interface must implement.

Part 2 specifies extended services that can be optionally implemented.

Part 3 provides guidance for testing a RTOS conformity to the standard.

Part 4 includes a subset of Arinc 653 for simplified application domains, enabling formal methods for analysis.

Part 5 specifies the core software recommended capabilities.

On systems with shared resources, there are a number of ways one function can do damage to another one of higher criticality, resulting in an unexpected failure. These include: a function erroneously writing on memory belonging to another function; stealing processor time from a critical function, or crashing the processor; corrupting I/O, by either outputting data appearing to come from the critical function or corrupting it before that function uses it.

To remove these risks, the Arinc 653 specification resides on the concept of robust partitioning, which consists on spatially and temporally isolating applications, referred to as partitions. These are portions of software which bundle avionic functionality, and are analogous to processes in regular operating systems. However, a partition has internal processes, which divide functionality into smaller logical units, and the analogous of these in regular operating systems would be threads.

The partitioning is two-dimensional: space partitioning ensures protection of the program's data, dedicated I/O and registers, and time partitioning ensures unrestricted access to the processor and sufficient communication bandwidth in a given time window.

2.2.1 Space partitioning

Space partitioning should ensure that software in one partition cannot change data from another partition, either in memory or in transit, nor interact with private devices of another partition, i.e. peripherals [32]. This is achieved by providing each partition with exclusive access to certain memory regions. The implementation is software based, via high-integrity write protection and separate virtual memory spaces managed by the RTOS. The two key principles are: any persistent storage location must only be writeable by one function, and any temporary storage location used by a function, for example processor registers, must be saved when control is transferred away from that function and restored before resuming.

2.2.2 Time partitioning

Time partitioning is required so that a function has the necessary access to hardware resources for whatever time it requires, and doing so in a predictable way. The way this is achieved is by deterministically scheduling processor time to functions. At the partition level, static schedules are created at the system integration phase, assigning time windows or frames to partitions in such a manner that its correct functioning is assured. This is the origin of scheduling problem on which this dissertation focuses. On the process level, scheduling is done at runtime, via fixed priority task scheduling algorithms, namely rate monotonic scheduling.

2.2.3 Interface

The standard interface is known as APEX, and is composed of several main components [26, 36]:

- partition management,
- process management,
- time management,
- memory management,
- inter-partition communication,
- intra-partition communication,
- health monitoring.

Partition management services are related to running modes, and allow the system to start, restart or stop a partition when needed. The RTOS selects the process with the highest priority to run within a partition,

and APEX provides services to manage processes, like creating processes and collecting their status, changing priorities, and changing preemption status.

Time management allows managing deadlines, periodicity and budget times given to processes, as well as time-outs for communications.

Memory management is a major component of any operating system, however, in order to enforce space partitioning, APEX does not provide any memory management services. Instead, the partition memory space is statically reserved at build time.

Inter-partition communications allows for exchanging messages between partitions on the same or on different modules. These services preserve message ordering and hide any underlying division of large messages in smaller ones from the application. Communications are based on channels, which are logical links between a source and one or more destination partitions, and determine the characteristics of the messages being sent. APEX provides partitions access to channels via ports, which are configured by the system integrator and not the application developer. Two different kinds of ports exist: sampling ports store the message along with a freshness parameter (i.e. a timestamp) which can be queried by the destination application. Each message sent on that port overwrites the previous one, and these are meant for the kinds of channels that carry identical but updated data, such as sensor measurements. Queuing ports allow buffering of multiple messages in a First-In First-Out (FIFO) queue, which assures that every message is carried in the channel with preserved order, as long as the buffer memory is not full. Adding to these mechanisms, on certain cases, communication between partitions in the same module can be implemented with shared memory regions.

Finally, health monitoring periodically checks the system for errors or exceptions and dispatches the appropriate error handler, which can log the error, stop or restart the failed process or the whole partition.

2.3 Communications network

Communication buses for IMA systems have different requirements than older avionic systems. In particular, federated systems accept a one-to-one or one-to-many network topology, where any two nodes that must be connected require a dedicated cable with its own interface. The de-facto bus used until the adoption of IMA was Arinc 429. Since IMA systems eliminate many of these physical links between nodes, the data buses must offer higher bandwidth than Arinc 429, and the shared nature of these devices demands robustness and determinism.

Today, many competitive standardized data buses exist, with the most prevalent being Avionics Full-Duplex Switched Ethernet™ (AFDX). AFDX™ was developed by Airbus and later became standardized in the Arinc specification 664 [18], its main motivation was integration with IMA and usage of COTS components, namely Ethernet technology, while maintaining the required reliability. Determinism is guaranteed by the definition of virtual links, which emulate the point-to-point nature of an Arinc 429 bus, offering full-duplex capabilities. The virtual links are defined with a configuration table that specifies the network configuration, and through bandwidth reservation services, the bus guarantees bounded End-to-End (ETE) latencies in the network [12]. However, determining these worst-case ETE latencies still poses

a significant challenge in the system integration phase, and is also required for certification purposes; see Benammar et al. [44] for the problem of determining the worst case latencies, and Annighöfer and Thielecke [33] for an optimized approach to determining the network topology under AFDX™ and IMA.

Many other technologies with limited to considerable presence in the industry exist, we raise attention to Time-Triggered Ethernet (TTEthernet), which is used in distributed systems where tight synchrony is required. Messages on this network are statically scheduled, which guarantees they are correctly transmitted in the time slot they are given [34, 42].

2.4 Partition scheduling

An IMA system is composed of several CPMs with each hosting a set of partitions, with a static, cyclic partition schedule. The schedule is static because it is configured at build time, and does not change at runtime. A partition is forcefully stopped at the end of its allocated time window, and control transferred to the next one to ensure fault containment. It is cyclic because a representative unit is continuously repeated while the system is active.

2.4.1 Partition scheduling model

Scheduling partitions in IMA involves decisions in two domains. Firstly, IMA makes it possible that partitions are capable of running on many if not all available CPMs, but the schedule restricts that each must run on only one. Part of the scheduling problem consists in assigning partitions to different processors, verifying their real-time constraints. These can include memory, stack-size, bandwidth, as well as other constraints related to redundancy management. Also, each partition must be allocated time windows to execute while also verifying time segregation with other partitions in the same module, and being able to communicate with other elements in the avionic network.

Partitions are characterized by a period and an execution requirement, which are measured in integer units of time, noting that the highest precision for time measurements in real-time computing systems is the CPU clock period. Partitions are executed strictly periodically, which means that the time separating two consecutive execution windows (or instances, jobs) of the same partition is exactly the partition period. The period is defined based on functional requirements of the application, and is considered a step prior to scheduling. For information on this step, see for example Nasri and Fohler [39]. This strict periodicity is common in real-time systems as it is required by control loops for example, but it must be noted that the Arinc 653 standard does not enforce this. What is required is that there is a ‘periodic processing start’, a point in a partition schedule coinciding with the beginning of a window where the internal periodic process scheduling is allowed to start [36]. The execution requirement is taken as the Worst-Case Execution Time (WCET) of the partition and can be provided by the application developer, or determined through testing since it is dependent on the hardware. The smallest unit of repetition of the schedule in one CPM is called the Major Time Frame (MTF), or in other words the hyper-period of the partitions scheduled in that CPM. This is the smallest time window that is indefinitely repeated, and is equal to the Least Common Multiple

(LCM) of the partition periods, which guarantees that at least one partition time window be allocated to each partition in the duration of one MTF. The MTF concept is shown in figure 2.2, with all partition periods sharing a factor of 2, making the MTF equal to the largest one.

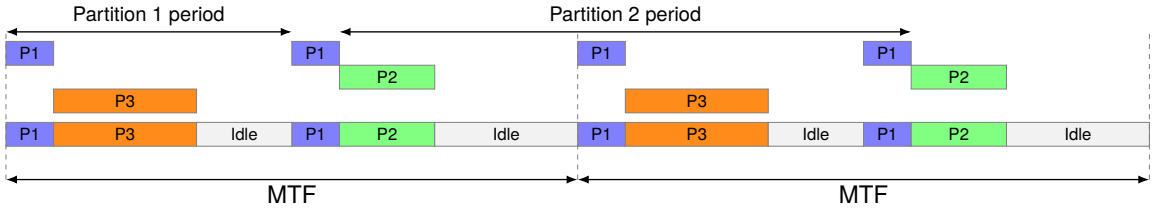


Figure 2.2: Example schedule with 3 partitions.

Given the strict periodicity of partition executions, once an execution window is defined, all subsequent ones are implicitly defined, thus the starting time offset with respect to the MTF is sufficient to fully describe the schedule of a partition. The typical partition scheduling problem is combining this with distributing the partitions among the available CPMs, complying with some constraints. This is part of the problem we aim to solve, and the constraints considered are exclusion, inclusion (or cohabitation), domain, memory, temporal segregation and communication constraints, these are described in detain in chapter 3. Furthermore, this is transformed into a Combinatorial Optimization Problem (COP), by the defining an optimization criterion based on flexibility, as introduced in [20], which intuitively consists in providing each partition with room to increase its execution window without interfering with other partitions. In practice, this is achieved by scheduling idle time in-between partition executions, and this is described in detail in section 3.3. In addition to this, we also investigate an extended problem variant explained next.

2.4.2 Extended model

This work expands on classical partition scheduling problem by allowing each partition instance to be split in multiple windows at the partition schedule, maintaining strict periodicity for the first window, as sketched in figure 2.3. This requires suspending a partition in the middle of its execution, which is ultimately preemption, therefore the cost associated with preemption must be evaluated and prevented from affecting the system behaviour. In general, preemption is undesirable in real-time systems due to the following issues:

- Preemption destroys program locality, increasing cache misses and ultimately the execution time, making WCETs harder to characterize [29].
- For control applications, the I/O delay and jitter should be minimized, and this is achieved when the process is allowed to run continuously and non-preemptively [29].
- The actual context-switching mechanism takes time, and is not negligible compared to the partition execution requirements.

However, it improves schedulability and allows for higher processor usage loads. The first issue on the previous list can be tackled by implementing cache restoration on context switches, and the third

by experimentally evaluating the time expended on context switching, similarly to how the WCETs are determined, and taking this time into account. Additionally, we can control exactly where preemptions are possible, thus forbidding interruption on critical sections and I/O, but the disadvantage is that the system integrator needs to have knowledge of the partition internal details. Control-related applications remain unsuitable to have their execution instances split into multiple windows.

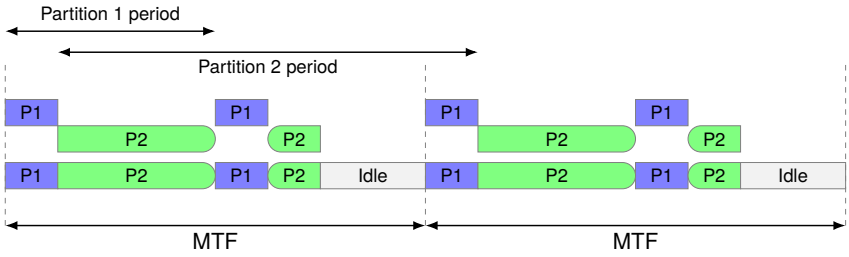


Figure 2.3: Partition schedule with some executions split in two windows.

One specific motivation for introducing multiple windows per instance is the case shown in figure 2.3, where the gaps left by the periodic execution of a partition with a small period are smaller than the execution requirement of another partition. In this case, maintaining both temporal segregation and strict periodicity is only achievable by using multiple windows per instance.

Using more execution windows as described adds complexity to the problem, for the solution will also need to specify the subdivision in windows, and any additional one can be freely scheduled as strict periodicity does not apply. Throughout this dissertation, this extension to the problem is considered separately.

2.4.3 Multiple schedules in DIMA

As stated in section 2.1.3, one goal in the future of DIMA is for the CPMs to autonomously react to a change in configuration, such as introduction of mission specific payloads, by loading the necessary applications and thus changing the partition schedule. Additionally, changing schedules can be a form of fault management, for example by having specific schedules that are loaded to compensate for partitions or CPMs failing.

However, all the possible configurations are static and still require independent certification, hence, from the point of view of this problem, they are considered distinct scheduling problems altogether.

2.5 Related work

2.5.1 Scheduling in real-time systems

The pioneering work of Liu and Layland [2] proves the completeness of the earliest deadline first scheduling algorithm for the dynamic scheduling of real time preemptive tasks on a single processor, that is, the algorithm always finds a valid solution if one exists. Considerable amount of work was devoted to

preemptive runtime scheduling, for a sample see Buttazzo [24], and Easwaran et al. [19] in the avionic domain.

Strictly periodic scheduling has received less attention in comparison. Korst [4] introduced this problem in the context of digital signal processing, deriving a necessary and sufficient condition for schedulability of two tasks. Following work by Korst et al. [7] proves that the problem is NP-complete even for the case of one processor.

Several authors [13, 21, 45, 47] approach real-time multi-processor scheduling as a case of the bin-packing problem, with the ultimate goal being to minimize the number of processors needed to schedule all tasks. Mayank and Mondal [45] focus on non-preemptive periodic tasks, their approach consists in sequentially assigning tasks to processors using variations of the best-fit and first-fit heuristics, followed by scheduling using a non-preemptive earliest deadline first algorithm. Zheng et al. [47] study the scheduling problem of non-preemptive strictly periodic tasks on multiple processors, modelling it with MILP which can yield optimal solutions. A heuristic sequential task assignment algorithm is proposed which compares favourably to the exact one, in the sense that it offers inferior time complexity at an acceptable cost in optimality.

Schedulability conditions for the strictly periodic non-preemptive case are derived in Marouf and Sorel [25], motivated by the avionics domain. The analysis is done separately for harmonic and non-harmonic periods, with the schedulability conditions for the former being necessary and sufficient. For the latter, only a sufficient condition exists, which can be used to assert if a set of tasks remains schedulable after the addition of a new task, thus a sequential process can prove schedulability for any task set.

2.5.2 Partition scheduling for Integrated Modular Avionics

Authors model the partition scheduling problem in different ways, including mixing process and partition-level scheduling. The prevalent modelling methods used are MILP and one instance of Satisfiability Modulo Theories (SMT), as well as other heuristic approaches. MILP or Integer Programming (IP) formulations are the preferred choice for general discrete COPs, and are able to reach optimal solutions via branch and bound or branch and cut algorithms. However, for intractable problems like this one, large dimension problems cannot be tackled by these methodologies in acceptable time, therefore there is a global effort in literature to employ heuristic algorithms for partition scheduling.

Lee et al. [10] represent the first efforts to automate scheduling for IMA platforms. They address the problem of scheduling partitions as well as the corresponding tasks in a two-level schedule, considering also message transmission between tasks. Tasks are scheduled with a fixed-priority preemptive schedule, and partitions are iteratively assigned a starting time until the requirements are met. The partitions are assumed to have harmonic, strict periods, and part of their execution time is dedicated to communications. Practical constraints to the IMA domain are suggested: the replication of partitions as a redundancy mechanism; the possibility to modify partition characteristics without redoing the schedule as a means to decrease recertification costs; and the time tick based schedule, which means that all variables with units of time must be integer numbers, usually of microseconds. Although this work laid the groundwork for the

formal definition of the requirements for partition scheduling, the iterative algorithm used is insufficiently robust for modern instances of the problem, given that it is essentially brute-force search.

Easwaran et al. [19] focus on the scheduling of partition processes, considering a pre-defined partition scheduling policy, using the deadline monotonic algorithm. Each process is subject to a periodic release time that is affected by a maximum amount of jitter, which consists in loose periodicity. The description is accurate such that it considers communications by limiting the time of processing chains, and considers the impact of preemption overheads and blocking times, yielding formal guarantees for certification purposes. However, it is incomplete as scheduling is carried out independently for each module, and a simplistic solution is taken for partition-level schedules.

Eisenbrand et al. [22] solve the problem of scheduling partitions with strict periodicity requirements in the minimum number of processing modules using an IP formulation. They show that their formulation optimally solves this bin-packing problem and outperforms similar implementations, with computation times taking less than 15 min for large examples provided by Boeing. The constraints supported are mostly Knapsack constraints, by limiting memory and bandwidth at each module, but there are also redundancy and cohabitation constraints. What is missing is inter-partition communications, as it is only considered that not exceeding the modules' bandwidth is considered sufficient to guarantee communication. Overall, this work is shown superior to other approaches to the problem of minimizing the number of modules.

This optimization is of undeniable interest for applications where weight minimization is paramount. However, often the hardware configuration is defined even before the scheduling phase, and in any case the resulting schedules are congested, leaving the system difficult to adjust.

Al Sheikh et al. [20] deal with partition scheduling with strict periodicity constraints and inter-partition communications. The optimization criterion consists in maximizing the worst-case scalability potential of every partition. The communication model is based on processing chains, and considers an asynchronous network. An exact MILP formulation is used which solves the problem for small scale examples, however, it fails to converge for fairly large problems in acceptable time. A preprocessing step based on graph theory is proposed, and it achieves a substantial reduction in computation times, provided all CPMs have identical characteristics.

This optimization criterion is valuable to avionic systems because it facilitates integration and maintenance, as small adjustments to partition execution budgets are possible. The approach is also extensive with respect to the constraints supported, however, the concept under the communication model is questionable, as supplementary delay is assumed for all communications to account for asynchrony between CPMs, even if the communication is performed within one CPM. Also, being a purely exact approach, it is unable to handle cases with modern dimension, notwithstanding the great reduction in solution time achieved by the preprocessing step.

On their following paper, Al Sheikh et al. [28] introduce an alternative method to perform local search based on the best response algorithm coming from Game Theory. An initial schedule is found from a greedy algorithm, then partitions update, one at a time, their allocations and starting times until no changes can be made that benefits any of them individually. It is shown that this algorithm converges in a finite number of steps to a local maximum, and that the optimal solution is found for a good enough

starting point. A multi-start method with Bayesian stopping rules is used to explore a greater portion of the search space and stop when some probabilistic certainty is obtained that the solution found is optimal. However, communication constraints are dropped to accommodate this heuristic.

Results show that the described heuristic compares favourably to the exact formulation, reaching optimal or close to optimal solutions much quicker. Since this essentially depends on the initial state, the authors attempt to increase the likelihood of finding the optimum by using more starting states. However, this consists in solving the whole problem many times, and could be improved.

On a completely different paradigm, Beji et al. [34] aim to minimize the cost of integration in IMA systems, considering the partition scheduling and network topology based on TTEthernet. They distinguish hard constraints, related to the network and the usual partitions segregation requirements, and soft constraints, related to the integration costs. Hard constraints should always be satisfied whilst soft constraints only impose a penalty if they are not. The approach is a holistic scheduling of partitions and messages in the TTEthernet network, with detailed constraints related to the TTEthernet technology. Distribution constraints are restricted to redundancy at the partition level, and in fact for some cases it is required that the different partition instances run synchronously, on different CPMs. The problem is encoded in SMT language, a form of constraint programming, and solved with an SMT-based tool for a relatively small example.

The optimization performed is interesting as it aims to minimize costs, and this work also showcases the relevancy of TTEthernet in the avionic domain. This is also the main disadvantage, as the model is specific to this technology, hence it is not portable to other IMA implementations.

Pira and Artigues [41] also use a Game Theory heuristic proposed in [28] for scheduling strictly periodic tasks on multiprocessors, and compare it to an exact MILP solution. In particular, they found that for large-scale examples, the heuristic could yield at least a feasible solution in minutes, while the number of variables was too large to even load the MILP solver. Their contribution consists of detailing efficient methods for computing the optimal partition offsets in this heuristic, which are based on linear programming, improving the performance of the methodology presented in Al Sheikh et al. [28].

Melani et al. [46] explore multicore scheduling in the context of IMA. The authors propose to mitigate the introduced temporal unpredictability by forcing synchronous context switching across cores, and splitting the partition execution times in a critical section and an optional section. The solution is a mixed pre-runtime and runtime scheduler, partitions are still assigned fixed slots in a conservative manner, but at runtime, a resource reclaiming mechanism redistributes the unused time budgets. The simulation results show that this approach is a promising solution for implementing IMA on multicore platforms, and a valid framework for describing these systems. On the other hand, a runtime component of this schedule is not anticipated in Arinc 653, and as a result, a solution like this cannot be implemented in the foreseeable future.

Blikstad et al. [49] approach the problem of pre-runtime scheduling of tasks with loose periods on an IMA platform with a MILP formulation. This is done at the process level and also considering communication between tasks, which are characterized explicitly by their release times, deadlines, and execution times, as well as knapsack and other distribution constraints. The approach consists on a comprehensive

mathematical model of the whole avionic system, with a constraint generation procedure to supply the MILP-encoded model. Given this, no particular optimization goal is given, and the execution time is significantly higher than the other approaches – ranging up to weeks for larger examples.

The main issue, in my opinion, is the excessive abstraction used, which does not clearly distinguish partitions and processes in the context of Arinc 653. Also, as with the other approaches that use IP/MILP, the solution time is substantial, although for the author's industrial application with Saab, this was deemed acceptable.

Table 2.1 summarizes the different approaches in literature, by the order they were introduced.

2.5.3 Relevant scheduling work in other domains

The present scheduling problem is rather specific to the avionic domain. Below we list two contributions in the general domain of scheduling that are somewhat similar to ours.

For scheduling in more general applications the reader can refer to Pinedo [16], providing a full overview of the topic, with special attention to the manufacturing and service industries. Stochastic models are used for dealing with uncertainty, something that sees no application in real time systems, except when determining worst-case values for quantities like communication delays and execution times. Special attention is called to heuristic methods for undertaking optimization problems in general, such as genetic algorithms, Ant Colony Optimization (ACO), Simulated Annealing (SA), Tabu-search, beam-search, and agent-based and machine learning methods, which inspired the approach in this dissertation.

At last, an alternative approach to periodic scheduling is done in Bar-Noy et al. [11], which represent the problem using trees (acyclic directed graphs). Their problem consists of a set of clients requesting from a service a fraction of the available bandwidth, for example in broadcast disks or Bluetooth 'park mode'. Every client should get a strictly periodic time slot to access the service, and the goal is to minimize the difference between the clients' assigned and requested bandwidths. It is shown how to construct periodic schedules from trees, where each client is a leaf node and its period proportional to the degree of the nodes leading to it. An optimal algorithm for constructing the optimal tree is given, which runs in exponential time, and several heuristic algorithms approximate the optimal solution in polynomial time. It is noted, however, that not every periodic schedule has a tree representation.

2.5.4 Contributions

This dissertation follows a problem model similar to Al Sheikh et al. [20] and with the same optimization goal, but considering synchronous communication between partitions, and additional constraints restricting on which CPMs partitions can be scheduled.

Identically, the MILP model is adapted to our variant of the problem, and the best response heuristic algorithm [28, 41] is adapted to perform local search. For global search, stochastic optimization algorithms, namely SA, Tabu-search and a genetic algorithm are used to complement global search, as presented by Pinedo [16].

We also include an extended model where partition execution can be divided in multiple windows, and

this component is novel in literature. The best response algorithm is adapted to solve this case as well. However, this analysis is done separately and is not included in the MILP model.

Table 2.1: Summary of related work in IMA partition scheduling.

Reference	Optimization	Methodology	Constraints		
			Timing	Distribution	Communication
Lee et al. [10]	–	Constraint-based	Strict periods	Redundancy	Process-level
Easwaran et al. [19]	–	Runtime scheduling with deadline monotonic, process-level	Loose periods with minimum jitter	–	Process-level
Eisenbrand et al. [22]	Minimize the number of CPMs	IP	Strict periods	Redundancy, cohabitation, Knapsack	–
Al Sheikh et al. [20]	Scalability	MILP	Strict periods	Redundancy, Knapsack	Partition-level
Al Sheikh et al. [28]	Scalability	MILP and best response heuristic	Strict periods	Redundancy, Knapsack	–
Beji et al. [34]	Integration cost	SMT	Strict periods, synchronous redundancies	Redundancy	Partition-level
Pira and Artigues [41]	Scalability	MILP and best response heuristic	Strict periods	–	–
Melani et al. [46]	–	Dedicated algorithms	Multicore, synchronous context switching	–	–
Blikstad et al. [49]	–	MILP	Loose periods	Redundancy, cohabitation, Knapsack	Partition-level
This dissertation	Scalability	MILP, best response heuristic, stochastic optimization	Strict periods	Redundancy, cohabitation, Knapsack	Partition-level

Chapter 3

Problem Model

This chapter presents the mathematical model for the partition scheduling problem. We define the problem variables, constraints, and an optimization criterion, describing how these relate to the motivation for this problem. Feasibility and schedulability conditions are given in section 3.2, and the problem is summarized in section 3.4 in the form of a Mixed Integer Linear Program.

3.1 Problem Definition

Consider a set of N_p partitions $\mathcal{P} = \{p_1, p_2, ... p_{N_p}\}$ to be scheduled in N_c modules $\mathcal{C} = \{c_1, c_2, ... c_{N_c}\}$. Partitions $p_i \in \mathcal{P}$ are characterized by:

- e_i – execution requirement in units of time, taken as its WCET.
- T_i – period, in units of time.
- s_i – memory requirement, in arbitrary units.

Modules $c_m \in \mathcal{C}$ are characterized by:

- S_m – memory capacity, in the same units as s.
- ε_m – context switching cost, in units of time.

This context switching cost is a time penalty added to the partition execution when it is divided in multiple windows, corresponding to the time taken to restore the execution state. In usual real life instances, these quantities would be the same for every module.

The assignment of partitions to modules is represented by variables $f_i,\ \forall p_i \in \mathcal{P}$, which denotes that partition p_i is assigned with the module with index $m = f_i$. For convenience, we also define $\mathcal{P}_m \subseteq \mathcal{P}$ as the subset of partitions scheduled in module c_m. That is,

$$\mathcal{P}_m \equiv \{p_i \in \mathcal{P},\ f_i = m\}. \tag{3.1}$$

The module MTF is the hyper-period of the partition periods hosted in each module:

$$H_m \equiv lcm\{T_i\},\ p_i \in \mathcal{P}_m, \tag{3.2}$$

where *lcm* denotes the least common multiple operator. Under this, a partition executes $K_i = H_m/T_i$ times, and these individual executions are called jobs (notice K_i is an integer due to the definition of H_m). Let now t_i be the starting offset for partition p_i, such that this partition is scheduled to start at strict periodic instants $t_i + kT_i,\ k = 0, 1, \ldots K_i - 1$. To verify that all K_i jobs fit in one hyper-period, one must have

$$t_i + (K_i - 1)T_i + e_i \leq H_m, \tag{3.3}$$

which represents the finishing time of the last job. Plugging $H_m = K_i T_i$ yields the important condition:

$$t_i \leq T_i - e_i. \tag{3.4}$$

Figure 3.1 shows the introduced timing notation in a schedule with non-harmonic periods. By definition, periods T_i, T_j are harmonic if and only if T_i divides T_j or T_j divides T_i.

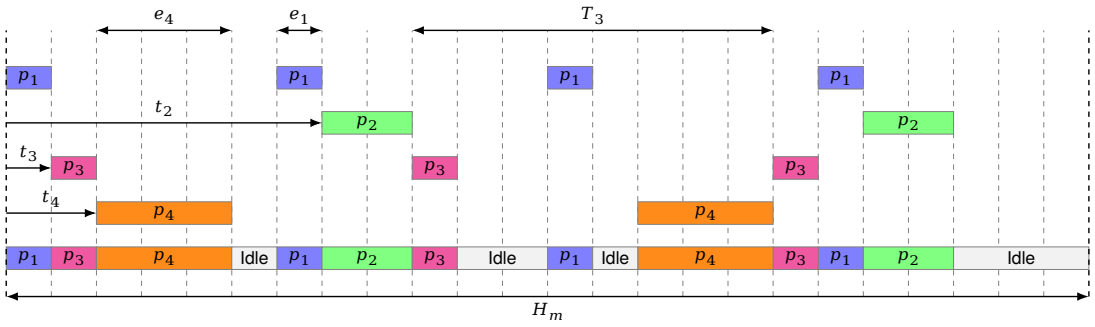


Figure 3.1: Schedule with annotated timing variables. Partitions have non-harmonic periods.

Assuming single execution windows, a partition executes in time windows $[t_i + kT_i,\ t_i + kT_i + e_i],\ k = 0, 1, \ldots K_i$, and temporal segregation requires that these windows do not overlap for partitions in the same module. Partition execution in multiple windows is considered a different problem and is described in section 3.1.3. All timing variables are integers, and in particular periods and durations are strictly positive.

3.1.1 Distribution constraints

The assignment of partition to modules is restricted by constraints which we call distribution constraints.

Exclusion – Two partitions are said to be in exclusion if they cannot be assigned to the same module. This covers, but is not restricted to, the redundancy requirements of an avionic system, where safety-critical functionality must be replicated in different machines. An exclusion constraint between partitions p_i, p_j is denoted by $f_i \neq f_j$.

Inclusion – Two partitions are said to be in inclusion if they must be placed in the same module, which

is useful for applications that are tightly coupled. Similarly to exclusion, an inclusion constraint between partitions p_i, p_j is denoted by $f_i = f_j$.

Domain – A partition can only be assigned to a subset of all modules, called the partition's domain. That is the case in some architectures where applications require specific hardware that is only installed in some modules, like peripherals. The domain D_i of a partition p_i is essentially a list of modules it can be assigned to. Formally, $D_i \subseteq \mathcal{C}$ such that:

$$c_m \notin D_i \implies f_i \neq m. \tag{3.5}$$

Memory – This is the only Knapsack constraint considered. For each module, the sum of the partitions' memory sizes much not exceed that module's memory capacity:

$$\sum_{p_i \in \mathcal{P}_m} s_i \leq S_m. \tag{3.6}$$

Uniqueness – This constraint simply imposes that each partition is assigned to exactly one module. Using f_i notation, this constraint is implicit, but with the abbreviated $\mathcal{P}_m$ notation it is expressed as:

$$\begin{cases} \mathcal{P}_1 \cup \mathcal{P}_2 \cup \ldots \cup \mathcal{P}_{N_c} = \mathcal{P} \\ \forall c_m, c_n : \ \mathcal{P}_m \cap \mathcal{P}_n = \emptyset. \end{cases} \tag{3.7}$$

3.1.2 Communication constraints

Accounting for inter-partition communications can initially be interpreted as precedence relations, where the execution of one partition depends on the previous execution of another partition. However, since the schedule is cyclic, these are always true given enough time.

More thoroughly, inter-partition communications are often represented as processing chains, consisting of some kind of data being treated by successive partitions. One can think of data originating from a sensor or user input, being processed by one or more partitions, then originating a certain response in its final destination [28].

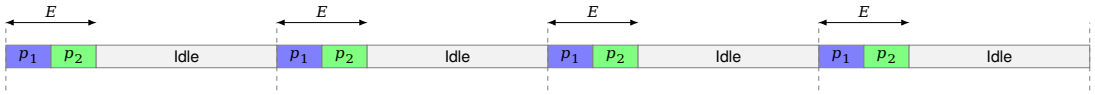
For this problem, we will consider such chains, but limit them to two partitions only, such that the time taken to process the data from its origin in the sender partition to its consumption in the receiver partition is bounded.

A chain linking p_i to p_j is subject to a maximum delay $E^{max}_{i,j}$, so we can describe all communication constraints by a matrix $[E^{max}]$, with entries being infinity when there is no communication between partitions. The chain processing time is denoted $E_{i,j}$, measured from the start of p_i to the end of p_j, and must verify

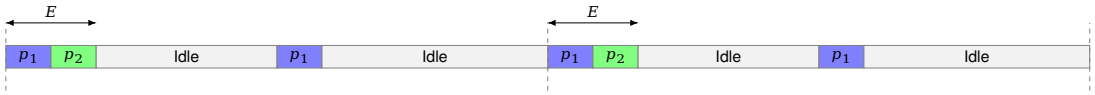
$$E_{i,j} \leq E^{max}_{i,j}. \tag{3.8}$$

This definition is agnostic to which jobs actually participate in the chain. If the two partitions have equal

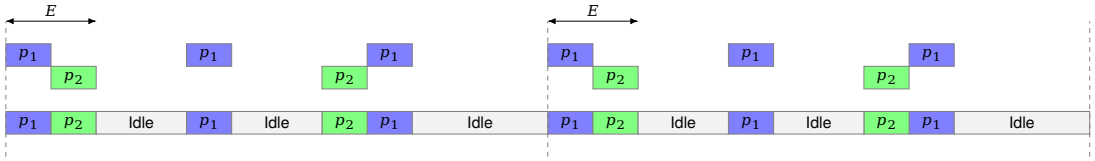
periods, then the delay between two consecutive jobs is constant, but if the periods are not equal but still harmonic, then $E_{i,j}$ is defined as the shortest delay, and the chain can occur with a period equal to the larger of the two partition periods. When the two partition periods are non-harmonic, then for simplicity we consider also the smallest delay, and the chain shall be repeated with a period equal to the hyper-period of these two partitions. See figure 3.2 for clarification.



(a) Chain with $T_1 = T_2$.



(b) Chain with $2T_1 = T_2$.



(c) Chain with $2T_1 = 3T_2$.

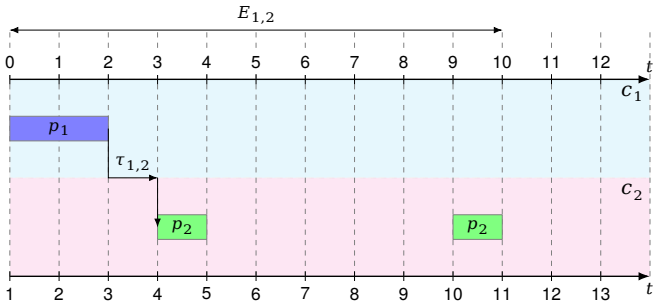
Figure 3.2: Duration of chains for different periods.

The examples in figure 3.2 show chains where the partitions involved are assigned to the same module. If they are assigned to different modules, the network delay between these two modules must be considered. The network is characterized by a matrix [T] with elements $\tau_{m,n}$ of maximum ETE delays, which are upper bounds to the actual communication delay between modules c_m, c_n. Communications between partitions in the same module are not subject to network delays, thus we define $\tau_{m,m} = 0,\ \forall m$. We can also assume $\tau_{m,n} \ll T_i,\ \forall m, n, i$, by at least one order of magnitude.

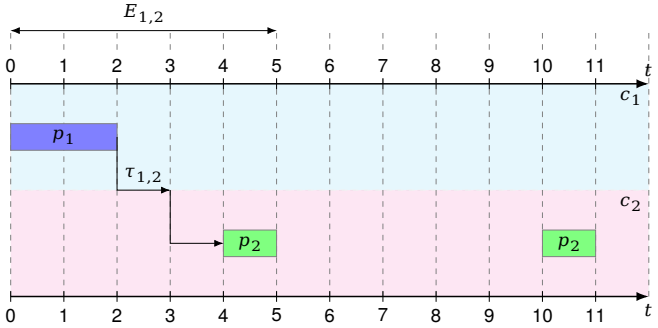
Communications between modules are fundamentally different depending on the specific implementation. If the modules run asynchronously, independent of the network delay, the message can arrive at an arbitrary instant in the second module's schedule, thus the worst case must be considered. This corresponds to the message arriving immediately after the second partition checks for messages, in which case the processing chain can only complete in the next job of this partition. These are the circumstances considered in Al Sheikh et al. [20].

The situation considered here is when the modules run synchronously. Intuitively, this means the two modules' schedules are aligned, and the instant when the message arrives in the second module is known. In this case, we do not need to consider the worst case at all times, the message is only processed in the next job when it arrives past the time the receiving partition checks for messages. It is also considered that all messages are sent and received in the end and beginning, respectively, of a partition's execution window. The earlier assumption $\tau_{m,n} \ll T_i$ is meant to prevent messages from being delayed for more than one period.

Figure 3.3 sketches both cases, where we have $e_1 = 2$, $e_2 = 1$, $T_1 = 12$, $T_2 = 6$, $t_1 = 0$, $t_2 = 4$, and $\tau_{1,2} = 1$. In figure 3.3a, the modules are operating with an arbitrary offset, thus the worst case is to consider module c_2 to be running 1 time unit ahead, such that the message arrives in the critical instant that p_2 is checking for messages, and must wait for the next job of this partition. In contrast, in figure 3.3b the modules are synchronized and the message arrives in time to be processed in the current job of p_2.



(a) Chain between modules running asynchronously with an arbitrary offset.



(b) Chain between synchronous modules.

Figure 3.3: Inter-module communications example.

To be noted that in the asynchronous variant, the actual schedule is irrelevant for chains when the two partitions are assigned to different modules, and it is sufficient that the network delay is compatible with the maximum processing time:

$$E_{i,j} = e_i + \tau_{m,n} + T_j + e_j, \qquad m \neq n. \tag{3.9}$$

This entails that schedules complying with the asynchronous model will also comply with the same requirements considering the synchronous model.

3.1.3 Multiple window model

For the extended model, it is considered that partition jobs (instances) can be divided in more than one window of execution, hence we introduce an extended model with specific constraints such that the real-time requirements of these partitions are verified. Henceforth, this is referred to as simply 'multiple windows'.

Consider now that, for each job k of a partition $p_i \in \mathcal{P}_m$, there are $M_{i,k}$ execution windows, with

lengths $(e_{i,k,1}, e_{i,k,2}, \dots, e_{i,k,M_{i,k}})$, such that:

$$\sum_{u=1}^{M_{i,k}} e_{i,k,u} = e_i + (M_{i,k} - 1)\varepsilon_m, \ \forall k. \tag{3.10}$$

The windows are represented as $\lambda_{i,k,u}$, and each has its own offset $t_{i,k,u}$, defined with respect to the hyper-period. Additionally, all windows that compose the partition are denoted by the set Λ_i. Since the first window at each job must be executed strictly periodically, its offset is not independent for all jobs, and is constricted by:

$$t_{i,k,1} = t_{i,1,1} + (k-1)T_i, \ \forall k. \tag{3.11}$$

The problem is greatly complicated because we require vector variables to completely represent the schedule, in particular because the number of jobs depends on the hyper-period, which is a function of the periods of all partitions assigned to that module, and the number and sizes of each window are also variable.

We require that the partition splitting be done only in predetermined points in order to limit the problem complexity, as well as due to reasons covered in section 2.4.2. These set of possible points for splitting is represented for each partition p_i as $\mathcal{B}_i$. Splitting a partition at job k in a subset of preemption points $\mathbf{b} \subseteq \mathcal{B}_i$ yields the window sizes $\mathbf{e}_{i,k} = (b_1, b_2 - b_1 + \varepsilon_m, \dots, b_N - b_{N-1} + \varepsilon_m)$, where we consider without loss of generality that $\mathbf{b}$ has N sorted elements.

The response time of a task is defined as the time taken from the task activation to when the task completes, and clearly, this concept is not relevant if the execution is made in a single window. With more than one window per job, the partition finishes executing in instants $\left\{t_{i,k,M_{i,k}} + e_{i,k,M_{i,k}}\right\}$, which prompts the definition of the response time, r_i, as

$$r_i = \max_k \left\{t_{i,k,M_{i,k}} + e_{i,k,M_{i,k}} - t_{i,k,1}\right\}, \tag{3.12}$$

restricted by a relative deadline, d_i:

$$r_i \leq d_i, \tag{3.13}$$

which is measured with respect to to the job start, $t_{i,k,1}$. All partitions have the implicit deadline $d_i \leq T_i$ to ensure that all windows finish before the next job starts.

Figure 3.4 sketches the new notation introduced in this section, where the windows corresponding to a subdivision of a job are represented with rounded edges.

3.2 Schedulability

The feasibility of the subproblem of assigning partitions to modules such that the distribution constraints are verified is analysed in section 4.1.

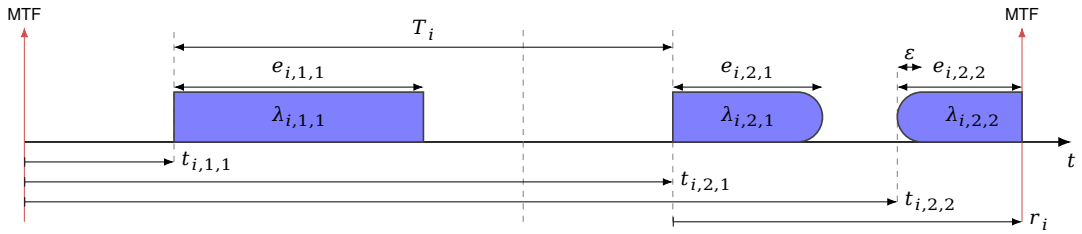


Figure 3.4: Multiple window execution notation.

A necessary and sufficient condition for two strictly periodic tasks to not overlap in time is derived in Korst et al. [7], which we reiterate for partitions. For two partitions $p_i, p_j \in \mathcal{P}_m$, let $g_{i,j}$ be the Greatest Common Divisor (GCD) of the two partition periods,

$$g_{i,j} = gcd\{T_i, T_j\}. \tag{3.14}$$

Then, the partitions do not overlap in time if and only if:

$$e_i \leq mod\{t_j - t_i,\ g_{i,j}\} \leq g_{i,j} - e_j, \tag{3.15}$$

where *mod* denotes the modulo operator (remainder after division). In particular, the unsigned modulo function is used, meaning that the result has the same sign as the divisor. In fact, the term

$$l_{i,j} \equiv mod\{t_j - t_i,\ g_{i,j}\}, \tag{3.16}$$

is the smallest separation between the start times of p_i and p_j, to which we call latency delay after Pira and Artigues [41]. From the properties of the modulo function and noting that $g_{i,j} = g_{j,i}$, equation 3.15 can be expressed as:

$$\begin{cases} l_{i,j} \geq e_i \\ l_{j,i} \geq e_j \end{cases}. \tag{3.17}$$

Figure 3.5 marks the latency delays for two partitions with $T_j = 2T_i$.

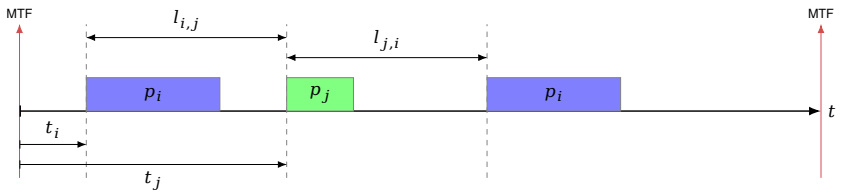


Figure 3.5: Latency delays, adapted from [28].

This condition is extendable to any set of partitions. For a set $\mathcal{S}$, we denote by $\mathcal{S}^2$ the set of every combination of two distinct elements belonging to $\mathcal{S}$. Then, a partition schedule for module m verifies temporal segregation if and only if equation 3.17 (or equivalently 3.15) is true $\forall p_i, p_j \in \mathcal{P}_m^2$.

From equation 3.15, it can be deduced that two partitions are schedulable in the same module if and

only if

$$e_i + e_j \leq g_{i,j}. \tag{3.18}$$

This becomes a sufficient condition for more than two partitions: $\mathcal{P}_m$ is schedulable if

$$\sum_{p_i \in \mathcal{P}_m} e_i \leq g, \tag{3.19}$$

where $g = gcd\{T_i, p_i \in \mathcal{P}_m\}$. In practice, this condition proves to be too strict, as there are many sets that do not verify this condition but that are easily schedulable. As an example, the schedule in figure 3.1 has $\sum e = 7$ and $g = 2$. Stronger schedulability conditions are derived in Marouf and Sorel [25]. The most general one asserts if a new candidate partition, p_j, can be added to an already schedulable set $\mathcal{P}_m$:

$$e_j \leq \sum_{p_i \in \mathcal{P}_m} e_i \cdot \delta\left[mod\{T_j, T_i\} \cdot \left(mod\{T_j, 2g\} + mod\{T_i, 2g\}\right)\right], \tag{3.20}$$

where δ is the Kronecker delta:

$$\delta(x) = \begin{cases} 1, & \text{if } x = 0 \\ 0, & \text{otherwise} \end{cases}. \tag{3.21}$$

Still, it remains a sufficient condition only, and it is unpractical since it should be applied iteratively to the set.

As a remark, an important motivation for the partition periods to be harmonic is schedulability, as this ensures that

$$g = \min_{p_i \in \mathcal{P}_m} \{T_i\}, \tag{3.22}$$

which is the maximum value it can take. In the other extreme case, if there are co-prime periods then $g_{i,j} = 1$, and these two tasks are not schedulable.

Schedulability can also be analysed with the processor usage fraction, U_m, given by:

$$U_m = \sum_{p_i \in \mathcal{P}_m} \frac{e_i}{T_i}. \tag{3.23}$$

This can be seen as the percentage of time that a processor is active, and for single core processors the set is clearly not schedulable when $U_m > 1$. For the complete problem, it is infeasible if:

$$\sum_{c_m \in \mathcal{C}} U_m > N_c. \tag{3.24}$$

Schedulability with communications

Using the definition of the latency delay, the chain processing time is given as:

$$E_{i,j} = \begin{cases} l_{i,j} + e_j, & \text{if } l_{i,j} - e_i \geq \tau_{m,n} \\ l_{i,j} + e_j + T_j, & \text{otherwise} \end{cases}, \tag{3.25}$$

with $p_i \in \mathcal{P}_m$, $p_j \in \mathcal{P}_n$. When there is enough leeway between the two partitions to accommodate the communication delay $\tau_{m,n}$, the chain can complete at the next instance of p_j, otherwise the chain prolongs for additional period of the second partition. We reiterate an assumption that $\tau \ll T$, which just prevents prolonging the chain by two or more periods instead.

From its definition, a chain is only possible when

$$e_i + e_j \leq E_{i,j}^{max}, \tag{3.26}$$

and additionally it is definitely possible if this is verified and p_i, p_j can be scheduled in the same module. On the other hand, the latency delay between two partitions is bounded by $g_{i,j}$, therefore a chain is always verified (independently from respective offsets) when:

$$g_{i,j} + e_j + T_j \leq E_{i,j}^{max}, \tag{3.27}$$

if scheduled in the same module, otherwise when:

$$e_i + e_j + max\{\mathrm{T}\} + T_j \leq E_{i,j}^{max}. \tag{3.28}$$

The term $max\{\mathrm{T}\}$ represents the maximum element of this matrix, which corresponds to the highest ETE delay in the network.

3.3 Optimization criterion

The optimization criterion chosen is one that aims to increase flexibility, by providing each partition a potential to increase its execution time. This is accomplished by leaving some idle time after each partition execution windows, and has two benefits,

- upon system maintenance or modification, one can add functionality to partitions, increasing their execution requirement, without having to recompute a new schedule,
- it mitigates uncertainty on the determined WCETs.

Possibly, it could also be viewed as allowing for the usage of slower, cheaper hardware, where the execution requirements would increase.

The evaluation function used is the α-parameter, which is the maximum factor that scales all partition execution requirements, such that the schedule becomes borderline valid [20]. It can be defined for each module, or for the whole system:

$$\alpha_m = min\left\{\frac{l_{i,j}}{e_i}\right\},\ \forall p_i, p_j \in \mathcal{P}_m,\ i \neq j \tag{3.29}$$

$$\alpha = \min_{c_m \in \mathcal{C}} \{\alpha_m\}. \tag{3.30}$$

This yields each partition further execution time in proportion to the original execution requirement, which is appropriate since more complex applications with longer execution requirements are more likely to need to be updated and/or expanded.

The single processor scheduling problem is formulated as:

$$\begin{aligned} max \quad & \alpha \\ \text{s.t.} \quad & \alpha \geq 0 \\ & e_i \alpha \leq l_{i,j} \\ & 0 \leq t_i \leq T_i - e_i \\ & p_i, p_j \in \mathcal{P}_m \ \ i \neq j \\ & t_i \in \mathbb{Z} \end{aligned} \tag{3.31}$$

A useful property of the α-parameter is that it is able to rate even invalid solutions. A value $\alpha = 1$ means a borderline valid schedule, a value $\alpha > 1$ a valid schedule that has some slack, and a value $\alpha < 1$ an invalid schedule which has overlaps. If we are not interested on the maximization problem but instead in the feasibility problem, plugging $\alpha \geq 1$ in the previous model transforms it into a CSP, for which many complete algorithms exist. Related work on this problem either uses the same optimization criterion [20, 28, 41], or instead aims to minimize the number of modules [21, 22, 47].

Figure 3.6 illustrates the optimization criterion. Note that in this figure, p_2 actually can increase its execution further, but the critical factor for the α-parameter is p_1. Also note the solution presented is sub-optimal, it is clear that shifting p_2 to the right increases α.

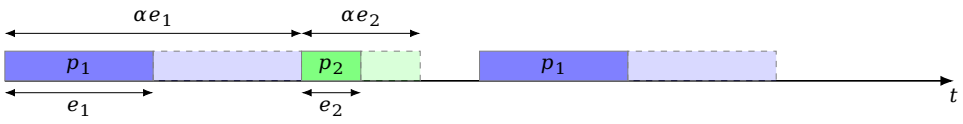


Figure 3.6: Effect of the α-parameter.

A concept that will be used in the next chapter is the partition utility, α_i, defined as follows,

$$w(i,j) = min\left\{\frac{l_{i,j}}{e_i}, \frac{l_{j,i}}{e_j}\right\} \tag{3.32}$$

$$\alpha_i(t_i) = \min_j \{w(i,j)\}, \tag{3.33}$$

for $p_j \in \mathcal{P}_m \setminus \{p_i\}$. It represents the module's α-parameter, as a function of the offset t_i, but considering

only the partition pairs where p_i is involved, and all other offsets fixed. It is distinct from simply the execution potential for that partition, because it also accounts for the effect on the remaining partitions in that module, as seen in expression 3.32.

Introducing chains and multiple windows adds some nuances to the α-parameter. With respect to chains, increasing partition executions also increases the chain processing time, so it would be reasonable to limit α not only to avoid overlapping, but also to avoid chains exceeding their predefined delays. However, the premise for α is adding functionality to partitions, so we can consider that the processes depending on the communication link still complete their tasks even if functionality is appended to the partition. This still entails that messages must be sent in places other than the end of an execution window, but this was only a simplification made for this model, and not a system requirement. The keynote here is that the α-parameter is not directly constricted due to chains.

Following the same idea, for partitions with multiple execution windows we consider that the execution potential is appended only to the last window at every job. The definition of α in this case is more complicated, as each execution window needs to be considered independently. In particular, note that since each partition job can have a different arrangement of execution windows, each window is only repeated with a period of H_m, the hyper period of the module where it is scheduled. A good abstraction to this is to define the window period to be this value, $T_{i,k,u} = H_m$.

We begin by redefining the latency delays for specific partition windows, abbreviating $a \equiv i, k_1, u_1$ and $b \equiv j, k_2, u_2$,

$$l_{a;b} \equiv mod\{t_b - t_a,\ g_{a;b}\}, \tag{3.34}$$

and for our purposes, the partitions will be assigned to the same module, thus $g_{a;b} = H_m$.

The utility is defined for each window as well:

$$w(a,b) = \begin{cases} \frac{l_{a;b}}{e_a}, & \text{if } u_1 < M_{i,k_1} \wedge l_{a;b} < e_a \\ \infty, & \text{if } u_1 < M_{i,k_1} \wedge l_{a;b} \geq e_a \\ \frac{l_{a;b} - e_a + e_i}{e_i}, & \text{if } u_1 = M_{i,k_1}, \end{cases} \tag{3.35}$$

$$\alpha_a(t_a) = min\{w(a,b_1), w(b_1,a), w(a,b_2), w(b_2,a), ...\}. \tag{3.36}$$

Additional execution is only appended to the last window of a job, so only the branch with $u = M_{i,k}$ in equation 3.35 computes the utility as expected. Here, the idle space is $l_{a;b} - e_a$, meaning that proportionally to the original partition execution, it can be scaled by $1 + (l_{a;b} - e_a)/e_i$. For all other windows, we are only interested in avoiding overlaps, which happen when $l_{a,b} < e_a$. The value in this case does not have any meaning, but it is convenient that it is less than 1.0 and proportional to the portion of the window that is overlapping in order to classify this as an invalid solution. Otherwise we can ignore the constraint, but note this will never cause the utility to be infinite, because it is defined as a minimum, and at least one value will be finite.

The simplest way to define the α-parameter of the module is as the minimum utility of all windows:

$$\alpha_m = min\{\alpha_{i,k,u}(t_{i,k,u})\}, \forall p_i \in \mathcal{P}_m, \forall \lambda_{i,k,u} \in \Lambda_i. \tag{3.37}$$

Figure 3.7 illustrates the concept of latency delays and utility when there are multiple windows. We labelled the regions A, B and C, which correspond to the three branches of equation 3.35, in the same order. Since there is overlap in region A, the utility (for either partition) is the minimum one between any two windows, which is a value less than 1.0.

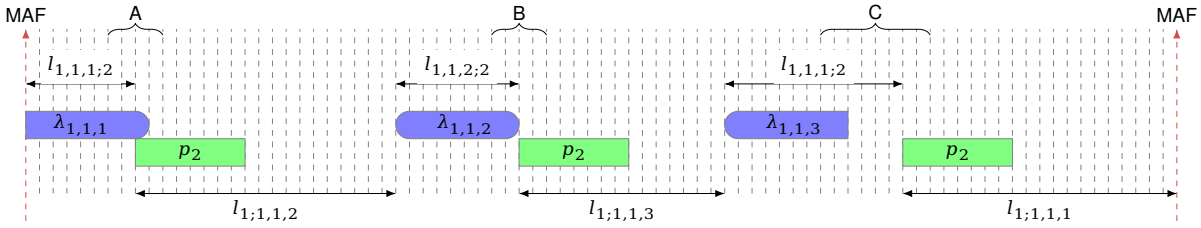


Figure 3.7: Partition utility with multiple windows.

3.4 MILP formulation

In this section we detail a MILP formulation that models the multiprocessor partition scheduling problem with communication constraints, but does not consider multiple execution windows. The classification as a Mixed Integer Linear Program comes from the fact that the formulation uses both integer variables (offsets and assignments) as well as real variables (the α-parameter) involved in linear constraints. This serves as a proof of concept that MILP can be used for combinatorial optimization problems, as every NP-complete problem is known to have a polynomial-sized ILP representation [14], and is commonly used for comparing with lighter, heuristic methods. The performance of a solver using this model is compared to other developed methods in chapter 5. The formulation follows closely that in [20], and uses the natural encoding of binary variables, 1-true and 0-false.

We begin by reformulating the assignment of partitions to modules. Let $a_{i,m}$ be a binary variable expressing that partition p_i is assigned to module c_m,

$$a_{i,m} = \begin{cases} 1, & \text{if } p_i \in \mathcal{P}_m \\ 0, & \text{otherwise} \end{cases}, \tag{3.38}$$

which requires $N_p \times N_c$ variables. The distribution constraints expressed with this new notation take the

following form:

$$\text{Uniqueness}: \quad \forall i \ \sum_{m} a_{i,m} = 1. \tag{3.39a}$$

$$\text{Memory}: \quad \forall m \ \sum_{i} a_{i,m} s_i \leq S_m \tag{3.39b}$$

$$\text{Exclusion}: \quad f_i \neq f_j: \ \forall m \ a_{i,m} \leq 1 - a_{j,m} \tag{3.39c}$$

$$\text{Inclusion}: \quad f_i = f_j: \ \forall m \ a_{i,m} = a_{j,m} \tag{3.39d}$$

$$\text{Domains}: \quad \forall i \sum_{c_m \in \{\mathcal{C} \setminus D_i\}} a_{i,m} = 0. \tag{3.39e}$$

Temporal segregation is ensured by equation 3.15, which is non-linear due to the modulo function. Linearisation requires the introduction as free variables of the quotient from the division,

$$q_{i,j} = \left\lfloor \frac{t_j - t_i}{g_{i,j}} \right\rfloor, \tag{3.40}$$

which enables rewriting condition 3.15 as

$$e_i \leq t_j - t_i - q_{i,j} g_{i,j} \leq g_{i,j} - e_j. \tag{3.41}$$

The α-parameter multiplies the partition executions, and these constraints should only apply to partitions in the same module, thus a 'Big-M' constant Z is used to trivially satisfy these inequalities when the partitions are not assigned to the same module. So, constraints 3.41 can be divided as follows:

$$\forall p_i, p_j \in \mathcal{P}, c_m \in \mathcal{C}: \quad t_j - t_i - q_{i,j} g_{i,j} \geq \alpha \cdot e_i - Z\left(2 - a_{i,k} - a_{j,k}\right) \tag{3.42a}$$

$$t_j - t_i - q_{i,j} g_{i,j} \leq g_{i,j} - \alpha \cdot e_j + Z\left(2 - a_{i,k} - a_{j,k}\right). \tag{3.42b}$$

This also shows that only one quotient q is needed for each pair of partitions, so to reduce the number of variables, it is defined only for $j > i$. The reciprocal variable would be $q_{j,i} = -1 - q_{i,j}$, thus the latency delays are:

$$\begin{aligned} l_{i,j} &= t_j - t_i - q_{i,j} g_{i,j} \\ l_{j,i} &= t_i - t_j + q_{i,j} g_{i,j} + g_{i,j}. \end{aligned} \tag{3.43}$$

Bounds for q can be supplied from

$$\frac{t_j - t_i}{g_{i,j}} - 1 < \left\lfloor \frac{t_j - t_i}{g_{i,j}} \right\rfloor \leq \frac{t_j - t_i}{g_{i,j}}, \tag{3.44}$$

noting that $t_i \in [0, 1, \ldots, T_i - e_i]$, which gives

$$\frac{e_i - T_i}{g_{i,j}} - 1 < q_{i,j} \leq \frac{T_j - e_i}{g_{i,j}}. \tag{3.45}$$

The strict inequality in 3.45 can be replaced by a regular inequality without compromising the solution.

Regarding the communications constraints, for each chain, let the new binary variable $x_{i,j}$ denote that the chain being delayed for one period. This is used to eliminate the branching in equation 3.25, having:

$$l_{i,j} + e_j + x_{i,j} T_j \leq E_{i,j}^{max}. \tag{3.46}$$

The network delay affecting the two partitions is denoted by the auxiliary variable $\hat{\tau}_{i,j}$, defined as:

$$\hat{\tau}_{i,j} = \sum_{c_m, c_n \in \mathcal{C}} a_{i,m} a_{j,n} \tau_{m,n}, \tag{3.47}$$

but this is non-linear in terms of our free variables. Linearisation is done by introducing new binary variables $y_{i,j,m,n}$:

$$y_{i,j,m,n} = \begin{cases} 1, & \text{if } p_i \in \mathcal{P}_m \wedge p_j \in \mathcal{P}_n \\ 0, & \text{otherwise} \end{cases}, \tag{3.48}$$

which yields:

$$\hat{\tau}_{i,j} = \sum_{c_m, c_n \in \mathcal{C}} y_{i,j,m,n} \tau_{m,n}. \tag{3.49}$$

Since $y_{i,j,m,n} = a_{i,m} \wedge a_{j,n}$, the logical 'and' can be expressed as:

$$y_{i,j,m,n} \geq a_{i,m} + a_{j,n} - 1 \tag{3.50a}$$

$$y_{i,j,m,n} \leq a_{i,m} \tag{3.50b}$$

$$y_{i,j,m,n} \leq a_{j,n}. \tag{3.50c}$$

Finally, when a chain is not delayed for one period, then it must verify $l_{i,j} + e_i - \hat{\tau}_{i,j} \geq 0$, so we introduce

$$l_{i,j} + e_i - \hat{\tau}_{i,j} + x_{i,j} Z \geq 0, \tag{3.51}$$

where again, the 'Big-M' constant Z is used to ignore this constraint when $E_{i,j}^{max}$ is respected in equation 3.46 with the chain being delayed one period.

The full model is:

$$\begin{aligned} \textit{max} \quad & \alpha \\ \text{s.t.} \quad & 0 \leq \alpha \leq \min_{p_i \in \mathcal{P}} \left\{ \frac{T_i}{e_i} \right\} \\ & \forall p_i \in \mathcal{P}: \quad \sum_{c_m \in \mathcal{C}} a_{i,m} = 1 \\ & \qquad\qquad\quad 0 \leq t_i \leq T_i - e_i \\ & \qquad\qquad\quad \forall c_m \in \{\mathcal{C} \setminus D_i\}: \; a_{i,m} = 0 \end{aligned}$$

$$\forall c_m \in \mathcal{C}: \quad \sum_{p_i \in \mathcal{P}} a_{i,m} s_i \leq S_m$$

$$\forall p_i, p_j \in \mathcal{P}^2: \quad f_i \neq f_j,\ \forall c_m \in \mathcal{C}:\ a_{i,m} \leq a_{j,m}$$

$$f_i = f_j,\ \forall c_m \in \mathcal{C}:\ a_{i,m} = a_{j,m}$$

$$\forall p_i, p_j \in \mathcal{P}, j > i: \quad \forall c_m \in \mathcal{C}:\ t_j - t_i - q_{i,j} g_{i,j} \geq \alpha e_i -$$

$$- Z\left(2 - a_{i,m} - a_{j,m}\right)$$

$$\forall c_m \in \mathcal{C}:\ t_j - t_i - q_{i,j} g_{i,j} \leq -g_{i,j} \alpha e_j -$$

$$- Z\left(2 - a_{i,m} - a_{j,m}\right)$$

$$\frac{e_i - T_i}{g_{i,j}} \leq q_{i,j} \leq \frac{T_j - e_i}{g_{i,j}}$$

$$0 \leq t_j - t_i - q_{i,j} g_{i,j} \leq g_{i,j}$$

$$\forall p_i, p_j \in \mathcal{P}, E_{i,j}^{max} < \infty: \quad t_j - t_i - q_{i,j} g_{i,j} + e_j + x_{i,j} T_j \leq E_{i,j}^{max}$$

$$t_j - t_i - q_{i,j} g_{i,j} + e_i - \sum_{c_m, c_n \in \mathcal{C}} \left(y_{i,j,m,n} \tau_{m,n}\right) +$$

$$+ x_{i,j} Z \geq 0$$

$$\forall c_m, c_n \in \mathcal{C}:\ y_{i,j,m,n} \geq a_{i,m} + a_{j,n} - 1$$

$$\forall c_m, c_n \in \mathcal{C}:\ y_{i,j,m,n} \leq a_{i,m}$$

$$\forall c_m, c_n \in \mathcal{C}:\ y_{i,j,m,n} \leq a_{j,n}$$

$$a_{i,m} \in \{0, 1\},\ t_i \in \mathbb{Z},\ q_{i,j} \in \mathbb{Z}$$

$$x_{i,j} \in \{0, 1\},\ y_{i,j,m,n} \in \{0, 1\}$$

Chapter 4

Methodology

This chapter describes the heuristic methods developed to solve the problem presented in chapter 3. As seen by the complexity of the MILP formulation in section 3.4, approaching the problem globally would be virtually infeasible, in the sense that solutions are not available in admissible time. Experimentation with solving this problem directly with generic optimization algorithms showed no promising results, therefore we opt to divide the problem into subproblems and provide specialized methods for solving these. The three subproblems are: assigning partitions to modules to verify the distribution constraints, performing local optimization on a single module, and performing global optimization, which are detailed in the following sections.

4.1 Partition assignment

The partition assignment problem aims to distribute the partitions among the available modules in a way that verifies the distribution constraints. For a problem with N_p partitions and N_c modules, there are ${N_c}^{N_p}$ possible configurations. Even if we assume that most of these are valid, it is infeasible to analyse each of these and obtain a schedule on top of that.

Following this, the first step is to find one viable assignment of partitions to modules, which essentially consists in assigning values to f_i, $\forall p_i \in \mathcal{P}$, such that an initial solution can be created. At the same time, tackling this reduced problem allows us to quickly prove infeasibility for certain problem instances without entering the additional complexity of considering the actual schedules.

Two different methods are implemented for this subproblem: a constraint programming approach, and a sequential assignment similar to those used for the bin-packing problem.

4.1.1 Constraint programming

Constraint programming is a generic framework to solve combinatorial problems like this one, modelling as a CSP.

The formulation as a CSP is straightforward from the MILP model, but here we rearrange it to use f nomenclature. The formulation is:

Variables : $f_i,\ \forall p_i \in \mathcal{P}$

Domains : $\{m\},\ \forall c_m \in D_i$

Constraints :

$$\text{memory:} \quad \forall m \sum_i \{s_i \cdot \delta\,(f_i - m)\} \leq S_m \tag{4.1}$$

$$\text{inclusion:} \quad f_i = f_j \tag{4.2}$$

$$\text{exclusion:} \quad f_i \neq f_j \tag{4.3}$$

This problem can be solved using a general purpose CSP search algorithm. Algorithm 1 is a recursive implementation of a backtracking search algorithm, where in each call a new variable is assigned a value in its domain. One advantage of CSPs is they are able to perform search using generalized heuristics not dependant on the problem structure.

Algorithm 1 Generic recursive backtracking search algorithm with forward checking, adapted from [43].

```
1:  procedure Backtracking-search(Problem)
2:      return Backtrack({}, Problem)
3:  end procedure
4:
5:  procedure Backtrack(Assignment, Problem)
6:      if assignment is complete then return assignment
7:      end if
8:      var ← unassigned-variable(Problem)
9:      for each value ∈ var.domain do
10:         if value is consistent with Assignment then
11:             add {var = value} to Assignment
12:             if Forward_check(Problem, Assignment, var) then
13:                 result ← Backtrack(Assignment, Problem)
14:                 if result ≠ failure then
15:                     return result
16:                 end if
17:             end if
18:             remove {var = value} from Assignment
19:         end if
20:     end for
21:     return failure
22: end procedure
```

Another important characteristic of using backtracking search to solve this CSP is completeness, as we are able to prove infeasibility of the whole partition scheduling problem if the partition assignment subproblem is infeasible. Therefore, the first step in the scheduling tool will be to find a valid solution to this subproblem.

4.1.2 Sequential assignment

On approaches to similar scheduling problems, the hardware available is not defined, and the goal is to distribute partitions in a way that minimizes the required number of modules. This can be considered an

instance of the bin-packing problem, common techniques for solving these include sequential assignment algorithms. In fact, Eisenbrand et al. [21] demonstrates that the simple `First-Fit` heuristic is a 2-approximation algorithm, i.e. it produces a solution with at most twice the cost of the optimal one, also showing that no polynomial-time algorithm can do better.

In this section we study a solution of the partition assignment subproblem using sequential algorithms. The differences in context are evident, notably that ours is a decision problem whereas these heuristics are aimed at an optimization problem, the existence of other kinds of constraints, and the fact that the modules (bins) are not identical. Additionally, bin-packing requires a measure of capacity. It would seem natural that memory be used as capacity, but for two reasons we choose to use the usage fraction instead, U from equation 3.23. The first is it makes every module similar in the sense of the bin-packing problem, as all have a limit usage fraction of 1; the second is it being a valid indicator for feasibility for the following scheduling problem.

Starting with `First-Fit`, the algorithm takes items in an arbitrary order, and assigns them sequentially to a bin, always choosing the first bin where it fits, and if the item does not fit in any, a new bin is opened. In our context, the items are partitions and the bins are modules, and since there is a known number of modules defined, the algorithm fails if the partition does not fit in any. Additionally, by *fitting*, it is meant that constraints 4.1 to 4.3 are verified, and the usage fraction is not exceeded.

Another heuristic algorithm with similar behaviour is `Best-Fit`. Here, for each item, all bins where it fits are determined, and the item is assigned to the bin that is left with lowest capacity after this assignment.

Other variations of these algorithms exist, however, since they are designed to minimize the number of bins, the solutions will naturally tend to pack more partitions in a few modules. For our optimization criterion, it is beneficial that the partitions are more evenly distributed among the hardware, which motivates a heuristic sequential assignment algorithm that takes this into account. We name it `Best-Fit-Inverse`, and similarly to `Best-Fit`, it determines all bins where it fits, but then chooses the one which is left with highest capacity after the assignment. This achieves our goal of favouring solutions with U_m being balanced among the modules. See algorithm 2 for an implementation for this specific problem.

Finally, the ordering which the partitions are assigned is significant in any of these algorithms. With experimentation, it is determined that ordering by number of constraints involved produces more consistent results.

The preferred choice for solving the partition assignment subproblem is via a CSP solver, due to it being more robust for highly constricted cases, and able to prove infeasibility. The performance advantage of sequential assignment with `Best-Fit-Inverse` is negated by the fact that this step only needs to be carried out once. Also, we empirically verify that the CSP approach is also able to produce a solution with balanced usage fractions if the variable domains are randomized, since algorithm 1 selects values for assignment in the order they are provided.

A note should be made that the MILP model presented in section 3.4 can be simplified to solve this subproblem, and in fact it would become an Integer Program. However, since this is a discrete combinatorial problem and optimization is not required, the CSP approach with its associated heuristics is superior.

Algorithm 2 `Best-Fit-Inverse` sequential assignment algorithm, applied to the partition distribution problem.

procedure best-fit-inverse($\mathcal{P}$, $\mathcal{C}$, $constraints$)
  $items \leftarrow$order_by_#_constraints($\mathcal{P}$, $constraints$)
  $bins \leftarrow \mathcal{C}$
  $assignment \leftarrow \{\}$
  for each $item \in items$ **do**
    for each $bin \in bins$ **do**
      $bin.fit \leftarrow$ fits?($item$, bin, $assignment$, $constraints$)
      $bin.capacity \leftarrow \sum_{i \in assignment} \{e_i / T_i \times \delta(bin.index - assignment[i])\}$
    end for
    $selected \leftarrow$ min_by_capacity(filter($bins$, $fit = true$))
    if $selected$ **then**
      add $\{item = selected\}$ to $assignment$
    else
      return $failure$
    end if
  end for
  return $assignment$
end procedure

4.2 Local optimization

Another subdivision of the overall problem is optimization of the schedule for one module. This procedure is done for a certain assignment of partitions to modules, and aims to find the optimal schedule for that configuration, which is why it is named local optimization. Until section 4.2.4, we focus the description on instances without communication between modules or multiple windows.

4.2.1 CSP formulation

For the sake of completeness, we list the constraint satisfaction variant of the problem:

Variables : $t_i, \ \forall p_i \in \mathcal{P}_m$

Domains : $\{0, 1, \dots, T_i - e_i\}$

Constraints :

$$\text{No-overlap :} \quad e_i \leq l_{i,j} \leq g_{i,j} - e_j, \ \forall p_i, p_j \in \mathcal{P}_m^2 \tag{4.4}$$

$$\text{Chains :} \quad E_{i,j} \leq E_{i,j}^{max}, \ p_i, p_j \in \mathcal{P}_m \tag{4.5}$$

This has the advantage of having only binary constraints between variables, but the fact that the offsets take integer values upwards to the partition period makes the search space very large. As such, this formulation serves no purpose.

4.2.2 Best response algorithm

Optimizing the offsets for all partitions such that α is maximized is a complex problem. However, optimizing the offset of one partition in the schedule while taking the other partition offsets as fixed is feasible. The strategy is to iteratively update the offset of each partition to a better value, and due to the similarities with game theory, the procedure is called the best response algorithm. This solution was studied separately in [28, 41].

Consider the partitions players in a game. The game is played in turns, each player updates its strategy knowing the current strategy for the other players, and the game is played until the strategies converge. In particular, each player chooses the offset that maximizes its utility (defined in equation 3.33), which is the factor by which all executions can be multiplied without overlapping with its own execution window. Since partitions choose their offset independently, this game is categorically non-cooperative, and the optimal solution lies on an equilibrium point, which in game theory is known as a Nash Equilibrium Point, from Nash [1]. However, a partition's utility maximizes not only the partition's window of execution, but also other partitions' interactions with its own, therefore, it has a cooperative trait. This is an important aspect that guarantees that, with a few nuances, this procedure converges to one of these equilibrium points, and additionally, this point will be at a local optimum with respect to the α-parameter. The converse is also true, any local optimum solution will be an equilibrium point.

Convergence is proved in the before-mentioned references for the simplest problem definition, without chains (equation 4.5), and with extended domains (t_i can take values up to T_i). This also requires that when choosing the best strategy, and if the player's previous strategy is equally good to the best one, then the player does not change its strategy. This prevents the game from entering loops, and speeds up convergence.

In general, problem instances have many equilibrium points, which are all locally optimal solutions to the scheduling problem. Finding the optimal solution consists in finding the best of these equilibrium points, and is achieved by providing different starting points to the best response algorithm.

The introduction of chains (with equation 4.5) will simply restrict which offsets are valid. This has the effect of speeding up convergence since it restricts the problem further, however, it also increases the number of equilibrium points, making the procedure more dependant on the initial state.

Algorithm 3 lists the best response algorithm. A counter for the number of *stable* partitions is kept, and is reset every time a partition changes its offset. Equilibrium is reached when no partition changes its offset, and this counter equals the number of partitions, terminating the algorithm. On line 6, the domain is restricted in order to verify communications constraints, and on line 7, the partition determines the best offset in this domain. Section 4.2.3 is entirely dedicated to procedures that determine this value. Partitions are cyclically visited, the `next()` operator on line 5 can be thought as getting the next element on a circular queue of partitions.

The ordering by which partitions are cycled would seem to affect the solution, by favouring the first partition. However, we corroborate the findings in [41] that no significant improvements in the solution process are found by introducing randomness in this ordering. In contrast, the randomness included in the initial state is the main factor that allows different solutions to be found.

Algorithm 3 Best response algorithm.

```
procedure Best_response_optimization(partitions, chains, offsets)
    N ←length(partitions)
    N_stable ← 1
    while N_stable < N do
        p ← next(partitions)
        domain ← valid_offsets(p, chains, offsets)
        t_new ← best_value(p, domain, offsets)
        if t_new = offsets[p] then
            N_stable ← N_stable + 1
        else
            offsets[p] ← t_new
            N_stable ← 1
        end if
    end while
    return offsets
end procedure
```

Figure 4.1 shows the progress of this algorithm for a particular case with no chains. In 4.1a, an initial state is given, which does not need to be a valid solution, as seen by the overlap between partitions 1 and 3. The iteration order is $p_2 \to p_3 \to p_1 \to p_2$ ..., and for this starting state, any other ordering leads to a solution with the same α-value. The first iteration (in 4.1b) maximizes the utility of p_2 by moving to $t_2 = 7$, and this is dominated entirely by the relation with p_1. Notice that $t_2 = 19$ would be an equally good move here, the selection between these is arbitrary. In 4.1c, the utility of p_3 is maximized, and again, the dominant relation is with p_1. Finally, p_1 does not change its offset in 4.1d, as its current position is already maximizing its utility. Convergence is reached in this step, and in fact p_2 and p_3 are checked again before the procedure terminates, these steps were omitted from the figure. The final solution has $\alpha = 7/3$, and is optimal.

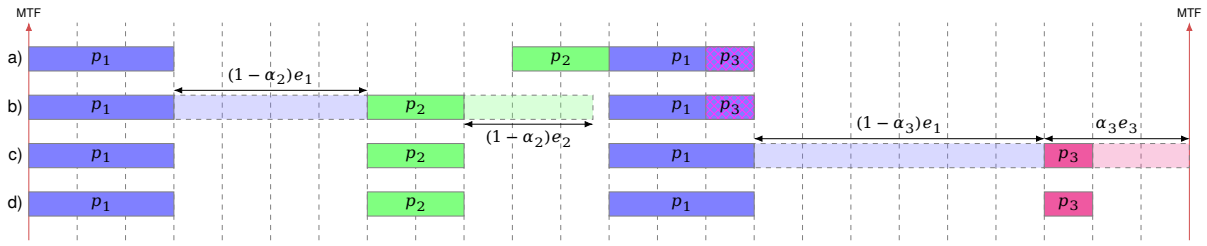


Figure 4.1: Progress of the best response algorithm.

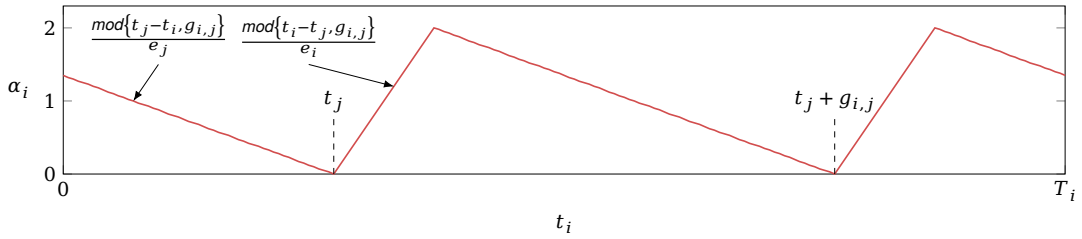
4.2.3 Linear search

The `best_value` procedure consists in finding t_i which maximize utility, by solving the following program:

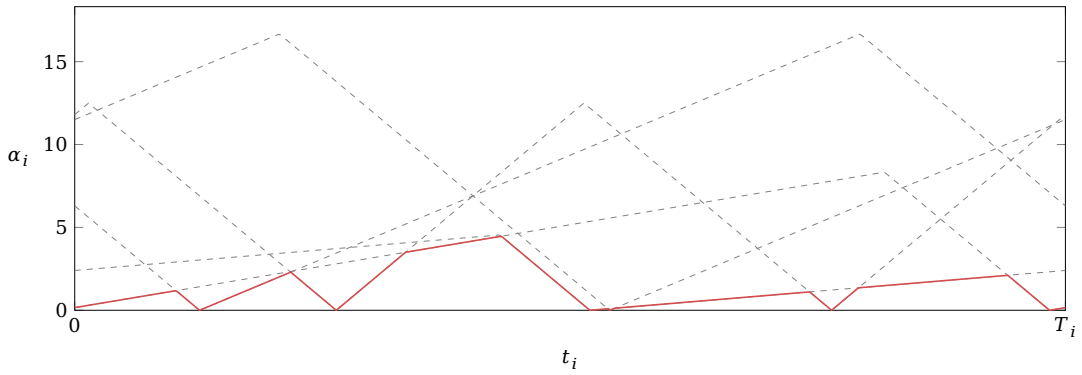
$$\begin{aligned} \max \quad & \alpha_i \\ \text{s.t.} \quad & \alpha_i \geq 0 \\ & \alpha_i e_j \leq mod\{t_i - t_j,\ g_{i,j}\} \leq g_{i,j} - \alpha_i e_i,\ \forall p_j \in \mathcal{P}_m \setminus \{p_i\} \\ & t_i \in \mathcal{T}_i, \end{aligned} \tag{4.6}$$

with $\mathcal{T}_i \equiv [0, 1, \ldots, T_i - e_i]$ being the valid offsets to consider. This is trivially solved by computing α_i for all possible values of t_i and choosing the best one. However, this procedure will be repeated many times inside the local and global search procedures, therefore a faster method is paramount. An efficient method is based on linear programming, and the previous program is in fact a linear program if we relax the offset to be a real number: $0 \leq t_i \leq T_i - e_i$.

First, we look into the structure of the solution. Considering only a pair of constraints, we have the line $\alpha_i = mod\{t_j - t_i,\ g_{i,j}\}/e_i$, which is discontinuous at its zeros, $t_i = t_j + g_{i,j}n,\ n \in \mathbb{Z}$, but strictly decreasing in other regions. Likewise for the other part of the same constraint, we have the line $\alpha_i = mod\{t_i - t_j,\ g_{i,j}\}/e_j$ that is strictly increasing but discontinuous at the same zeros (from the property $mod\{-a, b\} = b - mod\{a, b\}$). The overall value of α_i will then lie in the minimum of these two lines, as seen in figure 4.2a, and with multiple partitions, the solution will be the minimum of all these lines, as seen in figure 4.2b.



(a) Solution set with one other partition



(b) Solution set with five other partitions.

Figure 4.2: Structure of the `best_value` solution set, adapted from [41].

The solution set is composed of adjacent polyhedra, separated by zeros of the utility. Moreover, an important property in the solution set is that inside each polyhedron, the utility is strictly increasing until the local maximum, then strictly decreasing until the next zero. This in turn guarantees that every local maxima, including the global maximum, lie at the intersection of an ascending line and a descenting line. The solution to this problem, as proposed in Al Sheikh et al. [28], is to determine the subset of intersection points $\mathcal{I}_i \subset \mathcal{T}_i$, and compute the utility only for these points.

The intersection points can be determined for each polyhedron. Locally, the constraint $mod\{t_i - t_j, g_{i,j}\} \geq \alpha_i e_j$ is bounded by an increasing line, of the form $t_i - o_j^{inc} = e_j \alpha_i$. Given a reference offset

t_i^{ref} in the polyhedron, the value of o_j^{inc} is given as:

$$o_j^{inc} = t_i^{ref} - mod\{t_i^{ref} - t_j, g_{i,j}\}. \tag{4.7}$$

Identically, the decreasing line is $t_k - o_k^{dec} \geq \alpha_i e_i$, with $o_k^{dec} = o_k^{inc} + g_{i,k}$. Proof of 4.7 is derived from modulo properties, see Pira and Artigues [41] for details. With this, the intersection point between these two lines is:

$$\hat{t} = \frac{e_j o_k^{dec} + e_i o_j^{inc}}{e_j + e_i}, \tag{4.8}$$

with $p_j, p_k \in \mathcal{P}_m \setminus \{p_i\}$, possibly having $j = k$. The intersection point is in general a rational number, therefore for each $\hat{t}$, the points $\lceil \hat{t} \rceil$ and $\lfloor \hat{t} \rfloor$ must be added to $\mathcal{J}_i$. In order not to check repeated polyhedra, the reference offset t_i^{ref} can be taken at the zeros of the utility, which are located at $t_j + n g_{i,j}, \forall n \in \mathbb{Z}, \forall j \neq i$.

Other methods exist that allow to skip some polyhedra after a lower bound for the utility is found, using line search [41], and performance is superior to the method described above. To the best of my knowledge, these methods cannot be applied with communication constraints so they were not investigated. Another simple method that improves the simple solution of checking every possible offset consists in checking offsets only while the utility increases, and jump to the next zero when it starts to decrease, taking advantage of the structure of the problem.

The actual calculation of the utility is done in $\mathcal{O}(N_p)$ time (see equation 3.33), therefore both exhaustive search and its improved version run in $\mathcal{O}(N_p T_i)$. For the method based on intersection points, we first note that for N constraining partitions, there will be N^2 intersection points in each polyhedron, with the number of polyhedra being trivially bounded by T_i, so actually this algorithm runs in $\mathcal{O}(T_i {N_p}^3)$. Despite this asymptotic time complexity being clearly worse, for usual problem instances with limited N_p, we verify that this version is superior to exhaustive search, as will be seen in chapter 5.

On the implementation level, the utility at each intersection point can be computed directly with the intersection point, or the set of intersection points can be stored in memory and the utility calculated afterwards. The benefit of the latter is that we can avoid inspecting the utility at repeated intersection points, at the cost of using more memory. Experimentation showed that the former approach is superior.

4.2.4 Extending the search with chains

For chains, different approaches are possible, the most general is, for each partition, to consider all chains that affect it, which will possibly depend on partitions in modules other than the one we try to optimize. This is the approach followed here, and we maintain the same goal that is computing the offset that yields maximum utility, considering all other variables fixed.

As with the CSP approach, communication constraints do not affect the utility of a partition or the α-parameter of the whole schedule, their effect simply restricts the problem by forbidding certain invalid offsets. Equation 3.25 is the important one here, restricting $\mathcal{T}_i$ to a series of feasible intervals, as seen in

figure 4.3. The number of these intervals depends on the periods of the partitions involved in the chain. In the figure, two intervals are seen because in this example the chain is done with a partition with period $T_j = T_i/2$.

Clearly, only the intersection points that belong to the feasible region need to be considered, but in addition to these, the boundary points in each segment become interest points. This includes the boundary points of $\mathcal{T}_i$, $\{0, T_i - e_i\}$.

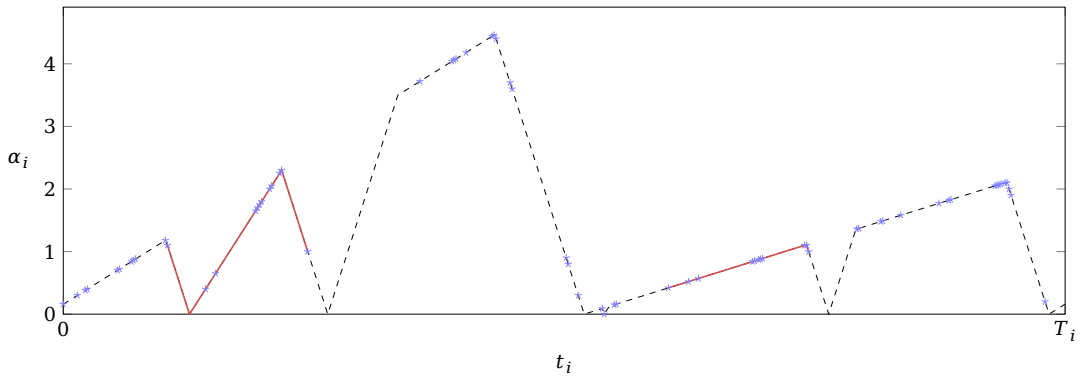


Figure 4.3: Feasible region (red) constrained by one chain, with all interest points (blue).

Computing these feasible regions is not straightforward because different cases apply when the delay allows for delaying one period. The easiest case is with harmonic partition periods and when delaying a period never happens, for which the feasible region consists in all intervals:

$$\left[a + k g_{i,j},\ b + k g_{i,j}\right],\ k \in \mathbb{Z}. \tag{4.9}$$

The values a, b depend on whether p_i is the sender or the receiver in the chain:

$$p_i \to p_j: \quad \begin{cases} a = t_j + e_j - E^{max}_{i,j} \\ b = t_j - \hat{\tau}_{i,j} - e_i \end{cases} \tag{4.10}$$

$$p_j \to p_i: \quad \begin{cases} a = t_j + \hat{\tau}_{j,i} + e_i \\ b = t_j - e_i + E^{max}_{j,i} \end{cases}. \tag{4.11}$$

Proof. We consider the case where the chain is processed as $p_i \to p_j$ and t_j being fixed, with the other case having an identical derivation. The first branch in equation 3.25 is linearised as:

$$t_j - t_i - q g_{i,j} + e_j \le E^{max}_{i,j}, \tag{4.12}$$

which yields an upper bound on t_i:

$$t_i \ge t_j - q g_{i,j} - E^{max}_{i,j} + e_j. \tag{4.13}$$

Identically, the constraint that prevents a chain from being delayed one period is:

$$l_{i,j} - e_i \geq \hat{\tau}_{i,j}, \tag{4.14}$$

is linearised as:

$$t_i \leq t_j - qg_{i,j} - \hat{\tau}_{i,j} - e_i. \tag{4.15}$$

The interval becomes:

$$\left[t_j + e_j - E^{max}_{i,j} - qg_{i,j},\ t_j - \hat{\tau}_{i,} - e_i - qg_{i,j}\right]. \tag{4.16}$$

It was seen in section 3.1.2 for the specific case of harmonic periods that the chain is repeated with a period equal to the greatest period among the two partitions, or equivalently, $g_{i,j}$. Therefore, the previous interval can be shifted by $g_{i,j}$ units as necessary, yielding:

$$\left[t_j + e_j - E^{max}_{i,j} + (k-q)g_{i,j},\ t_j - \hat{\tau}_{i,} - e_i + (k-q)g_{i,j}\right],\ \forall k \in \mathbb{Z}. \tag{4.17}$$

Finally, q can be removed since it is an integer value by definition, leaving an equivalent to expression 4.9. □

The limits of these intervals in expression 4.9 become interest points that possibly result in a local maximum of the utility. Evidently, each partition is not restricted to be involved in only one chain, and in fact all of them can be considered at this step. If this is the case, all regions from expression 4.9 should be intersected, and the interest points are the boundary points of the resulting region, plus the intersection points, $\mathcal{J}_i$, determined in section 4.2.3 that are contained in this region.

To reiterate, chains do not need to be considered for local optimization, although this is favourable in order to find a solution faster. Introducing chains restricts the search space and thus speeds the computation of the best offset, and it also promotes partitions to stick together faster, which speeds the convergence of the best response algorithm. However, this also leads to more equilibrium points, hence more restarts are required to find the optimal solution with this procedure when compared to not including chains. Also, issues arise in tightly constricted problems when there are no viable offsets for a partition, as will be discussed in section 4.2.7. Here it would make sense to either keep the current offset in order to promote convergence, or alternatively, select the best offset with respect to the α-parameter regardless of the constraining chains. Empirically, we found better results for the latter option, which can be explained by it allowing the solution to escape the current bad region, and also improving the utility for other partitions in the module.

4.2.5 Extending the search with multiple windows

The best response procedure is also able to optimize a module's schedule with multiple partition windows per job when using the more complex form of utility, from equation 3.36. However, this function does not share the same properties as the simpler version which allowed for efficient computation of the best offset. In particular, being defined by branches, the function is not continuous, and loses the interesting polyhedron-shape of the curves shown in figures 4.2 and 4.3, thus the linear programming approach discussed in section 4.2.3 is no longer applicable. For this reason, the best value calculation is done exhaustively for every offset when any of the partitions in the module has multiple windows of execution per job.

At the level of the best response procedure, the approach is to consider all windows independent players, but similarly to chains, some strategies will be tightly constrained by strategies of other players because we want to maintain the ordering of windows and respect the maximum response time. The exception is the first window at each job; seeing that the offsets for these are all strictly related (by the partition period), it makes sense to consider them a single player. For the purpose of computing the utility, the worst among the K_i jobs is counted. As for the ordering, there is no theoretical or empirical reason to enforce a particular playing order. For simplicity of implementation, we define that all windows for the same partition play in chronological order.

The more inefficient calculation of the best offset compared to the case with simple partitions is combatted by the restricted domains due to window ordering and response times. Window ordering is represented by having:

$$t_{i,k,u} < t_{i,k,u+1} \tag{4.18a}$$

$$t_{i,k,M_{i,k}} < t_{i,k+1,1}, \tag{4.18b}$$

so when determining the best offset for $\lambda_{i,k,u}$, $u > 1$, it is possible to restrict the calculation to offsets:

$$\left[t_{i,k,u-1},\, t_{i,k,u+1}\right]. \tag{4.19}$$

Additionally, the maximum response time is limited to d_i, so we can further restrict the offsets for all windows $u > 1$ to guarantee that the following ones can all be placed in the remaining time:

$$t_{i,k,u} \leq t_{i,k,1} + d_i - \sum_{v=u}^{M_{i,k}} \left\{e_{i,k,v} + \varepsilon_m\right\}. \tag{4.20}$$

In fact, since $d_i \leq T_i$ and $t_{i,k+1,1} = t_{i,k,1} + T_i$, inequality 4.18b is redundant.

Convergence of the best response algorithm for the restricted problem has been formally proven [28]. For the extended model presented here, it was experimentally verified that about 6 % of randomly generated cases do not converge. We cope with this by discouraging changing the offsets as the best response algorithm progresses. A threshold is introduced after a few iterations, and players only change their strategy if the gain in utility is greater than this threshold. However, the equilibrium solution might not

be at a local optimum anymore, although the penalty in terms of the α-parameter should be minimal.

Algorithm 4 lists these modifications to the best response algorithm (algorithm 3), but it should be viewed as an alternative to be used when needed rather than a generalization of 3, as it is clearly less efficient when the simpler one applies.

Algorithm 4 Adapted best response algorithm.

```
1:  procedure Best_response_multiple_windows(partitions, chains, offsets)
2:      players ← empty list
3:      for each p_i ∈ partitions do
4:          Add Λ_i to players
5:      end for
6:      N ←length(players)
7:      N_stable ← 1
8:      i ← 0
9:      threshold ← 0
10:     while N_stable < N do
11:         p ← next(players)
12:         domain ← valid_offsets(p, chains, offsets)
13:         t_new ← best_value_b(p, domain, offsets, threshold)
14:         if t_new = offsets[p] then
15:             N_stable ← N_stable + 1
16:         else
17:             offsets[p] ← t_new
18:             N_stable ← 1
19:         end if
20:         // Once every full cycle around the players, after the 15th
21:         if mod{i, N} = 0 ∧ i/N > 15 then
22:             threshold ← threshold + 0.005
23:         end if
24:         i ← i + 1
25:     end while
26:     return offsets
27: end procedure
28:
29: function best_value_b(p, domain, offsets, threshold)
30:     t_best ← offsets[p]
31:     α_now ← compute_utility(t_best, offsets)
32:     α_best ← α_now
33:     for each t ∈ domain do
34:         α_new ← compute_utility(t, offsets)
35:         if α_new − α_now > threshold and α_new > α_best then
36:             t_best ← t
37:             α_best ← alpha_new
38:         end if
39:     end for
40:     return t_best
41: end function
```

4.2.6 Parallel best response

Parallel best response is the procedure of applying local optimization to several modules simultaneously, an important component in our methodology for global optimization. Since it is based on the best response algorithm, we describe it now.

The previous description of the effects of chains in the best response algorithm focused on restricting the offsets a partition could select at each iteration. However, it is fair to say that when a chain is performed across two modules, then their respective schedules are not independent, hence performing local optimization independently for these does not, in general, result in an overall local optimum. The alternative approach which we explore here is to optimize the modules connected by chains in parallel, which means we take all partitions scheduled in those modules as our set of players, and apply the best response algorithm to this set.

Fundamentally, the best response algorithm is unchanged, the only nuance here is for computing the best offset (and therefore the utilities), only the partitions scheduled in the same module are considered, but the viable offsets are restricted by all partitions in the set. It was noted before that the best response algorithm has worst-case complexity that is exponential with the number of players, which seems a deterrent towards this strategy, however, this complexity arrives from having to verify the strategy against every other player, which is not the case here. In fact, players on separate modules do not influence each other's strategies, they just restrict which strategies are even possible, which is no different from solving the modules independently.

That is not to say the performance is unaffected, just not as critically as it may be apparent. When some modules schedules converge faster than others, we are still checking every best strategy on the converged players at every cycle around the players, which represents wasted computations. Additionally, there is an extra step needed, determining which modules need to be optimized in parallel and which can be optimized independently. If we form an undirected graph where the modules are vertices and any chain creates an edge between the modules where the partitions are assigned to, then this problem consists in enumerating all unconnected subgraphs, easily solved by either breadth-first or depth-first search. This is solvable in linear time since each node only needs to be visited once. In figure 4.4 we see an example of this, where modules c_2, c_3, c_4 form an unconnected subgraph and thus should be scheduled in parallel, while module c_1 can be scheduled independently.

Of course, in general, we can just optimize all modules in the system in parallel and skip the unconnected subgraph problem, especially considering that often in problems with many chains there are no unconnected subgraphs. However, the gains in performance by decreasing the number of players in the best response algorithm are appealing, so the previous solution was implemented.

4.2.7 Limitations

Performance of the best response algorithm is evaluated in chapter 5. Here we identify some conceptual limitations of this heuristic. Firstly, the non-cooperative nature leads to deadlocks, cases where partitions do not have a good option to move when considered individually, but one with an easy solution if the problem were considered globally. A simple example of this idea is shown in figure 4.5. If using the utility given by equation 3.33, one can see that none of these partitions have an option to improve it, so the best response procedure *converges* immediately for the current non-feasible solution, even though a valid solution is trivial. This issue escalates for similarly tightly packed cases ($U \approx 1$), and this algorithm can

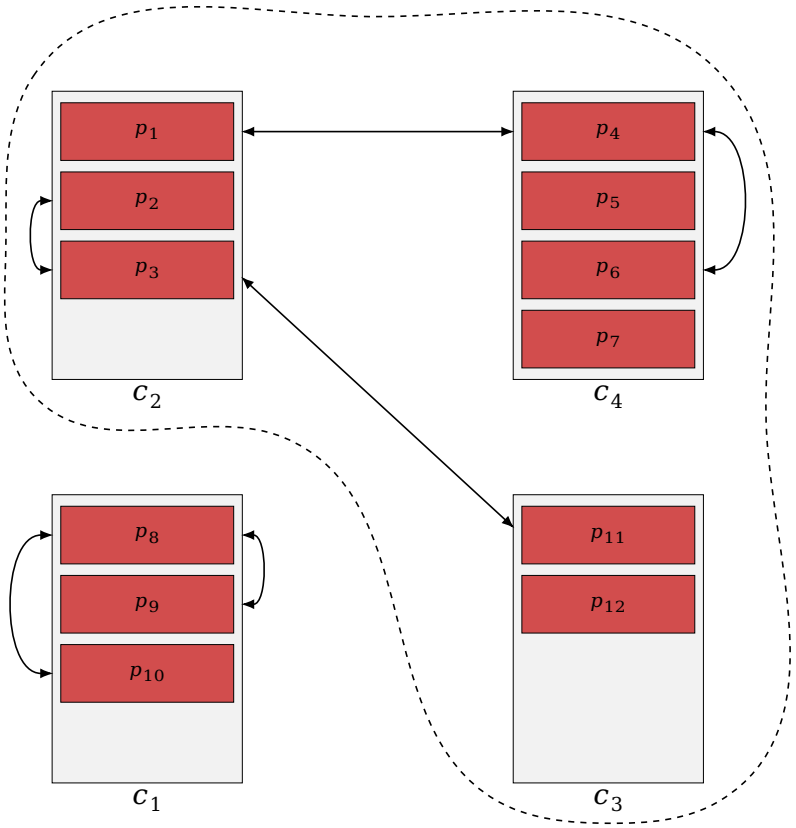


Figure 4.4: Modules scheduled in parallel due to chains, represented as arrows.

only solve these cases starting from already good starting points.

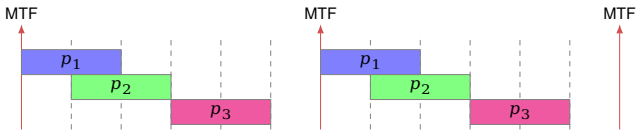


Figure 4.5: Special case where the best response algorithm fails.

Other deadlocks happen due to chains with tight delays. When partitions are involved in tight chain delays, it can happen than only a few offsets are feasible, and the partitions will not be able to leave this region. For instance, consider the extreme case where only one offset satisfies a certain chain, then, the partition is assigned this offset and will be *locked* there. We can move the two partitions together, but not one individually as required by the best response procedure.

Another limitation comes from the complexity of the problem when there are multiple windows of execution, since for this heuristic, each window is considered a different player altogether. The worst-case complexity of the standard best response algorithm is known to be $\mathcal{O}(NA^{N-1})$, with N being the number of players and A the number of strategies per player [40], evidencing that the performance is especially sensitive to the number of players.

4.3 Global optimization

Local optimization allows us to efficiently generate schedules for a single module after having defined the partitions that are assigned to this module as well as their configuration with respect to windows. In fact, the optimal schedule for each module is within reach of the local optimization procedure, given sufficient restarts from different starting configurations. Therefore, the remaining task is finding the partition assignment and window configuration that enables us to find an optimized solution to the overall problem. A viable distribution of partitions among modules can be found via methods described in section 4.1; furthermore the number of viable configurations can be as high as ${N_c}^{N_p}$ with no distribution constraints, so the goal in this section is to develop a strategy to explore a relevant portion of the search space.

The best response algorithm has been used to not only select the best offset, but also the best module for each partition [28, 41]. Generating initial states becomes more troublesome as a random assignment of partitions to modules in general will not produce a feasible assignment, and the CSP procedure from section 4.1 would need to applied at each restart. Also, the algorithm would not be applied to a single module but to the whole system, and as seen before, the complexity is exponential with the number of partitions. The present work decides to investigate a different solution, the main reason being that we consider more comprehensive distribution constraints, which would lead to different kinds of deadlocks in the best response algorithm.

The followed strategy is based on stochastic optimization algorithms, in particular SA, Tabu-search and a genetic (or evolutionary) algorithm. These are of course classified as local search algorithms, but the fact that we are applying them only to a subtask in our problem, namely exploring starting points for a dedicated local optimization procedure, makes it adequate to use them for global search. These algorithms operate on a complete solution and gradually improve it, and this allows for the usage of the modularity of the local optimization procedure to improve only the needed modules. Another reason for this choice is we lack a proper way to evaluate partial solutions, that is, not all problem variables being assigned a value, which means a constructive algorithm is not appropriate.

A set of operators are defined which navigate the search space, and a set of rules (meta-heuristics) for applying them are described in this section. The algorithms themselves are very similar, and have been implemented on a wide range of well known problems, so only the key differences and implementation details are described here; the full pseudo-code is included in appendix A.

4.3.1 Evaluation function

The evaluation function of a solution is the metric we try to optimize, which naturally corresponds to the α-parameter. However, the problem representation allows for invalid states, and the α-parameter does not account for unmet distribution constraints in its definition. For usage with the global optimization algorithms which can spontaneously generate infeasible solutions from a feasible one, it is necessary to penalize the infeasible solutions with a more generic evaluation function. We represent this by $ev(S)$,

where S is the state (or solution), and define it here as follows:

$$ev(S) = \begin{cases} \alpha, & n_c = 0 \\ \frac{\alpha}{n_c+1}, & \alpha < 1 \wedge n_c > 0 \\ \frac{1}{n_c+1}, & \text{otherwise,} \end{cases} \tag{4.21}$$

where n_c is the total number of unmet constraints.

This definition allows us to maintain the property of having a valid solution when $\alpha \geq 1$, which is now $ev(S) \geq 1$. Additionally, we also know that for $0.5 \leq ev(S) < 1$, we have all constraints (except for no execution overlap) satisfied.

4.3.2 Operators

A state is a possible solution to the problem, and consists of a complete assignment of the problem variables: offsets, partition assignment to modules, and partition windows. Note that on other problems or fields a state does not need to be complete, i.e., not all variables need to be assigned, but throughout our implementation we only deal with complete assignments thus we keep this definition of state. States have neighbourhoods, a number of other states that differ to the current one by only a few variables, and the process of moving to another state in the neighbourhood is called an operation.

The operators apply to states and produce a new state, as $op(S_1) \rightarrow S_2$. Six operators are defined and detailed below: the move operator, $Mov(S, \chi, m, n)$, the swap operator, $Sw(S, \chi_1, \chi_2, m, n)$, the shuffle operator, $Sh(S, m)$, the local optimization operator, $Lop(S, \mathcal{M})$, the slice operator, $Sl(S, i)$, and the crossover operator, $Cr(S_1, S_2)$, which is the only binary operator and combines two states to produce a new one.

These operators may change both the offsets and the assignments to modules, and without special consideration, it is possible to obtain a state which does not respect the distribution constraints from a state that does. A reduction in the dimension of the search space is thus achievable by restricting that the partition distribution always remains valid with respect to the distribution constraints when applying an operator, which essentially means we are only navigating valid solutions. In practice, this will apply to Mov and Sw, which change the partition assignments, while for Cr this is in general unavoidable. The disadvantage is this can lead to deadlocks if the distribution constraints are tight, where another valid solution cannot be reached from a given state by applying only one of these operators. Henceforth we refer to the operators as *coherent* if they require the distribution constraints to remain valid.

Operator Mov

The move operator changes the assignment of a group of partitions, $\mathcal{X} \subseteq \mathcal{P}$, essentially moving from one module to the other. The definition below uses the assignment notation from section 4.1:

$$
\begin{aligned}
& Mov(S_1, \mathcal{X}, m, n) \rightarrow S_2 \\
& \text{Precondition:} \quad \forall p_i \in \mathcal{X},\ f_i = m, \quad \text{in } S_1 \\
& \text{Effects:} \quad\quad\quad\quad\ f_i = n, \quad \text{in } S_2.
\end{aligned}
$$

A few heuristics are possible. Firstly, the α optimization goal is the minimum among the available modules, thus a move is more likely to improve a solution if it relieves this module, i.e. selecting m as this module. Also, if the operation must be coherent, all partitions that share an include constraint should be moved in the same operation. Other than that, the arguments can be chosen at random, but there is a nuance depending on whether we select a module from $\mathcal{C}$ at random and then a partition assigned to it, or if we just select a partition at random from $\mathcal{P}$ and take its assignee module as well. The advantage of the latter approach is the assignee modules are more likely to be the modules hosting more partitions, hence these moves will normally lead to more uniform distributions of partitions among modules, which is generally desirable.

Operator Sw

The swap operator swaps the assignment of two partitions, or two groups of partitions, virtually chaining two move operators:

$$
\begin{aligned}
& Sw(S_1, \mathcal{X}_1, \mathcal{X}_2, m, n) \rightarrow S_2 \\
& \text{Preconditions:} \quad \forall p_i \in \mathcal{X}_1,\ f_i = m, \quad \text{in } S_1 \\
& \qquad\qquad\qquad\quad \forall p_j \in \mathcal{X}_2,\ f_j = n, \quad \text{in } S_1 \\
& \text{Effects:} \quad\quad\quad\quad f_i = n,\ f_j = m, \quad \text{in } S_2
\end{aligned}
$$

This operator is useful for reaching certain states without passing through worse intermediary states, having either low α or just invalid distribution constraints, and also when we mean to keep all operations coherent, since it is applicable even in situations with tight distribution constraints. For instance, it can be directly applied to partitions that are in exclusion, or between modules that are near their memory limits, both cases where a move operator would not be coherent. Just as with the move operator, partitions that share an includes constraint must be moved as a group if the operator must be coherent.

Operator Sh

The shuffle operator provides a new start point for the local optimization procedure, by assigning random offsets to all partitions and all partition windows assigned to a certain module:

$Sh(S_1, m) \rightarrow S_2$

Preconditions:

Effects: $\forall p_i \in \mathcal{P}_m, \forall \lambda_a \in \Lambda_i : t_a = \textit{random}$ in S_2.

This random selection is wrapped to the possible values following equation 3.4, which in general do not contemplate communication constraints.

Operator Lop

The local optimization operator is applicable to a group of modules, $\mathcal{M} \subseteq \mathcal{C}$, as described in section 4.2.6, or a single module as described in section 4.2.

$Lop(S_1, \mathcal{M}) \rightarrow S_2$

Preconditions:

Effects: Modules $c_m \in \mathcal{M}$ in a local optimum in S_2.

Operator Sl

The slice operator changes the window configuration of a partition.

$Sh(S_1, i) \rightarrow S_2$

Preconditions: $\mathcal{B}_i \neq \emptyset$

Effects: $\Lambda_i = \textit{select},$ in S_2.

Here, *select* is the procedure of creating windows from a set of available preemption points, $\mathcal{B}_i$, as is exemplified in section 3.1.3. This is done randomly, but heuristically we can prefer the single window configuration more often when the execution is already sliced. Even though this configuration does not necessarily lead to an optimal result, it is important to decrease complexity.

Applying this operator can also follow heuristic rules, favouring partitions with larger executions, or partitions assigned to the module with smallest α.

Operator Cr

The crossover operator is a binary operator specific to the genetic algorithm that combines two states into one. It relies on a gene-based representation of the problem variables, and generally a new solution is made by combining at random some genes from one parent and some from the other. Alternatively, there

are ‘blend crossover’ operators that assign new values to genes, in between the values of those genes from the originating states, especially used for real-valued variables [5].

For this specific problem, the only viable gene representation considers tuples (f_i, t_i) for each partition, representing the module and the offset, respectively. Essentially, we combine a subset of partitions from one state with the remaining from the other state:

$$Cr(S_1, S_2) \rightarrow S_3$$

Preconditions: $\mathcal{X}_1 \cup \mathcal{X}_2 = \mathcal{P},\ \mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$

Effects: $\forall p_i \in \mathcal{X}_j :\ (f_i, t_i)$ *in* $S_3 = (f_i, t_i)$ *from* $S_j,\ j = 1, 2.$

In general, this operator is *incoherent*, and we can expect all modules to be affected. In order to use the strengths of genetic algorithms, the genes should represent good solution characteristics with some modularity, such that they can be transmitted to new solutions, and be gradually improved. In this regard, the chosen representation is flawed, as the optimization criteria fundamentally evaluates groups of partitions. To gain some value from the crossover operator, we can favour in the selection of $\mathcal{X}_1, \mathcal{X}_2$ that partition pairs constrained by chains are present in the same group. Another option which would benefit our optimization criterion would be to define genes from modules and the partitions assigned to them. However, this is not feasible since in the created solution some partitions can be missing or appear duplicated.

Also note that this operator has no effect when applied to two identical solutions.

Operator selection

The main idea is to apply one of the operators Mov, Sw, Sh, Sl, Cr at each iteration, followed by Lop only on the modules which were affected. The overlying meta-heuristic algorithm is responsible for choosing which operators are used, to which variables, and also whether or not to accept the resulting solution. The operator selection is done mostly randomly as is characteristic of the class of algorithms used, but the heuristics described for each operator affect the selection. Essentially, the operator is chosen according to a fixed probability vector:

- Mov a random partition: 20 %
- Mov a partition scheduled in the module with lowest α: 40 %
- Sw on two random partitions: 17 %
- Sw on two partitions in exclusion: 2 %
- Sw on the module with the least memory remaining: 1 %
- Sh on the module with lowest α: 10 %
- Sl on the module with lowest α: 10 %

Of course these values might not be the most favourable ones for every problem instance, however, a suitable rule for dynamically adjusting these probabilities was not found, so these values were determined empirically.

Strategy

Upon description of these operators, one should intuitively notice that these operators are sufficient for reaching any possible distribution of partitions among modules. In particular, just applying Mov at random visits every possible state with respect to the partition assignment to modules, given enough time. Furthermore, with Lop we can reach every equilibrium point corresponding to local maxima.

The implemented algorithms are identical in most aspects. A copy of the best state so far is kept at all times, and is returned if the stopping condition is verified or the algorithm is stopped early. Additionally, a current state (or a population of states) is kept, and the algorithm traverses the search space by applying the operators to generate new states. The new states can be better (in the sense of the optimization criterion) or worse than the current state, and the policy that decides whether to accept or reject a new state is the main distinguishing factor between the meta-heuristic algorithms.

4.3.3 Simulated Annealing

Simulated Annealing is a probabilistic optimization meta-heuristic with an analogy from the cooling of materials. Essentially conceived for minimization problems, states are characterized by their energy which is meant to be minimized. At each iteration, the algorithm generates a candidate state through a given operation. If this has lower energy, then the algorithm always moves to this state, but even if the energy is higher it will move to this state with probability P_a. This probability of accepting a higher energy state is a function of the increase in energy and of the temperature. The temperature is initially high and decreases with successive iterations, with higher temperatures allowing for more likely accepting higher energy states, which is analogous to how minerals form in cooling materials. For maximization problems it is usual to simply flip the sign of the evaluation function [3].

What this means is the algorithm is non-greedy at the start, but becomes progressively more greedy as it progresses. It is essential for any algorithm of this type that worse states are sometimes accepted, as this is important for the search to overcome local maxima. The algorithm is tuned by defining the initial temperature, cooling schedule, and acceptance function [15].

The initial temperature, T_0, should be high to allow exploration of the search space. If too low, the optimization procedure will be greedy and unable to escape local maxima, but if too high, the procedure is essentially random search, and initial iterations are useless. The most common cooling schedule which is used here is the geometric cooling schedule,

$$T(i) = T_0 q^{\frac{i}{N}}, \tag{4.22}$$

where N is the number of iterations, i the current iteration, and $q < 0$ the geometric ratio. With this schedule, the temperature decrease quickly and converges to qT_0.

The acceptance function used is the normalized exponential form [15],

$$P_a = exp\left\{\frac{c - c_{\text{new}}}{k \cdot T(i)}\right\}, \tag{4.23}$$

where c denotes cost ($c \equiv -ev(S)$), and k is a scaling factor. This acceptance function has the characteristic that a sideways move, $c = c_{\text{new}}$, is always taken. The parameters T_0, q, k can be determined experimentally to suit the specific problem. Some authors recommend that, at the highest temperature, the probability associated with the worst expected change in cost should be 50 % [15]. For reference, in this implementation at the starting temperature, a probability of $P_a = 50\,\%$ is verified for a decrease $\Delta\alpha = -0.4$.

Several stopping conditions can be used, such as thresholds for temperature and energy, or mechanisms to detect convergence such as consecutively rejecting too many states or not finding an improving solution in many iterations [15]. Here we use a maximum number of iterations, which is equivalent to defining a final temperature, and a target value for α as the stopping criterion. The implemented algorithm is shown in appendix A.1.

4.3.4 Tabu-search

Tabu-search is another generic optimization algorithm similar to SA. The main difference is that, at each iteration, Tabu-search generates many neighbour states through some permutation of the problem variables and moves to the best one [9]. Ideally, all neighbour states would be generated, however in problems like this one the neighbourhood is large and computationally expensive to explore, thus the search can be restricted to a beam of neighbours. To avoid getting stuck at a certain region of the state-space, the algorithm avoids revisiting solutions by keeping the most recently visited states in memory, in what is called the Tabu list. This is usually implemented as a Last-In First-Out (LIFO) queue of previous states, hashes of states or history of operations. It is updated at each iteration and has a size between 6 and 9 previous states [16].

The rule to always select the best neighbour characterizes this as a greedy algorithm, but the fact that only a limited number of neighbours are explored at a time also helps in combatting this, due to the possibility that all the randomly selected neighbours be worse than the current one. On the other hand, it also becomes possible to miss an improving solution with this mechanic.

The stopping condition is essentially the same as for SA, but one should note that this algorithm is steady-state and thus can run indefinitely without the converging nature of SA. The implemented algorithm is shown in appendix A.2.

4.3.5 Genetic algorithm

Genetic (or evolutionary) algorithms are inspired by the process of evolution through natural selection. Unlike SA and Tabu-search, genetic algorithms operate on multiple states at a time, here called individuals on a population [8].

The strength of a genetic algorithm comes from its exploration and exploitation capabilities. Exploration of the search space is achieved by maintaining a diverse population and inserting random mutations [6]. In this context, mutation consist in modifying states by applying the Mov, Sw, Sh, and Sl operators. Additionally, genetic algorithms perform exploitation or intensification of solutions by combining parts of different solutions, which is the crossover operator detailed in section 4.3.2.

There are different ways of implementing a genetic algorithm, mostly with respect to the population type and size, the choice of operators, the population replacement strategy and the selection of individuals for mating. The performance of the algorithm is greatly dependant on the previous criteria, and these are empirically determined for each specific problem. In problems with an extensive search space and a low ratio of feasible to infeasible solutions, a generic genetic algorithm does not perform better than random search [8].

For this problem, we apply a steady-state genetic algorithm, meaning the populations are overlapping between consecutive generations, as opposed to a classical algorithm where the population is entirely replaced by a new one at each iteration. With this, at each iteration, the worst elements are removed from the population and then the newly created individuals are inserted in their place. The ordering here matters because this way the new individuals cannot be immediately removed. The population size and number of individuals replaced at each iteration should ensure that the best individuals are kept every time, so that there is more opportunity for these to contribute to improving solutions. However, a common problem of this strategy is premature convergence, which happens when the population consists of copies of the same sub-optimal solution. To balance this, a similarity measure between states is introduced, and diversity is promoted by allowing a limited number of similar states in the population.

The similarity measure is based on the partition distribution. We define a cluster as a subset of individuals in the population that share the same partition distribution. Then, we define the maximum size for clusters, and preferentially remove the worst individual in the clusters with more individuals than allowed, and only when all clusters have the allowed size, the worst individuals in the entire population are removed. The implemented algorithm is shown in appendix A.3.

4.4 Summary

We conclude this chapter by summarizing the complete solution methodology, assisted by the flowchart in figure 4.6.

From the problem specification, we first apply the schedulability conditions given in equations 3.24 and 3.25. These are simple, lightweight conditions that can prove a problem infeasible right away, but we restate that these are not sufficient for schedulability. The CSP methodology described in section 4.1 follows, whose purpose is to assign values to f_i, $\forall p_i \in \mathcal{P}$ such that the distribution constraints are met. By using a backtracking algorithm for solving the CSP, which is complete, it is again possible to prove the problem to be infeasible if this procedure fails.

Having found a valid assignment, we apply the Lop operator to compute the offsets, t_i, $\forall p_i \in \mathcal{P}$, and find an initial solution S, which may or may not be valid already. From there, one of the meta-heuristic

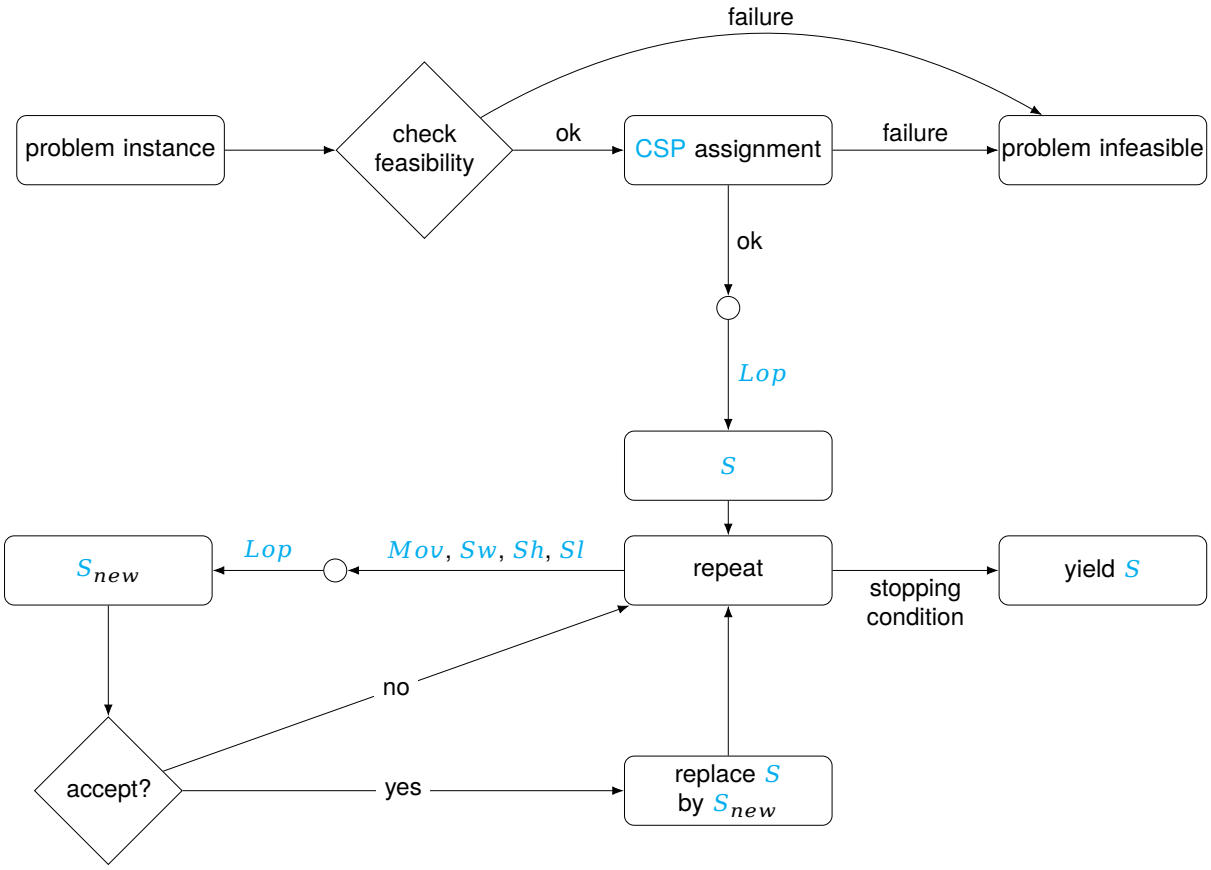


Figure 4.6: Problem solving methodology.

algorithms described in section 4.3 takes over. A new state S_{new} is found, by applying one operator, Mov, Sw, Sh, Sl on the current one, S, followed by local optimization (Lop) on the modules which were affected. Then the acceptance condition specific to the meta-heuristic determines whether or not this new state is accepted, and the process repeats.

However, the flowchart is not accurate for the genetic algorithm which operates on a population of states, for this one specifically consult appendix A.3. Also note Tabu-search generates multiple new states, S_{new} each iteration.

The solution process stops after a stopping condition is verified, which can be a maximum number of iterations, elapsed time, or a solution as good as demanded is found. For the analysis of results in the next chapter, the stopping condition used is this last one.

Chapter 5

Results

In this chapter, we evaluate the computational performance of the procedures detailed in the previous chapter, dividing this description between analysing some specific algorithms and evaluating the developed scheduling tool as a whole. We define test cases of different scales, ranging from a trivial example to an example with scale similar to contemporary industry instances, all generated randomly.

For benchmarking purposes, the results were taken in a machine equipped with an Intel® i7 CPU rated @ 3.6 GHz with 8 MB cache and 16 GB RAM memory, running Linux. The tool and all algorithms are implemented in Python 3.6 and the MILP-based solver used is the open source solver CBC [23]. All time measurements are taken as the sum of processor time in user model and kernel mode on behalf of the program, and every execution exceeding 24 h is aborted.

Since the meta-heuristics used are stochastic in nature and the search-space is large, the states visited and the time needed to find a solution with some target value varies greatly for separate runs. For this reason, the scheduler is run 20 times for each case, and performance evaluated via descriptive statistics. In particular, without inferring on the statistical distribution, the most relevant parameter to determine is the median solution time.

5.1 Local optimization algorithms

As a first analysis we evaluate the performance of some key algorithms that solve subproblems rather than the complete scheduling problem, namely the best response algorithm and its important component, the best value algorithm. Since these algorithms are deterministic, a statistical analysis is not relevant, although the values presented are an average of a few samples with the purpose of mitigating some inaccuracies present when measuring such short executions. In this section, we consider cases with a variable number of partitions whose timing characteristics are listed in table 5.1.

5.1.1 Best value

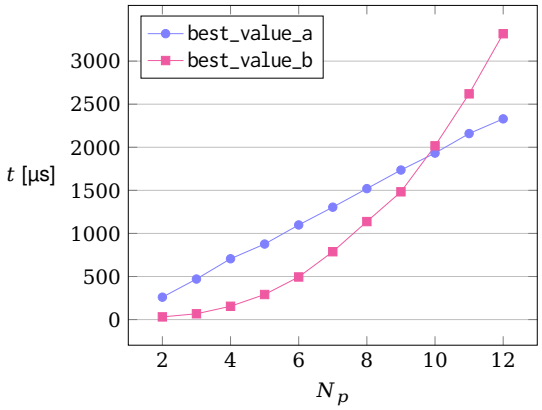
The best value procedure (described in section 4.2.3) is repeated many times inside each $Lop()$ method, therefore its efficiency is critical to the scheduler performance. We evaluate two different algorithms for

Table 5.1: Test case for algorithmic evaluation.

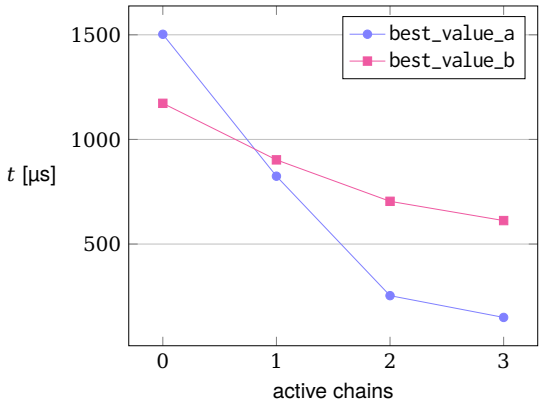
partition	T	e	partition	T	e
1	250	10	7	2000	10
2	1000	50	8	500	10
3	1000	20	9	250	20
4	250	10	10	1000	20
5	1000	100	11	2000	30
6	1000	10	12	500	40

determining the best value. Here, `best_value_a` is the exhaustive version which checks the partition utility at each valid offset, having time complexity $\mathcal{O}(T_i N_p)$, and `best_value_b` is the algorithm that computes the utility only in a set of interest points (not to be confused with the pseudo-code naming in algorithm 4), which has time complexity $\mathcal{O}(T_i {N_p}^3)$. Measurements are taken for several instances with 2 to 12 partitions, and the best value is computed with respect to the partition with index 1 from table 5.1.

Figure 5.1 presents these results. The results from figure 5.1a suggest that version 'a' has time complexity linear with the number of partitions, while version 'b' is asymptotically worse, and for the range where we evaluate it, it is consistent with the cubic complexity predicted in section 4.2.3. Still, for a moderate number of partitions, version 'b' is superior, and the turning point for this example is observed at 10 partitions.



(a) Performance as function of the number of partitions.



(b) Performance as function of the number of chains, $N_p = 8$.

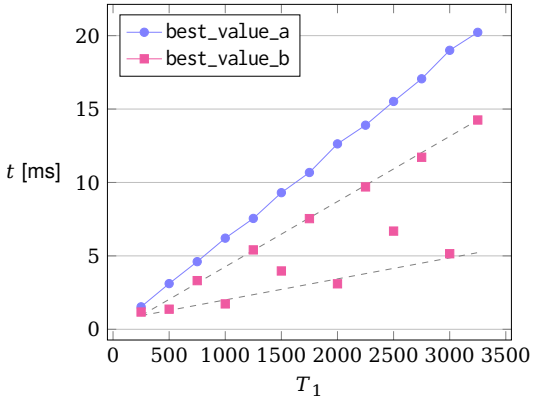
Figure 5.1: Performance of two methods for computing the best value for partition offsets.

Since both these versions are modular, it is possible to dispatch the most efficient algorithm at runtime depending on the conditions. However, this is hard to implement correctly because of the number of variables that have an impact on performance and the wide ranges of values these can take, and experimentation showed that this introduces overhead that ultimately decreases performance for the regular problem instances we consider.

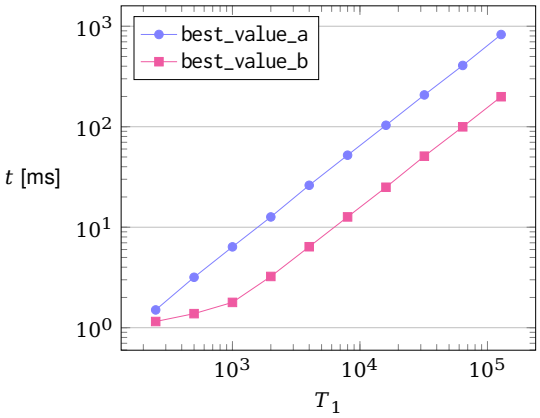
Regarding the effects of chains, it can be seen in figure 5.1b that these speed up both algorithms, but more significantly for version 'a'. When looking quantitatively at these values, it should be noted that the computation of the feasible region via expression 4.9 is done as a step prior to this algorithm. Also, it must be noted that these chains were cherry-picked for this example in order to gradually impose more

restrictiveness to the problem. In real-life instances, the chain delays are not particularly tight, therefore it is common that some chains do not restrict this subproblem at all.

Another factor that influences this algorithm is the partition period for which we compute the best value. To evaluate this, we measure performance when changing this period, T_1, as shown in figure 5.2. Earlier, we predicted linear complexity with respect to the partition period for version 'a', and this is what is verified in figure 5.2a. For version 'b', the same is verified, but additionally, this algorithm also performs better if the partition periods are harmonic, or otherwise if they share large factors with the remaining periods. The figure suggests different slopes, the smallest is when $T_1 = 2^n \cdot 250,\ n \in \mathbb{N}$ which is harmonic with the periods of the remaining partitions from table 5.1. In figure 5.2b, we present results taken only for these periods in a log-log plot, and this evidences the asymptotic linear complexity for both algorithms, and additionally, it shows that version 'b' is superior. In real-life cases we expect periods in the order of hundreds to hundreds of thousands, and we confirm for these values that version 'b' of the algorithm is superior, even for cases with more partitions than presented.



(a) Performance of the two methods for linearly increasing periods.



(b) Performance of the two methods for harmonic periods.

Figure 5.2: Effect of the partition period on the best response calculation, with $N_p = 8$.

Finally, we analyse only cases with up to 12 partitions because in real-life instances the number of partitions per CPM is consistently around this value. Even if future improvements to IMA motivate this number to increase, we present an algorithm that has linear time complexity, so this methodology is not invalidated in this case.

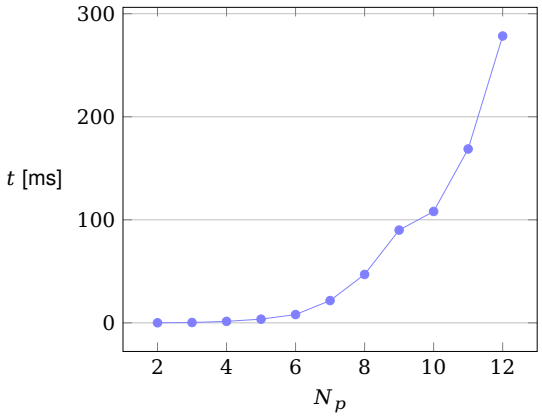
5.1.2 Best response

Evaluation of the performance of the best response algorithm is done for the single module case, as described in section 4.2.2, and for multiple modules using a parallel approach as described in section 4.2.6. For the single module case, we use the same test case from table 5.1, and measure the time taken for the best response algorithm to converge for randomly generated starting points, averaging the result. The optimal α-parameter is determined using the MILP formulation, and we list the percentage of starting points that lead to this optimal value, as well as the average final error in α-parameter and the average time to convergence. The results are listed in table 5.2, and the time to convergence is also plotted in

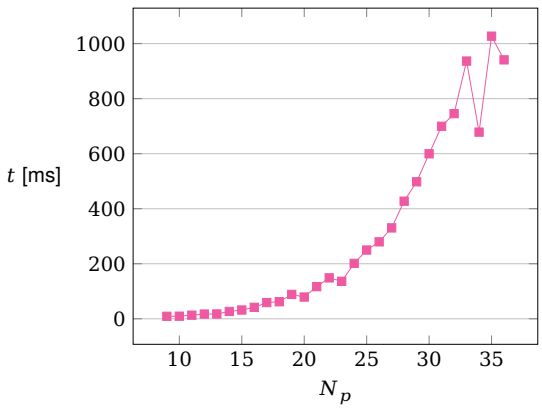
figure 5.3a.

Table 5.2: Performance of the best response algorithm as function of the number of partitions.

N_p	α_{opt}	MILP time [s]	convergence to α_{opt} [%]	average error [%]	average convergence time [ms]
2	4.16	0.74	100	0	0.07
3	4.16	0.72	100	0	0.34
4	3.56	1.23	98.4	1.34×10^{-2}	1.46
5	2.08	5.26	94.6	5.52×10^{-2}	3.58
6	2.08	19.10	96.0	2.58×10^{-1}	8.05
7	2.08	248.24	96.0	1.90×10^{-1}	21.61
8	2.08	936.18	92.2	6.39×10^{-2}	47.00
9	1.78	513.74	42.6	5.73×10^{-1}	90.02
10	1.78	4514.53	41.4	6.43×10^{-1}	108.10
11	1.78	3192.69	46.0	1.20	168.74
12	1.78	8048.55	10.4	14.70	278.35



(a) Single module case.



(b) Algorithm applied in parallel to 3 modules.

Figure 5.3: Performance of the best response algorithm as function of the number of partitions.

In terms of validity, the best response algorithm allows to find the optimal solution from a fraction of starting points, which decreases as the number of partitions increases. Even when the converged state is not an optimal solution, the error to the α-parameter remains small, but the optimal solution is still expected to be found by restarting the algorithm multiple times. Likewise, each restart converges in exponential time as was predicted, and corroborated by the results in figure 5.3a. The conclusion is this algorithm is very efficient for scheduling or optimizing the schedule of a single module, with a small compromise in optimality that is mitigated by repeating the algorithm for several starting points.

Let us analyse the hardest example shown, with 12 partitions. For each randomly generated starting point there is a 10.4 % probability that an optimal solution is found, taking on average 278.35 ms. To reach an optimal solution with a 99 % percent probability, 42 restarts are required, taking a total of 11.7 s. For comparison, the MILP solver takes over 2 h to prove optimality for this example.

For the multi-module approach, we use 3 modules and make similar measurements for 9 to 36 partitions, such that the average number of partitions scheduled in each module remains similar to the previous cases. Like in the previous analysis, the results are listed in table 5.3 and the time to convergence

is plotted in figure 5.3b, however, we elect to omit results for numbers of partitions 29 to 36 for succinctness and because for these large cases we are not able to prove optimality for the solutions found. For the cases shown in table 5.3, optimality is proven by inference. Notice that adding an additional partition to a problem with known α_{opt} creates a problem with at most this value for the α-parameter, thus if a solution with this value is found, it is necessarily optimal. Knowing this, we only need to use the MILP solver for the cases with $N_p = \{9, 18\}$.

Table 5.3: Performance of the parallel best response algorithm as function of the number of partitions.

N_p	α_{opt}	convergence to α_{opt} [%]	average error [%]	average convergence time [ms]
9	2.08	90.0	1.41	9.04
10	2.08	84.0	3.42	9.45
11	2.08	87.0	1.91	13.14
12	2.08	82.0	2.88	17.20
13	2.08	75.0	4.16	17.52
14	2.08	88.0	3.15	26.64
15	2.08	75.0	2.41	31.90
16	2.08	68.0	4.02	41.67
17	2.08	62.0	4.37	58.96
18	1.55	62.0	5.51	61.95
19	1.55	53.0	7.07	88.56
20	1.55	56.0	5.65	79.08
21	1.55	55.0	7.62	116.77
22	1.55	44.0	8.51	148.68
23	1.55	56.0	7.08	136.21
24	1.55	42.0	9.98	201.41
25	1.55	45.0	8.10	249.87
26	1.55	47.0	8.50	279.59
27	1.55	19.0	13.94	330.34
28	1.55	2.00	12.72	427.48

It is clear that solving several modules in parallel increases the total number of equilibrium points for the same total number of partitions, as there will be overall less restriction to each partition. However, we observe that the fraction of starting points that lead to an optimal solution does not decrease as one might expect. Instead, the main effect is the increased average error of the solutions found.

The most important remark is related to the time taken for convergence, which compares favourably against the time for a single module considering the same total number of partitions. In fact, the determining factor for the convergence speed of this algorithm, and consequently the overall scheduler, is the average number of partitions per module. This is further illustrated by the fact that the convergence time is not strictly increasing for this case, as seen in figure 5.3. Sometimes, adding a partition to a module with few partitions can restrict the problem and speed up convergence, however, the overall trend remains an exponential increase in convergence time.

5.2 Complete scheduling tool

5.2.1 Test cases

We define five test cases for the problem without multiple windows, which are identified by the number of modules and partitions, and one test case of moderate complexity for a problem where multiple windows are allowed. The test cases are generated randomly, choosing periods from the set $\{100, 200, 500, 1000\}$ which allows non-harmonic periods, and durations up to 15 % of the respective period. With respect to the distribution constraints, the memory requirements of the partitions are set to about 40 % of the modules' capacity, thus imposing some restriction. About 20 % and 5 % of partitions are subject to an exclusion and inclusion constraints, respectively, and the domains are not restricted. Regarding communications, chains are defined with maximum delays in the order of magnitude of the partition periods, but always in a way that some restriction is imposed. The network delays are in the order of magnitude of partition execution requirements and required to be independent of the direction. The summary of these test cases is presented in table 5.4 and the full characteristics for replication purposes are given in appendix B.

Table 5.4: Test cases definition.

Designation	N_c	N_p	Number of chains	α_{best}
$2M6P$	2	6	0	5.5
$4M10P$	4	10	3	6.403
$4M20P$	4	20	8	2.875
$8M40P$	8	40	15	2.984
$20M100P$	20	100	40	2.325
$3M15P$-S	3	15	3	1.26

The scheduling tool was repeatedly applied to these test cases in the course of gathering results, and the value α_{best} consists of the best solution ever found, but we do not guarantee that this is the optimal solution, with the exception of $2M6P$, for which the MILP model was able to prove optimality.

For the problem allowing multiple windows of execution, we define only one test case, $3M15P$-S, which is generated similarly to the previous ones but with cherry-picked preemption points for 4 partitions, and adjusted durations such that it is infeasible without multiple windows. This test case has 3 chains constraining the partitions that cannot execute in multiple windows, and $\varepsilon_m = 1 \ \forall m$.

5.2.2 Feasibility problem

As discussed, asserting feasibility is done by solving the problem until a solution with $ev(S) \geq 1$ is found, thus we begin by evaluating the scheduler for solving the test cases with an upper bound of 1 on the evaluation function.

This is compared to the MILP implementation by adapting the model from section 3.4. We do this by adding a constraint $\alpha \geq 1$ and removing the optimization goal, which in contrast to the heuristic approach consists in a rigorous method to determine feasibility.

Results are shown in table 5.5. This shows that both methods are effective for simple cases, but the

Table 5.5: Scheduler performance finding the first valid solution.

Instance	t_{MILP}	$t_{heuristic}$ (median)
$2M6P$	1.00 s	0.593 s
$4M10P$	1.34 s	0.595 s
$4M20P$	6.29 s	0.830 s
$8M40P$	310.7 s	1.155 s
$20M100P$	> 24 h	23.32 s
$3M15P$-S	NA	74.53 s

solution time increases more drastically for the MILP solver, finally not converging in useful time for the hardest test case. One must note that all examples are subject to an approximately constant time that is spent on parsing and validating the problem data. For example, the actual solution time for the MILP solver (as interfaced by the software) for $2M6P$ is 7.273 ms.

The performance of the scheduler is appropriate for the industry setting, where we can expect it to be able to verify feasibility in seconds even for complex problem instances. This is certainly useful because system integration depends on many interactions with the different suppliers to define all the parameters and requirements, which are then translated into constraints accepted by this model. In this stage, the problem changes quickly and having a tool to check feasibility and build a simple solution is of great benefit to the system integrator.

Again, it must be stressed that only the MILP approach is able to prove infeasibility, in spite of the heuristic approach failing to provide a valid solution being a strong indication that an instance is infeasible. The MILP model does not support multiple windows and as such is not applicable to $3M15P$-S, however, assuming single windows, the solver proves infeasibility for this case in 1.19 s.

These results already show the price of allowing multiple windows to the problem complexity, as we see the solution time for $3M15P$-S is superior to any of the other test cases that are substantially larger in scale.

5.2.3 Optimization problem

For the optimization problem, we are interested in finding the optimal solution. Here, the MILP approach does not converge in admissible time for any case other than $2M6P$, thus the analysis in this section focuses only on the heuristic scheduler, and we compare the implemented meta-heuristics.

Since it not possible to prove optimality with the heuristic scheduler, we settle for evaluating the solution time to a *good* solution, characterized by a value close to α_{best}. For convenience in gathering results, seen as the solution can take hours and must be repeated many times, we choose a target value of $0.9\alpha_{best}$ for the test cases of larger scale.

The results are presented in figure 5.4 in the form of boxplots. For the reader not familiar with these, boxplots present the quartiles of the data set, with the box detailing the first to third quartiles, divided by the median value. The whiskers can represent different things depending on the author. Here, the upper whisker details the highest data sample still within $1.5IQR$ of the lower quartile, IQR being the interquartile range, and the bottom whisker is defined identically, with any outliers marked in the plot.

Boxplots do not make assumptions of the underlying statistical distribution, but make it possible to visualize the spread and skewness in the data.

The results indicate that SA is best for smaller problems and Tabu-search more reliable for larger problems, while the genetic algorithm shows no clear advantage compared to these two for any of these test cases.

Tabu-search always computes a fixed number of different solutions at each iteration and only one is really used, meaning that the others are effectively wasted. When the problem is easy and an improving solution is likely to be found, this procedure wastes time computing many solutions at once. In contrast, SA immediately accepts improving solutions, hence outperforms Tabu-search for easy problems.

For harder cases, the benefits of computing many solutions at each iteration start to show, and Tabu-search shows better results for $8M40P$ and $20M100P$. Still, it is interesting to note that with many attempts, SA likely achieves the minimum solution time of any meta-heuristic, happening when the search progress is especially lucky, as is the case in figure 5.4e. The converse is also observed, Tabu-search presents a higher maximum solution time than SA on all test cases up to $8M40P$, and in general presents more outliers than any other meta-heuristic, which seems to happen when the search progress is particularly unlucky.

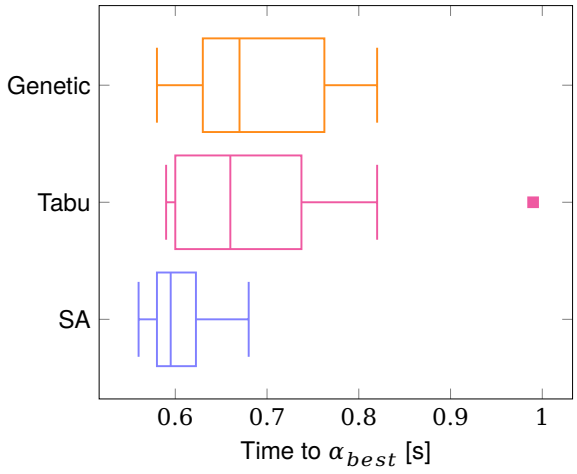
The genetic algorithm is comparable to the other two for easier cases, but for $8M40P$ it performs significantly worse, and for $20M100P$ it does not reach the target value in useful time. Since the distinguishing feature of the genetic algorithm as compared to the others is the crossover operator, the conclusion is that combining two solutions in the way which was defined is not beneficial to the search process.

As for the consequences to the industrial setting, based on the results for $20M100P$, we can expect to have a fairly optimized solution for the partition scheduling problem consistently in under an hour of processor time (over 75 % of attempts), which is an excellent result. We further expect that real-life cases may be slightly larger than $20M100P$, but given the moderate scaling of the solution time when going from $4M20P$ to $8M40P$ to $20M100P$, it should remain in the order of hours. Regarding the multiple window case, the solution time of $3M15P$-S is located in between $8M40P$ and $20M100P$, which demonstrates the expensive increase in complexity when we allow multiple windows, and this indicates that, from the scheduling point of view, multiple windows should be avoided unless absolutely necessary for problem feasibility.

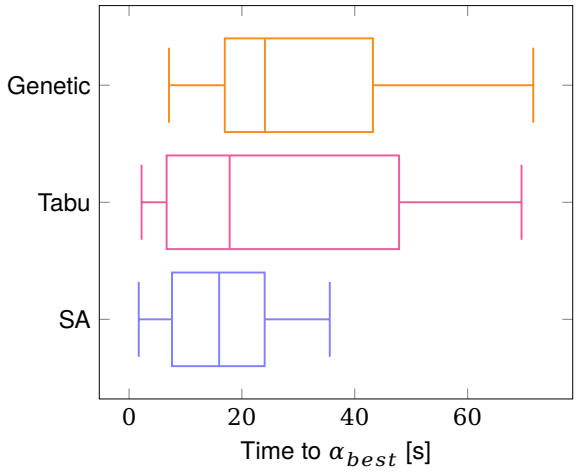
Comparison with the exact model

As mentioned, the exact model only proves optimality for $2M6P$ (subject to a 24 h timeout), taking 3.16 s. A passing mention should be made that there is currently a huge gap in the performance of open source MILP solvers and commercial ones [30], and even among the commercial ones, the performance varies greatly depending on the specific problem, thus it is common practice to experiment with different solvers to find the most suitable one for the specific problem. This means the MILP model can potentially perform much better than reported here, and be a viable solution for the more complex examples.

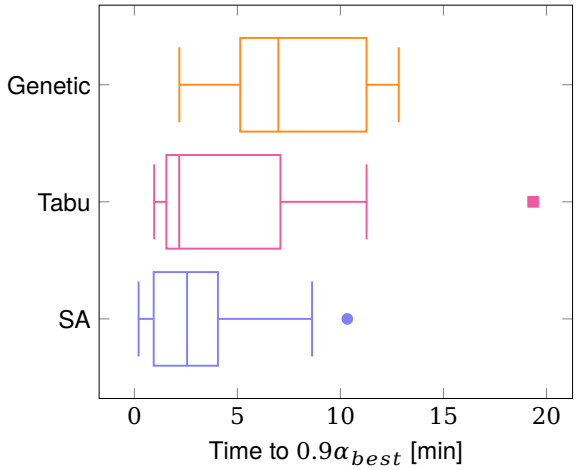
Another clear difference is the heuristic method guarantees that each individual partition is scheduled



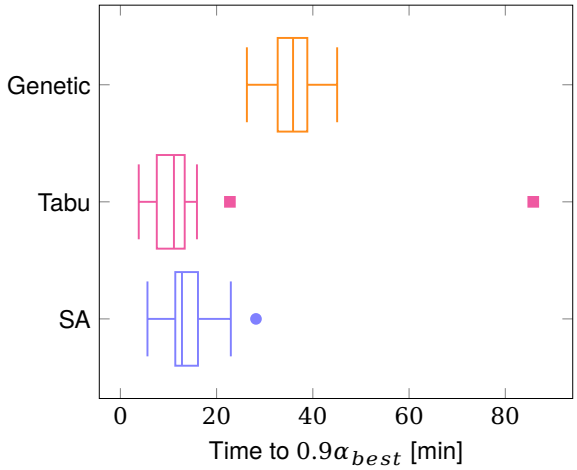
(a) $2M6P$



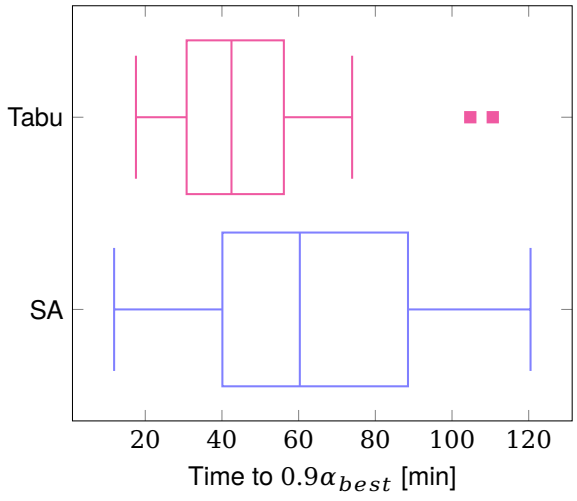
(b) $4M10P$



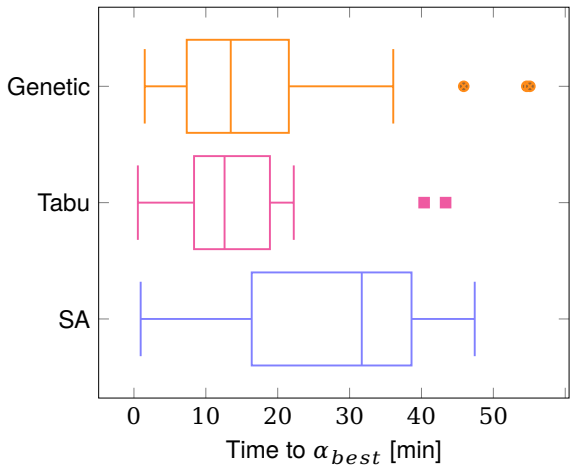
(c) $4M20P$



(d) $8M40P$



(e) $20M100P$



(f) $3M15P$-S

Figure 5.4: Performance of the scheduler with different meta-heuristics.

in a locally optimal slot that maximizes its own execution potential, whereas the exact solver settles for an execution potential equal to the worst one in the system. In other words, the MILP approach maximizes the partition utility in the worst case with no regard for the average case, as pointed out in Al Sheikh et al. [28]. Going around this would require a multi-objective evaluation function. Here we look at the example of $2M6P$, in table 5.6. The solutions found by the MILP solver and the heuristic scheduler have $\alpha = 5.5$, but the MILP model achieves an average partition utility $\bar{\alpha}_i = 5.87$, and the heuristic scheduler yields $\bar{\alpha}_i = 12.23$.

Since the heuristic scheduler relies on the best response algorithm, it suffers from the same limitations for cases with high processor usage fractions, while the exact model does not, being limited only by the problem dimension.

Table 5.6: Two optimal solutions for $2M6P$.

	MILP			Heuristic		
i	f_i	t_i	α_i	f_i	t_i	α_i
1	2	45	5.50	2	28	5.60
2	1	462	5.52	1	430	13.87
3	1	0	7.60	1	900	29.03
4	2	90	5.50	2	45	5.50
5	1	291	5.52	1	0	13.87
6	2	62	5.60	2	0	5.50

Comparison to related work

Comparison with related work, [28, 41], is not straightforward since there are differences in the models considered. Particularly, we consider more comprehensive distribution and communication constraints, and some sacrifices were taken to accommodate these differences.

Al Sheikh et al. [28] randomly generate several cases with 4 modules and 40 partitions and evaluate the CPU time, reaching results between 5 min to 50 min and averaging at 27.4 min. Their methodology does not consider communication constraints, multiple windows, inclusion and domains constraints, but considers memory and exclusion constraints. Also, the scheduler is stopped after a given percentage of the estimated equilibria are visited.

Since we are not able to replicate this stopping condition, we elect to use the same technique of stopping after a solution with evaluation function close to α_{best} is found. Given that this model is more loosely constricted, we choose to stop at $0.95\alpha_{best}$, and again it must be pointed out that the optimal value of the cases considered in these experiments is not known.

Therefore, we generate 8 test cases as described in Al Sheikh et al. [28], however, we decide to decrease the average partition execution time from 20 % of its respective period to 5 %, otherwise the generated test cases are infeasible due to the processor usage fraction being too high. We run these cases with the Tabu-search algorithm, whose results are listed in table 5.7.

Our results are similar, achieving almost identical minimum and maximum execution times, but we obtain a lower mean execution time of 15.25 min. Despite the disputable stopping condition used in this comparison, we can conclude that our methodology is an improvement on previous work.

Table 5.7: Results for problem instances based on Al Sheikh et al. [28].

Case	α_{best}	time to $0.95\alpha_{best}$ [min]	Case	α_{best}	time to $0.95\alpha_{best}$ [min]
$4M40P$	2.00	9.97	$4M40P$	2.54	13.36
$4M40P$	2.00	5.78	$4M40P$	1.29	23.67
$4M40P$	1.72	9.83	$4M40P$	1.42	8.34
$4M40P$	1.83	5.17	$4M40P$	1.72	45.85

Finally, Pira and Artigues [41] report results hundreds of times faster for their implementation that is also based on the best response heuristic, but this work does not consider any kind of distribution or communication constraints.

Chapter 6

Conclusions

This chapter presents an overview of the topics discussed in this thesis and of the conclusions obtained, and leaves suggestions for future work.

6.1 Overview

The research work presented in this dissertation focused on formally describing an avionics partition scheduling problem, and providing methods for efficiently solving it. The final product is to be integrated in GMV's tool suite for system configuration.

A mathematical model is provided, detailing how to express high-level system requirements in terms of problem variables and constraints. The overall problem is solved using different classes of algorithms. A MILP formulation optimally solves the problem in its simplest form, but it is incapable of handling large instances commonly encountered in this domain, thus heuristic methods are employed. We use a constraint-based approach to build initial resource allocations or prove infeasibility on this subtask, and general purpose stochastic optimization algorithms, namely SA, Tabu-search and a genetic algorithm are adapted to perform exploration around this starting solution. Tabu-search is found to be suitable for examples of modern scale, such as those encountered in modern avionic systems. In addition, we adapt an existing local optimization procedure, called the best response algorithm, borrowed from the field of game theory, that performs exploitation of solutions, and is shown superior to any other method for this task.

6.2 Achievements

Firstly, the described model remains compatible with most similar approaches to the problem, while supporting more kinds of constraints, which allow the user (the system integrator) to adequately specify platform requirements, with respect to resource usage as well as the redundancy architecture.

The different methods implemented also allow the scheduling tool to be used for multiple purposes. The heuristic methods are generally able to quickly provide a solution that verifies all constraints, and

this is useful for determining feasibility of certain instances in the early stages of integration. Additionally, in later phases of system integration, the heuristic methods are able to create optimized solutions in a moderate amounts of time. If on the other hand optimality is the goal and time is not an issue, then any external solver can take our MILP model and solve the problem to optimality.

The tool is also equipped with features that assist the user to make decisions along the project life cycle, and in general decreases manual effort in this task:

- Easy to maintain problem definition based on configuration files.
- Evaluation of solutions and logs detailing which specific constraints are not met, and which pose no restriction on the problem.
- Insight on the cause for infeasibility in certain cases.
- Visualisation of schedules in graphical format similar to the images included in this dissertation, and in table format.

In conclusion, the objectives stated in the introduction were accomplished. In addition, this work poses a contribution to academic research due to the novel changes imposed in our model, namely the synchronous communications model and the possibility for splitting a partition's execution in multiple windows.

6.3 Future work

Scheduling for IMA platforms remains an open area of research. One of the main difficulties faced is the non-existence of a standardized model that accurately expresses the functional requirements of these systems. Ultimately, some of these requirements are implementation-specific, and this means research effort is spread over many slightly different scheduling problems.

This model considers that the avionic architecture is already defined, including the available hardware and communications infrastructure. There are techniques for defining and optimizing IMA architectural topologies [33]; and parametrizing network delays [44], which are tasks that are tightly coupled with partition scheduling. However, these are currently approached independently, so a holistic approach integrating these tasks would be relevant.

Other smaller modifications to the described model are also interesting to consider. Namely, our optimization criterion aims to maximize the worst case partition utility, but a weighed sum of all partition utilities could be more relevant as it gives the user the decision of which partitions are more relevant to give larger execution budgets. Other criteria can also apply, namely minimizing the usage of the communications network. Finally, more comprehensive modelling of communication constraints is required, and this likely requires that we specify and schedule each individual message, rather than setting maximum time separation between partitions that exchange messages.

Finally, I consider that given the recent additions to Arinc specification 653 and the lacklustre performance observed in our scheduler with the addition of multiple partition execution windows, the most important future work on partition scheduling should be dedicated to multicore IMA systems.

Bibliography

[1] John Nash. 'Non-cooperative games'. In: *Annals of mathematics* (1951), pp. 286–295. DOI: 10.2307/1969529.

[2] Chung Laung Liu and James W Layland. 'Scheduling algorithms for multiprogramming in a hard-real-time environment'. In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61. DOI: 10.1145/321738.321743.

[3] Scott Kirkpatrick, C Daniel Gelatt and Mario P Vecchi. 'Optimization by simulated annealing'. In: *science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671.

[4] J.H.M. Korst. 'Periodic multiprocessor scheduling'. English. PhD thesis. Technische Universiteit Eindhoven, Department of Mathematics and Computer Science, 1992. DOI: 10.6100/IR388787.

[5] Larry J Eshelman and J David Schaffer. 'Real-coded genetic algorithms and interval-schemata'. In: *Foundations of genetic algorithms*. Vol. 2. Elsevier, 1993, pp. 187–202. DOI: 10.1016/B978-0-08-094832-4.50018-0.

[6] David B Fogel. 'An introduction to simulated evolutionary optimization'. In: *IEEE transactions on neural networks* 5.1 (1994), pp. 3–14. DOI: 10.1109/72.265956.

[7] Jan Korst, Emile Aarts and Jan Karel Lenstra. 'Scheduling periodic tasks'. In: *INFORMS journal on Computing* 8.4 (1996), pp. 428–435. DOI: 10.1287/ijoc.8.4.428.

[8] Matthew Bartschi Wall. 'A genetic algorithm for resource-Constrained scheduling'. PhD thesis. Massachusetts Institute of Technology, 1996.

[9] Fred Glover and Manuel Laguna. 'Tabu search'. In: *Handbook of combinatorial optimization*. Springer, 1998, pp. 2093–2229. DOI: 10.1007/978-1-4613-0303-9_33.

[10] Y-H Lee, Daeyoung Kim, Mohammed Younis and Jeff Zhou. 'Scheduling tool and algorithm for integrated modular avionics systems'. In: *19th DASC. 19th Digital Avionics Systems Conference. Proceedings (Cat. No. 00CH37126)*. Vol. 1. IEEE. 2000, pp. 1C2–1. DOI: 10.1109/DASC.2000.886885.

[11] Amotz Bar-Noy, Vladimir Dreizin and Boaz Patt-Shamir. 'Efficient algorithms for periodic scheduling'. In: *Computer Networks* 45.2 (2004), pp. 155–173. DOI: 10.1016/j.comnet.2003.12.017.

[12] DA Gwaltney and JM Briscoe. *Comparison of communication architectures for spacecraft modular avionics systems*. Marshall Space Flight Center, AL, US: NASA, 2006.

[13] Omar Kermia, Liliana Cucu and Yves Sorel. 'Non-preemptive multiprocessor static scheduling for systems with precedence and strict periodicity constraints'. In: *Proceedings of the 10th International Workshop On Project Management and Scheduling, PMS'06*. 2006.

[14] Oded Goldreich. 'Computational complexity: a conceptual perspective'. In: *ACM Sigact News* 39.3 (2008), pp. 35–39. DOI: 10.1145/1412700.1412710.

[15] Heikki Orsila, Erno Salminen and Timo D Hämäläinen. 'Best practices for simulated annealing in multiprocessor task distribution problems'. In: *Simulated annealing*. IntechOpen, 2008.

[16] Michael Pinedo. *Scheduling. Theory, Algorithms, and Systems*. 3rd ed. Springer, 2008. DOI: 10.1007/978-0-387-78935-4.

[17] Paul J Prisaznuk. 'ARINC 653 role in integrated modular avionics (IMA)'. In: *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*. IEEE. 2008, 1–E. DOI: 10.1109/DASC.2008.4702770.

[18] *Arinc Specification 653: Aircraft Data Network. Part 7: Avionics Full-Duplex Switched Ethernet Network*. Aeronautical Radio, Incorporated. 2009.

[19] Arvind Easwaran, Insup Lee, Oleg Sokolsky and Steve Vestal. 'A compositional scheduling framework for digital avionics systems'. In: *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2009, pp. 371–380. DOI: 10.1109/RTCSA.2009.46.

[20] Ahmad Al Sheikh, Olivier Brun and Pierre-Emmanuel Hladik. 'Partition scheduling on an IMA platform with strict periodicity and communication delays'. In: *18th international conference on real-time and network systems*. 2010, pp. 179–188.

[21] Friedrich Eisenbrand, Nicolai Hähnle, Martin Niemeier, Martin Skutella, José Verschae and Andreas Wiese. 'Scheduling periodic tasks in a hard real-time environment'. In: *International colloquium on automata, languages, and programming*. Springer. 2010, pp. 299–311. DOI: 10.1007/978-3-642-14165-2_26.

[22] Friedrich Eisenbrand, Karthikeyan Kesavan, Raju S Mattikalli, Martin Niemeier, Arnold W Nordsieck, Martin Skutella, José Verschae and Andreas Wiese. 'Solving an avionics real-time scheduling problem by advanced IP-methods'. In: *European Symposium on Algorithms*. Springer. 2010, pp. 11–22. DOI: 10.1007/978-3-642-15775-2_2.

[23] COIN-OR Foundation. *CBC – Coin-or branch and cut*. Version 2.9.0. 2010. URL: https://github.com/coin-or/Cbc (visited on 08/07/2019).

[24] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media, 2011. DOI: 10.1007/978-1-4614-0676-1.

[25] Mohamed Marouf and Yves Sorel. 'Scheduling non-preemptive hard real-time tasks with strict periods'. In: *ETFA2011*. IEEE. 2011, pp. 1–8. DOI: 10.1109/ETFA.2011.6059014.

[26] Slawomir Samolej. 'ARINC Specification 653 Based Real-Time Software Engineering.' In: *e-Informatica* 5.1 (2011), pp. 39–49.

[27] James Windsor, Marie-Hélène Deredempt and Regis De-Ferluc. 'Integrated modular avionics for spacecraft. User requirements, architecture and role definition'. In: *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*. IEEE. 2011, 8A6–1. DOI: 10.1109/DASC.2011.6096141.

[28] Ahmad Al Sheikh, Olivier Brun, Pierre-Emmanuel Hladik and Balakrishna J Prabhu. 'Strictly periodic scheduling in IMA-based architectures'. In: *Real-Time Systems* 48.4 (2012), pp. 359–386. DOI: 10.1007/s11241-012-9148-y.

[29] Giorgio C. Buttazzo, Marko Bertogna and Gang Yao. 'Limited preemptive scheduling for real-time systems. a survey'. In: *IEEE Transactions on Industrial Informatics* 9.1 (2012), pp. 3–15. DOI: 10.1109/TII.2012.2188805.

[30] Bernhard Meindl and Matthias Templ. 'Analysis of commercial and free and open source solvers for linear optimization problems'. In: *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS* 20 (2012).

[31] Cláudio Silva. 'Integrated modular avionics for space applications. Input/output module'. MA thesis. Instituto Superior Técnico, Universidade de Lisboa, 2012.

[32] CM Ananda, Sabitha Nair and GH Mainak. 'ARINC 653 API and its application–An insight into Avionics System Case Study'. In: *Defence Science Journal* 63.2 (2013), pp. 223–229. DOI: 10.14429/dsj.63.4268.

[33] Björn Annighöfer and Frank Thielecke. *Supporting the design of distributed integrated modular avionics systems with binary programming*. Deutsche Gesellschaft für Luft-und Raumfahrt-Lilienthal-Oberth eV, 2013.

[34] Sofiene Beji, Sardaouna Hamadou, Abdelouahed Gherbi and John Mullins. 'SMT-based cost optimization approach for the integration of avionic functions in IMA and TTEthernet architectures'. In: *Proceedings of the 2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society. 2014, pp. 165–174. DOI: 10.1109/DS-RT.2014.28.

[35] Claudio Silva and Cassia Tatibana. 'MultIMA-Multi-Core in integrated modular avionics'. In: *DASIA 2014-DAta Systems in Aerospace*. Vol. 725. 2014.

[36] *Arinc Specification 653: Avionics Application Software Standard Interface. Part1: Required Services*. Aeronautical Radio, Incorporated. 2015.

[37] Thomas Gaska, Chris Watkin and Yu Chen. 'Integrated modular avionics-past, present, and future'. In: *IEEE Aerospace and Electronic Systems Magazine* 30.9 (2015), pp. 12–23. DOI: 10.1109/MAES.2015.150014.

[38] Aamir Mairaj. 'Preferred choice for resource efficiency: Integrated Modular Avionics versus federated avionics'. In: *2015 IEEE Aerospace Conference*. IEEE. 2015, pp. 1–6. DOI: 10.1109/AERO.2015.7119127.

[39] Mitra Nasri and Gerhard Fohler. 'An efficient method for assigning harmonic periods to hard real-time tasks with period ranges'. In: *2015 27th Euromicro Conference on Real-Time Systems*. IEEE. 2015, pp. 149–159. DOI: 10.1109/ECRTS.2015.21.

[40] Stéphane Durand and Bruno Gaujal. 'Complexity and optimality of the best response algorithm in random potential games'. In: *International Symposium on Algorithmic Game Theory*. Springer. 2016, pp. 40–51. DOI: 10.1007/978-3-662-53354-3_4.

[41] Clément Pira and Christian Artigues. 'Line search method for solving a non-preemptive strictly periodic scheduling problem'. In: *Journal of Scheduling* 19.3 (2016), pp. 227–243. DOI: 10.1007/s10951-014-0389-6.

[42] Tiyam Robati. 'Model-based techniques for the future integration for formal verification of critical real-time software systems'. PhD thesis. École de technologie supérieure, 2016.

[43] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. 3rd ed. Malaysia; Pearson Education Limited, 2016.

[44] Nassima Benammar, Frédéric Ridouard, Henri Bauer and Pascal Richard. 'Forward End-to-End Delay for AFDX Networks'. In: *IEEE Transactions on Industrial Informatics* 14.3 (2017), pp. 858–865. DOI: 10.1109/TII.2017.2720799.

[45] Jaishree Mayank and Arijit Mondal. 'Non-preemptive multiprocessor scheduling for periodic real-time tasks'. In: *2017 7th International Symposium on Embedded Computing and System Design (ISED)*. IEEE. 2017, pp. 1–6. DOI: 10.1109/ISED.2017.8303931.

[46] Alessandra Melani, Renato Mancuso, Marco Caccamo, Giorgio Buttazzo, Johannes Freitag and Sascha Uhrig. 'A scheduling framework for handling integrated modular avionic systems on multicore platforms'. In: *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE. 2017, pp. 1–10. DOI: 10.1109/RTCSA.2017.8046314.

[47] Hongliang Zheng, Yuanju He, Lingyu Zhou, Yiou Chen and Xiang Ling. 'Scheduling of non-preemptive strictly periodic tasks in multi-core systems'. In: *2017 International Conference on Circuits, Devices and Systems (ICCDS)*. IEEE. 2017, pp. 195–200. DOI: 10.1109/ICCDS.2017.8120477.

[48] Miguel Barros, José Neves, José Negrão, Sérgio Penna and Marco Ortiz. 'Distributed Integrated Modular Avionics'. In: NATO. 2018.

[49] Mathias Blikstad, Emil Karlsson, Tomas Lööw and Elina Rönnberg. 'An optimisation approach for pre-runtime scheduling of tasks and communication in an integrated modular avionic system'. In: *Optimization and Engineering* 19.4 (2018), pp. 977–1004. DOI: 10.1007/s11081-018-9385-6.

Appendix A

Optimization algorithms

A.1 Simulated Annealing algorithm

Algorithm 5 SA algorithm for the partition scheduling problem.

Require: $N \leftarrow$maximum number of iterations

Require: $\gamma \leftarrow$ upper bound on the target function (α) ▷ Optional

Require: state $\leftarrow$ initial state

Require: $T_0 \leftarrow$ Initial temperature

1: **procedure** Simulated_Annealing(state)
2: best $\leftarrow$ state
3: $\alpha_{\text{best}} \leftarrow ev$(state)
4: $e \leftarrow -ev$(state) ▷ Energy
5: $i \leftarrow 0$
6: **while** $i < N$ **do**
7: $T \leftarrow T_0 q^{\frac{i}{N}}$
8: $op \leftarrow$random_choice(Mov, Sw, Sh, Sl)
9: candidate $\leftarrow op$(state)
10: Lop(candidate)
11: $e_c \leftarrow -ev$(candidate)
12: **if** $e_c < e$ **then**
13: state $\leftarrow$ candidate
14: $e \leftarrow e_c$
15: **else**
16: $P_a \leftarrow exp\left(\frac{e-e_c}{k \cdot T}\right)$
17: **if** true with probability P_a **then**
18: state $\leftarrow$ candidate
19: $e \leftarrow e_c$

Algorithm 5 SA algorithm for the partition scheduling problem. (cont.)

```
            end if
        end if
        if −e > α_best then
            best ← state
            α_best ← −e
        end if
        if α_best ≥ γ then
            return best
        end if
        i ← i + 1
    end while
    return best
end procedure
```

A.2 Tabu search algorithm

Algorithm 6 Tabu search algorithm for the partition scheduling problem.

Require: $N \leftarrow$ maximum number of iterations

Require: $\gamma \leftarrow$ upper bound on the target function (α) ▷ Optional

Require: state ← initial solution

Require: tabu_size ← size of the tabu-list

Require: beam ← beam width ▷ Number of neighbors expanded at each iteration

```
 1: procedure Tabu_Search(state)
 2:     best ← state
 3:     α_best ← ev(state)
 4:     current ← state
 5:     tabu_list ← a LIFO queue with size tabu_size, filled with dummy data
 6:     i ← 0
 7:     while i < N do
 8:         e_best ← −∞
 9:         for j ← 1 ... beam do
10:             op ←random_choice(Mov, Sw, Sh, Sl)
11:             candidate ← op(state)
12:             Lop(candidate)
13:             e ← ev(candidate)
14:             if e > e_best and candidate ∉ tabu_list then
15:                 next ← candidate
16:                 e_best ← e
```

Algorithm 6 Tabu search algorithm for the partition scheduling problem. (cont.)

```
            end if
        end for
        state ← next
        push state into tabu_list
        pop from tabu_list
        if e_best > α_best then
            best ← state
            α_best ← e_best
        end if
        if α_best ≥ γ then
            return best
        end if
        i ← i + 1
    end while
    return best
end procedure
```

A.3 Genetic algorithm

Algorithm 7 Genetic algorithm for the partition scheduling problem.

Require: $N \leftarrow$ maximum number of generations
Require: state ← initial solution
Require: $P \leftarrow$ population size
Require: $M \leftarrow$ maximum cluster size
Require: overlap ← population overlap between generations
Require: $p_m \leftarrow$ mutation probability

```
1:
2: procedure Genetic_Algorithm(state)
3:     population ← list of size P filled with copies of state
4:     n_rep ← ⌊(1 − overlap) × P⌋        ▷ number of new individuals created at each generation
5:     while i < N_gen do
6:         for j ∈ n_rep do
7:             S_1 ← random_choice(population)
8:             S_2 ← random_choice(population)
9:             child_j ← crossover(S_1, S_2)
10:            if true with probability p_m then
11:                op ←random_choice(Mov, Sw, Sh, Sl)
```

Algorithm 7 Genetic algorithm for the partition scheduling problem. (cont.)

```
12:                 child_j ← op(child_j)
13:             end if
14:             Lop(child_j)
15:         end for
16:         replace(population, child_1, child_2, ... )
17:         i ← i + 1
18:     end while
19:     return max_α (population)
20: end procedure
21:
22: function replace(population, children)
23:     clusters ← group population by similarity
24:     for each i ← 1 ... length(children) do
25:         c ← largest(clusters)
26:         if size(c) > M then
27:             Remove worst from c
28:         else
29:             Remove worst from population
30:         end if
31:     end for
32:     insert children into population
33: end function
```

Appendix B

Test Cases

B.1 $2M6P$

Table B.1: Problem specification for $2M6P$.

Partition	Period	Duration	Memory	Domain
1	1000	1	9	All modules
2	1000	31	9	All modules
3	500	5	5	All modules
4	100	3	4	All modules
5	100	10	1	All modules
6	100	5	1	All modules

(a) Partition information.

Module	Memory
1	36
2	36

(b) Module information.

B.2 $4M10P$

Table B.2: Problem specification for $4M10P$.

Partition	Period	Duration	Memory	Domain
1	1000	23	8	All modules
2	1000	41	7	All modules
3	1000	56	8	All modules
4	1000	77	6	All modules
5	1000	35	8	All modules
6	500	14	7	All modules
7	500	14	5	All modules
8	200	1	7	All modules
9	200	12	3	All modules
10	100	8	7	All modules

(a) Partition information.

Module	Memory
1	32
2	31
3	32
4	31

(b) Module information.

Chain	Delay
$p_8 \rightarrow p_7$	121
$p_3 \rightarrow p_1$	842
$p_8 \rightarrow p_6$	123

(c) Chain constraints.

m, n	1	2	3	4
1	-	25	5	12
2	25	-	6	6
3	5	6	-	12
4	12	6	12	-

(d) Network delays.

Other constraints: $f_8 \neq f_9$, $f_5 \neq f_{10}$.

B.3 $4M20P$

Table B.3: Problem specification for $4M20P$.

Partition	Period	Duration	Memory	Domain
1	1000	46	6	All modules
2	1000	29	8	All modules
3	1000	73	1	All modules
4	500	16	4	All modules
5	500	29	9	All modules
6	500	29	3	All modules
7	500	5	6	All modules
8	500	40	8	All modules
9	500	24	1	All modules
10	200	1	3	All modules
11	200	21	9	All modules
12	200	18	8	All modules
13	200	21	2	All modules
14	200	15	7	All modules
15	200	23	2	All modules
16	200	3	7	All modules
17	100	9	6	All modules
18	100	1	6	All modules
19	100	2	5	All modules
20	100	11	8	All modules

(a) Partition information.

Module	Memory
1	60
2	63
3	62
4	63

(b) Module information.

Chain	Delay
$p_7 \rightarrow p_{14}$	98
$p_{13} \rightarrow p_9$	125
$p_{18} \rightarrow p_5$	459
$p_9 \rightarrow p_2$	550
$p_{10} \rightarrow p_{11}$	134
$p_{19} \rightarrow p_3$	582
$p_{15} \rightarrow p_1$	1003
$p_5 \rightarrow p_{19}$	52

(c) Chain constraints.

m, n	1	2	3	4
1	-	13	10	13
2	13	-	11	18
3	10	11	-	16
4	13	18	16	-

(d) Network delays.

Other constraints: $f_6 \neq f_{19}$, $f_{14} \neq f_4$, $f_{12} \neq f_{18}$, $f_{15} \neq f_{14}$, $f_7 \neq f_2$, $f_{13} \neq f_{11}$.

B.4 $8M40P$

Table B.4: Problem specification for $8M40P$.

Partition	Period	Duration	Memory	Domain
1	1000	1	1	All modules
2	1000	64	4	All modules
3	1000	80	1	All modules
4	1000	42	6	All modules
5	1000	37	3	All modules
6	1000	49	8	All modules
7	1000	16	2	All modules
8	500	8	6	All modules
9	500	23	1	All modules
10	500	10	7	All modules
11	500	37	7	All modules
12	500	29	7	All modules
13	500	35	6	All modules
14	500	18	5	All modules
15	500	34	8	All modules
16	500	15	1	All modules
17	500	11	2	All modules
18	500	30	3	All modules
19	500	22	3	All modules
20	500	30	3	All modules
21	500	31	1	All modules
22	200	2	3	All modules
23	200	9	5	All modules
24	200	21	3	All modules
25	200	21	5	All modules
26	200	13	8	All modules
27	200	19	9	All modules
28	200	18	8	All modules
29	200	12	3	All modules
30	200	22	7	All modules
31	200	8	9	All modules
32	100	2	8	All modules
33	100	5	8	All modules
34	100	2	3	All modules
35	100	14	1	All modules
36	100	13	3	All modules
37	100	9	8	All modules
38	100	14	2	All modules
39	100	10	2	All modules
40	100	2	7	All modules

(a) Partition information.

Module	Memory
1	61
2	64
3	59
4	64
5	65
6	60
7	62
8	60

(b) Module information.

Chain	Delay
$p_{21} \rightarrow p_{26}$	252
$p_{23} \rightarrow p_{7}$	215
$p_{37} \rightarrow p_{18}$	424
$p_{1} \rightarrow p_{30}$	166
$p_{10} \rightarrow p_{25}$	50
$p_{27} \rightarrow p_{6}$	706
$p_{28} \rightarrow p_{34}$	119
$p_{30} \rightarrow p_{20}$	259
$p_{17} \rightarrow p_{14}$	321
$p_{21} \rightarrow p_{38}$	109
$p_{8} \rightarrow p_{18}$	309
$p_{5} \rightarrow p_{30}$	204
$p_{18} \rightarrow p_{23}$	205
$p_{8} \rightarrow p_{28}$	157
$p_{32} \rightarrow p_{24}$	243

(c) Chain constraints.

m, n	1	2	3	4	5	6	7	8
1	-	9	25	16	10	11	10	7
2	9	-	23	11	18	18	7	15
3	25	23	-	15	11	15	20	11
4	16	11	15	-	5	6	17	7
5	10	18	11	5	-	5	7	23
6	11	18	15	6	5	-	25	17
7	10	7	20	17	7	25	-	21
8	7	15	11	7	23	17	21	-

(d) Network delays.

Other constraints: $f_{31} \neq f_{29}$, $f_{34} \neq f_{9}$, $f_{15} \neq f_{25}$, $f_{11} \neq f_{40}$, $f_{38} \neq f_{29}$, $f_{35} \neq f_{10}$, $f_{32} \neq f_{9}$, $f_{4} \neq f_{25}$, $f_{12} \neq f_{30}$, $f_{16} \neq f_{25}$, $f_{28} = f_{29}$, $f_{2} = f_{8}$, $f_{4} = f_{15}$, $f_{31} = f_{38}$.

B.5 $20M100P$

Table B.5: Problem specification for $20M100P$.

Partition	Period	Duration	Memory	Domain
1	1000	55	2	All modules
2	1000	78	1	All modules
3	1000	28	4	All modules
4	1000	2	6	All modules
5	1000	75	6	All modules
6	1000	53	7	All modules
7	1000	47	7	All modules
8	1000	32	2	All modules
9	1000	46	1	All modules
10	1000	42	1	All modules
11	1000	68	6	All modules
12	1000	48	7	All modules
13	1000	79	2	All modules
14	1000	45	8	All modules
15	1000	74	9	All modules
16	1000	12	7	All modules
17	1000	46	7	All modules
18	1000	80	5	All modules
19	1000	24	6	All modules
20	1000	38	1	All modules
21	1000	45	8	All modules
22	1000	78	9	All modules
23	1000	21	8	All modules
24	500	24	1	All modules
25	500	23	9	All modules
26	500	37	2	All modules
27	500	15	4	All modules
28	500	40	4	All modules
29	500	40	8	All modules
30	500	11	1	All modules
31	500	13	8	All modules
32	500	7	2	All modules
33	500	17	7	All modules
34	500	9	3	All modules
35	500	24	6	All modules
36	500	40	5	All modules
37	500	4	6	All modules
38	500	24	4	All modules
39	500	34	7	All modules
40	500	11	6	All modules

(a) Partition information. (1/3)

Module	Memory
1	62
2	63
3	60
4	59
5	65
6	60
7	63
8	65
9	62
10	62
11	62
12	65
13	63
14	61
15	62
16	63
17	60
18	65
19	59
20	59

(b) Module information.

Partition	Period	Duration	Memory	Domain
41	500	5	4	All modules
42	500	31	1	All modules
43	500	16	1	All modules
44	500	17	2	All modules
45	500	6	3	All modules
46	500	30	2	All modules
47	500	23	6	All modules
48	500	3	6	All modules
49	200	11	4	All modules
50	200	9	4	All modules
51	200	19	2	All modules
52	200	12	2	All modules
53	200	16	7	All modules
54	200	15	2	All modules
55	200	15	7	All modules
56	200	13	4	All modules
57	200	16	9	All modules
58	200	21	1	All modules
59	200	17	2	All modules
60	200	9	2	All modules
61	200	18	6	All modules
62	200	17	9	All modules
63	200	17	6	All modules
64	200	12	3	All modules
65	200	16	5	All modules
66	200	12	9	All modules
67	200	23	9	All modules
68	200	3	6	All modules
69	200	4	2	All modules
70	200	23	9	All modules
71	200	20	4	All modules
72	200	18	4	All modules
73	100	8	5	All modules
74	100	4	4	All modules
75	100	12	4	All modules
76	100	8	2	All modules
77	100	1	4	All modules
78	100	1	6	All modules
79	100	3	6	All modules
80	100	8	2	All modules
81	100	15	3	All modules
82	100	1	9	All modules
83	100	3	4	All modules
84	100	10	9	All modules
85	100	10	2	All modules
86	100	2	2	All modules
87	100	5	5	All modules
88	100	14	4	All modules
89	100	10	8	All modules
90	100	7	3	All modules

(c) Partition information. (2/3)

Chain	Delay
$p_{85} \rightarrow p_{86}$	120
$p_{39} \rightarrow p_{54}$	143
$p_{41} \rightarrow p_{1}$	684
$p_{12} \rightarrow p_{50}$	217
$p_{78} \rightarrow p_{100}$	118
$p_{70} \rightarrow p_{68}$	56
$p_{83} \rightarrow p_{24}$	111
$p_{94} \rightarrow p_{79}$	96
$p_{6} \rightarrow p_{60}$	141
$p_{36} \rightarrow p_{56}$	87
$p_{3} \rightarrow p_{89}$	77
$p_{57} \rightarrow p_{67}$	130
$p_{95} \rightarrow p_{43}$	324
$p_{9} \rightarrow p_{64}$	242
$p_{73} \rightarrow p_{36}$	481
$p_{8} \rightarrow p_{33}$	285
$p_{21} \rightarrow p_{43}$	264
$p_{75} \rightarrow p_{25}$	384
$p_{29} \rightarrow p_{33}$	423
$p_{5} \rightarrow p_{30}$	260
$p_{53} \rightarrow p_{8}$	788
$p_{5} \rightarrow p_{44}$	215
$p_{4} \rightarrow p_{73}$	52
$p_{19} \rightarrow p_{96}$	100
$p_{38} \rightarrow p_{13}$	640
$p_{46} \rightarrow p_{19}$	528
$p_{67} \rightarrow p_{70}$	180
$p_{82} \rightarrow p_{33}$	460
$p_{56} \rightarrow p_{74}$	84
$p_{82} \rightarrow p_{99}$	98
$p_{1} \rightarrow p_{42}$	425
$p_{69} \rightarrow p_{99}$	118
$p_{60} \rightarrow p_{33}$	500
$p_{99} \rightarrow p_{86}$	61
$p_{30} \rightarrow p_{33}$	165
$p_{73} \rightarrow p_{70}$	229
$p_{6} \rightarrow p_{96}$	77
$p_{54} \rightarrow p_{1}$	642
$p_{68} \rightarrow p_{60}$	138
$p_{2} \rightarrow p_{21}$	704

(d) Chain constraints.

Partition	Period	Duration	Memory	Domain
91	100	15	5	All modules
92	100	1	6	All modules
93	100	13	4	All modules
94	100	7	5	All modules
95	100	14	1	All modules
96	100	7	9	All modules
97	100	3	8	All modules
98	100	1	2	All modules
99	100	6	6	All modules
100	100	2	3	All modules

(e) Partition information. (3/3)

m, n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	-	17	11	12	12	8	25	16	19	11	21	17	23	12	5	17	12	18	6	8
2	17	-	6	11	25	18	13	24	11	18	24	6	22	11	6	7	23	11	5	17
3	11	6	-	20	15	5	5	8	22	17	24	20	5	17	25	16	6	13	13	20
4	12	11	20	-	14	16	24	6	21	22	20	7	16	13	11	19	11	23	18	25
5	12	25	15	14	-	20	23	6	11	20	14	21	9	7	25	15	6	24	20	15
6	8	18	5	16	20	-	15	24	8	13	8	17	6	6	21	24	6	9	17	25
7	25	13	5	24	23	15	-	16	5	9	12	23	23	18	9	13	9	19	20	12
8	16	24	8	6	6	24	16	-	20	23	18	25	22	21	10	7	7	10	10	13
9	19	11	22	21	11	8	5	20	-	5	19	15	5	18	23	25	8	15	14	9
10	11	18	17	22	20	13	9	23	5	-	21	5	21	20	19	20	12	22	19	21
11	21	24	24	20	14	8	12	18	19	21	-	14	24	11	16	19	19	18	21	19
12	17	6	20	7	21	17	23	25	15	5	14	-	17	17	19	18	18	13	23	23
13	23	22	5	16	9	6	23	22	5	21	24	17	-	21	14	15	9	22	20	22
14	12	11	17	13	7	6	18	21	18	20	11	17	21	-	8	19	16	13	24	6
15	5	6	25	11	25	21	9	10	23	19	16	19	14	8	-	5	7	9	23	15
16	17	7	16	19	15	24	13	7	25	20	19	18	15	19	5	-	7	10	15	25
17	12	23	6	11	6	6	9	7	8	12	19	18	9	16	7	7	-	12	12	15
18	18	11	13	23	24	9	19	10	15	22	18	13	22	13	9	10	12	-	9	16
19	6	5	13	18	20	17	20	10	14	19	21	23	20	24	23	15	12	9	-	9
20	8	17	20	25	15	25	12	13	9	21	19	23	22	6	15	25	15	16	9	-

(f) Network delays.

Other constraints: $f_{10} \neq f_5$, $f_{39} \neq f_{53}$, $f_{49} \neq f_{40}$, $f_{74} \neq f_4$, $f_{92} \neq f_{31}$, $f_{40} \neq f_{12}$, $f_{46} \neq f_{19}$, $f_{35} \neq f_{23}$, $f_{95} \neq f_6$, $f_{32} \neq f_{45}$, $f_{59} \neq f_{76}$, $f_{51} \neq f_{37}$, $f_{87} \neq f_9$, $f_{41} \neq f_{61}$, $f_{58} \neq f_8$, $f_{24} \neq f_{56}$, $f_{22} \neq f_{11}$, $f_9 \neq f_{25}$, $f_{17} \neq f_{68}$, $f_{34} = f_{82}$, $f_{65} = f_{77}$, $f_{86} = f_{90}$, $f_{29} = f_{97}$, $f_{34} = f_{54}$, $f_{54} = f_{82}$, $f_{33} = f_{96}$.

B.6 $3M15P$-S

Table B.6: Problem specification for $3M15P$-S.

Partition	Period	Duration	Memory	Domain	Preemption points	Deadline
1	1000	100	5	All modules	{40, 80}	200
2	1000	50	3	All modules	{}	–
3	1000	100	3	All modules	{25, 40, 50, 60, 75}	300
4	1000	70	3	All modules	{20, 40}	220
5	500	35	4	All modules	{}	–
6	500	25	4	All modules	{}	–
7	500	50	2	All modules	{}	–
8	250	50	5	All modules	{20, 25, 30, 35}	175
9	250	20	6	All modules	{}	–
10	200	30	7	All modules	{}	–
11	100	10	4	All modules	{}	–
12	100	20	2	All modules	{}	–
13	100	5	5	All modules	{}	–
14	100	10	5	All modules	{}	–
15	50	2	6	All modules	{}	–

(a) Partition information.

Module	Memory
1	33
2	35
3	30

(b) Module information.

Link	Delay
$p_6 \rightarrow p_{15}$	48
$p_{10} \rightarrow p_5$	125
$p_{11} \rightarrow p_{12}$	119

(c) Link constraints.

Modules	1	2	3
1	-	12	7
2	12	-	14
3	7	14	-

(d) Network delays.

Other constraints: $f_8 \neq f_1$, $f_9 \neq f_5$, $f_{10} \neq f_8$, $f_1 = f_{12}$, $\varepsilon = 1$.

Partition Scheduling in Distributed Integrated Modular Avionics

João Miguel Fonseca Gonçalves