Processor and memory co-scheduling of embedded real-time applications on multicore platforms

Ikram Senoussaoui

UNIVERSITY OF LILLE
UNIVERSITY OF ORAN1

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

Discipline: Computer Science

Ikram SENOUSSAOUI

Processor and memory co-scheduling of embedded real-time applications on multicore platforms

Co-ordonnancement processeur et mémoire des applications temps-réel sur les plateformes multicœurs

Under the direction of: Pr. LIPARI Giuseppe
Pr. BENHAOUA Mohammed Kamel

December 14, 2023

Maryline CHETTO	Full Professor, LS2N, University of Nantes, France	Referee
Hadda CHERROUN	Full Professor, LIM, University of Laghouat, Algeria	Referee
Sidi Mohammed BENSLIMANE	Full Professor, LabRI, ESI-SBA, Algeria	Examiner
Emmanuel GROLLEAU	Full Professor, LIAS, ISAE-ENSMA Poitiers, France	Examiner
Bilel DERBEL	Full Professor, CRIStAL, University of Lille, France	President
Houssam-Eddine ZAHAF	Associate Professor, LS2N, University of Nantes, France	Supervisor
Mohammed Kamel BENHAOUA	Full Professor, LAPECI, University of Mascara, Algeria	Director
Giuseppe LIPARI	Full Professor, CRIStAL, University of Lille, France	Director

Résumé

La demande en puissance de calcul dans les systèmes embarqués temps-réel a considérablement augmenté ces dernières années. Les plateformes multicœurs qui sont généralement équipés d'un sous-système de mémoire partagé par tous les cœurs ont répondu dans une certaine mesure à ce besoin croissant en capacité de calcul. Cependant, dans les systèmes temps-réel, l'utilisation simultanée du sous-système de mémoire peut induire à des interférences mémoire significatives. Ces dernières peuvent rendre les pires temps d'exécution des tâches (WCET) très pessimistes et conduire à une sous-utilisation du système.

Cette thèse se concentre sur la réduction des interférences résultantes des conflits liés aux ressources partagées (par exemple les mémoires cache, les bus de communication et la mémoire principale) dans les systèmes multicœurs grâce au co-ordonnancement des calculs et des transferts de donnée des applications temps-réel. À cette fin, nous utilisons des modèles de tâches existants tels que le modèle *DFPP (Deferred Fixed Preemption Point)*, le modèle *PREM (PRedictable-Exécution-Model)* et le modèle *AER (Acquisition-Execution-Restitution model)*. Nous proposons un nouveau modèle de tâche réaliste et plusieurs algorithmes de co-ordonnancement et de partitionnement des tâches temps-réel. Nous montrons que de tels ordonnanceurs peuvent améliorer jusqu'à 50% le taux d'ordonnançabilité par rapport aux ordonnanceurs équivalents générés avec les méthodes de l'état de l'art. De plus, nous démontrons expérimentalement l'applicabilité de nos méthodologies sur la famille de processeurs multicœurs Infineon AURIX TC-397 en utilisant différents benchmarks.

Abstract

The demand for computational power in real-time embedded systems has increased significantly in recent years. Multicore platforms which are generally equipped with a single memory subsystem shared by all cores, have satisfied this increasing need for computation capability to some extent. However, in real-time systems, simultaneous use of the memory subsystem may result in significant memory interference. Such memory interference owing to resource contention may lead to very pessimistic worst-case execution time bounds (WCETs) and lead to under-utilization of the system.

This thesis focuses on reducing interference resulting from shared resource contention (e.g., caches, buses and main memory) on multicore systems through processor and memory co-scheduling for real-time applications. To this end, we use existing task models such as *DFPP (Deferred Fixed Preemption Point)* model, *PREM (Predictable-Execution-Model)* and *AER (Acquisition-Execution-Restitution)* model. We also propose a new realistic task model and several algorithms for task set allocation and for processor and memory co-scheduling. We show that our proposed methodologies can improve schedulability by up to 50% compared to equivalent schedules generated with the state-of-the-art methods. Furthermore, we experimentally demonstrate the applicability of our methodologies on the Infineon AURIX TC-397 multicore family of processors using different benchmarks.

Acknowledgements

I have always enjoyed reading the acknowledgments for dissertation manuscripts. This time, it's my turn to indulge.

Four years might seem like a short time in the grand scheme of things, but it certainly doesn't feel like it when your life is constantly changing, for worse or for better, and they definitely felt way longer to me during this experience as a Ph.D. From moving to another country to being stuck at home during the pandemic, there have been many ups and downs in this journey, so much so that the version of me a few years back feels almost like a stranger now. Despite all the struggles and hardships encountered along the way, I honestly think this has been a positive experience for me, with many teachable moments and growing ventures.

I would like to express my first thanks to my supervisors, who gave me this opportunity and mentored me through-out this experience. A big thank to Prof. Mohammed Kamel BENHAOUA for his help, advices, support and above all his trust and understanding. I express my gratitude to Prof. Giuseppe LIPARI, he has been a seemingly inexhaustible source of wisdom in all the subjects we worked on. I appreciated his scientific rigor and his human qualities. His immense knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life. Special thanks to Dr. Houssam-eddine ZAHAF who participated in the direction of the thesis and had a significant impact on the choices made during these years. He has been always available for suggestions and discussions. Thank you for being a friend and a big brother to me. Many thanks to (Dr, Senior Scientist) Richard OLEJNIK, who directed me during my first year of the thesis in France.

I would thank also the members of my jury. Firstly, Prof. Maryline CHETTO and Prof. Hadda CHERROUN for accepting and putting time even in their very full agenda, to read and evaluate my work. I thank also Prof. Sidi Mohammed BENSLIMANE, Prof. Emmanuel GROLLEAU and Prof. Bilel DERBEL to be members of my jury and accept to examine the research presented in this dissertation.

A big thank goes to everyone who worked at the LAPECI and CRIStAL laboratories during these years, especially to the members of the SYCOMORES team (Lille), who became like a second family to me, and to the ORTESE team

(Oran1). Each contributed to making this experience more enjoyable, sharing the hardships and the joys of this work. We are a relatively small group, but there would be too many people to name all of them here. Still, some people deserve special thanks for how they helped me get through these past few years. People like Julien FORJET, his reviewing was precious. I appreciated his support and encouragement. Also, I would like to express my gratitude to all Ph.D students, former Ph.D students and post-docs I met during these years.

I also want to thank all of my friends outside the workplace for all the fun we had together throughout these years.

First and foremost, I would like to thank and express my gratitude to my dear parents, my brother Yacine and my sisters Manel and Bouchra, for their tireless support during these four years of my Ph.D and my whole life, and for the unconditional love they have shown me, which prevented me from giving up.

Contents

Abstract ii

Introduction 2

I Motivation, Background and Related work 5

1 Multiprocessors and Parallel Systems 6
- 1.1 Introduction . . . 7
- 1.2 Uniprocessor vs multiprocessor systems . . . 7
- 1.3 Classification of multiprocessor systems . . . 8
  - 1.3.1 Based on their structure . . . 8
  - 1.3.2 Based on their architecture and microarchitecture . . . 8
  - 1.3.3 Based on their memory architecture . . . 9
- 1.4 Scratchpad vs cache memories in multicore platforms . . . 10
- 1.5 Programming parallel architecture . . . 11
  - 1.5.1 POSIX Threads . . . 11
  - 1.5.2 OpenMP framework . . . 12
- 1.6 Conclusion . . . 13

2 Real-time Systems 14
- 2.1 Introduction . . . 15
- 2.2 Task models . . . 15
  - 2.2.1 Liu and Layland model . . . 16
  - 2.2.2 PREM and AER models . . . 17
  - 2.2.3 Directed Acyclic Graph (DAG) . . . 18
- 2.3 Priority assignment for scheduling . . . 19
  - 2.3.1 Scheduling characteristics . . . 19
- 2.4 Time- vs event-triggered real-time scheduling . . . 20
- 2.5 Preemptive vs non-preemptive scheduling . . . 21
- 2.6 Earliest Deadline First (EDF) . . . 21
  - 2.6.1 Preemptive real-time scheduling . . . 21
  - 2.6.2 Non-preemptive real-time scheduling . . . 23
- 2.7 Preemptive multiprocessor real-time scheduling . . . 24
- 2.8 Conclusion . . . 25

3 Processor and Memory Co-scheduling in Multicore Systems 26
3.1 Introduction . . . 27
3.2 Cache related delays in multicore systems . . . 27
3.3 Shared resources contention: main memory and buses . . . 29
3.4 Related work . . . 30
3.5 Conclusion . . . 34

II Contributions 35

4 Allocation of Real-time Tasks Onto Identical Core Platforms Under Deferred Fixed Preemption-point Model 36
4.1 Introduction . . . 37
4.2 System model . . . 37
4.2.1 Task model . . . 37
4.2.2 Architecture model . . . 38
4.3 Limited Preemption analysis for single-processor . . . 39
4.3.1 Maximum non-preemptive execution-time . . . 39
4.3.2 Selection of effective preemption points . . . 40
4.4 Task allocation . . . 40
4.4.1 Enumerating algorithm . . . 41
4.4.2 Branch and Bound . . . 42
4.4.3 Computational complexity . . . 46
4.4.4 Allocation heuristics . . . 47
4.5 Results and discussions . . . 48
4.5.1 Task generation . . . 48
4.5.2 Simulation results and discussions . . . 48
4.6 Conclusion . . . 52

5 Contention-free Scheduling of PREM Tasks on Partitioned Multicore Platforms 53
5.1 Introduction . . . 54
5.2 System model . . . 54
5.2.1 Architecture model . . . 54
5.2.2 Task model . . . 55
5.3 Offset based processor and memory co-scheduling . . . 55
5.3.1 Task-level offsets : sufficient condition . . . 56
5.3.2 Integer-Linear-Programming based offset assignment . . . 57
5.4 Deadline based processor and memory co-scheduling . . . 60
5.5 Results and discussions . . . 61
5.5.1 Task set generation . . . 62
5.5.2 Results of synthetic task set experiments . . . 62
5.6 Conclusion . . . 64

6 Memory-processor Co-scheduling AECR-DAG Real-time Tasks on Partitioned Multicore Platforms with Scratchpads 65
6.1 Introduction . . . 66
6.2 System model . . . 66
6.2.1 Architecture model . . . 66
6.2.2 Task model . . . 66
6.3 DAG tasks allocation and transformation . . . 69
6.3.1 Decision variables and objective function . . . 71
6.4 Deadline based DAG memory-processor co-scheduling . . . 73
6.4.1 Fair and proportional deadline assignment . . . 74
6.4.2 GA-based intermediate deadline assignment . . . 75
6.4.3 Evaluation Strategy . . . 79
6.4.4 Creating the next generation . . . 84
6.5 Results and discussions . . . 85
6.5.1 Task generation . . . 86
6.5.2 Simulation results and discussions . . . 87
6.6 Conclusion . . . 92

7 Conclusion and perspectives 93
7.1 Conclusion . . . 94
7.2 Limitations and perspectives . . . 94

List of Figures

1.1 Shared memory model. 9
1.2 Distributed memory model. 9
1.3 Hybrid memory model. 10
1.4 Example of Fork-Join Model 13

2.1 Periodic constrained-deadline task parameters. 17
2.2 DAG task example 18
2.3 Partitioned scheduling vs global scheduling. 25

4.1 Example of task parameters 38
4.2 Example of branch and bound 46
4.3 Schedulability for optimal solutions against BF, WF and FF . . . 49
4.4 The analysis time as a function of number of task 50
4.5 Schedulability at different taskset size. 50
4.6 Performance of BF, WF and FF using sorted tasks by density . . 51
4.7 Performance of BF, WF and FF using sorted tasks by laxity . . . 51

5.1 Multicore target platform. 54
5.2 Example of task parameters. 56
5.3 Schedulability of ILP vs heuristics approaches 62
5.4 Heuristics algorithms performances 63

6.1 Multicore architecture featuring 4 cores, and its interconnection buses. 67
6.2 DAG task example, computation subtasks are mapped on a dual-core platform. 68
6.3 DAG task transformation. 71
6.4 Example of offset and local deadline. 74
6.5 Example of an individual. 77
6.6 Schedulability rate for large DAG: $\mathsf{U}^{\mathsf{max}} = 0.7$ vs $\mathsf{U}^{\mathsf{max}} = 0.85$ . . . 88
6.7 Schedulability rate for long DAG: $\mathsf{U}^{\mathsf{max}} = 0.7$ vs $\mathsf{U}^{\mathsf{max}} = 0.85$ . . . 88
6.8 Schedulability rate for large DAGs: small vs large population . . . 89
6.9 Schedulability rate for long DAGs: small vs large population . . . 89
6.10 GA-ILP's performances VS (GA-WF and GA-BF) algorithms . . 90
6.11 GA-ILP's performances VS (FAIR and PROP) algorithms 91
6.12 GA-ILP's performances VS the literature 92

List of Tables

4.1 Task parameters 46

6.1 Example of Individual 78
6.2 Swapping local deadlines after a crossover point 85

List of Acronyms

AD Autonomous driving. 15

AECR Acquisition Execution Communication Restitution Model. 4

AECR-DAG Acquisition Execution Communication Restitution Model-DAG. 24, 69, 79, 87, 92

AER Acquisition Execution Restitution Model. 3, 17, 29, 34, 66

BB Branch and Bound. 40

BF Best-Fit. 25, 47–49

COTS Commercial off-the-shelf software. 29

CRPD cache-related preemption delays. 3, 28, 31

DAG Directed Acyclic Graph. 4, 17, 33, 66

dbf Demand bound function. 22

DFPP Deferred Fixed Preemption Point. 3

DLA Deep learning accelerators. 9

DM Deadline Monotonic. 19

DPM deferred preemption model. 30

DSP Digital signal processors. 9

EDF Earliest Deadline First. 19, 21, 22, 39, 40, 54, 59, 60, 64, 73, 82, 84

ET Event-triggered. 20

FCFS First-Come First-Served. 33

FF First-Fit. 47–49

FIFO First-In-First-Out. 29

FPP Fixed Preemption Points. 28

LLF Least Laxity First. 19

LPM Limited Preemptive Models. 28

NFP Floating Non-Preemptive Regions. 28

PREM PRedictable Execution Model. 3, 17, 29, 32, 54, 64

PT Preemption Thresholds. 28

PTS Preemption Thresholds Scheduling. 31

RM Rate Monotonic. 19

SMP Homogeneous or Symmetric Multiprocessors. 8

SPM Scratchpad memory. 11

TDMA Time Division Multiple Access. 32

TT Time-triggered. 19, 20

WCET Worst-case execution time estimation. 2, 10, 29

WF Worst-Fit. 25, 47, 48

Introduction

Context and motivation

Real-time systems are computing systems that must react with predictable time behavior to events in the environment. As a consequence, their correct behavior depends not only on the logical correctness of the computations but also on the time at which the results are produced [132]. Nowadays, real-time computing plays a crucial role in our society, as more and more complex systems rely, in part or completely, on computer control. Automotive applications, robotics, medical systems and flight control systems are examples of applications that require real-time computing.

Worst-case execution time estimation (WCET) is the key element of timing analysis of real-time systems. The WCET is defined as the maximum length of time that a task may take to completely execute on a given hardware platform. For single-core platforms, the WCET estimate is based on the time spent by the longest execution path of the process.

The demand for computational power in real-time embedded systems has increased significantly, making multicore and heterogeneous systems attractive in the real-time domain. Multicore platforms can provide significantly more processing power than traditional single-core platforms, however, due to the underlying complexity and interference in accessing shared resources, schedulability and predictability should not be guaranteed.

In multicore settings, where different applications running concurrently on different cores compete for the access to shared memories, the combination of interference for accessing a shared memory (i.e. the main memory) and local cache related delays can be large and highly variable depending on the platform architecture and the number of parallel access requests. These runtime-related delays inflate significantly the WCETs of tasks especially when preemption is enabled.

Shared memory interference represents a big challenge for the predictability of real-time embedded systems. In general, it is difficult to accurately compute the worst-case interference profile, which likely leads to include scenarios that might never occur, therefore over-estimating the worst-case interference.

In cache-based preemptive real-time systems, the preempted and the preempting tasks may interfere on the cache memory. This interference leads to cache misses in the preempted task which result in additional delays [64]. In fact, when a preemption occurs, the analysis must account for the time needed to reload cache blocks that have been evicted by preempting tasks. These delays

are known as *cache-related preemption delays (CRPD)*. CRPD may be large in fully preemptive systems as a given task can be preempted by any other higher priority task. Non-preemptive systems, on the other hand, may suffer from long blocking times: when a low priority task starts its execution, higher priority tasks are blocked waiting for the low priority task to finish, causing priority inversion.

We believe that reducing the impact of memory interference on WCET estimation can improve application performance in multi-core architectures. Models such as the *Deferred Fixed Preemption Point (DFPP)*, the *Predictable Execution Model (PREM)* and the *Acquisition-Execution-Restitution (AER)* model can help us in improving predictability in these systems.

In this thesis, we tackle the problem of reducing/avoinding shared memory contention for real-time task sets expressed with different predictable models on several types of multicore architectures (i.e. cache-based architectures and scratchpad-based architectures). The next section briefly describes our contributions and the organization of the subsequent chapters of this document.

Contributions

Our contributions consist in proposing efficient scheduling approaches for different predictable task models on multicore architectures. The outline of this manuscript and the main contributions are summarized in this section.

The thesis is divided into seven chapters.

In Chapter 1 & 2 we introduce real-time systems, multicore and parallel systems. In Chapter 3, we describe the memory interference problem in multicore architectures and we place the problem in the state of the art.

In Chapter 4, we explore techniques to allocate a set of real-time task onto an identical multicore platform with caches so to reduce the preemption costs. To this end, we present optimal algorithms and heuristics to solve this problem. This chapter present i) an exact allocation algorithm for a set of real-time tasks presenting fixed-preemption points onto an identical core platform, (ii) several techniques to reduce the time and space complexity of computing the exact solution as well as (iii) efficient allocation heuristics; and finally (iv) an exhaustive evaluation of the proposed approaches using a large set of synthetic experiments. In the model we use in this chapter, each fixed-preemption point is characterized by an architecture related cost, it is defined as the cumulative execution overhead due to the combination of processor components effects such as the cache-related effect, which results in very high costs due to the unpredictability of caches. We adopt in Chapter 5 the Predictable Execution Model (PREM) in order to achieve a higher degree of predictability.

In Chapter 5, we tackle the problem of contention in multicore architecture with scratchpad memories. The goal is to avoid shared memory contention for a set of tasks modeled using PREM. In this chapter, we explore and compare different designs for scheduling memory phases: a time-triggered-based approach and an on-line scheduling approach for memory phases. We show that obtaining an optimal solution of this problem is very time consuming even for small task set size. We propose heuristics that allows to find a feasible solution in a reasonable

time. We compare the proposed approaches against the state of the art using a set of synthetic experiments in terms of schedulability and analysis time and we show that our techniques improve up to 50% the schedulability. The different approaches are implemented on an Infineon AURIX TC397 multicore microcontroller and validated using a set of tasks extracted from well-known benchmarks from the literature.

In the sixth chapter, we proposed a new model named *Acquisition-Execution-Communication-Restitution-DAG (AECR)*. An AECR-DAG application is a *Directed Acyclic Graph (DAG)* of communicating subtasks respecting AER model. We use a genetic algorithm to derive scheduling parameters for a set of AECR-DAG tasks. Subtasks are partitioned onto the multicore platform while their memory requests and relative communications are scheduled onto the shared buses, in order to prevent interference and ensure predictability. We propose an ILP formulation to solve the mapping problem. Specifically, all subtasks and communications are assigned appropriate intermediate offsets and deadlines to guarantee that they comply with the system's timing constraints.

In all the proposed approaches of the two precedent chapters (5 & 6), we consider a multicore platform with scratchpads and preemptive partitioned scheduling at the core level for computations phases.

We conclude with an overview of our findings, their impact, and future directions in the thesis conclusion (Chapter 7).

Part I

Motivation, Background and Related work

Chapter 1

Multiprocessors and Parallel Systems

Contents

1.1 Introduction 7
1.2 Uniprocessor vs multiprocessor systems 7
1.3 Classification of multiprocessor systems 8
1.3.1 Based on their structure 8
1.3.2 Based on their architecture and microarchitecture . . . 8
1.3.3 Based on their memory architecture 9
1.4 Scratchpad vs cache memories in multicore platforms 10
1.5 Programming parallel architecture 11
1.5.1 POSIX Threads 11
1.5.2 OpenMP framework 12
1.6 Conclusion 13

1.1 Introduction

The concept of algorithms and modern computers is defined by Alan Turing [138] who presented the Turing Machine. He proved that mathematical computations can be performed by machines if they are represented by an algorithm. Turing machines are considered as the central concept of modern computers. Modern computer systems are built on ICs (*Integrated Circuits* (1958-1971)). Depending on the number of available processing units, modern computer systems can be classified as uniprocessor and multiprocessor systems.

1.2 Uniprocessor vs multiprocessor systems

A uniprocessor platform is a computer system with a single Central Processing Unit (CPU). This unit performs all computations sequentially, with just one task executing at any given time instance. As a result of the hardware constraint, no true (physical) parallelism is accomplished on such systems; yet, multitasking can be supported in such systems. Multitasking means that more than one task can be executed on the processor on a time-sharing basis.

In 1965, Moore's observation [137] states that the number of transistors in an IC doubles each two years, hence the performance of chips doubles as well. We can see that the evolution of Intel processors roughly parallels Moore's law. The number of transistors incorporated in their CPUs has increased dramatically over the years. According to Moore's law, processor performance advances exponentially over time as the number of transistors in integrated circuits doubles, increasing clock speed and allowing computers to accomplish tasks quicker. Since clock rates of processors are not going faster in the same previous rate, platform performance is improved by duplicating the number of CPUs, therefore, the multicore/multiprocessor solution was presented.

Multiprocessor systems are those that include more than one processing unit, allowing computations to be done in parallel. Until recently, uniprocessor systems predominated over multiprocessor systems in industrial execution platforms. It's either because system designers were afraid of the risk that came with modifying their previous reliable uniprocessor designs, or because uniprocessor performance was sufficient for most embedded system applications.

However, as the demand for processing power and effectiveness for massive computations and applications grows, a move from uniprocessor to multiprocessor systems is on the horizon. For example, in 2011, the percentage of uniprocessor applications was greater than 30% of total industrial applications. However, this ratio is likely to fall to 15% in the next few years as multiprocessor platforms gain popularity [34].

Multiprocessor systems offer software-level parallelism, hence, concurrent tasks and processes can execute simultaneously on different processing units.

1.3 Classification of multiprocessor systems

Multiprocessor are differentiated and classified according to different criteria. In this section, we present the most important classes.

1.3.1 Based on their structure

Multiprocessing systems can be classified based on their structure as follows:

- Multiprocessor system. As stated before, it consists of more than one processing unit. Applications can choose to benefit from the performance improvement provided by this architecture. Otherwise said, sequential applications can opt to execute sequentially on such systems without modifying their programming techniques. Parallelism can be achieved by splitting processes/threads of an application among several processors in the system, thereby increasing execution speed. Processors can also share system memory through a communication bus (see the classification according the memory architecture 1.3.3).

- Multicore system. A system's core is made up of more than one logical unit. These units (often 2 to 8 cores) have their own individual memory cache or scratchpad as well as another level of cache memory shared by the processor's cores (LLC for Last Level memory Cache). Application programming distinguishes multiprocessor from multicore systems. An application has to change its sequential architecture in order to scale up its performance while executing on multicore systems. The abovementioned systems are usually found in general-purpose devices. Examples are Intel, ARM and AMD processors;

- Many-core system. This system, which typically refers to extremely large multicore platforms, has the same technical architecture as multicore systems. The number of cores ranges from dozens to hundreds per single processor. These systems implement parallel architecture, hence, software has to be adapted to such systems in order to get advantage of hardware capabilities. For example, the TILE-GX 3 72-core processor from Tilera (2009).

1.3.2 Based on their architecture and microarchitecture

Based on the type of processors, multicores (multiprocessor) systems can be divided into categories as follows:

- Homogeneous or Symmetric Multiprocessors (SMP). The system's processors are all of the same type (multiple central processing units CPUs or multiple graphics processing units (GPUs)). They have typically identical architectures, instruction sets, and the execution rate of tasks is the same on all of them;

- Uniform Processors. In these systems, each processor has a speed or computational capacity that determines the rate at which a task is executed;
- Heterogeneous Processors. They incorporate multiple processing units of different types, each optimized for specific tasks in a single system. These systems can include a mix of CPUs, GPUs, deep learning accelerators (DLAs), digital signal processors (DSPs), and other specialized processors. Hence, the execution rate of a task is determined by the type of processor and the task itself.

1.3.3 Based on their memory architecture

According to memory architecture, multiprocessor systems can be divided into categories as follows:

- Shared memory. A multicore (multiprocessor) with shared memory provides a single memory that is shared by all processors. Any processor can physically access data at any location in memory (see Figure 1.1). In all the contributions of this work, we used a shared memeory multicore architecture.

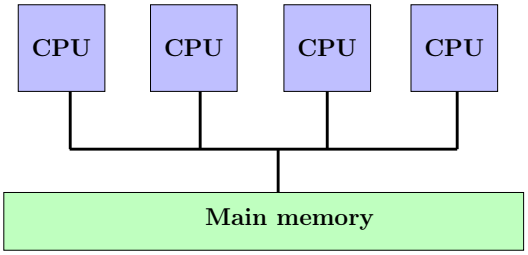


Figure 1.1: Shared memory model.

- Distributed memory. In a multicore (multiprocessor) with distributed memory, each processor has its own private memory. Processings can only operate on local data and must communicate with one or more remote processors, if remote data is needed (see Figure 1.2).

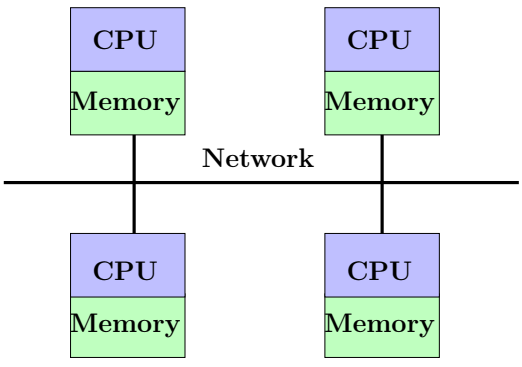


Figure 1.2: Distributed memory model.

- Hybrid memory. In a multiprocessor with hybrid memory, the distributed and shared memory models are combined. Each unit is a shared memory system (there are two or more cores sharing the same memory resources). Then, shared memories are connected through networks, forming a distributive system. Today, the fastest computers in the world employ both shared and distributed memory architectures (see Figure 1.3).

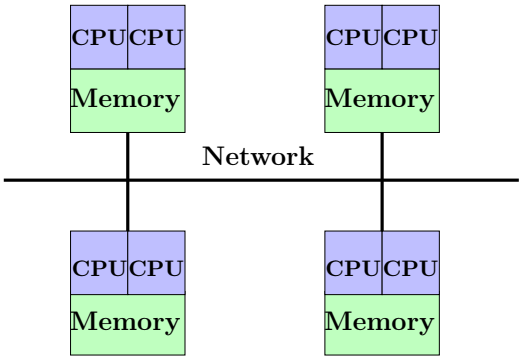


Figure 1.3: Hybrid memory model.

1.4 Scratchpad vs cache memories in multicore platforms

The increasing performance gap between the processor and the off-chip memory has made it important to use on-chip memory in real-time embedded systems. CPU caches in a modern multi-core platform typically consist of two or three cache memory levels between the core and the main memory. Usually, each core has a private smaller Level 1 (L1) cache memory. All cores share a larger Level 2 (L2) and/or a Level 3 (L3) cache memory. Caches have been extensively used to bridge that gap.

Data transfers between main memoty and caches are managed by hardware, in a transparent manner to the programmer and compiler. Unfortunately, caches are source of predictability problems in hard real-time systems. The cache replacement strategy might introduce unpredictability of the cache behavior. Instruction prefetching, out-of-order execution, and control speculation introduce interferences between processor components, e.g. caches, pipelines and prefetch queues.

A lot of progress has been achieved in the last ten years to statically predict worst-case execution times (WCETs) of tasks on architectures with caches [100, 108, 113, 120]. However, cache-aware WCET analysis techniques are not always applicable due to the lack of documentation of hardware manuals concerning the cache replacement policies.Moreover, caches are sources of timing anomalies in dynamically scheduled processors [112] (a cache miss may in some cases result in a shorter execution time than a hit). In such situations, cache locking and cache partitioning techniques are of interest [89].

Scratchpad memory (SPM) is a popular choice in real-time embedded systems. It is a small software-managed on-chip static RAM that has been widely accepted as an alternative to cache memory, as it offers better timing predictability compared to caches. The scratchpad memory is mapped into the address space of the processor, and is accessed whenever the address of a memory access falls within a predefined range. Contrary to caches, the compiler and/or the programmer explicitly controls the allocation of instructions and data to the scratchpad memory. This operating principle makes the latency of each memory access, and thus program execution time, more predictable. Significant effort has been invested in developing efficient static and dynamic allocation techniques for scratchpad memories [75, 76, 101, 105].

Many of the modern multicore platforms offer scratchpad memories, for example: NXP S32, Renesas R-car, STM Stellar, and Aurix Infineon platform series.

1.5 Programming parallel architecture

We define parallel applications as those applications that perform their computations simultaneously on multiple processors. Parallelism is important in order to get advantage of hardware advancement of processor architecture such as multiprocessor. It is crucial to remember that adding more processing units to execution platforms to increase performance is worthless if the designed software is incompatible with the hardware. Software parallelism is classified into the following classes:

- Inter-task parallelism: in which tasks execute in parallel. An example of such parallelism is a set of tasks which executes on multiple processors.
- Intra-task parallelism: or the inter-subtask parallelism, here a parallel task consists of subtasks which execute in parallel.
- Intra-subtask parallelism: a subtask of a parallel task consists of a set of threads running in parallel. In this case, a subtask requires more than one processor to execute.

A thread is an execution portion of a parallel task (or a subtask). Task's threads are usually all activated at the same time and have to terminate their execution at the same time. There are multiple programming languages and frameworks that allow the code to be executed in parallel on different threads. Two frameworks are available to the C programming language that we will present in this manuscript: *POSIX Threads(Pthreads)* and *OpenMP*.

1.5.1 POSIX Threads

POSIX Threads is a common threading model when working in C or C++, usually referred to as Pthread, it is a standardized model to work with threads. POSIX Threads can be use to parallelism task of a program in order to increase

the execution speed. The POSIX standard states that threads must share a number of informations, for example the threads must share:

- Process ID
- Parent Process ID
- User and group IDs
- Open FD's (File Descriptors)

POSIX Threads offers a header file and a library that must be included at compilation time and a set of functions to create, manipulate, synchronise and kill threads in a program. The starting point for threads in a Pthreads program is the *pthreads_create* function, this function takes a pointer to a function that is the starting point for the new thread. Then, the *pthread_exit* function is used to kill the thread. In the main thread, the *pthread_join* function is used to wait for the executing threads [3]. There are several other Pthread procedures for example those used for synchronization (with locks and barriers) and Mutexes.

1.5.2 OpenMP framework

OpenMP [42] is a parallel programming model for shared-memory multiprocessor systems developed in 1990 by SGI.The goal of OpenMP is to provide a standard and portable API for writing shared-memory parallel programs. OpenMP works in conjunction with either standard Fortran or C/C++. It is comprised of a set of compiler directives used in the source code. These directives are instructional notes to any compiler supporting OpenMP. They take the form of source code comments (in Fortran) or *#pragmas* (in C/C++). Collectively, these directives and library routines are formally described by the application programming interface (API) now known as OpenMP.

OpenMP is designed to support incremental parallelization, or the ability to parallelize an application a little at a time at a rate where the developer feels additional effort is worthwhile [104].

To create the threads when an parallel directive is encountered, OpenMP uses the Fork-Join model. Figure 1.4 shows an example of a Fork-Join model. When a executing thread encounters a parallel construct it will create a team of threads(fork) and become the master thread of the team. Then the team executes the code assigned to it and before the initial or master thread continue to execute the code after the parallel construct all the threads in the team are terminated(join).

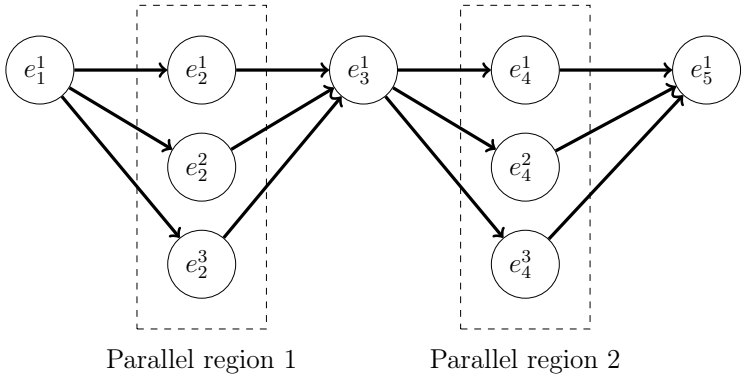


Figure 1.4: Example of Fork-Join Model

The first sequential execution in Figure 1.4 is e_1^1 called the source thread. Three parallels threads e_2^1, e_2^2 and e_2^3 gathered in "parallel region 1" are forked from e_1^1. The second sequential execution is e_3^1 and it joints the three threads forked by e_1^1, it forks at its turn three parallel threads e_4^1, e_4^2 and e_4^3 gathered in "parallel region 2". The last sequential task joins the threads created by e_3^1 and continue till the task ends.

Compared to using pthreads and working with mutex and condition variables, OpenMP is much easier to use because the compiler takes care of transforming the sequential code into parallel code according to the directives [33, 65].

1.6 Conclusion

In this chapter, we presented an overview of multicore architectures, the different types of memory used in these architectures and reviewed two parallel programming languages. These are notions on which our work will be based. We will consider a set of tasks with timing constraints to be partitioned on a multicore platform with the goal of reducing/avoiding shared resources contention. Before that, we will give a quick overview on real-time systems in the next chapter.

Chapter 2

Real-time Systems

Contents

2.1 Introduction 15
2.2 Task models 15
2.2.1 Liu and Layland model 16
2.2.2 PREM and AER models 17
2.2.3 Directed Acyclic Graph (DAG) 18
2.3 Priority assignment for scheduling 19
2.3.1 Scheduling characteristics 19
2.4 Time- vs event-triggered real-time scheduling 20
2.5 Preemptive vs non-preemptive scheduling 21
2.6 Earliest Deadline First (EDF) 21
2.6.1 Preemptive real-time scheduling 21
2.6.2 Non-preemptive real-time scheduling 23
2.7 Preemptive multiprocessor real-time scheduling . . . 24
2.8 Conclusion 25

2.1 Introduction

Real-time systems are computing systems that must react within precise time constraints to events in the environment. As a consequence, the correct behavior of these systems depends not only on the value of the computation but also on the time at which the results are produced [57].

A real-time system interacts with its environment through the use of sensors and/or actuators. Given the dynamic nature of the environment, the system's response to a change in the environment cannot be postponed indefinitely. The state of the system at any point in time must be correct in relation to the state of the environment. Any state of the system that causes an accident in the environment or in the system itself, such as the destruction of system components, is regarded as a failure. The cost of failure classifies these systems into 3 classes:

- Hard real-time systems: are systems in which a failure to meet even a single deadline may lead to complete or appalling system failure. This is particularly true for the autonomous driving (AD) systems, which incorporate complex functions along with strong safety requirements. Its software functions must coexist and share resources with other software vehicle functions without sacrificing real-time safety requirements;

- Soft real-time systems: are systems in which one or more failures to meet the deadline are not considered complete system failure, but that performance is considered to be degraded. Web browsing is an example of soft real-time system;

- Firm real-time systems: are not hard-real time systems, but results delivered after a deadline that has been violated, are ignored. Video conferencing and satellite based tracking are good examples of firm real-time system.

2.2 Task models

Real-time systems applications generally consist of a set of concurrent functionalities called tasks with timing-related properties. A task is a *thread*, i.e. a sequential piece of code executing on a multi-threading operating system. The pseudo-code of a typical real-time task in a POSIX-like environment is shown in Listing 2.1. A task can be activated several times during the system life. Each activation is called a "job" or "instance". In this thesis, we consider a set of real-time tasks $\mathcal{T}$, each task τ_i is described as a infinite succession of jobs. We denote j_i^l the l^{th} job of task τ_i. Each job j_i^l is characterized by three parameters $(\mathsf{a}_i^l, \mathsf{d}_i^l, \mathsf{c}_i^l)$. a_i^l represents the job arrival time, it corresponds to the time instance when the job j_i^l becomes ready to start its execution. d_i^l is the job absolute deadline, and c_i^l represents the execution time of j_i^l.

According to the recurrence problem, we can distinguish three types of real-time tasks:

- Periodic task: two consecutive releases (jobs) of the same task are separated by a fixed inter-arrival time;
- Sporadic task: two consecutive releases (jobs) of the same task are separated by a minimum inter-arrival time;
- Aperiodic task: There is no correlation between the activation of two successive instances of an aperiodic task.

```
void* task()
{
    // Initialization
    while (true) {

        // task code

        wait_for_next_activation() ;
    }
}
```

Listing 2.1: Example of real-time task code

2.2.1 Liu and Layland model

The classic model by Liu and Layland [135] modeled a real-time task by $\tau_i = (\mathsf{C}_i, \mathsf{D}_i, \mathsf{T}_i)$:

- C_i: is the worst-case execution time of the task, it is equal to:

$$\mathsf{C}_i = \max_{\forall l}(\mathsf{c}_i^l) \tag{2.1}$$

  It represents the maximum elapsed time between the processor acquisition and the task completion without considering any interruption. Several techniques are proposed in the literature to upper-bound and estimate this value, a good overview can be found in [109].

- D_i: is the task's relative deadline, it represents the time within the task have to end it's execution (i.e. the lenght of any interval $[\mathsf{a}_i^l, \mathsf{d}_i^l]$);
- T_i: is the task period, it represents the minimum inter-arrival time between two jobs if we consider sporadic real-time tasks.

If D_i is equal to the task period T_i then, the deadline is said to be *implicit*. If D_i is less than or equal to the task period T_i, then the deadline is *constrained*(2.1).

It is important to distinguish between two classes of periodic task sets, regarding the activation time of the 1^{st} instance of tasks (called the offset ϕ_{τ_i}): *synchronous* and *asynchronous* task sets. Synchronous task sets are those in which all offsets are equal, for example $\phi_{\tau_0} = \phi_{\tau_1} \cdots \phi_{\tau_n} = 0$. Asynchronous task sets in contrast are those where the offsets of two distinct tasks might be not

equal [111]. In this context, if the offsets of the jobs of a single task are equal, then we say that the offset is static. On the other hand, if at least one job offset of the task is different from the offset of the other jobs, then the offsets are dynamic.

Tasks may be characterized by other derived parameters. First, the task utilization, given by $\mathsf{u}_i = \mathsf{C}_i/\mathsf{T}_i$, which indicates the extent to which a task utilizes the computing resources. Thus, the total utilization is given by $\mathsf{U}_{\mathcal{T}} = \sum_{\tau_i \in \mathcal{T}} \mathsf{u}_i$. The maximum possible utilization for a single computing unit is assumed to be equal to 1. Second, the worst-case response time $\mathsf{R_i}$ which is defined as the longest time from a job arriving to its completion. Finaly, the hyperperiod H which is defined as the least common multiple of all task periods.

All the task parameters cited above are graphically represented in Figure 2.1.

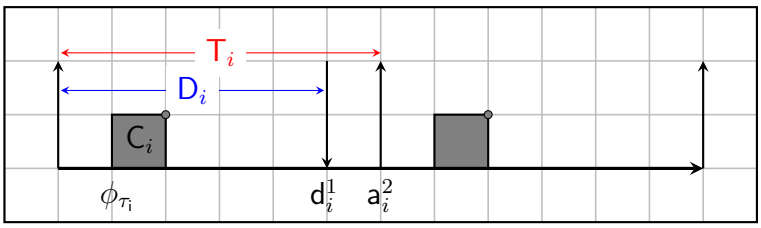


Figure 2.1: Periodic constrained-deadline task parameters.

If jobs of different tasks can execute in any order, they are said to be independent. Dependencies between tasks can be classified into: control dependencies (mutual exclusion and precedence constraints) or data dependency. When job j_i^l can start the execution only after another job j_h^k finishes its execution in a task model, then such constraint is called precedence constraint. These precedences can be modeled using a precedence graph. The latter, is a directed acyclic graph which represents the precedence constraints among a set of jobs. In real-time systems, jobs communicate via shared ressources, hence, data of one job depends on the results produced by others.

Several models that embody these dependencies were proposed in the literature. In this thesis we focus in particular on phased models (i.e. PREM and AER) and the common model of parallelism which is the Directed-Acyclic-Graph (DAG) model.

2.2.2 PREM and AER models

Under PREM model [61], the task code is divided into a set of scheduling intervals executed sequentially at run-time: compatible intervals and predictable intervals. Predictable intervals are divided into two phases: (i) a non-preemptive *memory phase* and a *computation phase*. In the memory phase, the core/processor accesses to the main memory to perform data fetches and replacements [50, 61]. At the end of the memory phase, all required data is available in the core local memory. Therefore, during the computation phase, the task can perform computations without any need to access to the main memory. Instead, OS activity is confined to compatible intervals.

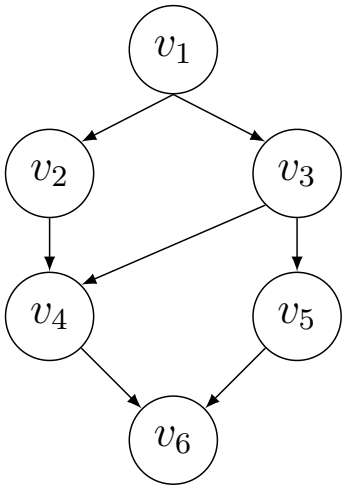


Figure 2.2: DAG task example

The AER model [39] is an extension of the PREM model. Under this model, tasks are divided into three distinct phases, namely: *acquisition* (A) and *restitution* (R) which are memory phases and a pure *computation*(E) phase.

These models where data transfers and computations are separated allow to both mitigate delays related to shared access to the main memory and make them easier to analyze. Since the memory and the computation phases are decoupled, it is possible to exploit parallelism. In fact, the computation phases of different tasks can execute in parallel with any other phase of any other task. These models ere used in Chapter 5 & 6.

2.2.3 Directed Acyclic Graph (DAG)

A task τ_i in task set $\mathcal{T}$ is represented by a tuple $(\mathsf{G_i}, \mathsf{D}_i, \mathsf{T}_i)$, where $\mathsf{G_i}$ is a Directed Acyclic Graph (DAG) that describes the internal structure of τ_i, D_i is its end-to-end relative deadline, and T_i is its period. A DAG is acyclic, i.e. there is no closed cycle between vertices of the graph, it can be periodic or sporadic.

Task τ_i consists of a set of subtasks (execution portions) under precedence constraints that determine their execution flow. Each subtask is characterized by a specific WCET and disposes a set of predecessors and a set of successors. Each DAG may have one or more than one *source* subtask and one or more than one *sink* subtask. A source subtask has no predecessor and it is activated by the activation of its DAG. A sink subtask is an ending subtask in the DAG and has no successors. An example of a DAG task is given in Figure 2.2 which shows the structure of the DAG model.

This model is used in Chapter 6 of this thesis along with the AER model to build our new task model named $AECR - DAG$.

2.3 Priority assignment for scheduling

In real-time systems, we assume that real-time tasks are arbitrated using a scheduler in order to be execute on computing resources such as unicore or multicore CPUs. Similar to general-purpose operating systems such as Linux, real-time operating systems also associate a notion of priority to tasks to determine the order in which they should be scheduled. We can define a scheduler as the algorithm which determine which of the task jobs is run on the core(s) at each moment of time.

According to the nature of priority assignment, three broad categories of real-time scheduling are identified: (i) fixed-task priority scheduling, (ii) fixed-job priority scheduling and (iii) dynamic-job priority scheduling. Under fixed-task priority scheduling, the priority of the task and all its jobs does not change during the whole system life; The algorithms Rate Monotonic (*RM*, Liu and Layland, [135]) and Deadline Monotonic (*DM*, Leung and Whitehead, [133]) belong to this first class. Under fixed-job priority scheduling, the priority of the task is allowed to change but not the priority of jobs. More specifically, a job has a fixed priority during its execution, but jobs of the same task may have different priorities. Earliest Deadline First (*EDF*, Liu and Layland, [135]) represents perfectly this class of scheduling algorithm. In the last class of scheduling algorithm, the priority of a job can changes at each moment of time during the execution. Least Laxity First (*LLF* Dertouzos, [134]) is the perfect example of dynamic-job priority scheduling algorithm.

The various scheduling algorithms cited above are *online*. They will schedule the tasks during the execution of the system. There exist also *off-line* schedulers, that will determine which task have to be executed based on an offline schedule. An example of an offline scheduler is the *time-triggered (TT)* discussed in Section 2.4.

2.3.1 Scheduling characteristics

Before diving into details, it is necessary to give more definitions [28].

Preemption

Preemption is the act of temporarily interrupting an executing job and invoke a scheduler to determine which process should execute next. Therefore, allowing higher priority jobs to acquire the preemptible resource. The interrupted job resumes its execution at some later time point.

Schedulability

A task is referred to as schedulable according to a given scheduling algorithm if its worst-case response time under that scheduling algorithm is less than or equal to its deadline. Similarly, a task set is referred to as schedulable according to a given scheduling algorithm if all of its tasks are schedulable.

Feasibility

A task set is said to be feasible with respect to a given system if there exists some scheduling algorithm that can schedule all possible sequences of jobs that may be generated by the task set on that system without missing any deadlines.

Optimality

A scheduling algorithm is referred as optimal if it can schedule all the task sets that can be scheduled by any other algorithm.

Sufficient tests

A schedulability test is said to be sufficient, with respect to a scheduling algorithm and a system, if all of the task sets that are deemed schedulable according to the test are in fact schedulable.

Necessary tests

Similarly, a schedulability test is said to be necessary if all of the task sets that are deemed un-schedulable according to the test are in fact unschedulable.

Exact tests

Finally, a schedulability test is termed as exact, if it is both sufficient and necessary.

Predictability

predictability represents an important property in real-time systems where task execution times are variable up to some worst-case value.

A scheduling algorithm is said to be predictable if the response times of jobs cannot be increased by decreasing their execution times, with all other parameters remaining constant.

2.4 Time- vs event-triggered real-time scheduling

Time- and event-triggered are the two major approaches for designing and scheduling real-time systems. In time-triggered (TT) systems, a pre-elaborated, offline static plan dictates the exact points in time when each task must execute and for how long it is allowed to execute. In event-triggered (ET) systems on the other hand, tasks are released for execution as a consequence of events, whose times of occurrence are not necessarily known in advance [17]. These events may indeed be related with time, such as the recurrent expiration of a timer cycle, but they can also be asynchronous, such as entering a particular system state, or receiving particular sensor or user inputs. At runtime, the scheduler makes online decisions about which task to execute, typically based on the priorities assigned to tasks whose activation events have occurred and need be handled.

More extensive comparisons between time- and event-triggered systems are given in [95, 97].

2.5 Preemptive vs non-preemptive scheduling

Preemption is a key factor in real-time scheduling algorithms, since it allows the operating system to immediately allocate the processor/core to incoming tasks that have higher priority to complete [77]. Task priorities are determined by the scheduling algorithm used. For instance, under dynamic priority scheduling such as EDF, higher priority task corresponds to higher urgency due to its earlier deadline.

Through fully preemptive scheduling, the scheduler can suspend a running task on a processor/core at any time. The context of the preempted task is saved, and be replaced with the one of the higher priority task. Whenever a preemption occurs, diffrent sources of overhead must be considered. We will go into more detail regarding these overheads in the next chapter.

In the contrast, a task can not be interrupted in the non-preemptive scheduling. Once the task starts its execution, it executes until completion, even when higher priority tasks arrive, which will allow for more predictable execution behavior. However, the system utilization may be degraded since additional blocking delays must to be considered in this case. This is a popular problem in the non-preemptive scheduling called the *priority inversion*. The priority inversion problem occurs when a higher priority task cannot execute (remains blocked) waiting for a low priority task to complete its execution.

Indeed, when the preemption cost is neglected in the analysis, fully preemptive scheduling is more efficient in terms of processor utilization than non-preemptive scheduling.

2.6 Earliest Deadline First (EDF)

In this document, we limit ourselves to fixed-job priority scheduling.

As mentioned previously, Earliest Deadline First (EDF) is a fixed-job priority scheduler. It assigns the higher priority to the job having the smallest value of absolute deadline. EDF's scheduling analysis changes depending on whether it is preemptive or non-preemptive scheduling. In this chapter, the scheduling analyses for EDF are presented for systems with and without preemptions.

2.6.1 Preemptive real-time scheduling

Synchronous tasks

EDF is an optimal scheduling algorithm on preemptive uniprocessor, that is, if there exists an real-time scheduler for a set of tasks, EDF can also schedule these tasks while respecting their timing constraints. The schedulability analysis of a preemptive system using EDF algorithm is based on the use of the processor demand bound analysis proposed by Baruah et al. [130].

We start from the concept of *demand function*. The demand function (df) for a given task set $\mathcal{T}$ is the amount of time demanded by the tasks in an interval $[t_1, t_2)$ that the core/processor must execute. The demand function is defined as:

$$df(t_1, t_2) = \sum_{i=1}^{|\mathcal{T}|} \Delta_i(t_1, t_2) \cdot \mathsf{C}_i \tag{2.2}$$

where:

$$\Delta_i(t_1, t_2) = \left(\left\lfloor \frac{t_2 - \phi_i - \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor - \left\lceil \frac{t_1 - \phi_i}{\mathsf{T}_i} \right\rceil + 1 \right)_0 \tag{2.3}$$

is the number of jobs of task τ_i with release time greater than or equal to t_1 and deadline less then or equal to t_2 [130].

Considering a fully preemptive single core/processor scheduler, a necessary and sufficient condition for a set of tasks to be schedulable by EDF consists in checking that the demand never exceeds the length of the interval.

Lemma 1 (Baruah et al. [130]). The taskset $\mathcal{T}$ is feasible on a single core/processor ($\mathsf{U}_\mathcal{T} \leq 1$) if and only if:

$$\forall 0 \leq t_1 < t_2 \leq 2\mathsf{H} + \phi_{\mathsf{max}}, \quad df(t_1, t_2) \leq t_2 - t_1 \tag{2.4}$$

where ϕ_{max} is the largest offset.

The worst case demand is found for intervals starting at 0. Thereby, we can define the *demand bound function (dbf.* For a set of synchronous periodic tasks and of a set of sporadic tasks $\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$, the demand bound function $\mathsf{dbf}(\mathcal{T}, t)$ corresponds to the maximum cumulative worst-case execution time of all jobs having their arrival time and absolute deadline within any interval of time of length t. It can be computed as follow:

$$\mathsf{dbf}(t) = \sum_{i=1}^{|\mathcal{T}|} \mathsf{dbf}_i(t) \tag{2.5}$$

Where $\mathsf{dbf}_i(t)$ is the demand bound function of a task τ_i in the interval of time of length t computed as follow:

$$\mathsf{dbf}_i(t) = df(0, t) \tag{2.6}$$

An exact schedulability test for a set of synchronous tasks was proposed by Baruah et al. [130]: if for each interval of length t, the demand bound of the task set is less than t, then the system is schedulable.

Theorem 1 *Given a set of tasks $\mathcal{T}$. $\mathcal{T}$ is schedulable on a single core if and only if:*

- $\mathsf{U}_\mathcal{T} \leq 1$ *and,*
- $\forall t \leq L^*, \mathsf{dbf}(t) \leq t$

where L^* is an estimated upper bound on the first idle time.

We can continue testing until H, instead of ending at L^*.

Asynchronous tasks

The previous theorem 1 of Baruah et al. [130] does not hold in the case of asynchronous task sets. It still gives a sufficient condition, in the sense that if the hypothesis holds for the corresponding synchronous task set, than the original asynchronous task set is feasible. However the condition is no longer necessary. Authors in [96] proposed an approximate schedulability test with pseudo-polynomial complexity for asynchronous task sets:

$$\forall t \leq L^*, \mathsf{dbf}(t) = \max_{\forall \tau_i \in \mathcal{T}} \left\{ \sum_{\tau_j \in \mathcal{T}} \left(\left\lfloor \frac{t - \overline{\phi}_{j,i} - d_j}{\mathsf{T}_i} \right\rfloor + 1 \right)_0 \cdot \mathsf{C}_j \right\} \tag{2.7}$$

where $\overline{\phi}_{j,i} = (\phi_j - \phi_i) \mod \mathsf{T}_i$. To understand Equation (6.15), consider that all the other tasks in $\mathcal{T}$ are activated with an offset relative to the arrival of τ_i. Therefore, we need to align the offset of one task to the beginning of an interval of length t and compute the workload generated in the interval. We do this for every task, and then we take the maximum. The resulting dbf is an upper bound to the actual dbf in that interval.

2.6.2 Non-preemptive real-time scheduling

Synchronous tasks

Jeffay et al. [124] proposed a schedulability test for non-preemptive systems. This test is valid only for systems with tasks that have implicit deadlines. For a task set $\mathcal{T}$ where tasks are sorted in non-decreasing order by period (i.e., for any pair of tasks τ_i, τ_j, if $i > j$, then $\mathsf{T}_i \geq \mathsf{T}_j$). If $\mathcal{T}$ is schedulable then:

1)

$$\sum_{\tau_i \in \mathcal{T}} \frac{\mathsf{C}_i}{\mathsf{T}_i} \leq 1; \tag{2.8}$$

2)

$$\forall i, 1 < i \leq |\mathcal{T}|;\ \forall t, \mathsf{T}_1 < t < \mathsf{T}_i : \quad \mathsf{C}_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t-1}{\mathsf{T}_j} \right\rfloor \cdot \mathsf{C}_j \leq t \tag{2.9}$$

In the first condition, we verify that the processor's global utilization does not exceed its computing capacity; Then, the second condition ensures that task blocking does not compromise adherence to time constraints.

For tasks with constrained deadline, the demand bound function can be expressed as follow [92]:

$$\mathsf{dbf}(t) = \sum_{i=1}^{|\mathcal{T}|} \mathsf{dbf}_i(t) + \mathsf{B}\ (t) \tag{2.10}$$

where:

$$\mathsf{B}\ (t) = \max\{\ \mathsf{C}_j \mid \mathsf{D}_j > t\ \}. \tag{2.11}$$

is the blocking time due to non-preemptive scheduling.

Asynchronous tasks

Schedulability analysis of tasks with offsets has been studied in [96]. Authors proposed a theorem which deals with blocking time due to mutually exclusive resources:

Theorem 2 *Given an asynchronous task set $\mathcal{T}$. The tasks are schedulable if the cumulative utilisation is less than 1 ($\mathsf{U}_{\mathcal{T}} \leq 1$) and,*

$$\forall t < L^*, \quad \sum_{\tau_i \in \mathcal{T}} \mathsf{dbf}_i(t) + \mathsf{B}\ (t) \leq t \tag{2.12}$$

This theorem will be extended in Chapter 6 to the AECR-DAG task sets.

Further considerations had to be taken into account for a multiprocessor settings [1, 103, 131]. As our focus is on working with modern cyber-physical systems, which typically involve multicore CPUs, in the next section, we provide background for multiprocessor scheduling on real-time systems.

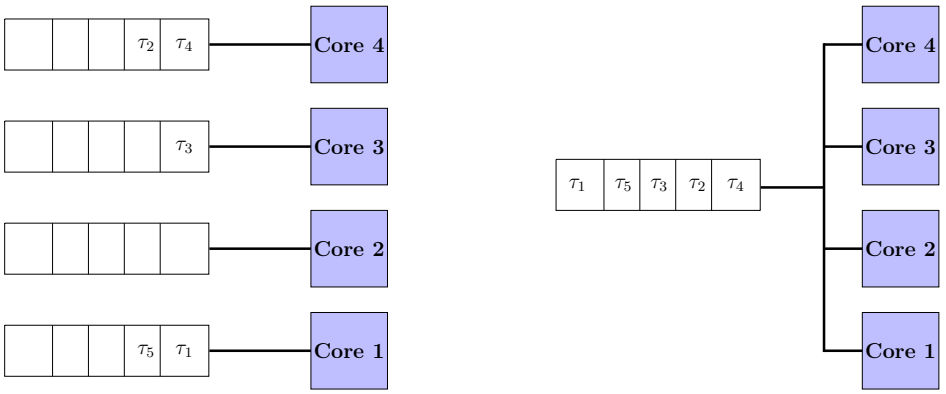
2.7 Preemptive multiprocessor real-time scheduling

The problem of scheduling real-time applications on multiprocessor systems is more complicated and challenging than real-time scheduling on uniprocessor systems. It is because there are more decisions to be taken in the case of multiprocessor scheduling and more issues to be considered.

On a multiprocessor, which we deal with throughout this thesis, real-time scheduling can be further classified into two categories: (i) Global and (ii) Partitioned (Semi-Partitioned and Federated scheduling are not considered in this work).

- Under global scheduling, tasks may migrate across processors.Two diffrent migration types are found:
  - Job level migration: a job can start its execution on a processor and be interrupted to continue its execution in an other processor.
  - Task level migration: a job is executed on only one processor/core, but jobs of the same task may be executed on different processors.
- Under partitioned scheduling, tasks are pinned to specific processor/core and may not migrate even if other cores are idle. Mostly, partitioning algorithms passes through three steps: (i) sorting tasks according to some criteria (for example sorting by deadline); (ii) Assign tasks to a processor; (iii) and the use of uniprocessor scheduling algorithms on each processor to schedule the processor's tasks after each task-to-processor assignment.

To make the difference between the two categories, we present in the following an illustration example of a platform with 4 cores (Figure 2.3). In Figure 2.3a representing the partitioned scheduling, each core has its own ready-queue. In the other hand, in Figure 2.3b representing the global scheduling, all cores have the same ready-queue and the m-highest priority jobs are run at the same time on m-processors.



(a) Partitioned Scheduling. (b) Global Scheduling.

Figure 2.3: Partitioned scheduling vs global scheduling.

While it may seem intuitive to always prefer global scheduling due to higher utilization benefits, previous works have shown that partitioned scheduling is much easier to analyze and can be used for most practical cases instead because it effectively boils down to the uniprocessor case for each core, allowing a large body of existing analyses to be applicable [25]. On the contrary, while global scheduling offers fast average-case response times, it is relatively harder to analyze and has high overheads in practice [67, 81]. The worst-case performance remains comparable to partitioned scheduling, which makes it less gainful for real-time systems. As a result, we focus on partitioned fixed-job priority task scheduling in our work.

An important consideration under partitioned scheduling is the manner in which tasks are partitioned across multiple computing resources. As this is proved to be a NP-hard problem, existing real-time literature suggests the use of heuristics such as *Best-fit (BF* and *Worst-fit (WF)* [1, 103] to allocate tasks on processor cores. In this thesis, we propose several exact methods for real-time task partitioning on a multicore platform.

2.8 Conclusion

In this chapter, we presented an overview of real-time systems and scheduling theory for uniprocessor and multiprocessor systems. We assume that all ingredients are ready to enter into our problem analysis. In the next chapter, we will discuss in details the runtime overheads in multicore architecture.

Chapter 3

Processor and Memory Co-scheduling in Multicore Systems

Contents

3.1 Introduction 27
3.2 Cache related delays in multicore systems 27
3.3 Shared resources contention: main memory and buses 29
3.4 Related work 30
3.5 Conclusion 34

3.1 Introduction

Real-time operating systems were initially designed for single-core platforms, where the use of real-time scheduling policies, efficient inter-process communication, and prioritized interrupt handling is enough to ensure temporal predictability. Support for multicore platforms was later introduced without a substantial change in design, resulting in a new set of challenges. For example, in an automotive environment, new safety functionalities like "automatic emergency breaking" and "night view assist" must read and fuse data from sensors, process video streams, and rise warnings when an obstacle is detected on the road all under real-time constraints [32, 60]. All these functionalities demand large computational power which can be ensured by multicore platforms. As mentioned, multicore processors generally feature shared resources including main memory, buses, caches, and input/output (I/O) peripherals. This has a great impact on the predictability of real-time systems which is critical for correctly estimating the worst-case execution time of tasks. In fact, a fundamental assumption common to all schedulability analysis techniques found in the literature is that an upper bound on the worst-case execution time of each task is given prior the analysis.

Several sources for temporal unpredictability could be noticed, a deep study of the sources is proposed in [43]. According to [43], these sources can be categorized into: primary sources including caches, main memory, and memory controller; and secondary sources including hardware-prefetching, power saving strategies and system management interrupts. These sources of temporal unpredictability increases considerably WCET estimates.

3.2 Cache related delays in multicore systems

The first main factor of temporal unpredictability in a multicore processor that we consider in this work is the cache memory hierarchy[51, 73, 85].

In fully preemptive systems, preemption can introduce a significant runtime overhead and may cause high fluctuations in task execution times. In fact, in such systems, preemption is allowed at any time during task execution. It is not possible to control the number of preemption points thus, degrading system predictability. In particular, at least four different types of costs need to be taken into account at each preemption [47]:

- Scheduling cost. It is the time taken by the scheduling algorithm to suspend the running task, insert it into the ready queue, switch the context, and dispatch the new incoming task;
- Pipeline cost. It accounts for the time taken to flush the processor pipeline when the task is interrupted and the time taken to refill the pipeline when the task is resumed;
- Bus-related cost. It is the extra bus interference for accessing the RAM due to the additional cache misses caused by preemption;

- Cache-related cost (CRPD). It is the time taken to reload the cache lines evicted by the preempting task.

It has been shown in [82] that, in some cases, the cumulative preemption overhead may increase a task's execution time up to 33%.

Since preemption destroys program locality, WCET estimates of preemptive tasks are computed by assuming worst-case cache related delays, given by the extra operations needed for refilling the cache lines evicted by the preempting task. In small embedded systems, such an extra cost results in longer and more variable response times that can also significantly affect the overall energy consumption [56].

Limiting preemptions is often possible without jeopardizing schedulability. In this context, the *Limited Preemptive Models (LPM)* has emerged in order to increase the predictablity and reduce the overall preemption overhead of such systems, and, consequently, to increase their schedulability [47]. There are different mechanisms for limiting preemption, among which we cite [63]:

- Fixed Preemption Points (FPP). Each task, under this model, is divided into a number of non-preemptive blocs separated by predefined preemption points inserted in the task code. If a higher priority task arrives between two preemption points of the running task, preemption is deferred until the next preemption point.

- Floating Non-Preemptive Regions (NFP). This approach consists in considering for each task τ_i a number of NPRs each with a maximum length, whose location is unknown. In this model, NPRs can be considered to be floating in the task code.

- Preemption Thresholds (PT). The concept of preemption thresholds, first proposed by [114] under fixed priority systems, is the basis of this approach. Using this method, a task is able to disable preemption up to a specified priority called preemption threshold. Each task is assigned a regular priority and a preemption threshold, and the preemption is allowed only when the priority of the arriving task is higher than the threshold of the running task. This work has been later improved by [102].

It is important to note that, in the FPP model, the length of the final non-preemptive block plays a crucial role in reducing the task response time. A large final non-preemptive block allows reduce in the interference the task may suffer from higher priority tasks. However, the length of the final non-preemptive block cannot be arbitrarily large to limit the blocking time imposed to higher priority tasks [54]. In the floating NPR model, instead, the exact location of each non preemptive region is not known a priori, so that a task could be preempted even by an arbitrarily small amount of time before the end of the execution, increasing the resulting response time. From a practical point of view, using fixed preemption points allows achieving higher predictability. In fact, by properly selecting the preemption points in the code, it is possible to reduce cache misses and context switch costs, therefore improving the estimation of preemption overheads

and worst-case execution times [63, 88]. For this reason, we will use the FPP model in Chapter 5 in order to reduce the preemption overhead.

3.3 Shared resources contention: main memory and buses

The second main factor of temporal unpredictability in a multicore processor that we consider in this work is contentions in particular those related to the main memory and the communication buses.

Currently, most COTS architectures feature a single port main memory that is shared for among all cores and I/O peripherals. The active components in these architectures (e.g. cores) can independently initiate access to shared resources, which cause contentions. In particular, contention for access to main memory can significantly delay data fetch, increasing the real-time task's worst case execution time (WCET) [74]. These shared resources are managed exclusively in hardware and they treat requests coming from different tasks running on different cores as if they were all requests from one single source (core) [51].

Two approaches are popular: (i) estimating the worst-case interference profiles and deriving safe execution time bounds; (ii) avoiding interference at the system design level. In the first approach, the hardware platform and the task execution must be modeled accurately: the memory access profiles of all tasks are extracted and combined with each other to estimate the worst-case profile. In general, it is difficult to precisely compute the worst-case interference profile, which likely leads to include scenarios that might never occur, therefore over-estimating the worst-case interference cost. The second approach tends to prevent interference by enforcing time isolation (e.g. time partitioning schemes like MemGuard [36]).

An intermediate approach is to use appropriate application models such as the Acquisition-Execution-Restitution model (AER) [39], or the PRedictable Execution Model (PREM) [61]. In the latter, a task is modeled by two distinct phases: a memory phase and computation phase. In the memory phase, data is exchanged between main memory and local memory. This includes write-back of the computed data from the local memory to main memory of the previous job, and fetching new data for the activated job from the main memory to the local memory. In a computation phase, loaded data is processed and all access to the main memory is forbidden.

Finding the proper way of scheduling memory phases is not straightforward. One difficulty is the lack of hardware support for real-time scheduling on the bus. Typical bus controllers available on commercial platforms support very simple policies like First-In-First-Out (FIFO) or slot-based time-triggered scheduling. FIFO safe-response time bounds are known to be very large, whereas fixed slots are not flexible enough to efficiently support the variety of application requirements. The problem is even more complex, as the respect of timing constraints requires to tightly co-schedule and analyze the memory phases and the computation phases.

3.4 Related work

In this section, we briefly present the main algorithms and results regarding real-time runtime overhead-free scheduling in multicore systems which are found in the state-of-the-art. This chapter is divided into two sections. The first Section 3.4 concentrates on preemption overheads on multicore systems and we review the main task models and scheduling algorithms proposed to reduce preemption overheads. Section 3.4 describes the contention problem and its effect on the WCET estimation. Then, we present the different task models proposed in the litterature to solve the problem of contention in particular those related the communication bus and main memory.

Preemption overheads in multicore systems

Many authors [31, 38, 55, 59, 78, 86, 94, 123] have focused on reducing the cost of preemptions in the real-time systems scheduling theory and practice from different perspectives. A good survey overviewing limited preemption models can be found in [48].

In Chapter 4, we focus on the deferred preemption model (DPM). In such model, a task is divided into a set of non-preemptive chunks separated by preemption-points. At runtime, preemption can only happen at the boundaries of non-preemptive chunks: if a higher priority task arrives while a lower priority task is executing, the preemption is delayed until the next preemption point. DPM increases predictibility and reduces preemption overhead compared to fully preemptive systems, and reduces blocking time compared to non-preemptive scheduling. Two models of preemption handling have been proposed: the floating preemption point model, and the *fixed preemption point model*. In the first model, a preemption point can be inserted at any place of the task code, whereas in the second model the preemption points are fixed prior to analysis.

DPM has been first introduced in [123]. Baruah [94] proposed techniques to compute the maximum length of any interval of time where a given task can execute non-preemptively without violating the schedulability for EDF under the floating preemption points model. The same bounds for fixed priority scheduling has been proposed in [80]. Response time analysis has been improved in [78] by considering special cases where the last non-preemptive chunk can delay the execution of higher priority tasks. Authors in [68] proposed schedulability analysis for both fixed priority and EDF.

In [68], authors tackled the problem of finding the best possible placement of preemption points, they assumed an identical preemption cost for all preemption points. Davis and Bertogna [49] introduced an optimal algorithm for fixed priority scheduling with deferred preemption. Authors in [63] propose schedulability analysis for the fixed preemption model under fixed priority scheduling by considering preemption points with different costs. The latter has been extended in [55] for EDF and using an optimal algorithm to select preemption points while respecting all timing constraints.

Another alternative, called hybrid preemption model, has been presented

in [114], based on the Preemption Thresholds Scheduling (PTS) approach in which a task can disable preemption up to a specified priority level(preemption threshold). Each task is assigned a preemption threshold and regular priority also, and it is allowed to preempt only when its priority is higher than the threshold of the preempted task. An exact schedulability analysis for FP with preemption thresholds has been presented in [72].

In order to compute the *Cache Related Preemption Delay* (CRPD), we need to consider different factors: (i) the preemption point PP in the code of the preempted task where the preemption occurs, (ii) the cache blocks used until PP and that may be reused by the preempted task after preemption, and finally, (iii) the evicting cache blocks of the preempting task [52]. Therefore, the CRPD is bounded by $(R \times BRT)$ where R is an upper bound on the number of reloaded cache blocks and BRT is a cache block reload time.

Among the cache-aware schedulability analyses, Altmeyer et al [45] proposed ECB/UCB-Union Multiset approaches. These approaches account for a more precise number of nested preemptions that can occur during a resource access, comparing to the first exact analyses: ECB/UCB-Union appraoches [91], [52] which consider that the CRPD can be computed by intersecting the *useful cache blocks* and *evicting cache blocks.*

Minimizing cache overhead using limited optimal preemption point placement algorithm using dynamic programming is presented in [30]. The authors have proposed a novel method to calculate the CRPD taking into account the selected preemption points resulting in greater accuracy. Complementary work by Bril et al [40] recently integrated CRPD costs into fixed priority preemption threshold schedulability analysis for taskset with arbitrary deadlines. Optimal priority thresholds are assigned via a CRPD cost minimization.

Both approaches presented in [116, 121] adopt a FP scheduler using deferred preemptions, with the common goal of reducing the preemption overhead by properly placing the preemption points, these approaches uses heuristic strategy for the placement of preemption points.

All researches which are described before consider only single core platforms. When considering multiprocessor scheduling, the allocation problem must be taken into account. For example, in global scheduling, a single job may execute onto more than one core, if it is preempted, leading to higher CRPD.

Authors in [38] studied the schedulability of the DPM under global EDF. In this work, a pseudo-polynomial time schedulability test has been proposed for limited-preemption scheduling under global multiprocessor platforms. In [31] authors propose schedulability analysis for Global Fixed Priority Scheduling with Deferred Preemption for identical cores platforms. They showed that the algorithm introduced in [49] is not optimal in the multiprocessor case.

When considering partitioned scheduling, the schedulability analysis of single core described above can be used to ensure the respect of real-time constraints as a single core level. However, the allocation problem is NP-HARD as it is a particular case of the bin-packing problem. Authors in [31] addressed the allocation problem for partitioned multiprocessor systems using deferred preemption. They used the First-Fit (FF) heuristic to allocate tasks to different processors.

Their experimental evaluation showed that partitioned scheduling using deferred preemption provides significantly improved performance over fully preemptive partitioned scheduling and non-preemptive partitioned scheduling.

Contentions in multicore systems

Contention on memory resources was the subject of many research works [19, 23, 37, 50, 90]. Some of them propose new execution models making use of pre-fetching techniques [90]. It has been shown in [119] that these techniques improve the cache/scratchpad locality and reduce in average worst-case execution times.

Another possibility to minimize contention in the shared memory resources is to decouple the memory requests from the actual application execution, so to guarantee that these requests are performed exclusively in isolation. This can easily be ensured by using a Time Division Multiple Access (TDMA) based protocol to enforce timing isolation among all the tasks in the system. The AER model proposed in [39] follows the same approach, with the aim of increasing the predictability of applications executing on COTS-based platforms.

PREM was introduced in [61] to co-schedule both memory requests from CPU and I/O with computations on uniprocessor platforms with multi-level caches. In this model, tasks are modeled in two phases: (i) a memory phase where all data are transferred from/to the main memory to/from a local memory, and (ii) a pure execution phase where the loaded data are processed. The model greatly reduces the variability of memory-contention latency by explicitly controlling memory accesses during memory phases. In [50], PREM has been extended to partitioned multicore/processor platforms, where isolation is provided through a coarse-grained TDM memory schedule. The scheduling policy of each core favors the priority of its pending memory phases above computation phases in order to ensure better utilization of the TDMA slots. The AER model is a generalization of the PREM model from [61]. It introduces two main advantages: (i) contention on memory resources is avoided by requiring memory phases to execute exclusively, by having only one memory phase (A or R) running at any time instance. (ii) the task's parallel execution in each core is allowed since the memory phases are decoupled from the computation phase.

In [24], Becker et al. presented an approach to time-triggered scheduling for automotive *runnables* on a many-core platform. Memory bank privatization is used to avoid contention between memory accesses from different cores. Each core has a private memory bank and runnables are assumed to have read-execute-write phases (AER model). The scheduling algorithm executes each runnable non-preemptively, and ensures that accesses to the shared memory bank made in read and write phases do not overlap, thus avoiding contention on the shared bus. The task allocation problem is formulated as an Integer Linear Programming (ILP) problem, whose solution is the optimal time-triggered schedule for the on-core execution as well as for the access to shared memory.

Recent research on memory-centric scheduling is presented in [12]. The authors proposed a fixed-priority memory-centric scheduler for predictable memory

management on COTS multiprocessor platforms without the need for any hardware support. The work in [10] focuses on bus contention for the 3-phases task models and assumes First-Come First-Served (FCFS) bus arbitration. Works in [10, 12, 24] are the closest to the first part of our work.

Several task models have been proposed to express data dependency within real-time task, most of them are based on DAGs, *e.g.*,[20, 26, 29, 46]. Nodes represent subtasks, and edges define communications and precedence constraints between nodes. In this manner, a subtask becomes ready for execution as soon as all its communications and precedence constraints are satisfied. In such model, memory transfers can take place on two different levels: (i) between cores and main memory and/or (ii) between cores' local memories.

Partitioned scheduling is generally more adapted to this type of application than global scheduling. In fact, in global scheduling tasks may migrate from one core to another at any time, and this implies large migration overheads, which are much greater in architectures with scratchpads since both needed data and code of the subtask have to migrate.

In partitioned scheduling, all subtasks of all DAGs are first allocated on cores, and then a separate scheduler is used for each core. Therefore, the problem is transformed into an allocation problem plus a number of separate and independent scheduling problems.

A number of partitioning algorithms that map dependent tasks to cores have been presented in the literature. Most of them are heuristics and operate on a single DAG task. They allocate each node of a DAG to a processor or a core. In [7], a partitioning heuristic and graph reduction techniques are proposed for DAG tasks. They consider identical cores and fixed-priority preemptive schedulers. In this paper, we propose an ILP formulation to compute the DAG task mapping that minimizes the communication delays and allows us to turn-off maximum communications.

The authors of [14] have studied the probabilistic response time analysis of DAG tasks on multicore platforms using partitioned fixed-priority scheduling. They proposed a priority assignment algorithm at the subtask level to define the execution order between different nodes from the same graph in order to reduce the response time of the entire DAG task while considering communication times for the subtasks scheduled on different cores. Although this work tries to minimize the inter-subtask communications costs, they do not consider scratchpads nor communication scheduling.

Another effective technique to schedule DAG tasks on multicore platforms is to assign intermediate deadlines and offsets to subtasks in order to enforce the precedence constraints. By completing the deadline and offset assignments, a DAG task is transformed into a set of independent subtasks. Two heuristic algorithms are popular: *Fair distribution* and *Proportional distribution*. These techniques were used in [11, 15, 20, 44].

3.5 Conclusion

In this chapter, we first presented our main problem. We show that it is of a paramount importance to focus on reducing the runtime overheads, in particular shared resource contention, in order to increase temporal predictability on multicore platforms. We give the different sources of temporal unpredictability, and we detail each of the sources. Secondly, we reviewed the different related works in the literature. In the rest of this thesis, we will present three main contributions to improving temporal predictability in multicore real-time systems by mitigating contention on shared resources. We introduce a new task-graph model based on the AER model and a set of processor-memory co-scheduling algorithms and their corresponding schedulablity analysis.

Part II

Contributions

Chapter 4

Allocation of Real-time Tasks Onto Identical Core Platforms Under Deferred Fixed Preemption-point Model

Contents

4.1 Introduction 37
4.2 System model 37
4.2.1 Task model 37
4.2.2 Architecture model 38
4.3 Limited Preemption analysis for single-processor . . 39
4.3.1 Maximum non-preemptive execution-time 39
4.3.2 Selection of effective preemption points 40
4.4 Task allocation 40
4.4.1 Enumerating algorithm 41
4.4.2 Branch and Bound 42
4.4.3 Computational complexity 46
4.4.4 Allocation heuristics 47
4.5 Results and discussions 48
4.5.1 Task generation 48
4.5.2 Simulation results and discussions 48
4.6 Conclusion 52

4.1 Introduction

In this chapter, we adress the problem of allocating a set of real-time tasks on m-identical cores with the goal of reducing the preemption cost. We use the deffered fixed preemption point model to represent each task. This model allows for more predictability and help us to reduce shared memory contention. In the first part of this chapter, we present the task and the architecture models, then we present the different approaches proposed for task allocation. We propose enumerative and branch-and-bound optimal algorithms, along with techniques to reduce their computation time. Further, we propose a set of heuristics to solve the same problem.

4.2 System model

4.2.1 Task model

Let $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of n tasks. Each task τ_i is a (infinite) sequence of jobs. Each task τ_i is characterized by $(\Gamma_i, \Lambda_i, \mathsf{D}_i, \mathsf{T}_i)$ where :

- $\Gamma_i = \{\gamma_i^1, \ldots, \gamma_i^{\mathsf{np}_i+1}\}$: is the *basic block list*. A basic block $\gamma_i^j \in \Gamma_i$ is a sequential non-preemptive chunk of code of task τ_i. That is, once a basic block starts executing, it can not be preempted by any other higher priority task. Each block γ_i^j is characterized by its worst-case execution time denoted by $\mathsf{C}(\gamma_i^j)$. There are $\mathsf{np}_i + 1$ basic blocks in a task, where np_i is the number of preemption points.

- $\Lambda_i = \{\lambda_1, ..., \lambda_{np_i}\}$: is the list of preemption points. Preemption point $\lambda \in \Lambda_i$ is the boundary between blocks γ_i^λ and $\gamma_i^{\lambda+1}$. $\mathsf{C}(\tau_i, \lambda)$ is the cost that must be taken into account when preempting task τ_i between basic blocks γ_i^λ and $\gamma_i^{\lambda+1}$.

- D_i: is the task's relative deadline.

- T_i: is the task period. We consider sporadic tasks.

In this part, we consider constrained deadlines, that is $\mathsf{D}_i \leq \mathsf{T}_i$. Tasks are considered to be independent.

The goal here is to select a subset of Λ_i, denoted by $\overline{\Lambda}_i = \{\overline{\lambda}_i^1, \ldots, \overline{\lambda}_i^s\}$, for every task τ_i, such that preemption is allowed only at preemption points in $\overline{\Lambda}_i$, while meeting all deadlines and minimizing preemption costs. Preemption points in $\overline{\Lambda}_i$ are called *effective-preemption points*.

Let $s = |\overline{\Lambda}_i|$ be the number of selected preemption points. The task is then divided into a set of $s + 1$ *non-preemptive regions* $\mathsf{NPR}_i^1, \ldots, \mathsf{NPR}_i^{s+1}$. A non-preemptive region is the union of consecutive non-preemptive blocks between which no preemption point has been selected. They can be expressed as follows:

$$\forall k = 1, \ldots, s \;\; \mathsf{NPR}_i^k = \bigcup_{j=\overline{\lambda}_i^k}^{\overline{\lambda}_i^{k+1}} \gamma_i^j \;\text{ and }\; \mathsf{NPR}_i^{s+1} = \bigcup_{j=\overline{\lambda}_i^k}^{\mathsf{np}_i+1} \gamma_i^j \tag{4.1}$$

Clearly, for any $\lambda \notin \overline{\Lambda}_i$, $\mathsf{C}(\tau_i, \lambda) = 0$. We define the worst-case execution time of a non-preemptive region to the sum of the execution times of its blocks, plus the preemption cost of a preemption before:

$$C(\mathsf{NPR}_i^k) = \sum_{j=\overline{\lambda}_i^k}^{\overline{\lambda}_i^{k+1}} C(\gamma_i^j) + \mathsf{C}(\tau_i, \overline{\lambda}_i^{k-1}) \tag{4.2}$$

(without loss of generality we assume that the cost of $\mathsf{C}(\tau_i, \overline{\lambda}_i^0)$ is equal to 0).

Therefore, the total execution time of task τ_i, including the cost of preemption, can expressed as:

$$\mathsf{C}(\tau_i, \overline{\Lambda}_i) = \sum_{j=1}^{\mathsf{np}_i+1} \mathsf{C}(\gamma_i^i) + \sum_{\lambda \in \overline{\Lambda}_{\tau_i}} \mathsf{C}(\tau_i, \lambda) \tag{4.3}$$

We define by $\mathsf{NPR}_i^{\max}$ the non-preemptive region of τ_i having the largest execution time and we determine the task utilization as a function of the selected preemption points as follows:

$$u_i(\overline{\Lambda}_i) = \frac{\mathsf{C}(\tau_i, \overline{\Lambda}_i)}{\mathsf{T}_i} \tag{4.4}$$

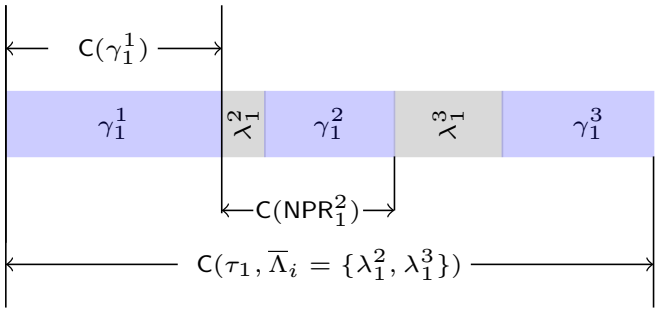


Figure 4.1: Example of parameters for a task with 3 basic blocks and 2 *effective-preemption points*.

Example 1 Figure 4.1 illustrates an example of a task with 3 basic blocks and 2 preemption points. We assume that all preemption points are effective. On the figure, we show the blocks, the non-preemptive regions and the execution times.

4.2.2 Architecture model

In this contribution, we consider a platform composed of m identical cores with partitioned scheduling. Tasks are allocated (partitioned) to the available cores

before execution. As mentioned previously, compared to global scheduling, partitioned scheduling reduces the overhead due to task migrations. It also simplifies the analysis, because it transforms the scheduling problem on m-identical core platform to an allocation problem and m independent single-processor scheduling problems, for which well-known efficient preemption-point selection techniques exist.

4.3 Limited Preemption analysis for single-processor

In this section, we review existing techniques to select preemption points, as well as schedulability analysis for a set of deferred preemption real-time tasks on a single core platform. The schedulability analysis is done in two steps: (i) first, it computes the maximum feasible non-preemptive execution time, (ii) then it selects preemption points using the algorithm proposed in [55].

4.3.1 Maximum non-preemptive execution-time

Algorithm 1 computing the length of the non-preemptive region

Require: $\mathcal{T}$: Task set
1: $\mathsf{deadlines} = \mathsf{compute\_and\_sort\_absolute\_deadlines}(\mathcal{T})$
2: $\mathsf{slack} = \mathsf{deadlines}[0] - \mathsf{dbf}(\mathcal{T}, \mathsf{deadlines}[0])$
3: **for** ($\forall d \in \mathsf{deadlines} - \{\mathsf{deadlines}[0]\}$) **do**
4: $\quad \mathsf{check\_feasibility}(d, \mathsf{H})$
5: $\quad \mathsf{slack} = \min(\mathsf{slack}, d - \mathsf{dbf}(\mathcal{T}, d))$
6: $\quad$ **if** ($\mathsf{slack} < 0$) **then**
7: $\quad\quad$ **return** false
8: $\quad$ **end if**
9: $\quad$ **if** ($d = \mathsf{D}_j$, for some task $\tau_j \in \mathcal{T}$) **then**
10: $\quad\quad \mathsf{Q}(\tau_j) = \mathsf{slack}$
11: $\quad$ **end if**
12: **end for**
13: **return** true;

Let $\mathsf{Q}(\tau_i)$ be the largest non-preemptive interval for task τ_i. It represents the maximum time that task τ_i may execute non-preemptively without violating any timing constraint. Baruah et al. in [94] have proposed techniques to calculate $\mathsf{Q}(\tau_i)$ for EDF, similarly in [80] for fixed priority. The algorithm to compute $\mathsf{Q}(\tau_i)$ for EDF is reported in Algorithm 1.

The algorithm starts by listing all deadlines in the interval $[0, \mathsf{H}]$ (line 1). Then computes the slack as the maximum difference between each deadline and the demand bound function, until the relative deadline of each task. If the slack is negative (Line 6) for some deadline, then the taskset is not schedulable under EDF.

4.3.2 Selection of effective preemption points

At this level, we assume that $\mathsf{Q}(\tau_i)$ has been already computed, and the goal of this second step is to select effective preemption-points. Bertogna et al.

Theorem 3 (Bertogna et al. [55]) *A task set $\mathcal{T}$ is schedulable if:*

$$\forall \tau_i \in \mathcal{T}, \;\; \mathsf{C}(\mathsf{NPR}_i^{\max}) \leq \mathsf{Q}(\tau_i) \tag{4.5}$$

[55] proposed a sufficient schedulability test (Theorem 3): a task set is schedulable if the execution time of any non-preemptive region is less than the maximum non-preemptive interval.

Several solutions may verify Condition (4.5). Bertogna's algorithm based on dynamic programming selects a set of effective preemption-points with the goal of optimally reducing the overall preemption overhead [55].

1. We noticed that the optimality of preemption points selection algorithm [55] leads to a lower preemption cost for a more relaxed non-preemptive interval. This property is used to prove Theorem 5 in Section 4.4.

Note that Condition (4.5) is necessary and sufficient under EDF [55].

4.4 Task allocation

In this section, we present how tasks can be allocated to cores to reduce the overall preemption costs. Let $\mathcal{T}$ be a set of n tasks to allocate on m identical cores. In the rest of this section, we will first present two exact algorithms: (i) an enumerating algorithm able to eliminate branches and solutions either for schedulability or optimality concerns and (ii) an exact algorithm based on Branch and Bound (BB). Further, we present a set of heuristics to solve the allocation problem.

Our allocation algorithms (exact and heuristic) use a list of not yet-allocated task ordered according to a given criterion. The algorithm selects a task and a core, and attempts to allocate the selected task to the selected core. According to the state of the not yet-allocated task list, the current solution can either be called an *allocation* or a *branch*.

Definition 1 Let $\mathcal{T}$ be a set of n tasks to be allocated onto m identical cores. Assignement $\mathcal{S} : \mathcal{T} \rightarrow P \cup \{\mathsf{n_a}\}$ is a mapping function defined by:

$$\mathcal{S}(\tau_i) = \begin{cases} \mathsf{p} & if \tau_i \text{ is allocated to core } \mathsf{p} \in P \\ \mathsf{n_a} & otherwise \end{cases} \tag{4.6}$$

$\mathcal{S}(\,\cdot\,)$ is called an allocation, if: $\forall \tau_i \in \mathcal{T}, \mathcal{S}(\tau_i) \neq \mathsf{n_a}$, otherwise, it is called a branch.

We define $Alloc : \mathcal{T} \rightarrow \{\mathcal{S}_1(\,\cdot\,) \ldots \mathcal{S}_x(\,\cdot\,)\}$, as the set of x possible allocations, where x is a finite number of allocations.

Definition 2 Let $\mathcal{S}_k(\cdot)$ be an allocation (resp. a branch). We denote by $\mathsf{cost}(\mathcal{S}_k(\cdot))$ the preemption cost of allocation (resp. branch) $\mathcal{S}_k(\cdot)$. It can be computed as follows:

$$\mathsf{cost}(\mathcal{S}_k(\cdot)) = \sum_{j=1}^{m} \sum_{\tau_i \in \mathcal{T}_j} \sum_{\overline{\lambda}_i^l \in \overline{\Lambda}_i} \frac{\mathsf{C}(\overline{\lambda}_i^l)}{\mathsf{T}_i} \tag{4.7}$$

4.4.1 Enumerating algorithm

The exact enumerating algorithm explores the space solution by solution. It is able to either cut a branch or to eliminate an allocation and preserve the optimal solution. An overview of our enumerating procedure is disclosed in Algorithm 2.

Algorithm 2 generate_evaluate_solutions($\mathcal{T}, \mathcal{P}, \mathcal{S}_{\mathsf{curr}}$)

```
1: if (T = ∅) then
2:     if (cost(S_curr) < cost(S_best)) then
3:         S_best = S_curr
4:         return
5:     end if
6: end if
7: τ_i = select the shortest relative deadline task from T
8: for ( p ∈ P ) do
9:     allocate τ_i to p for the current allocation S_curr
10:    if (schedulable(S_curr, τ_i, p)) then
11:        S_new = S_curr
12:        generate_evaluate_solutions(T \ τ_i, P, S_new)
13:    end if
14: end for
```

The algorithm generates all allocations recursively, it takes as input the set of all tasks in $\mathcal{T}$ sorted by relative deadlines in increasing order, the set of cores $\mathcal{P}$ and the current branch/allocation, and it returns the solution with the minimum preemption cost $\mathcal{S}_{best}$. In the beginning, it selects a task and a core and it tries to allocate the selected task to the selected core (Line 9). Therefore, it tests the schedulability (Line 10) for the concerned core, using Algorithm 3. Further, the algorithm removes the studied task from $\mathcal{T}$, and executes the recursive call (Line 12) for the new branch/allocation $\mathcal{S}_{new}$. If this test fails then, the branch/allocation is aborted. Once $\mathcal{T}$ is empty, the algorithm saves the best solution and continues to evaluate the next one. The algorithms repeats the latter operations for every task on all cores.

Algorithm schedulable (Algorithm 3), which tests the schedulability of the selected core p, uses properties of algorithm 1 and Property 1. It takes as input the selected task, the concerned core p and the task set already allocated to p. First, it tests the schedulability using a fast necessary utilisation based-test (Line 1). If the test is successful, then it computes the maximum of the *non-preemptive interval* of tasks already allocated to p using Algorithm 1 (Line 2). The latter

checks the schedulability until the hyper-period using dbf-based test (lines 5-8 in Algorithm 1). When schedulability test of algorithm 1 is performed then, our algorithm selects the effective preemption points by invoking the algorithm presented in [55] to determine the *non-preemptive regions* only for the studied task τ_i (Line 4). If condition (4.5) is respected during all the effective preemption points selection process, then, the algorithm updates the total execution time of the task τ_i, including the cost of preemption (Lines 6 and 7). If these tests fail, then the algorithm aborts on fail.

Therefore, when adding a new task to the current branch, the already computed maximum non-preemptive region and the selected points for the already allocated tasks do not need to be recomputed.

Algorithm 3 schedulable($\mathcal{S}_{\text{curr}}, \tau_i, \text{p}$)

```
1: if (fast_utilization_test(p)) then
2:     res_Q = compute_max_length_NPR(S_curr) using Algo 1
3:     if (res_Q) then
4:         res_npr = compute_NPR(τ_i, Q(τ_i)) using Algo in [55]
5:         if (res_npr) then
6:             update_C(τ_i)
7:             update_total_cost(p)
8:             return true
9:         end if
10:    end if
11: end if
12: return false
```

Optimality of the enumeration algorithm

To prove the optimality of the enumeration algorithm, we prove Theorem 4 below.

Theorem 4 *The enumeration algorithm explores all schedulable allocations and selects the optimal one.*

Proof 1 The proof derives from the recursive structure of the algorithm. For every task, all possible cores are tried (Line 8), and the algorithm invokes itself every time with a smaller set $\mathcal{T}$ (Line 12). Moreover, all schedulable solutions are explored at (Line 1), thus the best solution is selected.

4.4.2 Branch and Bound

In this section, we present a recursive algorithm based on branch and bound (Algorithm 4), and prove its correctness.

The algorithm uses a set of non allocated tasks $\mathcal{T}$ in the current branch/allocation $\mathcal{S}_{curr}$ and a list $\mathcal{L}$ of not-yet-finished branches. At each iteration, it selects a branch and tries to allocate the shortest relative deadline task in $\mathcal{T}$ (selected

in Line 8) to every core in the platform. Thus, it creates m new branches at each iteration.

For each new branch, the schedulability is tested : (i) without a complete evaluation of the branch (allocation) using Theorem 5 and Lemma 2 detailed below (Line 10); (ii) If not possible, the maximum non-preemptive region length is evaluated and the *effective-preemption points* are selected only for the selected task (Line 11) using dbf-based test according to Algorithm 3. If these tests fail, then the branch is discarded. In the opposite case, the branch is added to the list of not-yet-finished branches $\mathcal{L}$ (Line 12). Once, the non-yet-allocated task list $\mathcal{T}$ for the current branch is empty, the algorithm compares the allocation to the best known solution. If it improves it, the solution is saved and the lower-bound is updated (Lines 3-5). The latter is used to eliminate all branches having a preemption cost greater than the new lower-bound (Line 17).

While the not-yet-finished branches $\mathcal{L}$ is not empty, the algorithm selects a new branch to be explored in the next iteration (Line 21) according to two alternative criteria:

- the branch having the least preemption cost is selected first (depth-first);.
- the branch having the least number of tasks in its non-yet-allocated list, is selected first (breadth-first).

In the rest of this section, we will show how schedulability can be tested without a complete evaluation (i.e. without invoking the preemption-points selection process).

Definition 3 Let $\mathcal{S}_i(\,\cdot\,)$ and $\mathcal{S}_j(\,\cdot\,)$ be two distinct allocations, (resp. branches) $(i \neq j)$.

We define the relation order $>$ as follows:

$$\mathcal{S}_i(\,\cdot\,) > \mathcal{S}_j(\,\cdot\,) \implies \mathsf{cost}(\mathcal{S}_i(\,\cdot\,)) < \mathsf{cost}(\mathcal{S}_j(\,\cdot\,)) \tag{4.8}$$

Relation $>$ orders allocations (resp. branches) according to their preemption costs. Note that in the case of equal costs, we can not judge if $\mathcal{S}_i(\,\cdot\,)$ is better than $\mathcal{S}_j(\,\cdot\,)$.

Definition 4 Let $\mathcal{S}_1(\,\cdot\,)$ and $\mathcal{S}_2(\,\cdot\,)$ be two distinct allocations. We denote by τ_i a task in allocation $\mathcal{S}_1(\,\cdot\,)$, and by τ_i' the same task in allocation $\mathcal{S}_2(\,\cdot\,)$.

We define the relation order $\gg$ as follows :

$$\mathcal{S}_1(\,\cdot\,) \gg \mathcal{S}_2(\,\cdot\,) \implies \forall i, \mathsf{Q}(\tau_i) \geq \mathsf{Q}(\tau_i') \tag{4.9}$$

The relation order $\gg$ allows to define a dominance relation between two allocations by calculating only maximum non-preemptive regions lengths $\mathsf{Q}(\tau_i)$, rather than a complete selection of preemption points.

Theorem 5 *Let $\mathcal{S}_1(\,\cdot\,)$ and $\mathcal{S}_2(\,\cdot\,)$ be two distinct allocations.* $\mathcal{S}_1(\,\cdot\,) \gg \mathcal{S}_2(\,\cdot\,) \implies \mathcal{S}_1(\,\cdot\,) > \mathcal{S}_2(\,\cdot\,)$

Algorithm 4 branch_and_bound(S_{curr}, bound)

Require: global variables: $\mathcal{L} = \emptyset$, S_{best}
1: $\mathcal{T}$ = set of non allocated tasks in S_{curr}
2: **if** ($\mathcal{T} == \emptyset$) **then**
3: **if** (cost($\mathcal{S}_{curr}$) < bound) **then**
4: $S_{best} = S_{curr}$
5: bound = cost(S_{curr})
6: **end if**
7: **else**
8: τ = select the shortest relative deadline task from $\mathcal{T}$
9: **for** ($\forall p \in \mathcal{P}$) **do**
10: **if** (not_dominated(S_{curr}, S_{best})) **then**
11: **if** schedulable(S_{curr}, τ, p) **then**
12: $\mathcal{L} = \mathcal{L} \cup \{S_{curr}[\tau$ allocated on $p]\}$
13: **end if**
14: **end if**
15: **end for**
16: **end if**
17: $\mathcal{L}$ = eliminate_branches($\mathcal{L}$, bound)
18: **if** ($\mathcal{L} == \emptyset$) **then**
19: **return** S_{best};
20: **end if**
21: S = select_the_minimum_branch($\mathcal{L}$)
22: branch_and_bound(S, bound);

Proof 2 The proof is derived from Property 1 and Definitions 3, 4. Let consider $\mathsf{Q}(\cdot)$ be the maximum non-preemptive region for a given task on allocation $\mathcal{S}_1(\cdot)$ and $\mathsf{Q}(\cdot)'$ be the maximum non-preemptive region for the same task on allocation $\mathcal{S}_2(\cdot)$. We assume $\mathsf{Q}(\cdot) > \mathsf{Q}(\cdot)'$. From Property 1, it follows that the preemption cost in allocation $\mathcal{S}_1(\cdot)$ is less than the preemption cost in allocation $\mathcal{S}_2(\cdot)$. According to Definition 3 $\mathcal{S}_1(\cdot) > \mathcal{S}_2(\cdot)$, proving the theorem.

According to Theorem 5, it is not necessary to compute the preemption points to test domination between two allocations. In fact, we can avoid computing the preemption points for the dominated solution, further reducing the execution time of the algorithm.

Lemma 2 Let consider n tasks to allocate to m processors, with: $n > m$

Any feasible solution $\mathcal{S}(\cdot)$, having at least one free processor (without any task), is dominated.

Proof 3 The proof is straightforward from Theorem 5. In fact, having a free processor implies that for any non-empty processor, a task can be selected and reallocated to the free processor, thus producing a higher maximum-non-preemptive region length. Therefore, the new solution dominates the one with an empty processor according to Theorem 5.

According to this lemma, it is not necessary to evaluate the allocations having at least an empty processor: the algorithm needs not to compute the maximum non-preemptive region length as it will not lead to optimal solution.

The theorems and lemmas described in this section are used to eliminate allocations at (Lines: 10 and 17) in Algorithm 4.

Branch and Bound optimality proof

First, we prove that all branches generated by the branch and bound algorithm having a preemption cost greater than the lower bound are dominated by the best known solution. Further, we demonstrate that our algorithm preserves the optimal solution at each branch level.

Lemma 3 Let $S(\cdot)$ be a branch in the set of not-yet-finished branches $\mathcal{L}$, and let $\mathsf{cost}(\mathcal{S}(\cdot)) > bound$. Then all solutions belonging to the branch $S(\cdot)$ have larger cost than the currect bound.

Proof 4 When new branches are explored, tasks can only be added to cores. Since tasks are sorted by deadlines in non-decreasing order, the preemption cost can only increase. Thus, the branch $S(\cdot)$ having a preemption cost greater than the lower bound, can not lead to a better solution than the best-known.

Let us demonstrate now that all branches discarded using our algorithm are not optimal.

Theorem 6 *Function* eliminate_branch *never eliminates the optimal solution.*

Proof 5 From the previous lemma, eliminate_branch eliminates all solutions with cost greater than the lower bound, and since we have already found a solution S_{best} whose cost is equal to the lower bound, then it follows that eliminate_branch cannot eliminate the optimal solution.

To better clarify how the proposed branch and bound algorithm works, we propose an example in the following section.

Example 2 Let $\mathcal{T}$ be the task set to be allocated onto 2 cores. Tasks characteristics are described in Table 4.1. In Figure 4.2, we report a sub-tree of the branches eveluated by our branch and bound algorithm. Each node in the figure describes a branch. It contains the branch identifier, its cost and the already allocated tasks for the two cores.

Before starting, the algorithm sorts the tasks by increasing relative deadline. It then starts by selecting the first task τ_4, because it has the least deadline 1277. Thus two branches are created (2 and 3 in Figure 4.2), each of cost equal to 0. For the 2^{nd} iteration, task τ_1 is selected. As branch 2 and branch 3 have the same cost (equal to 0), we select arbitrarily branch 2. Task τ_1 can be either allocated along with τ_4 onto the same core as shown in branch 5. It can also be allocated to the other empty core. In both branches, the preemption cost is equal to 0, therefore we select branch 5 as it leaves an empty core, therefore

task	D_i	T_i	Γ_i	Λ_i
τ_1	1413	1500	{212,171,344,66,249}	{ 0,46,78,14,47 }
τ_2	5673	6000	{17,54,490,101,418,74}	{0,14,94,21,74,13}
τ_3	1498	1500	{146,347,136,37,121}	{0,90,32,7,19}
τ_4	1277	1500	{17,31,43,3,24,6}	{ 0,6,8,0,3,0 }

Table 4.1: Task parameters

giving us more chances to lead to an optimal solution. The next task to allocate is task τ_3. The algorithm has the two possibilities: (i) allocate τ_3 together with τ_4 and τ_1 on the same core or (ii) allocate τ_3 alone on the empty core. In the first solution, schedulability fails, therefore without further exploring, the branch it is eliminated. In the second, the schedulability cost is equal to 0 as the task is allocated on an empty core. Further, the algorithm selects branch 7. The next task to evaluate is task τ_2. It can be either allocated with τ_4 and τ_1 on the same core, having a preemption cost of 0.008 (Allocation 9) or with τ_3 onto its core, having a preemption cost of 0.0058 (Allocation 8). The not yet allocated task list is now empty, therefore the bound is updated to 0.0058 and the allocation is saved as the best known. All branches having preemption costs greater than 0.0058 are eliminated.

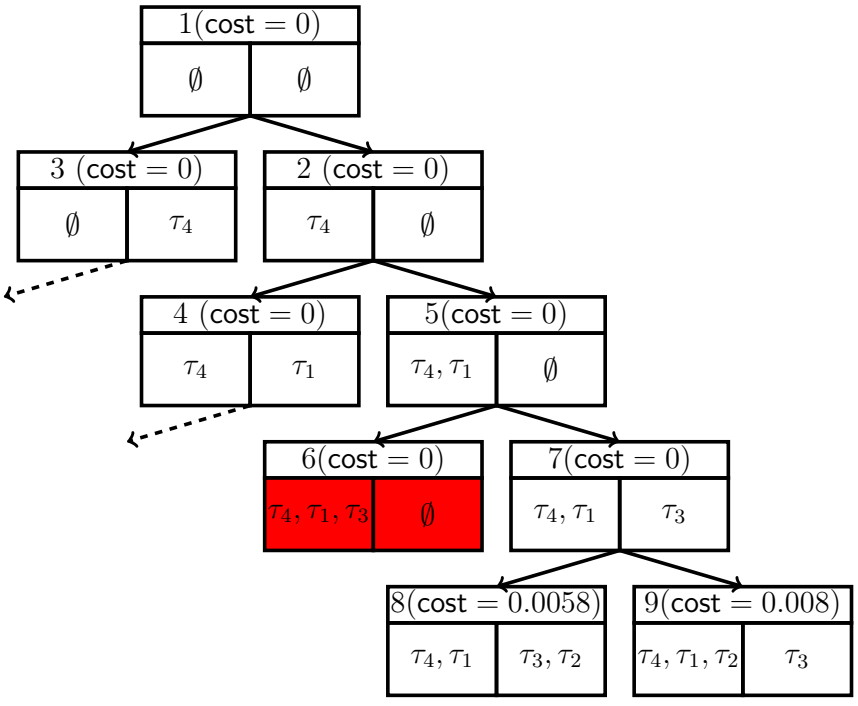


Figure 4.2: Example of branch and bound

4.4.3 Computational complexity

Regarding to runtime complexity, the proposed exact algorithms evaluate the maximum non-preemptive region length and selects the preemption points for

the selected task at every tree level except the root, yielding to a time complexity of $\mathsf{O}(n! \times m)$ in the worst case. However, the runtime of the branch-and-bound algorithm in the average case is likely less.

4.4.4 Allocation heuristics

Task allocation problem is known to be NP-hard in the strong sense. Thus, even with the proposed optimizations and dynamic programming, the complexity of finding an optimal solution is high. Therefore, we use classical bin packing heuristics as an alternative allocation.

Algorithm 5 Heuristics($\mathcal{T}$, p, alloc[FF, BF, WF], ORDER)

```
1: sort_tasks(ORDER)
2: for (∀τ ∈ 𝒯) do
3:     allocated = false
4:     sort processors for BF and WF
5:     for (∀p ∈ 𝒫) do
6:         S_curr = select tasks on p ∪ {τ}
7:         if (total_schedulable(S_curr, τ, p)) then
8:             allocated = true
9:             allocate task into core p
10:            break;
11:        end if
12:    end for
13:    if (allocated = false) then
14:        No task allocation is found
15:    end if
16: end for
17: Return tasks allocation
```

In practice, First-Fit (FF), Best-Fit (BF) and Worst-Fit (WF) operate in a similar fashion. First, tasks are sorted before the allocation according to their deadline, density, or laxity (Line 1). Then, the algorithm selects the tasks on the top of the order relation. Unlike the algorithm FF, BF and WF sort the cores by capacity. For BF the cores are sorted in a decreasing order of their utilizations, whereas in the case of WF, they are sorted in increasing order of utilization. The heuristic tries to allocate the selected tasks to the first processor. If the allocation fails (for schedulability), the next processor is investigated. When all processors are investigated and none of the allocations have been found feasible, the heuristics aborts on fail. The algorithm 5 describes FF, BF and WF heuristics. The schedulability is checked at the same time when calculating maximum-non-preemptive region length and when selecting preemption points using Algorithm "*total_schedulable*".

When tasks are sorted according to their deadline, at each allocation, total_schedulable algorithm is similar to Algorithm 3, as it is not necessary to recompute the maximum-non-preemptive region length and preemption points

for already allocated tasks, However, total_schedulable algorithm recomputes it for all other sorting mechanisms as they do not ensure that only tasks with shorter deadlines have been allocated before.

4.5 Results and discussions

In this section, we evaluate the performance of our schedulability analysis and allocation strategies. We compare our optimal algorithms against classical allocation heuristics: First Fit (FF), Best Fit (BF) and Worst Fit (WF).

Due to the complexity of computing the exact solution, and for sake of fast evaluation, we consider a hardware platform compound of 3 identical processing units, and a large set of synthetic task sets, each comprising 24 tasks.

4.5.1 Task generation

We apply our heuristics on a large number of randomly generated synthetic task sets.

The task set generation process takes as input n (the number of tasks) and $U_{\mathcal{T}}$ (the target total utilization). First, we start by generating the utilization of the n tasks by using the UUniFast-Discard [71] algorithm. Further, for every utilization u_i, we generate randomly the number of basic blocks k between 8 and 15. We generate block utilization by generating randomly k utilizations using UUniFast algorithm with total utilization equal to the task utilization.

To avoid intractable hyper-periods, the period of every task is generated randomly according to values taken from a list where the minimum is 120 and the maximum is $120,000$ by step of 500.

Further, we inflate each block utilization by the task period to generate the block execution time. Then, we generate the block preemption cost by generating a random value P between 0.1 and 0.2. P is the percentage of the block utilization that represents the block preemption cost. The first block preemption cost is equal to 0. The task deadline is generated randomly between $[0.75 \cdot \mathsf{T}_i, \mathsf{T}_i]$.

4.5.2 Simulation results and discussions

We varied the baseline utilization from 0.25 to 3.75 with a step of 0.25. In all the graphs presented in this section, each point is the average value of 100 executions. We reports the results of the schedulability and the timing complexity of optimal solutions and heuristics.

In Figure 4.3 we report the results of the schedulability of the optimal allocation algorithms (OPT) against the best fit (BF-DD), worst fit (WF-DD) and first fit (FF-DD) algorithms as a function of total utilization. For all the algorithms, the input tasks are sorted according to their relative deadlines in non-decreasing order. At low total utilization values, all algorithms are able to schedule all task sets. It is clear that any version of the optimal algorithm dominates the heuristics. As utilization increases, FF-DD and BF-DD algorithms outperform

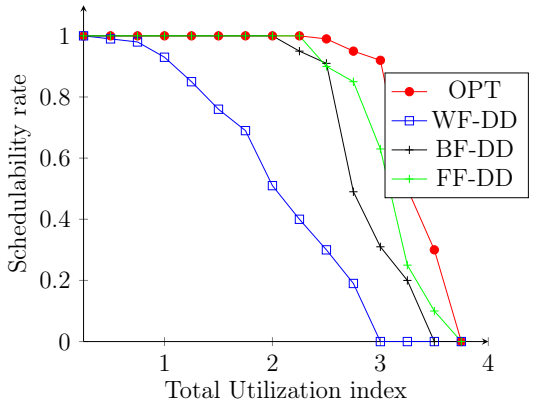


Figure 4.3: Schedulability for optimal solutions against BF, WF and FF

the WF-DD algorithm, the latter schedulability falls drastically as utilization increases.

In fact, the allocation process depends on the already allocated tasks. As in WF the tasks are allocated according to the worst-case utilization, tight deadline tasks may be allocated along with large deadline tasks. Therefore, the latter do not have enough slack to be executed non-preemptively, causing schedulability failure. In contrast, when using BF or FF algorithms, tight deadlines tasks are allocated together, allowing to have the same tight slacks, but as deadlines are closer, their execution requirements are closer (deadlines are generated as 0.75 of the task period, based on which task execution time is generated). Although we have only three cores, the optimal algorithm is able to achieve high schedulability rates even greater than the maximum number of cores.

To explain this fact, we remark that the preemption cost is inflated from the task execution time. Therefore the maximum schedulable utilization, when selecting all preemption points is equal to the number of cores, however when not selecting all preemption points, the actual utilization is less than the maximum theoretical generated utilization, which is used in the plots.

We show in Figure 4.4 a comparison between optimal solutions and heuristics as a function of the required time to complete the analysis. The required time for analysis using the enumerative algorithm is very large compared to those of the others algorithms. The optimal branch-and-bound algorithms require more time than heuristics, as expected. In fact, at any scenario, the enumerative algorithm will explore all the design space, thus it is very time-consuming, however the two branch-and-bound implementations may cut a branch without evaluation, therefore they are faster. The heuristic algorithms explore only a subset of the design space, and hence are the fastest. Algorithm OPT-P1 denotes the branch-and-bound algorithm where the next explored branch is the branch having the least cost, while in OPT-P2 the branch having the least size of the not-yet allocated tasks is selected first. Algorithm OPT-P1 has a lower average complexity compared to OPT-P2. In fact, OPT-P2 tries to find a low bound faster than OPT-P1, therefore has a better capacity to cut branches.

As the enumerative algorithm has a very high average complexity, the number of tasks in this setup has been limited to 8 tasks and the number of processors

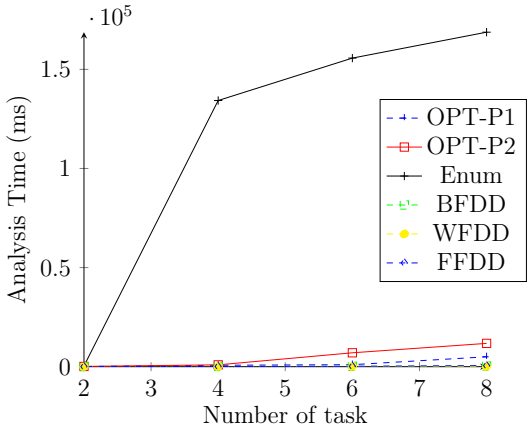


Figure 4.4: The analysis time as a function of number of task

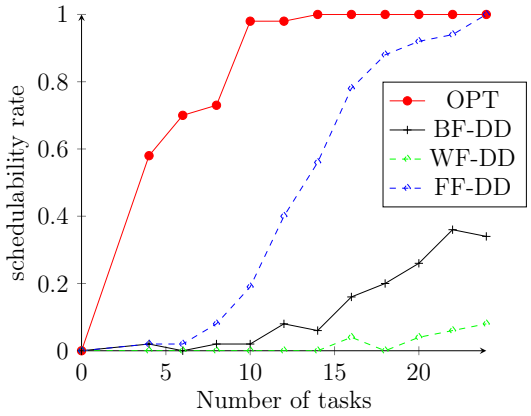


Figure 4.5: Schedulability at different taskset size.

to 3 to be able to achieve a large number of simulations.

In Figure 4.5, we evaluate the impact of the task set size on the effectiveness of the analysis to determine the task set schedulability. In this setup, the task set utilization was fixed at $U_{\mathcal{T}} = 3$, and the evaluated task set size ranges from 4 to 24. We notice that the schedulability rates is higher for task set with large number of tasks. In fact, tasks present more *effective-preemption points* in large task set, therefore the total preemption cost of each task increase allowing the execution time to be reduced, hence, schedulability increase as shown in Figure 4.5.

In Figure 4.6 and Figure 4.7 we focus on the impact of the sorting algorithms on FF, BF and WF heuristics.

Figure 4.6 reports schedulability as a function of total utilization when input tasks are sorted in increasing (respectively descreasing) order of task density. Please, notice that sorting tasks in decreasing order of density allows to achieve higher schedulability rates compared to the increasing density order. In fact, when using increasing order, FF and BF tend to allocate the small tasks, having a small density on the same core, and the more heavy tasks to be allocated in a small number of cores, thus reducing schedulability. WF presents the opposite behavior, but it also leaves the heaviest tasks to be allocated at the end, thus

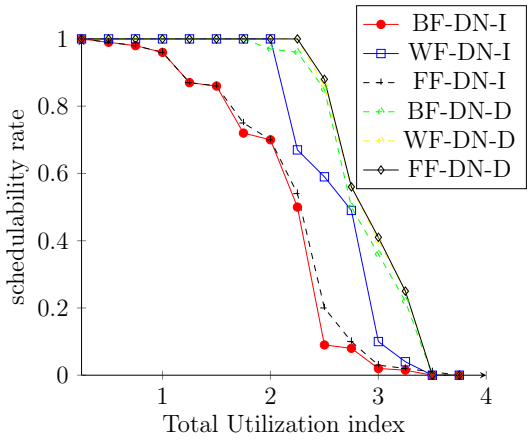


Figure 4.6: Performance of BF, WF and FF using sorted tasks by density

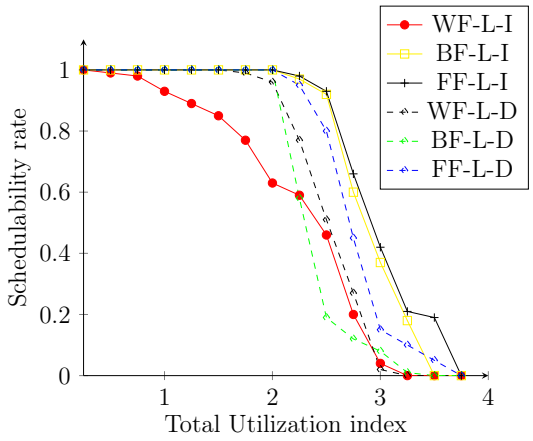


Figure 4.7: Performance of BF, WF and FF using sorted tasks by laxity

leading to schedulability failure. When ordering by decreasing order, BF, WF and FF performances have similar behavior as they start by allocating heavy tasks first, and further smaller tasks are inserted on the cores where they can be schedulable.

Figure 4.7 reports schedulability as a function of total utilization when input tasks are sorted in increasing (respectively descreasing) order of task laxity.

When ordering by increasing order, FF and BF are equal, the same performance behavior is noticed for BF and WF when tasks are sorted by decreasing order of laxity. Noticed that sorting tasks by laxity allows to have slightly better performances compared to density sorting.

4.6 Conclusion

In this chapter, we presented allocation algorithms for real-time tasks with fixed preemption points on an identical core platform. We have proposed two optimal algorithms and a set of heuristics to effectively achieve allocation while meeting all deadlines and minimizing the preemption costs. The task model used here as well as the processor and cache memory co-scheduling allow for improved predictability since the preemption occurs on predefined preemption points, thus reducing shared memory contention. The branch-and-bound implementations have shown a good compromise between computational time and the quality of the produced solution. We have presented the performances of the proposed approaches using a large set of synthetic experiments

Chapter 5

Contention-free Scheduling of PREM Tasks on Partitioned Multicore Platforms

Contents

5.1 Introduction 54
5.2 System model 54
5.2.1 Architecture model 54
5.2.2 Task model 55
5.3 Offset based processor and memory co-scheduling . . 55
5.3.1 Task-level offsets : sufficient condition 56
5.3.2 Integer-Linear-Programming based offset assignment . 57
5.4 Deadline based processor and memory co-scheduling 60
5.5 Results and discussions 61
5.5.1 Task set generation 62
5.5.2 Results of synthetic task set experiments 62
5.6 Conclusion 64

5.1 Introduction

In this chapter, we aim at avoiding contention for a set of tasks modeled using the Predictable Execution Model (PREM), i.e. each task execution is divided into a memory phase and a computation phase, on a hardware multicore architecture where each core has its private scratchpad memory and all cores share the main memory. We consider non-preemptive scheduling for memory phases, whereas computation phases are scheduled using partitioned preemptive EDF. We present in this chapter, three novel approaches to avoid contention in memory phases: (i) a task-level time-triggered approach, (ii) job-level time-triggered approach, and (iii) on-line scheduling approach. We compare the proposed approaches against the state of the art using a set of synthetic experiments in terms of schedulability and analysis time. The different approaches were implemented on an Infineon AURIX TC397 multicore microcontroller and validated using a set of tasks extracted from well-known benchmarks from the literature.

5.2 System model

5.2.1 Architecture model

In this chapter, we consider a multicore platform composed of m cores. Each core has a single local scratchpad memory. Memory copy operations between main and scratchpad memories are performed via a shared bus. The tasks explicitly trigger memory copies between main and scratchpad memories before starting the computation. In many cases, this separation of code between memory phase and computation phase can be performed automatically, for example when compiling code from high-level programming languages like Prelude [16], or by modifying existing compilers [18]. We assume that the separation between the two phases has been done either manually by the programmer, or by an appropriate code generation tool.

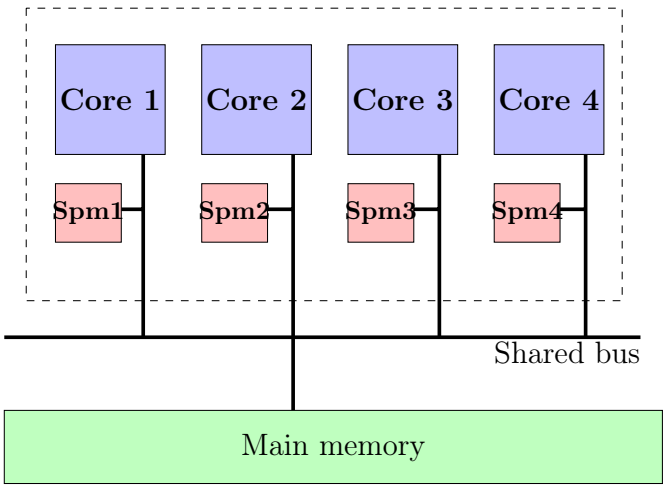


Figure 5.1: Multicore target platform.

Figure 5.1 depicts a multicore platform with 4 cores. Each core is directly connected to its own scratchpad memory and to the main memory. We assume that all memory (main memory and local scratchpads) is directly accessible to all cores via different address spaces. An example of such architecture is the Infineon Aurix TC397 [2].

5.2.2 Task model

Let $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ be a set of n periodic tasks. Each task τ_i has two phases: (i) a *memory phase* in which all data required (resp. produced) by τ_i is loaded (resp. stored) in memory, and (ii) a *computation phase* where preloaded data is processed, without any access to the main memory. The computation phase is not allowed to start before the completion of the memory phase. Therefore, task τ_i is characterized by the tuple $\tau_i = (M_i, C_i, \mathsf{D}_i, \mathsf{T}_i)$, where:

- M_i is the task worst-case memory access time. It represents an upper bound to the time during which the task τ_i perform data transfers from/to memory and/or I/O devices. Once this phase starts, it cannot be preempted.
- C_i is the task worst-case computation time. In contrast to the memory phase, the computation phase can be preempted.
- D_i is the task's relative deadline.
- T_i is the task period. We consider strictly periodic tasks.

We denote by $u_i^m = \frac{M_i}{\mathsf{T}_i}$ (resp. $u_i^c = \frac{C_i}{\mathsf{T}_i}$) the memory phase (resp. computation phase) utilization. Therefore, the task utilization is given by $U_i = u_i^m + u_i^c$ and the total utilization of task set $\mathcal{T}$ is computed as $U_{\mathcal{T}} = \sum_i^n U_i$.

Each task τ_i generates an infinite sequence of jobs, however the pattern repeats every $\mathcal{H}$ intervals. Therefore, we are interested in the set of released jobs $\mathcal{J}_i$ between time instance 0 and $\mathcal{H}$, i.e. $\mathcal{J}_i = \{j_i^0, j_i^1, \cdots, j_i^{\frac{\mathcal{H}}{T_i}}\}$. Each job j_i^l is released exactly at time instant $a_i^l = l \cdot \mathsf{T}_i$ and must complete no later than $d_i^l = a_i^l + \mathsf{D}_i$.

5.3 Offset based processor and memory co-scheduling

In this chapter, we tackle the bus contention problem by avoiding conflicting bus access altogether. In this first part, we assign offsets to memory phases so that they do not overlap at runtime, while all deadlines are met.

The task model with offsets is exemplified in Figure 5.2. We denote by ϕ_{M_h} the *memory offset* of job j_h. By design, memory phases will never compete with each other on the bus, therefore every memory phase starts its execution exactly at ϕ_{M_h} time instant from its activation a_h. The computation phase of job j_h becomes ready to execute exactly at time $\phi_{C_h} = a_h + \phi_{M_h} + M_h$, called

computation offset, regardless of the actual transfer time of the memory phase. In other words, the computation phases are activated by a timer programmed to fire an interrupt at ϕ_{C_h}. In this way, we can schedule computation phases on the different cores separately from the memory phase, using classical single core scheduler and classical single core analysis in the presence of offsets to assess schedulability.

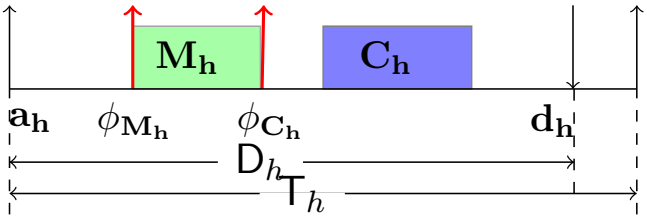


Figure 5.2: Example of task parameters.

In this chapter, we consider two types of memory phases offsets : *task-level* and *job-level* offsets. In the first, all jobs of the same task have the same offset, while in the second approach, different job of the same task might have different offsets.

5.3.1 Task-level offsets : sufficient condition

In the following, we present a technique to assign a fixed offset to the memory phase of a task, so that all jobs of the tasks will have the same offset, and all deadlines are met.

Theorem 7 (Jan Korst et al [126]) *Let τ_1 and τ_2 be two periodic tasks. τ_1 and τ_2 can be scheduled on the same core, without any overlap if and only if :*

$$\mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2) \geq C_1 + C_2 \tag{5.1}$$

Where gcd *is the greatest common divisor of* T_1 *and* T_2 .

The schedulability test of Theorem 7 allows to execute two tasks without any overlap. It is extended in to support tasks with offsets in [126] as follows :

Lemma 4 Let τ_1, τ_2 be two periodic tasks having offsets ϕ_{τ_1} and ϕ_{τ_2} respectively, τ_1 and τ_2 can be scheduled on the same core, without any overlap if and only if:

$$C_1 \leq (\phi_{\tau_2} - \phi_{\tau_1}) \bmod \mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2) \leq \mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2) - C_2 \tag{5.2}$$

We use Lemma 4 to compute the offsets of the different memory phases.

Lemma 5 A set of periodic memory phases $\mathcal{M} = \{M_1, M_2, \cdots, M_n\}$ can be scheduled without any overlap if:

$$\mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2, \cdots, \mathsf{T}_n) \geq \sum_{i=1}^{n} M_i \tag{5.3}$$

Proof 6 Since the minimum time distance between any activation time of any task τ_i and the successive activation time of another task τ_j is a multiple of the gcd of the task periods, then, if $\mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2, \cdots, \mathsf{T}_n) \geq \sum_{i=1}^{n} M_i$ all memory phases can be executed in interval of τ_i and τ_j, which proves the sufficiency of 5.3.

Theorem 8 *Let $\mathcal{T}$ be a set of n periodic tasks and let $\mathcal{M} = \{M_1, M_2, \cdots, M_n\}$ their memory phases. The time distance between any activation time of a memory phase M_i and its release time (the start time of $M_1 = 0$) so that any two memory phases do not overlap can be computed as follows:*

$$\phi_{M_j} \in \left[\sum_{i=1}^{j-1} M_i, \;\; g - \sum_{i=j+1}^{n} M_i \right] \tag{5.4}$$

Where $g = \mathsf{gcd}(\mathsf{T}_1, \mathsf{T}_2, \cdots \mathsf{T}_n)$.

Proof 7 From Lemma 8, it is sufficient that the sum of all memory phases to be less or equal to the gcd, so to schedule memory phases without overlapping. Therefore, it is sufficient to find any distribution of the bus-time equal to the gcd of task periods. By choosing the offsets (tasks are in any arbitrary order): $\phi_{M_j} = \sum_{i=1}^{j-1} M_i$ and $\phi_{M_1} = 0$ we can garantee the non overlapping of all memory phases, having Equation (5.4).

According to Theorem 8, to compute the task-level offsets, it suffices to compute the sum of the length of all memory phases. If it does not exceed the gcd of the periods of all tasks, then the offsets can be easily assigned. In this part, we order tasks in a non-decreasing order of relative deadline, therefore they get the smallest memory phase offset, to maximize the slack time to computation phases. Of course, this may be very pessimistic, and many schedulable tasks sets cannot be assigned task-level offsets in this way. A better solution will be proposed in the following.

5.3.2 Integer-Linear-Programming based offset assignment

In this section, we present a modular ILP design that is able to compute both *task-level* and *job-level* offsets.

The ILP must verify the following properties : (i) a schedule is found for all memory phases (prop1), (ii) two memory phases do not overlap on the bus (prop2), (iii) each memory phase receives sufficient bus-time to complete (prop3), and (iv) the schedulability of the different computation phases is granted (prop4).

Property (prop1): task-level and job-level offsets

The output of our ILP is the set of all memory phases offsets, or *fail* if no solution can be found. Our ILP is optimal, therefore if a solution exist, it will not fail. We define decision variable ϕ_{M_j} as the offset of the memory phase for job j. Our ILP builds $\sum_{\tau_i \in \mathcal{T}} \frac{\mathcal{H}}{\mathsf{T}_i}$ decision variable of type ϕ_{M_j}, representing offsets of all jobs, verifying therefore Property prop1.

Our ILP is able to build both task-level and job-level offsets by manipulating offsets decision variables as follows:

1. **Task-level offsets**. To enforce the ILP to select task-level offset, we set the offset decision variable of all jobs of the same task to be equal, as in Equation (5.5). One may even replace all appearances of ϕ_{M_j}, for every job of task τ by a single decision variable ϕ_τ, avoiding therefore to generate a variable per job per task. For sake of simplicity, and without loss of optimality for task-level offsets, we consider the first option, that is:

$$\forall i \in n, \forall j \in [1 \cdots \frac{\mathcal{H}}{\mathsf{T}_i}], \phi_{M_{j+1}} - \phi_{M_j} = 0 \tag{5.5}$$

2. **Job-level offsets**. To enforce the ILP to select job-level offset, we relax the constraints of Equation (5.5), therefore the ILP is free to select different offsets for different jobs of the same task.

Properties (p*rop*2) and (p*rop*3): Non-overlapping and sufficiency constraints

We introduce new constraints to verify Property prop2. The memory phase of job j starts at ϕ_{M_j} and completes exactly at $\phi_{M_j} + M_j$. During this interval, we must ensure that memory phase of any other job h cannot start or complete within $[a_j + \phi_{M_j}, a_j + \phi_{M_j} + M_j)$. Therefore, job h memory interval either completes before $a_j + \phi_{M_j}$ (case 1), **or** starts later to $a_j + \phi_{M_j} + M_j$ (case 2). Therefore, for every couple of jobs of different tasks, we introduce 2 new binary decision variables x_{jh}, and y_{jh} verifying which case the ILP solver has selected, as defined in Equations (5.6), (5.7).

$$x_{jh} = \begin{cases} 1 \ , & if \quad a_h + \phi_{M_h} + M_h \leq a_j + \phi_{M_j} \\ 0 \ , & otherwise \end{cases} \tag{5.6}$$

or

$$y_{jh} = \begin{cases} 1 \ , & if \quad a_j + \phi_{M_j} + M_j \leq a_h + \phi_{M_h} \\ 0 \ , & otherwise \end{cases} \tag{5.7}$$

Due to the mutual exclusion of both cases, the above constraints are not linear. Equation (5.8) shows linearization of these constraints:

$$\begin{aligned} &(a_j + \phi_{M_j}) - (a_h + \phi_{M_h} + M_h) - R \cdot x_{jh} \leq 0 \\ &(a_j + \phi_{M_j}) - (a_h + \phi_{M_h} + M_h) + R \cdot (1 - x_{jh}) \geq 0 \\ &(a_h + \phi_{M_h}) - (a_j + \phi_{M_j} + M_j) - R \cdot y_{jh} \leq 0 \\ &(a_h + \phi_{M_h}) - (a_j + \phi_{M_j} + M_j) + R \cdot (1 - y_{jh}) \geq 0 \\ &x_{jh} + y_{jh} = 1 \\ &\text{where } R \text{ is a large positive integer.} \end{aligned} \tag{5.8}$$

These constraints do not only allow to verify Property prop2, but as well Property prop3. It enforces all memory phases other than the one of job j to start no earlier to the completion of the memory phase of j.

Property (prop4) **(Feasibility constraints)**

We guarantee the respect of timing constraints for the EDF scheduling on every core, by incorporating the offsets induced by the memory phases to the classical processor demand schedulability analysis within our ILP. The exact condition for a set of jobs to be scheduled by EDF within the interval $I = [t_1, t_2]$ is that the cumulative computation time of all jobs with release time greater than or equal to t_1 and deadline less than or equal to t_2 not exceed the length of the interval $|I|$.

In order to avoid checking the schedulability for all values of t_1 and t_2 with a high complexity, we check only the intervals where the demand function might change. That is, we verify all intervals where t_1 is selected in the set of computation release offsets *i.e.* $t_1 \in \{\forall j, a_j + \phi_{M_j} + M_j\}$ and t_2 is selected from the set of absolute deadlines *i.e.* $t_2 \in \{\forall j, d_j\}$. We denote t_1 as $t(j)$ (referring to the start time of computation phase of job j) and t_2 as $d(h)$ (referring to the absolute deadline of job h).

Therefore, for every couple of $t(j)$ and $d(h)$ and for every job l, we introduce the decision variable $z_{l,j,h}$ that expresses if job l is released and has its absolute deadline in the interval $[t(j), d(h)]$. As we consider partitioned scheduling, j and l must be allocated to the same core without loss of optimality:

$$z_{l,j,h} = \begin{cases} 1, & \text{if } t(j) \leq a_l + \phi_{M_l} + M_l \textbf{ and } d(h) \geq d_l \\ 0, & \text{otherwise} \end{cases} \tag{5.9}$$

As before, we linearize the evaluation of $z_{l,j,h}$, and replace $t(j)$ by its value as follows:

$$\begin{aligned} &(a_l + \phi_{M_l} + M_l) - t(j) - R \cdot z_{l,j,h} \leq 0 \\ &(a_l + \phi_{M_l} + M_l) - t(j) + R \cdot (1 - z_{l,j,h}) \geq 0 \end{aligned} \tag{5.10}$$

The feasibility can be tested for all the intervals by computing the cumulative execution time for every couple of $t(j)$ and $d(h)$ using the following constraints:

$$\begin{aligned} &\forall t(j) \in \{\forall j, a_j + \phi_{M_j} + M_j\} \\ &\quad \forall d(h) \in \forall h, d_h \\ &\qquad \sum_l C_l \cdot z_{l,j,h} \leq (d(h) - t(j)). \end{aligned} \tag{5.11}$$

Jobs are sorted by deadlines, so that every job is considered in only a single couple, reducing the number of the constraints without loss of optimality.

Objective function. A realisable solution allows to respect all the constraints defined in our ILP. Therefore, it is not mandatory to our ILP to define an objective function as any realisable solution can be accepted from real-time perspective. Therefore, our objective function can be set to 0, so that solvers will stop at the first solution respecting all constraints making the ILP faster. We can as well set the objective function to minimize as much as possible the memory phases offsets, as follows:

$$\text{Minimize} \sum_{i \in n, j \in J_i} \phi_{M_j} \tag{5.12}$$

Once the ILP has been formulated, it is submitted to the CPLEX ILP-solver ([4]).

5.4 Deadline based processor and memory co-scheduling

The ILP-based approaches proposed in the previous section find the optimal solution when a feasible one exists. Unfortunately, they suffer from high computational complexity. For this reason, we followed a different approach to manage contention on the memory bus. The basic idea is to have a centralized yet partitioned scheduler, located on one of the cores, which performs the data transfers from/to local scratchpads according to an on-line scheduling algorithm. We consider data transfers as non-preemptive tasks, each one with a period and an *intermediate deadline*, to be scheduled on the single resource "bus" by a non-preemptive on-line scheduler (EDF). These assigned intermediate deadlines δ_i are used as offsets for scheduling computation phases on the cores.

The main difference with the ILP-based scheduling of memory phases is that the ILP assigns "slots" to memory phases, each slot is an interval of size equal to the length of the data transfer, while the deadline-based approach assigns interval which may be larger than the length of the data transfer, and memory phases are scheduled as non-preemptive tasks with an on-line single core policy. The system is correct, if all memory phases respect their intermediate deadline, and that the computation phase respect the task deadline with the intermediate deadline of memory phase as an offset.

For every phase, the intermediate deadline δ_i must be greater than the memory phase duration M_i, considered as a lower bound, and no greater to $\mathsf{D}_i - \mathsf{C}_i$, considered as an upper bound. Setting the intermediate deadline to the lower bound will enforce every memory phase to start its execution at its arrival, otherwise it misses its deadlines. Setting the deadline to the upper bound $\mathsf{D}_i - \mathsf{C}_i$ will enforce the computation phase to start its execution at its activation. The problem is to find a compromise between the slack time assigned to both memory and computation phase for every task.

Algorithm 6 implements a binary search technique to assign the intermediate deadlines to the memory phases. Therefore, lower bound lb_i and upper bounds ub_i are computed for every task (Line 2). Then, the algorithm sets the intermediate deadlines to the middle between lb_i and ub_i (Lines 4-8). Further, the non-preemptive schedulability test for EDF is applied [124], to assess the feasibility on the bus (Line 9). If the set of memory phases is schedulable on the bus using the computed intermediate deadlines, schedulability is checked on all cores, using Pellizzoni and Lipari [96] approximate test with pseudo-polynomial complexity (Line 11). In the case of success, Algorithm 6 exists on SUCCESS. If the schedulability on cores fails, it modifies only the intermediate deadlines of the tasks that are allocated on the cores that were deemed not schedulable by assigning shorter deadlines to the memory phases (moving the upper bounds to the computed intermediate deadlines - Line 14). Further, we iterate until the

system is schedulable on both bus and cores. If the schedulability fails on the bus, the intermediate deadlines are increased by setting the lower bounds to the intemediate deadlines (Line 17). When the upper bounds and the lower bounds are equal, the binary search fails to find a feasible schedule for both bus and cores and Algorithm 6 exits on FAIL.

Algorithm 6 Binary search guided by core schedulability

1: **Input** $\mathcal{T}$: set of tasks
2: $\forall \tau_i \in \mathcal{T}$: $\mathtt{lb}_i \leftarrow M_i$; $\mathtt{ub}_i \leftarrow \mathsf{D}_i - \mathsf{C}_i$;
3: **repeat**
4: $\mathcal{S} \leftarrow \emptyset$ ▷ The set of memory phases
5: **for** $\tau_i \in \mathcal{T}$ **do**
6: $\delta_i \leftarrow \frac{\mathtt{ub}_i + \mathtt{lb}_i}{2}$ ▷ Computes the intermediate deadlines
7: add (M_i, δ_i) to $\mathcal{S}$
8: **end for**
9: **if** dbf_analysis_np($\mathcal{S}$) **then**
10: $\forall \tau_i \in \mathcal{T}$: $\phi_{\mathsf{C}_i} \leftarrow \delta_i$
11: **if** dbf_offset_analysis($\mathcal{T}$) **then**
12: **return** $\mathcal{S}$ ▷ Returns feasible solution
13: **else**
14: $\forall \tau_i \in$ UnSchedulable : $\mathtt{ub}_i \leftarrow \delta_i$
15: **end if**
16: **else**
17: $\forall \tau_i \in \mathcal{T}$: $\mathtt{lb}_i \leftarrow \delta_i$
18: **end if**
19: **until** $\forall \tau_i \in \mathcal{T}$: $\mathtt{ub}_i = \mathtt{lb}_i$
20: **return** FAIL

5.5 Results and discussions

In this section, we present the performances of the proposed approaches with respect to the state of the art. First, we conducted experiments with randomly generated workloads to evaluate the proposed approaches using different task partitioning heuristic. We study the impact of the task memory stall ($\frac{M_i}{C_i+M_i}$) and the workload size on schedulability and the required time to complete the analysis. We compared the proposed approaches with a similar analysis from the state-of-the-art [10]. Finally, we evaluate the practicality of the proposed approaches and the overall system performances with a set of real benchmarks running on the Infineon Aurix TC397 microcontroller: MIbench [1] , FFTbench [2] and Mälardalen [3]. .

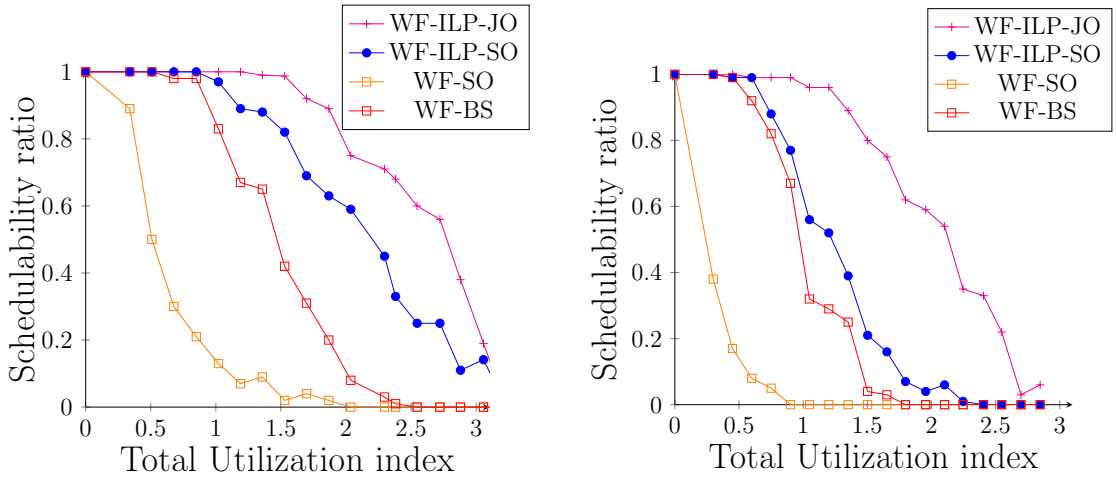
[1] https://github.com/embecosm/mibench
[2] https://github.com/ZiCog/fftbench
[3] https://github.com/TRDDC-TUM/wcet-benchmarks

5.5.1 Task set generation

The synthetic task set generation takes as input n the number of tasks and the target total utilization $U_{\mathcal{T}}$. It starts by generating the utilizations of the n tasks by using UUniFast-Discard [71] algorithm. We varied the baseline utilization from 0.4 to P (number of available cores) with a step of 0.2. For every utilization U_i, the algorithm generates the memory phase utilization u_i^m using a random stall value. The random value is either selected in $lev_1 = [0.10, 0.20]$ or in $lev_2 = [0.20, 0.30]$ according to the selected scenario. The generated utilization comprises computation and memory phases, therefore the total computation utilization on the cores is smaller than $U_{\mathcal{T}}$. For each scenario, we generate 100 task sets per utilization and per memory stall. We generate 10 tasks per task set for the offset-based approaches. As their complexity is high, they are evaluated under limited settings. For heuristic approaches, we generate 32 tasks per taskset. To avoid intractable hyper-periods, the period of every task is selected randomly from the list of periods : {80,100,200,240,400,600,800,1200}. When memory phase utilization is very low, periods are multiplied by 10, so that every task have at least M_i greater or equal to 1. The task deadline is set to 70% of the task's period.

5.5.2 Results of synthetic task set experiments

In this section, we evaluate the performance of three offset based methods against our heuristic (Algorithm 6): the task-level offset (Theorem 8) denoted as SO; the ILP-based task-level offset denoted as ILP-SO, and the job-level offset denoted as ILP-JO. The tasks are allocated on 4 cores by either Worst-fit (WF) or Best-fit (BF). Therefore, each algorithm is labeled by a combination of these techniques.

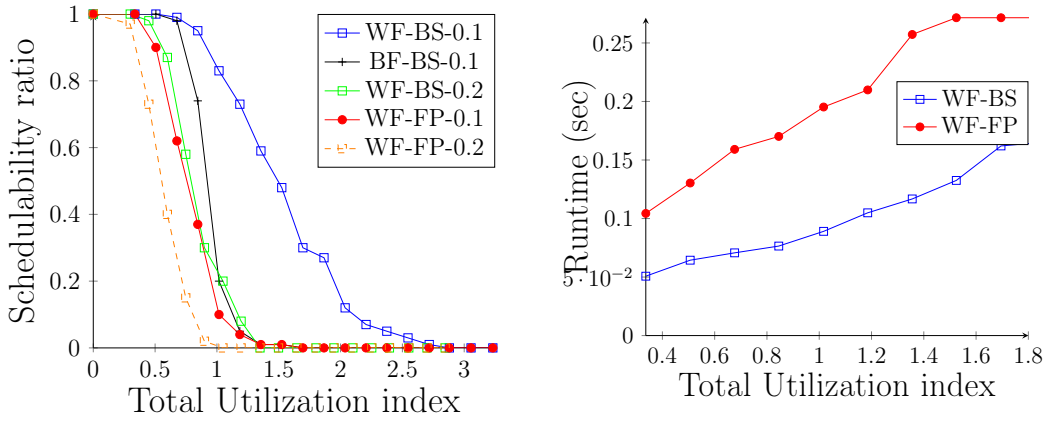


(a) stall $\in [0.10, 0.20]$ (b) stall $\in [0.20, 0.30]$

Figure 5.3: Schedulability of ILP vs heuristics approaches

In Figure 5.3, we report the schedulability ratio as a function of total utilization for the two classes of memory stalls. Consistently with previous results in the literature, WF outperforms BF in all simulated scenario, therefore, we only

report the results for WF for clarity of presentation. At low total utilization values, all algorithms easily schedule all tasks sets. As the utilization increases, the ILP-based offset assignment algorithms outperform the *task-level* offset assignment algorithm. We remark that, given a task allocation, ILP for job-level offset is a relaxed version of the ILP at task-level, therefore, it naturally outperforms the latter. The schedulability falls sharply for WF-SO algorithm because Condition 5.3 becomes quickly not satisfied for large memory phases. Please notice, that our intermediate deadlines approach performances are still very acceptable regarding the ILP-based approaches in terms of schedulability, even with their hugely shorted analysis time compared to ILP approaches. Indeed, each simulation using ILP takes around 3 hours to complete on a 40-cores Intel(R) Xeon(R) CPU E5-2630 v4 at 2.20GHz, with 130 GB of RAM using CPLEX ILP solver. In the other hand, the analysis time of WF-ILP-SO is acceptable (i.e. a few seconds). The difference between the ILP and our heuristic is even reduced when the stall is larger. Naturally, the schedulability of task-level offset based sufficient test falls sharply as total utilization (i.e. memory phases length increase).



(a) Schedulability rates

(b) Required analysis time

Figure 5.4: Heuristics algorithms performances

Our heuristic (Algorithm 6) is compared to related work on large settings composed of 32 tasks. The results in terms of schedulability and required analysis time are reported in Figure 5.4. We simulated two different memory stalls $[0.1-0.2)$ (denoted as 0.1) and stall $[0.2-0.3)$ (denoted as 0.2). We compare our heuristics against the response time analysis for FIFO-bus scheduler and fixed priority core scheduler found in [10], (denoted as FP in the algorithm label). As the utilization increases, the WF algorithms present better performances in terms of schedulability than BF algorithms, for [10] and our approaches. Therefore, we kept only the best results of BF algorithm, which is using our heuristic (BS) at stall $[0.1 - 0.2)$. Our heuristic algorithms perform much better than the approach analyzed in [10], especially when workload is high. Our algorithm has pseudo-polynomial complexity, however, their run-time is acceptable even for large task sets as shown in Figure 5.4b. Please observe that the memory stall has an impact on schedulability as it drops sharply, when the latter increases.

5.6 Conclusion

In this chapter, we proposed several techniques for contention avoidance on a multicore platform. We used the well-known PREM task model for its predictability. In the first part of this chapter, we presented two time-triggered-based approaches for memory requests scheduling. Then, we proposed a binary search guided by core schedulability approach as an alternative to the first approach, where memory requests were scheduled using the EDF scheduler. Our experiments show a significant improvement in the system performances compared to state-of-the-art. We demonstrate the applicability of our techniques with an implementation on a real hardware platform and on realistic benchmarks. In the next chapter, we extend our objective to task-graph task sets.

Chapter 6

Memory-processor Co-scheduling AECR-DAG Real-time Tasks on Partitioned Multicore Platforms with Scratchpads

Contents

6.1 Introduction 66
6.2 System model 66
6.2.1 Architecture model 66
6.2.2 Task model 66
6.3 DAG tasks allocation and transformation 69
6.3.1 Decision variables and objective function 71
6.4 Deadline based DAG memory-processor co-scheduling 73
6.4.1 Fair and proportional deadline assignment 74
6.4.2 GA-based intermediate deadline assignment 75
6.4.3 Evaluation Strategy 79
6.4.4 Creating the next generation 84
6.5 Results and discussions 85
6.5.1 Task generation 86
6.5.2 Simulation results and discussions 87
6.6 Conclusion 92

6.1 Introduction

In this chapter, we aim to improve the usability of scratchpad memories and exploit their predictability to hide access latency to shared resources. We use a genetic algorithm to derive bus scheduling parameters for a set of directed acyclic task-graphs (DAGs). We extend the DAG task model to include the communications following the Acquisition-Execution-Restitution (AER) model. We use this model in the second part of the chapter where we first partition subtasks onto the multicore platform while scheduling their memory requests and relative communications onto the shared buses, in order to prevent interference and ensure predictability. Specifically, all subtasks and communications are assigned appropriate intermediate offsets and deadlines to guarantee that they comply with the system's timing constraints. Here we focus on architectures featuring inter-core memory transfers dedicated buses such as the Infineon Aurix TC397 microcontroller [2]. Therefore, the same architecture of the previous chapter is used in this part with a small consideration: rather than using only one communication bus, we use an inter-core communication bus to perform the data transfers between scratchpad memories and a second bus to perform data transfers between cores and the main memory.

6.2 System model

6.2.1 Architecture model

We consider a multicore platform $\mathcal{A}$ composed of m cores, i.e., $\mathcal{A} = \{\mathsf{p}_1, \cdots, \mathsf{p}_\mathsf{m}\}$. Each core p_i has its own local scratchpad memory, onto which data and instructions are stored. All cores share a single main memory. Two types of memory copy operations are possible: (i) between scratchpad memories of different cores and (ii) between the main memory and a core's scratchpad memory. The two types of memory operations are performed by different buses: one bus, denoted as S2SB, interconnects all the scratchpad memories; a second separate bus, denoted as M2SB, interconnects the main memory and the scratchpads. The topology of our architectural model is illustrated in Figure 6.1. From a software point of view, we assume that all memories (main memory and local scratchpad memories) are directly accessible to all cores via different address spaces. The Infineon Aurix TC397 [2] is an example of a real architecture that can be modeled as described above.

Memory copy operations are explicitly triggered by the software, therefore it is easier to schedule them. The proposed model allows us to reduce the complexity of the schedulability analysis by separating the memory operations between local and global memories.

6.2.2 Task model

We consider a task set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ consisting of n sporadic tasks. Each task $\tau_i \in \mathcal{T}$ is represented by a tuple $(\mathsf{G}_\mathsf{i}, \mathsf{D}_i, \mathsf{T}_i)$, where G_i is a Directed Acyclic Graph

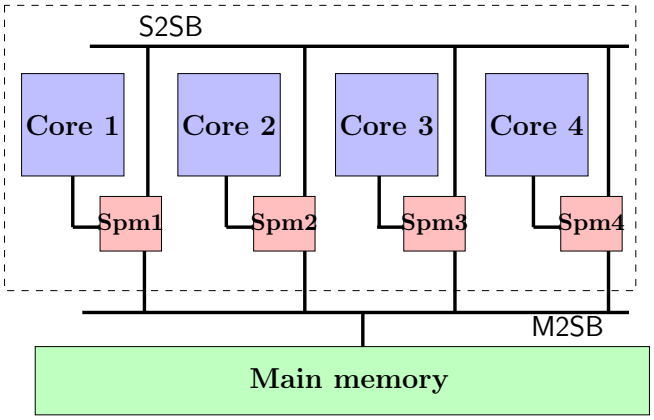


Figure 6.1: Multicore architecture featuring 4 cores, and its interconnection buses.

(DAG) that describes the internal structure of τ_i, D_i is its end-to-end relative deadline, and T_i is its period. We consider a constrained deadline task set, that is $\mathsf{D}_i \leq \mathsf{T}_i$ for all tasks.

Each task-graph $\mathsf{G_i}$ is defined by $(\mathcal{V}_i, \xi_i)$, where $\mathcal{V}_i$ is a set of n_i *subtasks*, and ξ_i is the set of precedence constraints between them. A subtask $v_{i,j} \in \mathcal{V}_i$ can be one of four types (see Figure 6.2):

- an *acquisition subtask*, denoted as $v^A_{i,j}$;
- a *restitution subtask* denoted as $v^R_{i,j}$;
- a *communication subtask* denoted as $v^m_{i,j}$;
- a *computation subtask* denoted as $v^e_{i,j}$.

The upper index is omitted when referring to a subtask without any consideration of its type.

Edge $e(v_{i,j}, v_{i,k}) \in \xi_i$ represents the precedence constraint between subtasks $v_{i,j}$ and $v_{i,k}$, i.e., $v_{i,k}$ can not released before $v_{i,j}$ has completed its execution. Thus, $v_{i,k}$ is an *immediate successor* of $v_{i,j}$, and $v_{i,j}$ is an *immediate predecessor* of $v_{i,k}$.

We denote by $\mathsf{i\_succ}(v_{i,j})$ the set of all immediate successors of $v_{i,j}$ (i.e., $\mathsf{i\_succ}(v_{i,j}) = v_{i,k} \mid (v_{i,j}, v_{i,k}) \in \xi_i$); we denote by $\mathsf{succ}(v_{i,j})$ the set of subtasks reachable from $v_{i,j}$. In the same way, we denote by $\mathsf{i\_pred(v_{i,j})}$ the set of all immediate predecessors of $v_{i,j}$ (i.e., $\mathsf{i\_pred(v_{i,j})} = v_{i,k} \mid (v_{i,k}, v_{i,j}) \in \xi_i$.

A *communication subtask* is a memory copy operation that takes place between the scratchpad memories of the cores where its immediate predecessor and its immediate successor are allocated. Therefore, it has only one immediate successor and one immediate predecessor.

An *acquisition subtask* represents a memory copy operation from the main memory to one or more of the local scratchpads. Without loss of generality, in this paper we consider that a task-graph has only one acquisition subtask with no predecessors, and it is therefore the source node of the graph.

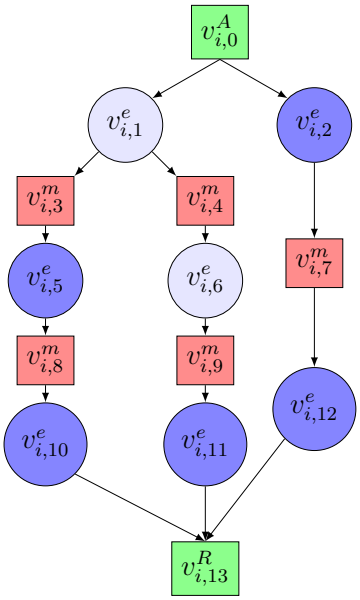


Figure 6.2: DAG task example, computation subtasks are mapped on a dual-core platform.

A *restitution subtask* represents a memory copy operation from one or more of the local scratchpads to the main memory. Without loss of generality, in this paper we consider that a task-graph has only one restitution subtask with no successors, and it is therefore the sink node of the graph.

A *computation subtask* represents code executing on one of the cores. While executing, the computation subtask can only access data and code on the corresponding local scratchpad memory. The immediate successors and immediate predecessors of a computation subtask are memory operations (either communication subtasks or acquisition or restitution subtasks). More generally, we consider that two subtasks of the same type cannot be immediate successors to each other.

Later on, after we allocate computation subtasks to cores, we will simplify the graph and allow two computation subtasks allocated on the same core to be directly linked by an edge without any communication needed between them.

A sequence of vertexes consisting of one computation subtask followed by a communication subtask followed by another computation subtask is called a *triplet*. We denote the triplet of subtasks $v_{i,j}, v_{i,k}, v_{i,h}$ as $\widehat{i : jkh}$.

Each subtask is characterized by its worst-case execution time, denoted by $\mathsf{C}_{v^e_{i,j}}$ for computation subtasks. When a subtask is an acquisition $\mathsf{C}_{v^A_i}$ (resp. restitution $\mathsf{C}_{v^R_i}$), its worst-case execution time represents an upper bound on the time required for the task to perform data transfers from (resp. to) the main memory and the scratchpad memories of the cores where its immediate successors (resp. predecessors) are allocated. If an acquisition (resp. restitution) has multiple successors (resp. predecessors), multiple memory copy operations may be performed by the subtask.

If a subtask is of the communication type, its worst-case execution time $\mathsf{C}_{v^m_{i,j}}$ represents an upper bound on the time required to perform data transfers be-

tween the scratchpad memories of the cores where its immediate predecessor and immediate successor are allocated.

Once a memory copy subtask starts its execution, it cannot be preempted. In contrast to memory copy subtasks, computation subtasks can be preempted, and they are partitioned among the platform cores.

We consider just one acquisition subtask (restitution subtask) in our model, in order to simplify the deadline assignment phase. If there are several data transfers for acquisition or for restitution, then we assume that these transfers are carried out sequentially (merged into one node A or R).

We define π_i^k as the k^{th} path of task τ_i, where $\pi_i^k = \langle v_{i,j}, v_{i,j+1}, \cdots, v_{i,|\pi_i^k|} \rangle$. The first subtask in a path represents the acquisition phase, while the last one represents the restitution phase. The set of all paths of task τ_i is denoted as Π_i.

We define π_i^k as the k^{th} path of task τ_i. π_i^k is a sequence of vertexes, $\pi_i^k = \langle v_{i,j}, v_{i,j+1}, \cdots, v_{i,|\pi_i^k|} \rangle$ such that, $\forall j \in [0, |\pi_i^k|)$, $e(v_{i,j}, v_{i,j+1}) \in \xi_i$. The first subtask in a path represents the acquisition phase, while the last one represents the restitution phase. The set of all paths of task τ_i is denoted as Π_i. Figure 6.2 depicts a task-graph with three paths.

We define task utilization as the occupancy ratio of the task on all the shared resources (cores and communication buses). It is computed as follows:

$$\mathsf{u}(\tau_\mathsf{i}) = \frac{1}{\mathsf{T_i}} \cdot (\mathsf{C}_{\mathsf{v_i^A}} + \mathsf{C}_{\mathsf{v_i^R}} + \sum_{\mathsf{v_{i,j}^m} \in \mathcal{V}_\mathsf{i}} \mathsf{C}_{\mathsf{v_{i,j}^m}} + \sum_{\mathsf{v_{i,j}^e} \in \mathcal{V}_\mathsf{i}} \mathsf{C}_{\mathsf{v_{i,j}^e}}) \tag{6.1}$$

The total $\mathsf{U^{max}}$ utilization of the task set is computed as follows:

$$\mathsf{U^{max}} = \sum_{\tau_i \in \mathcal{T}} \mathsf{u}(\tau_\mathsf{i}) \tag{6.2}$$

We denote by $\mathcal{V}_i^M$ the set of all communication subtasks of task τ_i, and by $\mathcal{V}_i^C$ the set of all computation subtasks of task τ_i. When we describe a behavior that relates to a single task, its index might be removed to avoid overcharging the symbols.

6.3 DAG tasks allocation and transformation

In this section, we consider the problem of partitioning a set of AECR-DAG tasks on an identical core platform. Our approach consists of two distinct stages: the allocation stage and the schedulability analysis stage.

Computation subtasks are allocated to the different cores; inter-core communications are scheduled on the intercommunication bus; and acquisition and restitution phases are scheduled on the memory bus. The subtask-to-core allocation process has a significant impact on inter-core communications. Communicating subtasks that are allocated on the same core will not generate traffic on the shared inter-core communication bus, while those allocated onto different cores may generate traffic that may jeopardize the schedulability. Rather than using classical bin packing heuristics that optimize core utilization, such as

Best-Fit (BF) that maximizes utilization per core or Worst-Fit (WF) that favors load balancing, we use Integer Linear Programming (ILP) for the subtask-to-core allocation. The goal of the ILP is to reduce the workload on the inter-core communication bus by favoring the allocation of communicating subtasks to the same core when possible or desirable.

Definition 5 (null communication) A communication subtask $v^m_{i,j}$ is called a *null-communication subtask* when its only immediate predecessor subtask and its only immediate successor subtask are allocated to the same core p.

A null-communication subtask does not generate traffic on the shared inter-core bus, therefore, its worst-case execution time can be set to 0, and hence it can be eliminated from the graph.

Definition 6 A reduced task, denoted as $\overline{\tau_i}$ of task τ_i is a DAG where all the null-communication subtasks are removed.

Let $v^m_{i,k}$ be a null-communication subtask, and let $\widehat{i : jkh}$ be the triplet of subtasks consisting of the immediate predecessor of $v^m_{i,k}$ namely $v^m_{i,j}$ and its immediate successor, namely $v^m_{i,h}$. Then in the reduced task $\overline{\tau_i}$, $v^m_{i,k} \notin \overline{\mathcal{V}_i}$, edges $e(v_{i,j}, v_{i,k}) \notin \overline{\xi_i}$ and $e(v_{i,k}, v_{i,h}) \notin \overline{\xi_i}$, and there exist an edge $e(v_{i,j}, v_{i,h}) \in \overline{\xi_i}$, where $\overline{\mathcal{V}_i}$ and $\overline{\xi_i}$ are respectively the nodes set and edges set of $\overline{\tau_i}$.

Therefore, in the reduced task we allow a computational subtask to be an immediate successor of another.

To produce a reduced task, we need only to replace all triplets containing a null-communication subtask with a simple edge between the two computation subtasks of the triplet.

We denote by $\mathbf{cost}(\overline{\tau})$ the ratio of the communication workload that the reduced task $\overline{\tau}$ will require on the share inter-core bus. It can be computed as follows:

$$\mathsf{cost}(\overline{\tau}) = \sum_{v \in \mathcal{V}^M(\overline{\tau})} \frac{C(v)}{\mathsf{T}(\tau)} \tag{6.3}$$

Example 3 In this example, subtasks are colored according to core where they are allocated (Figure 6.3). Therefore, we have four null-communication triplets: $\mathcal{V}^1_\tau(p_1) = \{v^e_5, v^e_{10}\}$, $\mathcal{V}^2_\tau(p_1) = \{v^e_{11}\}$, $\mathcal{V}^3_\tau(p_1) = \{v^e_2, v^e_{12}\}$ and $\mathcal{V}^4_\tau(p_2) = \{v^e_1, v^e_6\}$, as indicated by the red boxes in the left-hand side graph.
In the right-hand side graph, the memory subtasks of the different null-communication triplets have been dropped as their execution time is set to 0, resulting in a reduced task denoted as $\overline{\tau}$. In this example, the execution time of v^m_8 is set to 0 for $\mathcal{V}^1_\tau(p_1)$, and the execution time of v^m_7 is set to 0 for $\mathcal{V}^3_\tau(p_1)$ and also the execution time of v^m_4 is set to 0 for $\mathcal{V}^4_\tau(p_2)$.

Our ILP formulation has the objective to reduce the total communication cost over all tasks. In the remainder of this section, we detail the ILP formulation to optimally reduce the DAG tasks.

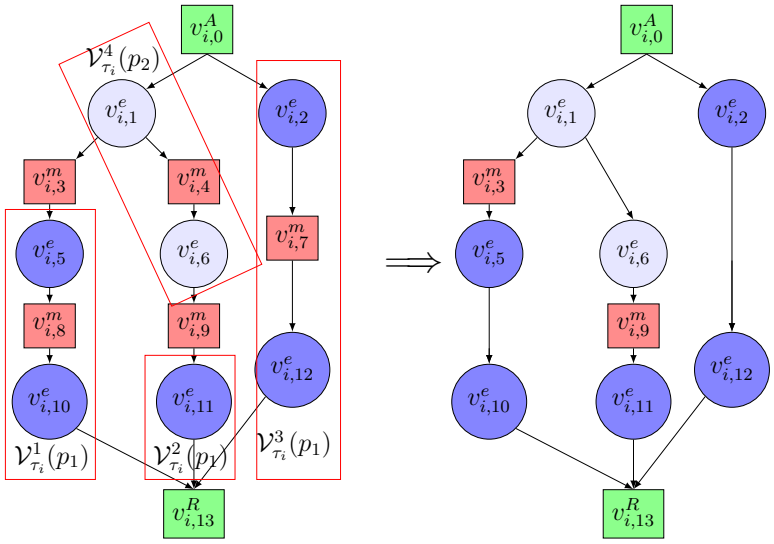


Figure 6.3: DAG task transformation.

6.3.1 Decision variables and objective function

Let $\mathsf{a}^{\mathsf{p}}_j$ be a binary decision variable to express that computation subtask $v_{i,j}$ is allocated to core p[1], *i.e*:

$$\mathsf{a}^{\mathsf{p}}_j = \begin{cases} 1, & \text{if } v_{i,j} \text{ is mapped to core } \mathsf{p} \\ 0, & otherwise \end{cases} \tag{6.4}$$

Let us consider the triplet $\widehat{i : jkh}$. If v_{ij} and v_{ih} are allocated to the same core, the execution time of subtask v_{ik} is set to zero as it does not generate traffic on the communication bus. Therefore, for each triplet $\widehat{i : jkh}$, we define the decision variable $\mathsf{cost}(\widehat{i : jkh})$ as follows:

$$\mathsf{cost}(\widehat{i : jkh}) = \begin{cases} 0, & \text{if } \forall \mathsf{p},\ \mathsf{a}^{\mathsf{p}}_j = \mathsf{a}^{\mathsf{p}}_h \\ C(v_{ik}), & \text{otherwise} \end{cases} \tag{6.5}$$

The objective function consists in minimizing the total communication cost of all tasks by summing over all triplets $\widehat{i : jkh}$

$$\mathsf{Minimize} \sum_{\tau_i \in \mathcal{T}} \sum_{\widehat{i:jkh} \in \Delta_i} \mathsf{cost}(\widehat{i : jkh}) \tag{6.6}$$

We will illustrate further in this section the different techniques to linearize these constraints.

Constraints. First, we describe the evaluation of the cost variables. The conditional construct and the different inequalities involved in computing the variable cost require linearization.

For every triplet $\widehat{i : jkh}$ and for every core p, we introduce two artificial binary decision variables: $x(\widehat{i : jkh}, \mathsf{p})$ and $y(\widehat{i : jkh}, \mathsf{p})$. These variables indicate

[1]Please notice that this decision variable is generated only for the computation subtasks

whether subtasks v_{ij} and v_{ih} are allocated to the same core, and are defined as follows:

$$x(\widehat{i:jkh},\mathsf{p}) = \begin{cases} 1 & \text{if} \quad a_j^{\mathsf{p}} > a_h^{\mathsf{p}} \\ 0, & \text{otherwise} \end{cases} \tag{6.7}$$

$$y(\widehat{i:jkh},\mathsf{p}) = \begin{cases} 1 & \text{if} \quad a_j^{\mathsf{p}} < a_h^{\mathsf{p}} \\ 0, & \text{otherwise} \end{cases} \tag{6.8}$$

If x or y are equal to 1, then the subtasks are not allocated to the same core. If both x and y are equal to 0, then the subtasks are allocated to the same core. x and y cannot both be equal to 1 at the same time, therefore we compute the cost as follows:

$$\mathsf{cost}(\widehat{i:jkh}) = C(v_k) \cdot \sum_{\mathsf{p}\in\mathcal{A}} (x(\widehat{i:jkh},\mathsf{p}) + y(\widehat{i:jkh},\mathsf{p})) \tag{6.9}$$

Once again, the conditional construct to compute $x(\widehat{i:jkh},\mathsf{p})$ requires linearization. First, we express the inequality in Equation (6.7) as follows:

$$a_j^{\mathsf{p}} - a_h^{\mathsf{p}} - 1 \geq 0$$

Let M be a very large constant. We linearize Equation (6.7) as follows:

$$\begin{aligned} a_j^{\mathsf{p}} - a_h^{\mathsf{p}} - 1 + M - M \cdot x(\widehat{i:jkh},\mathsf{p}) \geq 0 \\ a_j^{\mathsf{p}} - a_h^{\mathsf{p}} - 1 - M \cdot x(\widehat{i:jkh},\mathsf{p})) < 0 \end{aligned}$$

Similarly, we linearize Equation (6.8) as follows :

$$\begin{aligned} a_h^{\mathsf{p}} - a_j^{\mathsf{p}} - 1 + M - M \cdot y(\widehat{i:jkh},\mathsf{p}) \geq 0 \\ a_h^{\mathsf{p}} - a_j^{\mathsf{p}} - 1 - M \cdot y(\widehat{i:jkh},\mathsf{p})) < 0 \end{aligned}$$

In order to ensure that a subtask is allocated to one and only one core, we generate the following constraints:

$$\forall \tau \in \mathcal{T}, \forall v \in \mathcal{V}(\tau) : \sum_{\mathsf{p}\in\mathcal{A}} a_v^{\mathsf{p}} = 1 \tag{6.10}$$

We can enforce the utilization per core to not exceed a given bound U^{max} using the following constraints:

$$\forall \mathsf{p} \in \mathcal{A}, \sum_{\tau} \sum_{v\in\mathcal{V}^C(\tau)} a_v^{\mathsf{p}} \cdot \frac{C(v)}{\mathsf{T}_\tau} \leq U^{\mathsf{max}} \tag{6.11}$$

In the following listing, we report our complete ILP formulation.

$$\text{Minimize} \sum_{\tau_i \in \mathcal{T}} \sum_{\widehat{i:jkh} \in \Delta_i} \mathsf{cost}(\widehat{i:jkh})$$

under the constraints:

$$\forall \tau \in \mathcal{T}, \forall \widehat{i:jkh} \in \Delta_i :$$
$$\mathsf{cost}(\widehat{i:jkh}) = C(v_k) \cdot \sum_{\mathsf{p} \in \mathcal{A}} (x(\widehat{i:jkh}, \mathsf{p}) + y(\widehat{i:jkh}, \mathsf{p}))$$

$$\forall \tau \in \mathcal{T}, \forall \widehat{i:jkh} \in \Delta_i :$$
$$a_j^{\mathsf{p}} - a_h^{\mathsf{p}} - 1 + M - M \cdot x(\widehat{i:jkh}, \mathsf{p}) \geq 0$$
$$a_j^{\mathsf{p}} - a_h^{\mathsf{p}} - 1 - M \cdot x(\widehat{i:jkh}, \mathsf{p})) < 0$$
$$a_h^{\mathsf{p}} - a_j^{\mathsf{p}} - 1 + M - M \cdot y(\widehat{i:jkh}, \mathsf{p}) \geq 0$$
$$a_h^{\mathsf{p}} - a_j^{\mathsf{p}} - 1 - M \cdot y(\widehat{i:jkh}, \mathsf{p})) < 0$$

$$\forall \tau \in \mathcal{T}, \forall v \in \mathcal{V}(\tau) : \sum_{\mathsf{p} \in \mathcal{A}} a_i^{\mathsf{p}} = 1$$

$$\forall \mathsf{p} \in \mathcal{A}, \sum_{\tau} \sum_{v \in \mathcal{V}(\tau)} a_v^{\mathsf{p}} \cdot \frac{C(v)}{\mathsf{T}_\tau} \leq U^{\mathsf{max}}$$

Finally, the ILP is submitted to the CPLEX ILP-solver [4]. When subtask-to-core mapping is computed, reduced tasks are derived by removing the null-communication subtasks.

6.4 Deadline based DAG memory-processor co-scheduling

The second stage of our approach is memory and processor co-scheduling. This work considers partitioned scheduling, where subtasks are assigned to different cores and scheduled using a fully preemptive EDF scheduler per core. Acquisition and restitution subtasks are scheduled non-preemptively on the memory-to-scratchpad bus (M2SB), while communication subtasks are scheduled on the inter-core bus (S2SB). Scheduling on the system's buses is achieved using a non-preemptive EDF scheduler. To simplify dealing with precedence constraints, we impose intermediate offsets and deadlines on each subtask. In this way, precedence constraints are automatically respected if each subtask is activated after its offset and completes no later than its assigned deadline.

Several techniques have been proposed in the real-time systems literature to assign intermediate deadlines, such as *fair* and *proportional* deadline assignments. However, these techniques have been designed for the scheduling of computation tasks only and are not tuned to handle delays that may occur due to contention on the shared communication buses. In this section, we present our approach to assign intermediate deadlines to both memory and computation sub-

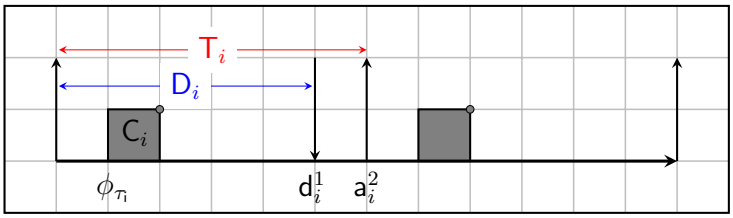


Figure 6.4: Example of offset and local deadline.

tasks, in order to respect the system's timing constraints and take into account the co-scheduling of both types of subtasks.

In the following, every subtask will be additionally characterised by its *intermediate deadline* d_v and *offset* ϕ_v. The offset of a subtask is the distance between the activation of the task-graph and the activation of the subtask. The intermediate deadline of a subtask represents its relative deadline with respect to its offset. We define the subtask *local deadline* dl_v as the sum of its intermediate deadline and its offset. Figure 6.4 illustrates the relationship between the activation, end-to-end deadline, intermediate deadline, offset, and local deadline of a subtask.

In this context, the activation time of the acquisition subtask corresponds to the activation of the task itself. The local deadline of a subtask represents the interval between the task-graph activation and the subtask's absolute deadline.

One strategy to solve the problem of assigning intermediate deadlines is to exhaustively search among all possible intermediate deadline combinations. While this method provides an optimal solution, it suffers from a high computational complexity. In our previous work on the scheduling of PREM tasks [9], we proposed a binary search-based method to compute and assign intermediate deadlines to memory phases, where each task consists only of a memory subtask and a computation subtask. However, it is not straightforward to extend this approach to DAG tasks, as the approach proposed in [9] already has a high complexity for a simpler problem. Therefore, in this section, we propose to use a genetic algorithm (GA) to explore the assignments of intermediate deadlines to subtasks. We will describe our approach in the rest of this section.

6.4.1 Fair and proportional deadline assignment

First, we review the *fair* and *proportional* deadline assignment techniques. The idea is to divide the slack time along a path π_i^k in the graph between all the subtasks of the path. We consider paths in decreasing order of their cumulative execution time, therefore, we start with the critical, *i.e.* having the largest cumulative execution time.

We first define the slack function $\mathsf{Sl}(\pi_i^k, \mathsf{D}_i)$ along path π_i^k of τ_i as:

$$\mathsf{Sl}(\pi_i^k, \mathsf{D}_i) = \mathsf{D}_i - \sum_{v \in \pi_i^k} \mathsf{C}_v \tag{6.12}$$

- **Fair distribution:** distribute slack as the ratio of the original slack by the number of subtasks along the path:

$$\mathsf{calculate\_share}(v_{i,j}, \pi_i^k) = \frac{\mathsf{Sl}(\pi_i^k, \mathsf{D}_i)}{|\pi_i^k|} \tag{6.13}$$

- **Proportional distribution:** distribute slack according to the contribution of the subtask execution time in the path:

$$\mathsf{calcule\_share}(v_{i,j}, \pi_i^k) = \frac{\mathsf{C}_{v_{i,j}}}{\mathsf{C}(\pi_i^k)} \cdot \mathsf{Sl}(\pi_i^k, \mathsf{D}_i) \tag{6.14}$$

  where $\mathsf{C}(\pi_i^k)$ represents the total cumulative execution time of the subtasks in π_i^k.

Once the relative deadlines of the subtasks along the critical path have been assigned, we select the next path in order of decreasing cumulative execution time, and assign the deadlines to the remaining subtask by appropriately subtracting the already assigned deadlines. The complete procedure is not reported here and can be found in [41].

6.4.2 GA-based intermediate deadline assignment

In this paper, we use a genetic algorithm to assign intermediate deadlines to memory and computation subtasks for a set of reduced tasks $\overline{\mathcal{T}}$. In a genetic algorithm, a population of candidate solutions (called individuals) is evolved towards better solutions. The goal is to move from unschedulable solutions to at least one schedulable solution.

Each candidate solution has a set of *chromosomes*, which in our case is the task set with intermediate deadlines assigned, that can be mutated and crossed over. The evolution process starts from multiple solutions, called the initial population, each having intermediate deadlines generated randomly.

The iterative process then evaluates the **fitness** function for every individual in the population. In each generation, a portion of the existing population is selected to reproduce and create a new generation through three operations: selection, crossover, and mutation. The new generation of candidate solutions is then used as input for the next iteration. The algorithm terminates when a maximum number of generations has been produced or a schedulable intermediate deadline assignement task set is found.

In the following, we describe the general structure of the genetic algorithm, the representation of a solution, the generation of the initial population, the fitness function, the selection, the crossover, and the mutation operations.

Algorithm 7 presents our approach for assigning intermediate deadlines. The algorithm starts by generating the initial population (Line 4). It goes through several iterations until a schedulable task set is found, namely: (i) population evaluation (Line 6), (ii) selection (Line 8), (iii) crossover (Line 9), and (iv) mutation (Line 10). If the algorithm finds a feasible schedule, it terminates with SUCCESS, otherwise, if a maximum number of iterations is reached, it aborts with FAIL.

Algorithm 7 GA intermediate deadlines assignment

```
1: function DeadlinesGA(T, p_size)                ▷ The DAG task set, population size
2:     found = false
3:     P_l ← ∅                                    ▷ The initial population
4:     ip_generation(T, p_size)                   ▷ The initial population generation
5:     while (not (found)) do
6:         found = ga_evaluation(P_l, A)
7:         if (not found) then
8:             selection(P_l)
9:             crossover(P_l, η_cr)
10:            mutation(P_l, η_mu)
11:        else
12:            return SUCCESS                     ▷ If schedulable at all levels
13:        end if
14:    end while
15:    return FAIL
16: end function
```

Individual representation

Each individual in the genetic algorithm consists of a set of subtasks, each with an intermediate deadline assigned. In this work, we represent an individual as an ordered list of pairs, where each pair consists of a subtask and its corresponding local deadline. In our approach, we only consider subtasks that are part of reduced tasks while constructing the ordered list of subtask-deadline pairs for an individual.

Definition 7 Individual An *individual* is denoted as

$$\begin{aligned}\mathsf{ind} =&(\langle v_{1,1}, dl_{1,1}\rangle, \langle v_{1,2}, dl_{1,2}\rangle, \cdots,\\ &\langle v_{1,n_1}, dl_{1,n_1}\rangle, \langle v_{2,1}, dl_{2,1}\rangle, \cdots,\\ &\langle v_{m,n_m}, dl_{m,n_m}\rangle)\end{aligned}$$

where:

- Subtasks of the same task are a sub-sequence of ind
- Subtasks of the same task are ordered in a topological order;
- $dl_{i,j}$ represents the local deadline of subtask $v_{i,j}$;
- All individuals present the same subtasks order.

Example 4 In Figure 6.5, we present a task set composed of two tasks, τ_1 and τ_2. The former has an end-to-end deadline of 50, while the latter has an end-to-end deadline of 80. Each task is comprised of seven subtasks. Table 6.1 shows the representation of an individual in our genetic algorithm.

The subtasks in an individual are sorted according to their topological order. For example, subtask $v^e_{1,3}$ appears earlier in the individual than $v^e_{1,4}$, $v^m_{1,5}$, and $v^e_{1,6}$, while $v^e_{1,6}$ appears earlier than $v^R_{1,7}$. It should be noted that we consider local deadlines, and the restitution subtask's local deadline is always equal to the task's end-to-end deadline.

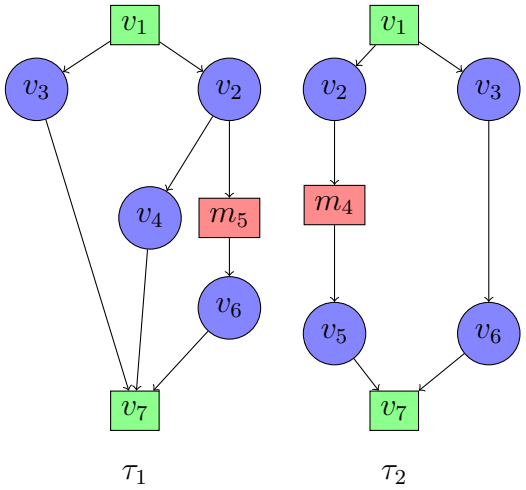


Figure 6.5: Example of an individual.

Initial population generation

The first step of a genetic algorithm is the generation of an initial population. It has been recognized that if the initial population provided to the genetic algorithm is of "high quality", the algorithm is more likely to find a sub-optimal solution [87, 98, 110]. Algorithm 8 summarizes the different steps of our initial population generation process.

Algorithm 8 Initial population generation

```
1:  function IP_GENERATION(T̄, p_size)
2:      P_l ← ∅                                   ▷ The initial population
3:      assigned = true
4:      while |P_l| < p_size do                   ▷ p_size is the maximum number of generations
5:          S ← ∅                                 ▷ An individual
6:          for τ̄_i ∈ T̄ do
7:              assigned = inter_dline(τ̄_i)       ▷ Assign τ̄_i intermediate
8:                                                ▷ deadlines
9:              if !assigned then
10:                 break
11:             end if
12:             add τ̄_i intermediate deadlines to S
13:         end for
14:         if assigned then                      ▷ If all subtasks have assigned a deadline
15:             if S ∉ P_l then
16:                 add S to P_l
17:             end if
18:         end if
19:     end while
20:     return P_l
21: end function
```

For each task $\overline{\tau_i} \in \overline{\mathcal{T}}$, the algorithm invokes the `inter_dline` function (Line 7) to assign intermediate deadlines to subtasks. Within this function, all task paths are sorted by non-increasing cumulative execution time. For each path,

v	dl_v
$v^A_{1,1}$	10
$v^e_{1,3}$	40
$v^e_{1,2}$	25
$v^m_{1,5}$	35
$v^e_{1,4}$	40
$v^e_{1,6}$	40
$v^R_{1,7}$	50
$v^A_{2,1}$	20
$v^e_{2,2}$	30
$v^e_{2,3}$	45
$v^m_{2,4}$	50
$v^e_{2,5}$	70
$v^e_{2,6}$	70
$v^R_{2,7}$	80

Table 6.1: Example of Individual

the `random_dlines` function (Algorithm 9) is invoked in its turn to distribute randomly the slack among all subtasks on that path. The `inter_dline` function aborts on FAIL if slack sharing fails on at least one path, otherwise, it ends on SUCCESS. If the intermediate deadline assignment succeeds for all tasks in $\overline{\mathcal{T}}$, the under generation individual denoted as $\mathcal{S}$ is inserted into the population.

The share of each subtask that has not yet been assigned a deadline on a given path is computed in Algorithm 9. We denote the set of subtasks that have not yet been assigned a deadline by $\mathcal{T}_{nd}$. The intermediate deadline assignment is divided into two parts.

In Part 1, Algorithm 9 computes the intermediate deadlines of subtasks that have an offset and where at least one of their immediate successors has been assigned an offset (line 3). If multiple immediate successors are found with different offsets, the algorithm computes the non-null minimum offset. Intermediate deadlines for these subtasks are assigned using Algorithm 10. The intermediate deadline of each subtask is computed as the difference between its offset and the minimum offset of its immediate successor. Each time a deadline is assigned to a subtask, Algorithm 10 modifies: (i) its local deadline, (ii) the offsets of its immediate successors, (iii) and removes it from $\mathcal{T}_{nd}$ (lines 9-11).

Algorithm 9 computes the intermediate deadlines for the rest of the subtasks in $\mathcal{T}_{nd}$ in Part 2. It computes the slack time on the analyzed path, generates a set of random values whose sum is equal to 1 (line 6) and uses each value to compute the intermediate deadline of a subtask as the sum of $\mathsf{SI} \cdot \mathtt{U}[\mathtt{v}]$ and the subtask worst-case execution time (line 9). The offset of each subtask is computed as the maximum local deadline of its immediate predecessors (except the offset of the acquisition subtask, which is always equal to 0).

Definition 8 (Valid assignment) An intermediate deadline assignment $\Omega(\tau_i)$ of a task-graph τ_i is valid if:

Algorithm 9 Compute the intermediate deadlines on a path

```
1:  function INTER_DLINES(π_i^k)
2:      T_nd ← compute_contributors(π_i^k)
3:      if !non_zero_offset_subtasks_dlines(T_nd, π_i^k) then
4:          return FAIL
5:      end if
6:      Sl ← compute_slack(π_i^k)                    ▷ Compute the slack in path π_i^k
7:      if Sl < 0 then
8:          return FAIL
9:      end if
10:     U ← random_rates()
11:     for v ∈ π_i^k do
12:         if v ∈ T_nd then                          ▷ v has no deadline
13:             dline = C_v + (Sl · U[v])
14:             d_v ← dline
15:             dl_v ← φ_v + dline
16:             update_isucc_offset(v)                ▷ update v's immediate
17:                                                   ▷ successors offset
18:             remove(v, T_nd)
19:         end if
20:     end for
21:     return SUCCESS
22: end function
```

1. the intermediate deadline of each subtask is greater than or equal to its worst-case execution time:
   $\forall v \in \mathcal{V}_i, \quad d_v \geq \mathsf{C}_v$;

2. and the local deadline of the restitution subtask is less than or equal to the end-to-end deadline of τ_i : $dl_{v^R} \leq \mathsf{D}_i$.

Lemma 6 Consider an AECR-DAG task (resp. AECR-reduced DAG task) τ_i. The intermediate deadline assignment $\Omega(\tau_i)$ computed by function *random_dlines* is valid.

Proof 8 The deadline of a subtask is computed either in Part 1 or in Part 2 of Algorithm 9. In the first part, the deadline of the subtask is assigned only if it is greater than or equal to its worst-case execution time, otherwise, it is computed in the second part of the algorithm as $\mathsf{C}_v + (\mathsf{Sl} \cdot \mathtt{U}[\mathtt{v}])$, ensuring Condition 1 of Definition 8. Moreover, the sum of the rates U generated in part 2 of Algorithm 9 is equal to 1; rates are by definition positive values. The slack in its turn, is positive, otherwise, the schedulability fails. The offset of each subtask is computed as the maximum among all predecessors offset, therefore, the local deadline of the restitution subtask cannot be greater than the task end-to-end deadline, confirming Condition 2 of Definition 8.

6.4.3 Evaluation Strategy

The most important step of a genetic algorithm is the evaluation of the population. In this section, we assume that all computation subtasks have been

Algorithm 10 Non zero offset subtasks deadline assignment

```
1: function NON_ZERO_OFFSET_SUBTASK_DLINES(&T_nd, p_i)
2:     for v ∈ p_i do
3:         if v ∈ T_nd then                          ▷ The subtask deadline is not yet computed
4:             min = min_offset(i_succ(v))                          ▷ get the minimum non-null
5:                                                  ▷ offset of the immediate successors of v
6:             if φ_v > 0 and min > 0 then
7:                 dline = min − φ_v
8:                 if dline ≥ C_v then
9:                     d_v ← dline
10:                    dl_v ← φ_v + dline
11:                    update_isucc_offset(v)                ▷ update v's immediate
12:                                                          ▷ successors offset
13:                    remove(v, T_nd)
14:                else
15:                    return FAIL
16:                end if
17:            end if
18:        end if
19:    end for
20:    return SUCCESS
21: end function
```

allocated (partitioned) on the platform's cores and that subtasks have already been assigned offsets and intermediate deadlines. We apply the processor demand criterion [129] to evaluate each individual in the population. Algorithm 11 summarizes and clarifies our evaluation strategy.

The algorithm takes as input the generated population and an empty set $\mathcal{S}$. Each individual ψ_i in the input population needs to be awarded a score to indicate how close it is to meet the overall schedulability. This score is called the *fitness score* and it is calculated by the fitness function (lines 3-4), which will be detailed later in Section 6.4.3.

Algorithm 11 Population evaluation

```
1: function POPULATION_EVALUATION(P_l, S)                          ▷ The population
2:     for s ∈ P_l do
3:         score<A, S2SB, M2SB> = dbf_dag_analysis(s, A)
4:         if (score<A, S2SB, M2SB> == 0) then
5:             return s                                            ▷ A solution is found
6:         else
7:             f_score = α_1 · score(A) + α_2 · score(S2SB) + α_3 · score(M2SB)
8:             s.set_score(f_score)                                ▷ Set the score of s
9:             S ← s                                               ▷ Add s to S
10:        end if
11:    end for
12:    return
13: end function
```

Schedulability of task-graphs

In this paper, we consider a system of sporadic task graphs $\mathcal{T}$. When an instance of a task is activated, its subtasks are activated with an offset relative to the activation of the task-graph. To analyse the schedulability of the system, we proceed by analysing the schedulability on each processor and on the two communication buses. If all subtasks respect their deadlines, then the entire system is schedulable.

Definition 9 We denote by $\Lambda_{\mathsf{p}}(\mathcal{T})$ the subset of subtasks of $\mathcal{T}$ allocated on core p. By extension, $\Lambda_{\mathsf{S2SB}}(\mathcal{T})$ is the set of all communication subtasks, and $\Lambda_{\mathsf{M2SB}}(\mathcal{T})$ is the set of memory subtasks.

To analyse the schedulabity on each processor and on each bus, we use the *Demand Bound Criterion*(dbf) [129]. The original demand bound analysis of [129] only considers sporadic tasks. Instead, subtasks belonging to the same task-graph have offsets with respect to each other. Therefore, we use the approximated method of [96] to compute the dbf of a task-graph on a given core.

The dbf at L on core $\mathsf{p} \in \mathcal{A}$ of a task-graph can be computed as follows [96]:

$$\mathsf{dbf}(\tau_i, L, p) = \max_{\forall v_{i,j} \in \Lambda_{\mathsf{p}}(\mathcal{T})} \left\{ \sum_{v_{i,k} \in \Lambda_{\mathsf{p}}(\mathcal{T})} \left(\left\lfloor \frac{L - \overline{\phi}_{k,j} - d_{v_{i,k}}}{\mathsf{T}_i} \right\rfloor + 1 \right)_0 \cdot \mathsf{C}_{v_{i,k}} \right\} \quad (6.15)$$

where $\overline{\phi}_{k,j} = (\phi_{v_{i,k}} - \phi_{v_{i,j}}) \mod \mathsf{T}_i$. To understand Equation (6.15), consider that the subtasks of a task τ_i are activated with an offset relative to the arrival of the first subtask. Therefore, we need to align the offset of one subtask to the beginning of an interval of length L and compute the workload generated in the interval. We do this for every subtask, and then we take the maximum. The resulting dbf is an upper bound to the actual dbf in that interval.

Theorem 9 *Given a set of subtasks allocated on core* p*, the subtasks are schedulable if their cumulative utilisation is less than 1, and*

$$\forall L \leq L^* \quad \sum_{\tau_i \in \mathcal{T}} \mathsf{dbf}(\tau_i, L, \mathsf{p}) \leq L \quad (6.16)$$

where L^* *is the first idle time on core* p*.*

Proof 9 The proof descends directly from the proof of Theorem 2 in [96]: it suffices to notice that subtasks belonging to different task graphs have no offset relationship between each other, while subtasks beloning to the same task-graph are subject to offsets.

Let now consider the schedulability on the buses. In our model, a task-graph may contain memory nodes that represent either the A/R phases or the communication phases, and these nodes are executed non-preemptively: e.g. once a communication subtask starts, it completes the memory transfer without being interrupted by other communication subtasks. Therefore, we have to take

into account a blocking time in the schedulability analysis. This can be done by extending Theorem 6 in [96], which deals with blocking time due to mutually exclusive resources.

Theorem 10 *Given a set of memory subtasks to be scheduled on* bus*, the subtasks are schedulable if their cumulative utilisation is less than 1, and:*

$$\forall L < L^* \quad \sum_{\tau_i \in \mathcal{T}} \mathsf{dbf}(\tau_i, L, \mathsf{bus}) + \mathsf{B}\ (\tau_i, L, \mathsf{bus}) \leq L \tag{6.17}$$

where:

$$\mathsf{B}\ (\tau_i, L, \mathsf{bus}) = \max\{\ \mathsf{C}_{v_{j,k}} \mid \forall j \neq i, v_{j,k} \in \Lambda_{bus}(\mathcal{T}) \wedge d_{v_{i,j}} > L\ \}. \tag{6.18}$$

is the blocking time due to non-preemptive scheduling.

Proof 10 Since subtasks executed completely inside any interval of length L can be blocked only once by subtasks that started before the beginning of the interval and have deadlines after the end of the interval, the maximum blocking time can not be longer than the maximum worst-case execution time among all subtasks having a deadline greater than L.
Now, by contradiction. Let v be the first subtask to miss a deadline at time t_2 and let $t_1 < t_2$ be the last instant before t_2 when there is an idle time or a subtasks with absolute deadline at or before t_2 is executed. Then, from the properties of EDF it follows that :

- in interval $[t_1, t_2]$ there is no idle time;
- at most one subtask with deadline greater than t_2 can execute (the blocking subtask)
- the rest of the interval is executed by subtasks with arrivals no earlier than t_1 and deadline no later than t_2.

Since a subtask has missed its deadline, then the cumulative demand in $[t_1, t_2]$, including the blocking subtask, has exceeded the length of the interval $L = (t_2 - t_1)$ and this in contradiction with the hypothesis.

Theorem 11 *Given a task set $\mathcal{T}$, and a platform $\mathcal{A}$, where all subtasks have been assigned to cores, and where:*

- *the computation subtasks are scheduled on their assigned cores by preemptive EDF;*
- *the communication subtasks are scheduled by non-preemtpive EDF on the* S2SB *;*
- *the memory subtasks are scheduled by non-preemptive EDF on the* M2SB*;*

the task set is schedulable (i.e. every instance of every task-graph completes before its end-to-end deadline) if:

$$\forall \mathsf{p} \in \mathcal{A}, \forall L \leq L^*, \quad \sum_{\tau_i \in \Lambda_{\mathsf{p}}(\mathcal{T})} \mathsf{dbf}(\tau_i, L, \mathsf{p}) \leq L \tag{6.19}$$

and

$$\forall L \leq L^*, \quad \sum_{\tau_i \in \Lambda_{\mathsf{S2SB}}(\mathcal{T})} \mathsf{dbf}(\tau_i, L, \mathsf{S2SB}) + \mathsf{B}\ (\tau_i, L, \mathsf{S2SB}) \leq L \tag{6.20}$$

and

$$\forall L \leq L^*, \quad \sum_{\tau_i \in \Lambda_{\mathsf{M2SB}}(\mathcal{T})} \mathsf{dbf}(\tau_i, L, \mathsf{M2SB}) + \mathsf{B}\ (\tau_i, L, \mathsf{M2SB}) \leq L \tag{6.21}$$

Proof 11 1) For each core $\mathsf{p} \in \mathcal{A}$, and from Theorem 9, the first condition of the dbf ensures that each computation subtask allocated on p is schedulable (i.e. every instance completes before its intermediate deadline); 2) For both S2SB and M2SB buses, and according to Theorem 10, the second and the third conditions ensure that each memory operation completes before its intermediate deadline.

Therefore: a) the precedence constraints are respected, b) the local deadline of the restitution subtask of every task-graph is less than or equal to its end-to-end deadline. Hence, the system is schedulable.

Fitness function

The fitness score is an indicator of how "fit" a candidate solution is to meet the overall schedulability condition. The *fitness function* takes as input an individual candidate solution to our problem and specifies how far away this individual is from satisfying the schedulability condition.

We define $\mathtt{score}(\mathsf{r})$, with $\mathsf{r} \in \mathcal{R}$, as the schedulability score of the cores, the inter-core bus (S2SB) and the memory-to-scratchpad bus (M2SB), respectively (line 3).

Definition 10 (Fitness score) A schedulability score is computed for each resource $\mathsf{r} \in \mathcal{R}$ as follow:

$$\forall \mathsf{r} \in \mathcal{R}, \mathtt{score}(\mathsf{r}) = \max_{0<L\leq L^*} \left(\frac{\overline{dbf}(L, \mathsf{r}) - L}{L}, 0 \right) \tag{6.22}$$

where $\mathcal{R} = \mathcal{A} \cup \{\mathsf{S2SB}\} \cup \{\mathsf{M2SB}\}$, and $\overline{dbf}(L, r)$ is the left-hand side expression of equations (6.19), (6.20) and (6.21), respectively.

The *fitness score* of a solution is the weighted sum of the three schedulability scores:

$$\begin{aligned} \mathsf{f\_score} = \alpha_1 \mathsf{score}(\mathcal{A}) + \alpha_2 \mathsf{score}(\mathsf{S2SB}) \\ + \alpha_3 \mathsf{score}(\mathsf{M2SB}) \end{aligned} \tag{6.23}$$

and

$$\mathsf{score}(\mathcal{A}) = \frac{\sum\limits_{\forall \mathtt{p} \in \mathcal{A}} \mathtt{score(p)}}{|\mathcal{A}|}$$

Where $\alpha_1 + \alpha_2 + \alpha_3 = 1$.

When the set of subtasks allocated on a given core $\mathtt{p} \in \mathcal{A}$ is schedulable by using preemptive EDF, then the score on $\mathtt{p}$ is equal to 0. Similarly, the score on both buses is equal to 0 if the memory subtasks are schedulable by non-preemptive EDF. The blocking time on the platform's cores is set to 0.

Algorithm 11 weights the fitness score for each non-schedulable individual. During our experimentation, it has been noticed that setting the fitness scores to give more importance to the schedulability on the inter-core bus (S2SB) yields better results, as they can vary greatly between individuals. In the experimental settings, we set $\alpha_1, \alpha_2, \alpha_3$ to $0.2, 0.6$ and 0.2 respectively. The algorithm adds non-schedulable individuals to S and proceeds to the selection step (lines 7-10).

6.4.4 Creating the next generation

Selection

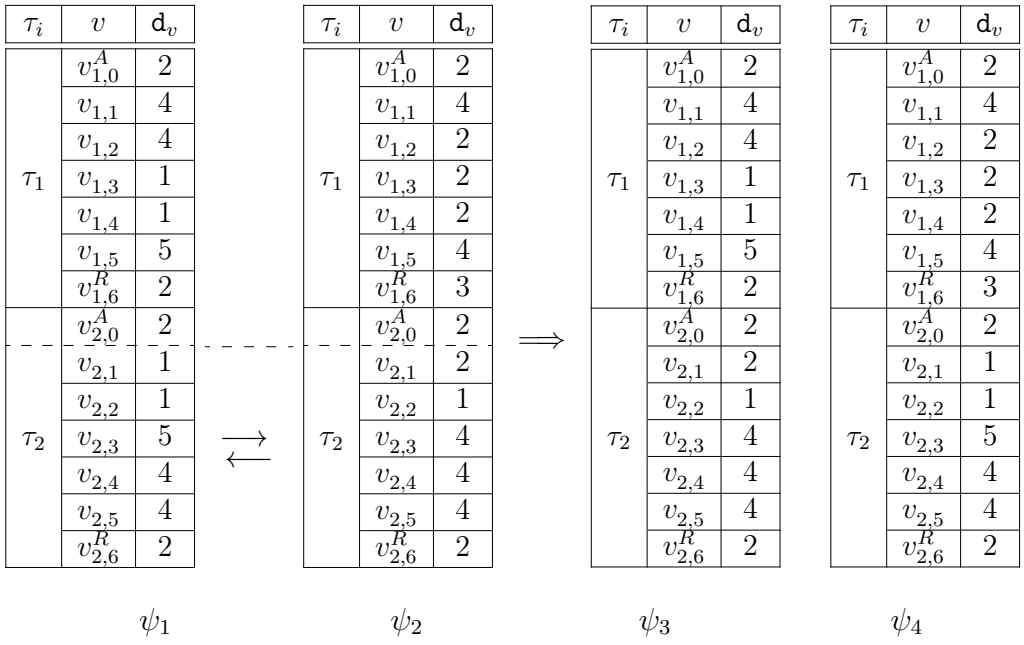
Before applying the genetic operators (crossover and mutation), the selection phase must be applied. Different approaches can be used to select the best individuals. A good comparative review of selection techniques in genetic algorithms can be found in [35]. We use rank selection in this work. One of the advantages of rank selection is its ability to maintain genetic diversity. Therefore, we order individuals by non-increasing order of their fitness score computed in Algorithm 11. Then, we select the 50% best individuals and replace the remaining 50% with new individuals created by applying the genetic operators.

Mutation and crossover

The mutation and crossover operations are applied to the best-selected individuals. The crossover creates new individuals and tries to improve the scheduling objective by exchanging partial information contained in two randomly selected individuals (parents). In our case, each child's individual Ψ_i receives local deadlines from the parents.

Several variants of the crossover are popular [106, 107, 127]. The original approach of the crossover operator is called `one-point` crossover: one crossover point on the two parent individuals is selected, and all local deadlines beyond that point are swapped between the two parents. This approach can be generalized to a multi-point operator, where the number of points is chosen randomly. It can be further generalised by copying local deadlines from the first parent with a probability p and from the second parent with a probability $1 - p$. The case $p = 0.5$ is called `uniform` crossover.

In this work, we use the `one-point` crossover operator. When two individuals are chosen for crossover, a crossover point is randomely selected at the same position for both individuals. This point divides each individual into two distinct



τ_i	v	$\mathtt{d}_v$
τ_1	$v^A_{1,0}$	2
	$v_{1,1}$	4
	$v_{1,2}$	4
	$v_{1,3}$	1
	$v_{1,4}$	1
	$v_{1,5}$	5
	$v^R_{1,6}$	2
τ_2	$v^A_{2,0}$	2
	$v_{2,1}$	1
	$v_{2,2}$	1
	$v_{2,3}$	5
	$v_{2,4}$	4
	$v_{2,5}$	4
	$v^R_{2,6}$	2

ψ_1

τ_i	v	$\mathtt{d}_v$
τ_1	$v^A_{1,0}$	2
	$v_{1,1}$	4
	$v_{1,2}$	2
	$v_{1,3}$	2
	$v_{1,4}$	2
	$v_{1,5}$	4
	$v^R_{1,6}$	3
τ_2	$v^A_{2,0}$	2
	$v_{2,1}$	2
	$v_{2,2}$	1
	$v_{2,3}$	4
	$v_{2,4}$	4
	$v_{2,5}$	4
	$v^R_{2,6}$	2

ψ_2

τ_i	v	$\mathtt{d}_v$
τ_1	$v^A_{1,0}$	2
	$v_{1,1}$	4
	$v_{1,2}$	4
	$v_{1,3}$	1
	$v_{1,4}$	1
	$v_{1,5}$	5
	$v^R_{1,6}$	2
τ_2	$v^A_{2,0}$	2
	$v_{2,1}$	2
	$v_{2,2}$	1
	$v_{2,3}$	4
	$v_{2,4}$	4
	$v_{2,5}$	4
	$v^R_{2,6}$	2

ψ_3

τ_i	v	$\mathtt{d}_v$
τ_1	$v^A_{1,0}$	2
	$v_{1,1}$	4
	$v_{1,2}$	2
	$v_{1,3}$	2
	$v_{1,4}$	2
	$v_{1,5}$	4
	$v^R_{1,6}$	3
τ_2	$v^A_{2,0}$	2
	$v_{2,1}$	1
	$v_{2,2}$	1
	$v_{2,3}$	5
	$v_{2,4}$	4
	$v_{2,5}$	4
	$v^R_{2,6}$	2

ψ_4

Table 6.2: Swapping local deadlines after a crossover point (ψ_3 and ψ_4 are new individuals)

parts: a head and a tail. The head of the first individual is combined with the tail of the second individual, and a similar operation is performed for the second individual. This process results in the creation of two new individuals (see Table 6.2).

The mutation operator is used to introduce genetic diversity in the population. With this operator, either we randomly change the value of a gene or we swap the positions of two of them. For instance, [107] uses swap, insertion, and inversion mutations. In this work, we randomly select an individual ψ, a DAG task τ_i and a subtask v. The local deadline for v is randomly modified, creating a new child individual. The new deadline is selected randomly in the interval $\mathtt{I} = [\mathtt{b\_inf}, \mathtt{b\_sup}]$ where $\mathtt{b\_inf} = \phi_v + \mathsf{C}(v)$ and $\mathtt{b\_sup}$ is the subtask local deadline. We then derive the new intermediate deadline of v, and we adjust the offset of its immediate successors.

The mutation and crossover rates, η_{mu} and η_{cr} respectively, are used to compute the number of individuals to generate to replace those eliminated by the selection operation.

6.5 Results and discussions

In this section, we will evaluate the performance of the proposed approaches in comparison to related work. Our contributions include allocation, deadlines assignment, and offsets assignment. First, we will assess the performance of our allocation strategy compared to bin-packing allocation heuristics such as Best Fit (BF) and Worst Fit (WF). Additionally, we will compare our deadline

assignment techniques to classical deadline assignment heuristics, specifically *fair* and *proportional* approaches.

To evaluate the performance of these techniques, we measured the schedulability rate and the runtime on two different platforms. The first platform features 4 identical cores, while the second platform features 6 identical cores.

6.5.1 Task generation

Experiments have been conducted on a large number of randomly generated synthetic task sets. The task generation process takes as input the number of tasks n, the target baseline utilization of the task set, and graph generation parameters.

First, the algorithm generates n task utilization using the UUnifast algorithm [71] such that their total sum is equal to the target baseline utilization. For each task, we randomly select a period from a predefined list of periods: $\{15000, 12000, 20000, 24000, 30000, 10000, 40000, 60000\}$. This allows us to control the schedulability analysis complexity and avoid intractable hyper-periods. The task deadline is set equal to $0.8 \cdot \mathsf{T}$.

Next, we compute the utilization for acquisition and restitution subtasks by inflating the task utilization by the $\texttt{stall}_{\texttt{A/R}}$ parameter, which is set to 0.05 for each, that is tasks will spend 5% of their worst-case execution time (WCET) receiving and sending data from/to the main memory. We then use the UUnifast-discard algorithm again to distribute the remaining task utilization among the subtasks of the considered task. We set the number of computation subtasks to 8. The execution time of each subtask is computed by multiplying the subtask utilization by the task period. Furthermore, we generate precedence constraints between the different subtasks using the *layer-by-layer* method [70]. It is known that the bahavior of the classical deadline assignment depends on the graph structure. Therefore, we generate large-DAG or long-DAG. For large DAGs, we randomly select the number of subtasks per layer between 3 and 5. For long DAG tasks, the number of subtasks per layer is either 2 or 3. We consider a probability $\gamma = 0.2$[2] to create a precedence constraint between two subtasks from different layers. Precedence between subtasks belonging to non-consecutive layers is allowed. We guarantee that the task is weakly connected to ensure the absence of isolated subtasks.

Communication subtasks are automatically inserted between every two dependent computation subtasks. The execution time of a communication subtask $\texttt{stall}_{\texttt{m}}$ is set equal to 0.2 of the execution time of its immediate predecessor. This value is subtracted from the predecessor's execution time to maintain the baseline utilization for the task graph unchanged.

COTS platforms with scratchpads are typically microcontrollers with limited computing capacity. Therefore, they are not able to handle highly complex software composed of hundreds of subtasks, compared to more powerful processors. Consequently, we believe that a setting with hendreds, and thousands of subtasks can not be representative to the applications that can be supported by

[2]We used the same value as in the literature [7].

scratchpad-based platforms.

6.5.2 Simulation results and discussions

We conducted experiments where we varied the baseline utilization from 0 to the number of cores by a step of 0.4. For each utilization value, we generated 100 task sets, with each task set containing 8 AECR-DAG tasks. We set the upper bounds for core utilization as $\mathsf{U}^{\mathsf{max}} = 0.85$ or $\mathsf{U}^{\mathsf{max}} = 0.7$.

For the genetic algorithm, we explored two types of scenarios: a small population class with 50 individuals and a large population class with 150 individuals. The genetic algorithm stops after either 50 generations for small populations or 100 generations for the large population class.

The experiments were conducted on a 12th Gen Intel(R) Core(TM) i7-1255U processor with 16 GB of RAM. We used CPLEX as the ILP solver for our experiments.

Comparison of allocation approaches

First, we conducted a study to analyze the impact of subtask-to-core allocation on the elimination of unnecessary communications. We compared our proposed approach against two well-known allocation heuristics: Worst-Fit (WF) and Best-Fit (BF). The experiments were conducted on a platform with 4 identical cores, using the same settings for the genetic algorithm (mutation probability η_{mu} and crossover probability η_{cr} set to 0.5). The experiments were performed for both small and large populations, as well as for long and large DAGs. Each combination is labeled as ILP, BF, or WF, representing the ILP, Best-Fit, or Worst-Fit allocation strategy, with GA denoting the genetic algorithm. A number represents an upper bound of the processor workload (0.7 and 0.85), and optionally, S or L denotes the small or the large population class.

In Figure 6.6, we present the results for the schedulability rate of the large graph DAG as a function of the target baseline utilization. We are specifically interested in examining the impact of the processor workload upper bound on the schedulability ratio.

ILP-based allocation outperforms all other approaches with the same parameters, as it effectively eliminates unnecessary communications compared to the BF and WF approaches. The WF approach itself dominates the BF approach, as the BF approach tends to allocate a maximum number of subtasks on the same core, resulting in a highly loaded core compared to the other approaches. This jeopardizes schedulability, making it weaker compared to the other approaches.

When the processor workload upper bound is set to 0.7, all the allocation strategies outperform the same strategy with the upper bound set to 0.85. This unloads the different cores, providing more laxity to find feasible definitions of the intermediate deadlines. However, the total workload to schedule decreases to 2.8, which is 0.7 multiplied by the number of cores. This reduction is still reasonable since allowing more workload (0.85) per core does not improve schedulability at high workloads.

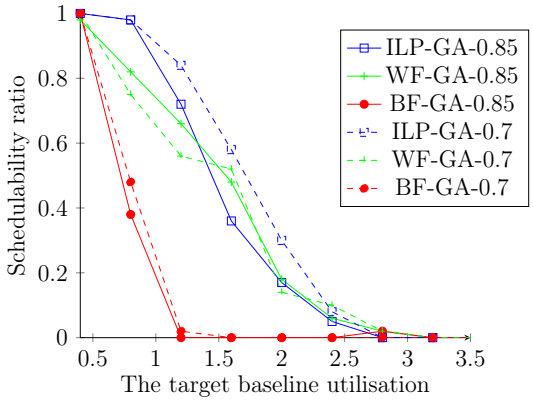


Figure 6.6: Schedulability rate for large DAG on Small poplation: $\mathsf{U}^{\mathsf{max}} = 0.7$ vs $\mathsf{U}^{\mathsf{max}} = 0.85$

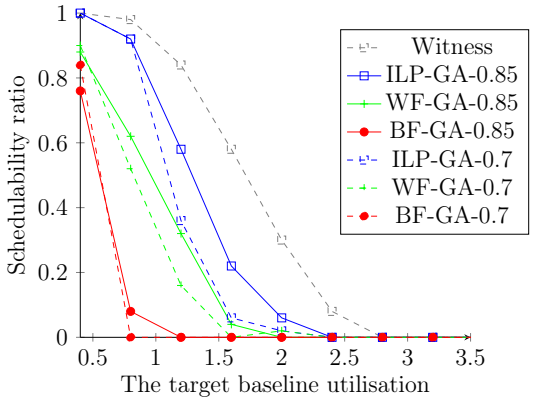


Figure 6.7: Schedulability rate for long DAG on Small poplation: $\mathsf{U}^{\mathsf{max}} = 0.7$ vs $\mathsf{U}^{\mathsf{max}} = 0.85$

Figure 6.7 presents the same experiments as Figure 6.6, but for long DAGs. To provide an indication, we include the best results from the previous figure in gray[3]. All allocation strategies perform similarly to those in the previous figure. However, the schedulability has slightly decreased. This reduction can be attributed to the presence of long graphs, which offer less flexibility in allocating intermediate deadlines. Although long graphs leverage parallel execution, they significantly impact the schedulability of individual tasks.

In Figure 6.8, we examine the impact of increasing the population size on schedulability as a function of the total workload. We compare different approaches with a workload threshold set to 0.7 for all cases. As expected, a larger population size (150 individuals) improves schedulability compared to an equivalent approach with a smaller population size (50 individuals). Even the WF-based allocation with a large population outperforms the ILP-based allocation with a smaller population (compared to the previous experimentations). This is due to the fact that a larger population allows for more diversification,

[3]It should be noted that the previous experiment was conducted on a different task set, so the comparison is made for the sake of indication rather than direct comparability

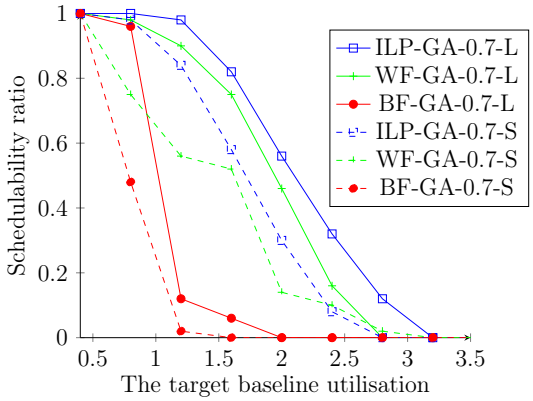


Figure 6.8: Schedulability for large DAGs at $\mathsf{U}^{\mathsf{max}} = 0.7$: small vs large population

and iterating for 150 generations instead of 50 generations allows for more intensive search. However, the Best-Fit approach still exhibits poor performance because it overly constrains certain cores compared to others.

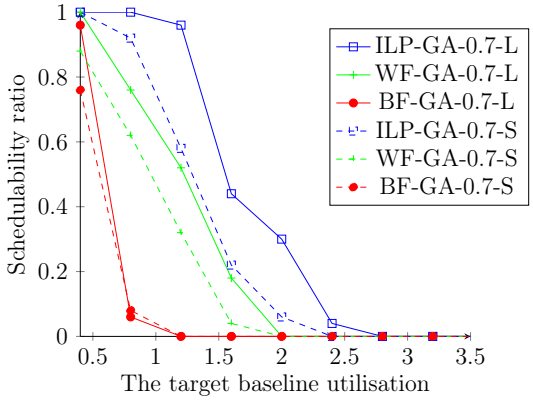
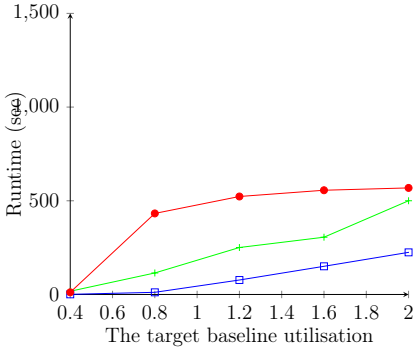


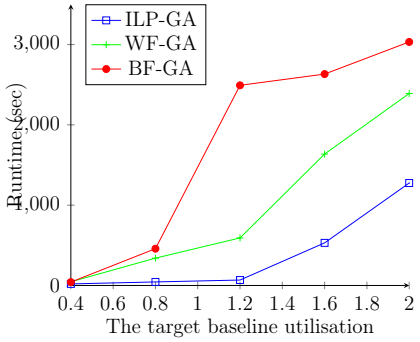
Figure 6.9: Schedulability rate for long DAGs: $\mathsf{U}^{\mathsf{max}} = 0.7$ small vs large population

In Figure 6.9, we present the results of the same experiments as Figure 6.8, but specifically for long DAGs. The results are comparable to the previous figure, with some notable differences. The ILP approach still outperforms the WF approach, even with a smaller population size. This is explained by the nature of long DAGs in our task set generation algorithm, where we have no more than two subtasks per layer, therefore, the slack time distributed over the different subtasks is smaller compared to large DAGs. In such a highly constrained system, the impact of diversified and intensive search for intermediate deadlines is minimal compared the subtask-to-core allocation, which has a more significant impact in determining schedulability. Therefore, based on the configuration of our tasks, it may be more important to prioritize intensive intermediate deadline search for large DAGs or focus on employing more intensive allocation strategies for long DAGs, to allow the deadline assignment techniques to be more efficient.

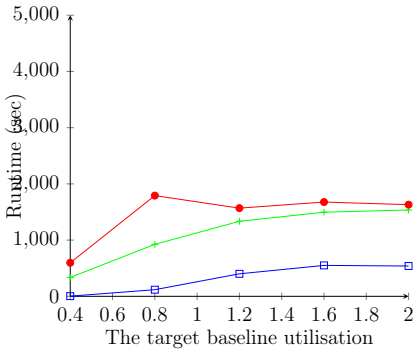
In Figure 6.10, we provide the time required to perform the allocation and



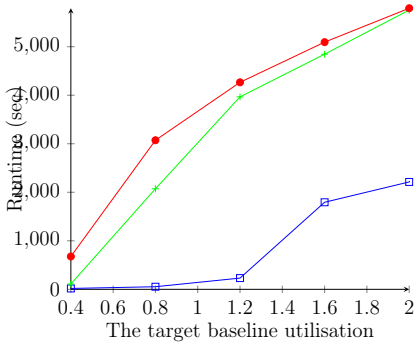
(a) Runtime for large-DAGs on small population($\mathsf{U}^{\mathsf{max}} = 0.85$, 50 generations)



(b) Runtime for large-DAGs on large population ($\mathsf{U}^{\mathsf{max}} = 0.7$, 100 generations)



(c) Runtime for long-DAGs on small population ($\mathsf{U}^{\mathsf{max}} = 0.85$, 50 generations)



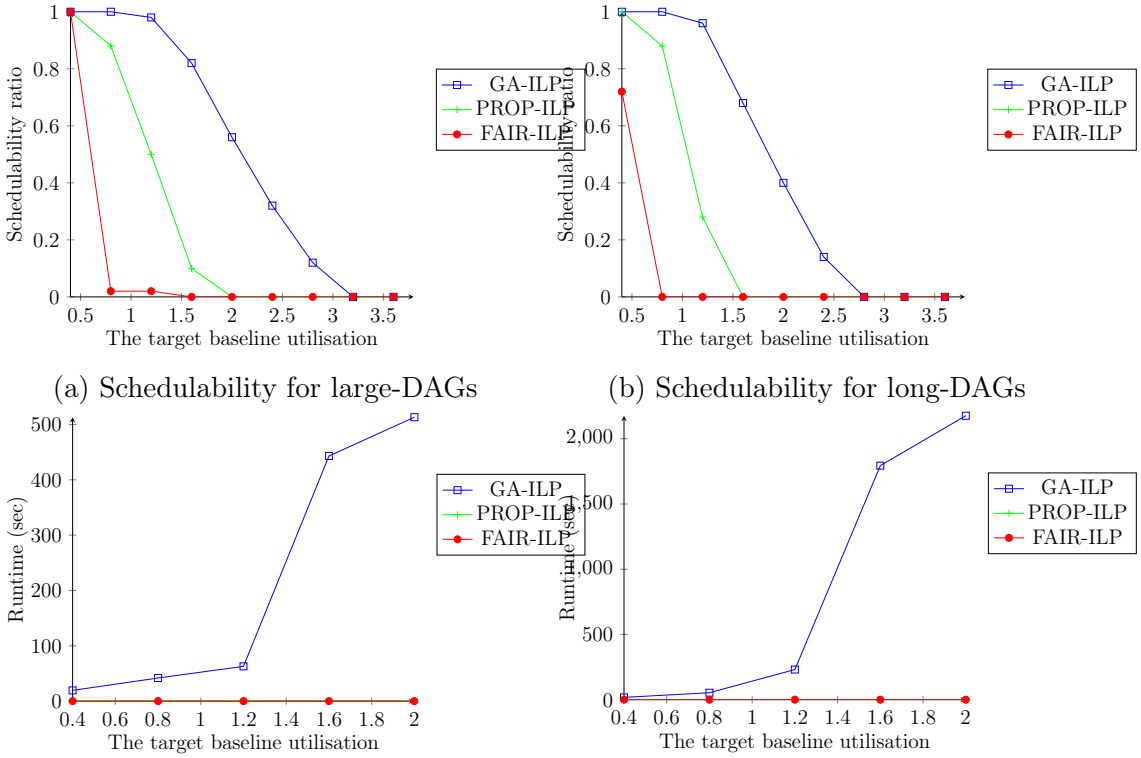
(d) Runtime for long-DAGs on large population($\mathsf{U}^{\mathsf{max}} = 0.7$, 100 generations)

Figure 6.10: ILP's performances VS (WF and BF) algorithms when using GA for deadline assignment

schedulability analysis for the previous experiments as a function of total utilization. We have limited the total utilization to 2, as having a smaller number of schedulable task sets beyond this limit does not allow drawing statistically significant and meaningful conclusions. Notably, the average time required to assess schedulability for large graphs (the two figures on the left-hand side) is generally shorter than that for the analysis of long DAGs (please note that the scales of the figures are different). On average, the BF heuristic takes more time because schedulability testing fails more frequently compared to the other approaches. Therefore, the analysis continues until the maximum number of iterations is reached, unlike the other approaches, which may terminate earlier if a schedulable solution is found in intermediate generations. Similarly, the ILP approach, which exhibits the best schedulability rates, completes its analysis faster than all the other approaches. Here, the required time to achieve a *good* allocation through the ILP is recovered by the effiency of the corresponding deadline assignment process.

Comparaison of Deadline Assignement Approches

In this section, we evaluate the performance of our genetic algorithm-based deadline assignment approach (Algorithm 7) against two deadline assignment methods: the fair deadline assignment heuristic denoted as FAIR and the proportional deadline assignment heuristic denoted as PROP. The tasks in this experience are allocated using the proposed ILP formulation, since it demonstrates the best performances. Each algorithm is, therefore, labeled by a combination of these techniques. The workload upper bound is set to $\mathsf{U}^{\mathsf{max}} = 0.7$. We compare the schedulability and the required analysis time for these approaches.



(a) Schedulability for large-DAGs

(b) Schedulability for long-DAGs

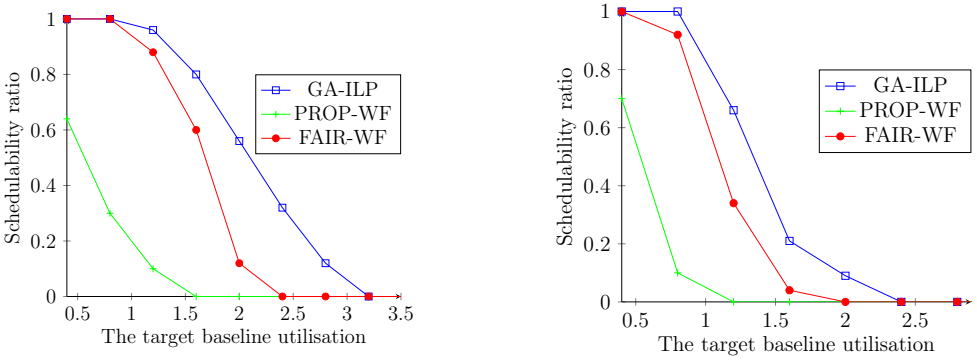
(c) Runtime for large-DAGs

(d) Runtime for long-DAGs

Figure 6.11: GA's performances VS (FAIR and PROP) when using ILP for task allocation on large population ($\mathsf{U}^{\mathsf{max}} = 0.7$)

In Figures (6.11a and 6.11b), we present the schedulability ratios of different deadline assignment techniques, namely GA, FAIR, and PROP, as a function of total utilization for large and long DAGs, respectively. Our genetic algorithm-based assignment technique significantly outperforms PROP and FAIR in both cases. The proportional approach assigns slack proportionally to the subtask execution time, which means that subtasks with longer execution times are more likely to receive additional slack compared to the FAIR approaches. As a result, these subtasks have more flexibility in terms of being scheduled without missing their deadlines. It is worth to notice that our approach is sensitive to the DAG topology, meaning that the structure and connections within the DAG can have an impact on the algorithm's performance, similarly to the related work.

Our approach improves the schedulability performance at an acceptable cost on the required time to achieve the analysis, *i.e.*, t requires more time to complete the analysis compared to the PROP and FAIR approaches (Figures 6.11c and 6.11d), which have negligible execution times in comparison to ours.



(a) Schedulability (Scenario 1)

(b) Schedulability (Scenario 2)

Figure 6.12: Compare GA's performances when using ILP for large-DAG tasks allocation with the literature (large population for GA, $\mathsf{U}^{\mathsf{max}} = 0.7$)

In Figures (6.12a and 6.12b), we compare our genetic algorithm with ILP-based allocation against the PROP and FAIR approaches using WF for subtask-to-core allocation, as a function of total utilization. In the left-hand side figure, we generate 10 DAGs, each consisting of 8 computational tasks, while in the right-hand side figure, we generate 3 DAGs, each consisting of 25 subtasks. The results indicate that our proposed approach consistently outperforms the approaches found in the literature. Our algorithm demonstrates that its performance is not dependent on the number of DAGs or the number of subtasks.

6.6 Conclusion

In this chapter, we presented an extension of our contribution presented in the previous chapter. The aim is to avoid contention for DAG tasks. In order to achieve this goal, we extended the DAG task model to include memory transfers, and we named it AECR-DAG. This model was used along with scratchpad memories. We proposed an ILP-based allocation strategy for AECR-DAG task sets allocation and a genetic algorithm-based technique for task scheduling parameter determination. We presented our experiments, which show a significant improvement in the system's performance compared to the state-of-the-art.

Chapter 7

Conclusion and perspectives

7.1 Conclusion

In multicore architectures, resources such as main memory or communication buses are shared and competed for among all CPU cores and I/O peripherals. Because of this memory bottleneck, contentions for access to main memory can significantly delay data fetch in architectures with caches, whenever a task suffers a cache miss. This effect greatly increases the task's worst-case execution time. Since more resources can simultaneously compete for access to main memory in a multicore architecture, execution time of memory intensive tasks can grow linearly with the number of cores in the system; in the worst case [74]. The use of task models proposed recently in the literature such as PREM or AER models where the task code is divided into (one or two) non-preemptive memory phases and a pure computation phase allow to reduce cache related delays since no preemption is allowed at memory phases level, however, it does not reduce memory contentions, in particular those related to the communication bus and main memory.

In this thesis, we had chosen to focus on the shared memory interference reduction through processor and memory co-scheduling for real-time applications. Firstly, we co-scheduled a set of real-time tasks on a multicore platform with shared cache memories, where the tasks were modeled using one of the limited preemption models, which is the DFPP model. To reach a higher level of predictability, phased task models (PREM and AER) were used in Chapter 5 and Chapter 6, where we addressed the problem on a multicore platform with scratchpads as an alternative to cache memories, as they (i.e., scratchpads) offer better timing predictability compared to caches. We co-scheduled memory phases in order to avoid contention. In Chapter 5, we explored and compared different designs for scheduling memory phases: a time-triggered memory scheduling approach and an on-line scheduling approach for memory phases of PREM task sets. We extended our scheduling to directed acyclic task-graphs respecting the AER model in Chapter 6. We proposed the AECR-DAG task model consisting of dependent subtasks and their respective communications, and we used a genetic algorithm to derive bus scheduling parameters for task sets.

The results obtained by testing a very large set of experiments have shown that our scheduling approaches are effective and improve up to 50% the schedulability compared to equivalent schedules generated with the state-of-the-art methods. We experimentally demonstrated the applicability of our methodology proposed in Chapter 5 on the Infineon AURIX TC-397 multicore family of processors using different benchmarks.

7.2 Limitations and perspectives

Global scheduling

All the contributions reported in this thesis consider only partitioned scheduling for which well-known analysis exist in the literature. We plan to extend our analysis to global scheduling.

Heterogenous architectures

The interest in autonomous vehicles is growing constantly. Given the high computational requirements of these systems, they rely on integrated heterogeneous multicore. However, their real-time requirements constitute an obstacle. In general, they couple the general purpose processors and different accelerators such as *GPUs*. Typically, they rely on a shared-memory organization, where the aforementioned compute units are interconnected through a shared bus to the main memory. As the number of compute engines grows, the main memory is subject to increasing contention.

For integrated devices, it has been shown in [22] that the GPU is highly susceptible to memory interference from the CPU. These memory interferences are significant even into a single GPU in particular into cache-based GPUs, where on-chip L2 cache memory is often shared across all the multiple *Streaming Multiprocessors (SMs)* composing the GPU, and hence, being susceptible to data evictions. In scratchpad-based GPUs, each SM has its own private scratchpad memory, which can be used by thread blocks that are launched in it. The number of thread blocks that are actually launched in an SM depends on the amount of scratchpad memory available in the SM.

It is clear that the SM-related interference needs to be mitigated in order to improve system's schedulability. We are convinced that the use of scratchpad memories in such systems combined with phased models to enforce a co-scheduling mechanism can help us to reduce memory interferences.

To get the PREM version, we can use *warp specialization* [53], presented in [21], since it provides a way to separate GPU programs into memory and computation phases. How to realize PREM on CPU is presented in [61]. In this context, we plan to extend our scheduling techniques, which are promising to significantly improve schedulability.

Personal publications

[1] I. Senoussaoui, G. Lipari, H.-E. Zahaf, and M. K. Benhaoua. "Memory-processor co-scheduling of AECR-DAG real-time tasks on partitioned multicore platforms with scratchpads". In: *under final revision to Journal Of System Architecture*. 2023, pp. 11–15.

[2] I. Senoussaoui, M. K. Benhaoua, H.-E. Zahaf, and G. Lipari. "Toward memory-centric scheduling for PREM task on multicore platforms, when processor assignments are specified". In: *2022 3rd International Conference on Embedded & Distributed Systems (EDiS)*. IEEE. 2022, pp. 11–15.

[3] I. Senoussaoui, H.-E. Zahaf, G. Lipari, and K. M. Benhaoua. "Contention-free scheduling of PREM tasks on partitioned multicore platforms". In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2022, pp. 1–8.

[4] I. Senoussaoui, H.-E. Zahaf, M. K. Benhaoua, G. Lipari, and R. Olejnik. "Allocation of real-time tasks onto identical core platforms under deferred fixed preemption-point model". In: *Proceedings of the 28th International Conference on Real-Time Networks and Systems*. 2020, pp. 34–43.

Bibliography

[1] J. el Goossens, U. L. de Bruxelles, S. Baruah, and S. Funk. "Real-time Scheduling on Multiprocessors". In: ().

[2] A. IT. "AURIX 32-bit microcontrollers for automotive and industrial applications". In: *Infineon Technologies AG* 1 ().

[3] A. Kernel-to-userspace. "Linux kernel". In: ().

[4] C. U. Manual. "Ibm Cplex Optimization studio". In: *Version* 12 ().

[5] I. Senoussaoui, G. Lipari, H.-E. Zahaf, and M. K. Benhaoua. "Memory-processor co-scheduling of AECR-DAG real-time tasks on partitioned multicore platforms with scratchpads". In: *under final revision to Journal Of System Architecture*. 2023, pp. 11–15.

[6] T. Thilakasiri and M. Becker. "Methods to Realize Preemption in Phased Execution Models". In: *ACM Trans. Embed. Comput. Syst.* 22.5s (Sept. 2023). DOI: 10.1145/3609132.

[7] S. Ben-Amor and L. Cucu-Grosjean. "Graph reductions and partitioning heuristics for multicore DAG scheduling". In: *Journal of Systems Architecture* 124 (2022), p. 102359.

[8] I. Senoussaoui, M. K. Benhaoua, H.-E. Zahaf, and G. Lipari. "Toward memory-centric scheduling for PREM task on multicore platforms, when processor assignments are specified". In: *2022 3rd International Conference on Embedded & Distributed Systems (EDiS)*. IEEE. 2022, pp. 11–15.

[9] I. Senoussaoui, H.-E. Zahaf, G. Lipari, and K. M. Benhaoua. "Contention-free scheduling of PREM tasks on partitioned multicore platforms". In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2022, pp. 1–8.

[10] J. Arora, C. Maia, S. Aftab Rashid, G. Nelissen, and E. Tovar. "Bus-Contention Aware Schedulability Analysis for the 3-Phase Task Model with Partitioned Scheduling". In: *29th International Conference on Real-Time Networks and Systems*. 2021, pp. 123–133.

[11] Z. Houssam-Eddine, N. Capodieci, R. Cavicchioli, G. Lipari, and M. Bertogna. "The HPC-DAG task model for heterogeneous real-time systems". In: *IEEE Transactions on Computers* 70.10 (2020), pp. 1747–1761.

[12] G. Schwäricke, T. Kloda, G. Gracioli, M. Bertogna, and M. Caccamo. "Fixed-priority memory-centric scheduler for COTS-based multiprocessors". In: *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020.

[13] I. Senoussaoui, H.-E. Zahaf, M. K. Benhaoua, G. Lipari, and R. Olejnik. "Allocation of real-time tasks onto identical core platforms under deferred fixed preemption-point model". In: *Proceedings of the 28th International Conference on Real-Time Networks and Systems*. 2020, pp. 34–43.

[14] B.-A. Slim, C.-G. Liliana, M. Mezouak, and Y. Sorel. "Probabilistic Schedulability Analysis for Real-time Tasks with Precedence Constraints on Partitioned Multi-core". In: *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE. 2020, pp. 142–143.

[15] H.-E. Zahaf, G. Lipari, S. Niar, et al. "Preemption-Aware Allocation, Deadline Assignment for Conditional DAGs on Partitioned EDF". In: *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE. 2020, pp. 1–10.

[16] F. Fort and J. Forget. "Code generation for multi-phase tasks on a multi-core distributed memory platform". In: *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE. 2019, pp. 1–6.

[17] J. Real, S. Sáez, and A. Crespo. "A hierarchical architecture for time- and event-triggered real-time systems". In: *Journal of Systems Architecture* 101 (2019), p. 101652. DOI: https://doi.org/10.1016/j.sysarc.2019.101652.

[18] M. R. Soliman and R. Pellizzoni. "Prem-based optimal task segmentation under fixed priority scheduling". In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. 2019.

[19] R. Tabish, R. Mancuso, S. Wasly, R. Pellizzoni, and M. Caccamo. "A real-time scratchpad-centric OS with predictable inter/intra-core communication for multi-core embedded systems". In: *Real-Time Systems* 55.4 (2019), pp. 850–888.

[20] H.-E. Zahaf, N. Capodieci, R. Cavicchioli, M. Bertogna, and G. Lipari. "A C-DAG task model for scheduling complex real-time tasks on heterogeneous platforms: preemption matters". In: *arXiv preprint arXiv:1901.02450* (2019).

[21] B. Forsberg, A. Marongiu, and L. Benini. "GPUguard: Towards supporting a predictable execution model for heterogeneous SoC". In: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*. 2017, pp. 318–321. DOI: 10.23919/DATE.2017.7927008.

[22] B. Forsberg, D. Palossi, A. Marongiu, and L. Benini. "Gpu-accelerated real-time path planning and the predictable execution model". In: *Procedia Computer Science* 108 (2017), pp. 2428–2432.

[23] C. Maia, G. Nelissen, L. Nogueira, L. M. Pinho, and D. G. Pérez. "Schedulability analysis for global fixed-priority scheduling of the 3-phase task model". In: *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE. 2017, pp. 1–10.

[24] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte. "Contention-free execution of automotive applications on a clustered many-core platform". In: *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE. 2016, pp. 14–24.

[25] B. B. Brandenburg and M. Gül. "Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations". In: *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2016, pp. 99–110.

[26] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. "Schedulability analysis of conditional parallel task graphs in multicore systems". In: *IEEE Transactions on Computers* 66.2 (2016), pp. 339–353.

[27] H. Swahn. *Pthreads and OpenMP: A performance and productivity study*. 2016.

[28] H. E. Zahaf. "Energy efficient scheduling of parallel real-time tasks on heterogeneous multicore systems". PhD thesis. Université de Lille 1, Sciences et Technologies, 2016.

[29] H.-E. Zahaf, A. E. H. Benyamina, G. Lipari, R. Olejnik, and P. Boulet. "Modeling parallel real-time tasks with di-graphs". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 339–348.

[30] J. Cavicchio, C. Tessler, and N. Fisher. "Minimizing cache overhead via loaded cache blocks and preemption placement". In: *2015 27th Euromicro Conference on Real-Time Systems*. IEEE. 2015, pp. 163–173.

[31] R. I. Davis, A. Burns, J. Marinho, V. Nelis, S. M. Petters, and M. Bertogna. "Global and Partitioned Multiprocessor Fixed Priority Scheduling with Deferred Preemption". In: *ACM Trans. Embed. Comput. Syst.* 14.3 (Apr. 2015), 47:1–47:28. DOI: 10.1145/2739954.

[32] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni. "A survey on cache management mechanisms for real-time embedded systems". In: *ACM Computing Surveys (CSUR)* 48.2 (2015), pp. 1–36.

[33] S. J. Kang, S. Y. Lee, and K. M. Lee. "Performance comparison of OpenMP, MPI, and MapReduce in practical problems". In: *Advances in Multimedia* 2015 (2015), pp. 7–7.

[34] M. Qamhieh. "Scheduling of parallel real-time DAG tasks on multiprocessor systems". PhD thesis. Paris Est, 2015.

[35] A. Shukla, H. M. Pandey, and D. Mehrotra. "Comparative review of selection techniques in genetic algorithm". In: *2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE)*. 2015, pp. 515–519. DOI: 10.1109/ABLAZE.2015.7154916.

[36] H. Yun, G. Yao, et al. "Memory bandwidth management for efficient performance isolation in multi-core platforms". In: *IEEE Transactions on Computers* 65.2 (2015), pp. 562–576.

[37] A. Alhammad and R. Pellizzoni. "Time-predictable execution of multithreaded applications on multicore systems". In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2014, pp. 1–6.

[38] B. Chattopadhyay and S. Baruah. "Limited-preemption scheduling on multiprocessors". In: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. ACM. 2014, p. 225.

[39] G. Durrieu, M. Faugère, S. Girbal, D. G. Pérez, C. Pagetti, and W. Puffitsch. "Predictable flight management system implementation on a multicore processor". In: *Embedded Real Time Software (ERTS'14)*. 2014.

[40] "Integrating Cache-Related Pre-emption Delays into Analysis of Fixed Priority Scheduling with Pre-emption Thresholds". English. In: *Proceedings Real-Time Systems Symposium (RTSS 2014)*. IEEE, Dec. 2014, pp. 161–172. DOI: 10.1109/RTSS.2014.25.

[41] Y. Wu, Z. Gao, and G. Dai. "Deadline and activation time assignment for partitioned real-time application on multiprocessor reservations". In: *Journal of Systems Architecture* 60.3 (2014), pp. 247–257.

[42] O. Board. "OpenMP application program interface version 4.0 http://www. openmp. org/mpdocuments". In: *OpenMP4. 0.0. pdf* (2013).

[43] D. Dasari, B. Akesson, V. Nelis, M. A. Awan, and S. M. Petters. "Identifying the sources of unpredictability in COTS-based multicore systems". In: *2013 8th IEEE international symposium on industrial embedded systems (SIES)*. IEEE. 2013, pp. 39–48.

[44] M. Qamhieh, F. Fauberteau, L. George, and S. Midonnet. "Global EDF scheduling of directed acyclic graphs on multiprocessor systems". In: *Proceedings of the 21st International conference on Real-Time Networks and Systems*. 2013, pp. 287–296.

[45] S. Altmeyer, R. I. Davis, and C. Maiza. "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems". In: *Real-Time Systems* 48.5 (2012), pp. 499–526.

[46] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. "A generalized parallel task model for recurrent real-time processes". In: *2012 IEEE 33rd Real-Time Systems Symposium*. IEEE. 2012, pp. 63–72.

[47] G. C. Buttazzo, M. Bertogna, and G. Yao. “Limited preemptive scheduling for real-time systems. a survey”. In: *IEEE transactions on Industrial Informatics* 9.1 (2012), pp. 3–15.

[48] G. C. Buttazzo, M. Bertogna, and G. Yao. “Limited preemptive scheduling for real-time systems. a survey”. In: *IEEE Transactions on Industrial Informatics* 9.1 (2012), pp. 3–15.

[49] R. I. Davis and M. Bertogna. “Optimal fixed priority scheduling with deferred pre-emption”. In: *2012 IEEE 33rd Real-Time Systems Symposium*. IEEE. 2012, pp. 39–50.

[50] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. “Memory-centric scheduling for multicore hard real-time systems”. In: *Real-Time Systems* 48.6 (2012), pp. 681–715.

[51] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. “Survey of scheduling techniques for addressing shared resources in multicore processors”. In: *ACM Computing Surveys (CSUR)* 45.1 (2012), pp. 1–28.

[52] S. Altmeyer and C. M. Burguière. “Cache-related preemption delay via useful cache blocks: Survey and redefinition”. In: *Journal of Systems Architecture* 57.7 (2011), pp. 707–719.

[53] M. Bauer, H. Cook, and B. Khailany. “CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’11. Seattle, Washington: Association for Computing Machinery, 2011. DOI: 10.1145/2063384.2063400.

[54] M. Bertogna, G. Buttazzo, and G. Yao. “Improving feasibility of fixed priority tasks using non-preemptive regions”. In: *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE. 2011, pp. 251–260.

[55] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazo. “Optimal selection of preemption points to minimize preemption overhead”. In: *23rd Euromicro Conference on Real-Time Systems*. 2011, pp. 217–227.

[56] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo. “Optimal selection of preemption points to minimize preemption overhead”. In: *2011 23rd Euromicro Conference on Real-Time Systems*. IEEE. 2011, pp. 217–227.

[57] G. Buttazzo. “A General View”. In: vol. 24. Sept. 2011, pp. 1–22. DOI: 10.1007/978-1-4614-0676-1_1.

[58] R. I. Davis and A. Burns. “A survey of hard real-time scheduling for multiprocessor systems”. In: *ACM computing surveys (CSUR)* 43.4 (2011), pp. 1–44.

[59] R. I. Davis and A. Burns. “A survey of hard real-time scheduling for multiprocessor systems”. In: *ACM computing surveys (CSUR)* 43.4 (2011), p. 35.

[60] S. Mohan, M. Caccamo, L. Sha, R. Pellizzoni, G. Arundale, R. Kegley, et al. "Using multicore architectures in cyber-physical systems". In: *Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems from Components*. 2011, p. 86.

[61] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, et al. "A predictable execution model for COTS-based embedded systems". In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2011, pp. 269–279.

[62] G. Yao, G. Buttazzo, and M. Bertogna. "Feasibility analysis under fixed priority scheduling with limited preemptions". In: *Real-Time Systems* 47.3 (2011), pp. 198–223.

[63] G. Yao, G. Buttazzo, and M. Bertogna. "Feasibility analysis under fixed priority scheduling with limited preemptions". In: *Real-Time Systems* 47.3 (2011), pp. 198–223.

[64] S. Altmeyer, C. Maiza, and J. Reineke. "Resilience analysis: tightening the CRPD bound for set-associative caches". In: *ACM sigplan notices* 45.4 (2010), pp. 153–162.

[65] B. Barney et al. "Introduction to parallel computing". In: *Lawrence Livermore National Laboratory* 6.13 (2010), p. 10.

[66] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. "Improved multiprocessor global schedulability analysis". In: *Real-Time Systems* 46.1 (2010), pp. 3–24.

[67] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. "An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers". In: *2010 31st IEEE Real-Time Systems Symposium*. IEEE. 2010, pp. 14–24.

[68] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo. "Preemption points placement for sporadic task sets". In: *2010 22nd Euromicro Conference on Real-Time Systems*. IEEE. 2010, pp. 251–260.

[69] S. Blagodurov, S. Zhuravlev, and A. Fedorova. "Contention-aware scheduling on multicore systems". In: *ACM Transactions on Computer Systems (TOCS)* 28.4 (2010), pp. 1–45.

[70] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner. "Random graph generation for scheduling simulations". In: *3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*. ICST. 2010, p. 10.

[71] P. Emberson, R. Stafford, and R. I. Davis. "Techniques for the synthesis of multiprocessor tasksets". In: *WATERS*. 2010.

[72] U. Keskin, R. J. Bril, and J. J. Lukkien. "Exact response-time analysis for fixed-priority preemption-threshold scheduling". In: *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*. IEEE. 2010, pp. 1–4.

[73] S. P. Muralidhara, M. Kandemir, and P. Raghavan. “Intra-application cache partitioning”. In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, pp. 1–12.

[74] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. “Worst case delay analysis for memory interference in multicore systems”. In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE. 2010, pp. 741–746.

[75] V. Suhendra, A. Roychoudhury, and T. Mitra. “Scratchpad allocation for concurrent embedded software”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32.4 (2010), pp. 1–47.

[76] Y. Yang, M. Wang, H. Yan, Z. Shao, and M. Guo. “Dynamic scratchpad memory management with data pipelining for embedded systems”. In: *Concurrency and Computation: Practice and Experience* 22.13 (2010), pp. 1874–1892.

[77] G. Yao, G. Buttazzo, and M. Bertogna. “Comparative evaluation of limited preemptive methods”. In: *2010 IEEE 15th Conference on Emerging Technologies and Factory Automation (ETFA 2010)*. 2010, pp. 1–8. DOI: 10.1109/ETFA.2010.5641199.

[78] R. J. Bril, J. J. Lukkien, and W. F. Verhaegh. “Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption”. In: *Real-Time Systems* 42.1-3 (2009), pp. 63–119.

[79] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang. “A Survey of WCET Analysis of Real-Time Operating Systems”. In: *2009 International Conference on Embedded Software and Systems*. 2009, pp. 65–72. DOI: 10.1109/ICESS.2009.24.

[80] G. Yao, G. Buttazo, and B. Marko. “Bounding the maximum length of non-preemptive regions under Fixed priority”. In: *2009 15th IEEE Internationnal Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2009, pp. 351–360.

[81] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson. “On the scalability of real-time scheduling algorithms on multicore platforms: A case study”. In: *2008 Real-Time Systems Symposium*. IEEE. 2008, pp. 157–169.

[82] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. “Impact of cache partitioning on multi-tasking real time embedded systems”. In: *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2008, pp. 101–110.

[83] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. “Coscheduling of cpu and i/o transactions in cots-based embedded systems”. In: *2008 Real-Time Systems Symposium*. IEEE. 2008, pp. 221–231.

[84] L. Sha, S. Gopalakrishnan, X. Liu, and Q. Wang. "Cyber-physical systems: A new frontier". In: *2008 IEEE international conference on sensor networks, ubiquitous, and trustworthy computing (sutc 2008)*. IEEE. 2008, pp. 1–9.

[85] V. Suhendra and T. Mitra. "Exploring locking & partitioning for predictable shared caches on multi-cores". In: *Proceedings of the 45th annual Design Automation Conference*. 2008, pp. 300–303.

[86] M. Bertogna and M. Cirinei. "Response-time analysis for globally scheduled symmetric multiprocessor platforms". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. IEEE. 2007, pp. 149–160.

[87] P. A. Diaz-Gomez and D. F. Hougen. "Initial population for genetic algorithms: A metric approach." In: *Gem*. Citeseer. 2007, pp. 43–49.

[88] G. Gebhard and S. Altmeyer. "Optimal task placement to improve cache performance". In: *Proceedings of the 7th ACM & IEEE international conference on Embedded software*. 2007, pp. 259–268.

[89] I. Puaut and C. Pais. "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison". In: *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE. 2007, pp. 1–6.

[90] J. Rosen, A. Andrei, P. Eles, and Z. Peng. "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. IEEE. 2007, pp. 49–60.

[91] Y. Tan and V. Mooney. "Timing analysis for preemptive multitasking real-time systems with caches". In: *ACM Transactions on Embedded Computing Systems (TECS)* 6.1 (2007), 7–es.

[92] S. K. Baruah. "Resource sharing in EDF-scheduled systems: A closer look". In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE. 2006, pp. 379–387.

[93] C. Berg. "PLRU cache domino effects". In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2006.

[94] S. Baruah. "The limited-preemption uniprocessor scheduling of sporadic task systems". In: *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. IEEE. 2005, pp. 137–144.

[95] H. Kopetz. "Event-triggered versus time-triggered real-time systems". In: *Operating Systems of the 90s and Beyond: International Workshop Dagstuhl Castle, Germany, July 8–12 1991 Proceedings*. Springer. 2005, pp. 86–101.

[96] R. Pellizzoni and G. Lipari. "Feasibility analysis of real-time periodic tasks with offsets". In: *Real-Time Systems* 30.1-2 (2005), pp. 105–128.

[97] A. Albert et al. "Comparison of event-triggered and time-triggered concepts with regard to distributed control systems". In: *Embedded world* 2004 (2004), pp. 235–252.

[98] E. K. Burke, S. Gustafson, and G. Kendall. "Diversity in genetic programming: An analysis of measures and correlation with fitness". In: *IEEE Transactions on Evolutionary Computation* 8.1 (2004), pp. 47–62.

[99] S. K. Baruah. "Dynamic-and static-priority scheduling of recurring real-time tasks". In: *Real-Time Systems* 24 (2003), pp. 93–128.

[100] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. "The influence of processor architecture on the design and the results of WCET tools". In: *Proceedings of the IEEE* 91.7 (2003), pp. 1038–1054.

[101] S. Udayakumaran and R. Barua. "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems". In: *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. 2003, pp. 276–286.

[102] J. Regehr. "Scheduling tasks with mixed preemption relations for robustness to timing faults". In: *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.* IEEE. 2002, pp. 315–326.

[103] B. Andersson, S. Baruah, and J. Jonsson. "Static-priority scheduling on multiprocessors". In: *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*. IEEE. 2001, pp. 193–202.

[104] R. Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[105] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. "Dynamic management of scratch-pad memory space". In: *Proceedings of the 38th annual Design Automation Conference*. 2001, pp. 690–695.

[106] L. Wang and D.-Z. Zheng. "An effective hybrid optimization strategy for job-shop scheduling problems". In: *Computers & Operations Research* 28.6 (2001), pp. 585–596.

[107] D. Koonce and S.-C. Tsai. "Using data mining to find patterns in genetic algorithm solutions to a job shop schedule". In: *Computers & Industrial Engineering* 38.3 (2000), pp. 361–374.

[108] F. Mueller. "Timing analysis for instruction caches". In: *Real-time systems* 18 (2000), pp. 217–247.

[109] P. Puschner and A. Burns. "Guest editorial: A review of worst-case execution-time analysis". In: *Real-Time Systems* 18.2-3 (2000), pp. 115–128.

[110] E. Zitzler, K. Deb, and L. Thiele. "Comparison of multiobjective evolutionary algorithms: Empirical results". In: *Evolutionary computation* 8.2 (2000), pp. 173–195.

[111] J. Goossens and R. Devillers. *General response time computation for hard real-time asynchronous periodic task sets using static schedulers*. Tech. rep. Technical Report 401, Universit e Libre de Bruxelles, Belgium, 1999.

[112] T. Lundqvist and P. Stenstrom. “Timing anomalies in dynamically scheduled microprocessors”. In: *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054)*. IEEE. 1999, pp. 12–21.

[113] T. Lundqvist and P. Stenström. “An integrated path and timing analysis method based on cycle-level symbolic execution”. In: *Real-Time Systems* 17 (1999), pp. 183–207.

[114] Y. Wang and M. Saksena. “Scheduling fixed-priority tasks with preemption threshold”. In: *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No. PR00306)*. IEEE. 1999, pp. 328–335.

[115] S. K. Baruah. “Feasibility analysis of recurring branching tasks”. In: *Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No. 98EX168)*. IEEE. 1998, pp. 138–145.

[116] S. Lee, C.-G. Lee, M. Lee, S. L. Min, and C. S. Kim. “Limited preemptible scheduling to embrace cache memory in real-time systems”. In: *Languages, Compilers, and Tools for Embedded Systems*. Springer. 1998, pp. 51–64.

[117] A. K. Mok and D. Chen. “A multiframe model for real-time tasks”. In: *IEEE transactions on Software Engineering* 23.10 (1997), pp. 635–645.

[118] L. George, N. Rivierre, and M. Spuri. “Preemptive and non-preemptive real-time uniprocessor scheduling”. PhD thesis. Inria, 1996.

[119] J. A. Hoogeveen, J. K. Lenstra, and B. Veltman. “Preemptive scheduling in a two-stage multiprocessor flow shop is NP-hard”. In: *European Journal of Operational Research* 89.1 (1996), pp. 172–175.

[120] Y.-T. Li, S. Malik, and A. Wolfe. “Cache modeling for real-time software: Beyond direct mapped instruction caches”. In: *17th IEEE Real-Time Systems Symposium*. IEEE. 1996, pp. 254–263.

[121] J. Simonson and J. H. Patel. “Use of preferred preemption points in cache-based real-time systems”. In: *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*. IEEE. 1995, pp. 316–325.

[122] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo. “Implications of classical scheduling results for real-time systems”. In: *Computer* 28.6 (1995), pp. 16–25.

[123] A. Burns. *Preemptive priority based scheduling: An appropriate engineering approach*. University of York, Department of Computer Science, 1993.

[124] K. Jeffay, D. F. Stanat, and C. U. Martel. “On non-preemptive scheduling of periodic and sporadic tasks”. In: *IEEE real-time systems symposium*. US: IEEE. 1991, pp. 129–139.

[125] K. Jeffay, D. F. Stanat, and C. U. Martel. “On non-preemptive scheduling of period and sporadic tasks”. In: *[1991] Proceedings Twelfth Real-Time Systems Symposium* (1991), pp. 129–139.

[126] J. Korst, E. Aarts, J. K. Lenstra, and J. Wessels. "Periodic multiprocessor scheduling". In: *PARLE'91 Parallel Architectures and Languages Europe*. Springer. 1991, pp. 166–178.

[127] T. Starkweather, S. McDaniel, K. E. Mathias, L. D. Whitley, and C. Whitley. "A Comparison of Genetic Sequencing Operators." In: *ICGA*. 1991, pp. 69–76.

[128] T. P. Baker. "A stack-based resource allocation policy for realtime processes". In: *[1990] Proceedings 11th Real-Time Systems Symposium*. IEEE. 1990, pp. 191–200.

[129] S. K. Baruah, A. K. Mok, and L. E. Rosier. "Preemptively scheduling hard-real-time sporadic tasks on one processor". In: *[1990] Proceedings 11th Real-Time Systems Symposium*. IEEE. 1990, pp. 182–190.

[130] S. K. Baruah, L. E. Rosier, and R. R. Howell. "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor". In: *Real-time systems* 2.4 (1990), pp. 301–324.

[131] K. Ramamritham, J. A. Stankovic, and P.-F. Shiah. "Efficient scheduling algorithms for real-time multiprocessor systems". In: *IEEE Transactions on Parallel and Distributed systems* 1.2 (1990), pp. 184–194.

[132] S. Biyabani, J. Stankovic, and K. Ramamritham. "The integration of deadline and criticalness in hard real-time scheduling". In: *Proceedings. Real-Time Systems Symposium*. 1988, pp. 152–160. DOI: 10.1109/REAL.1988.51111.

[133] J. Y.-T. Leung and J. Whitehead. "On the complexity of fixed-priority scheduling of periodic, real-time tasks". In: *Performance evaluation* 2.4 (1982), pp. 237–250.

[134] M. L. Dertouzos. "Control Robotics: The Procedural Control of Physical Processes". In: *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*. Ed. by J. L. Rosenfeld. North-Holland, 1974, pp. 807–813.

[135] C. L. Liu and J. W. Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[136] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[137] G. E. Moore. "Gramming more components onto integrated circuits". In: *Electronics* 38 (1965), p. 8.

[138] A. M. Turing et al. "On computable numbers, with an application to the Entscheidungsproblem". In: *J. of Math* 58.345-363 (1936), p. 5.