

Laura Elena Rubio Anguiano

Control techniques for thermal-aware energy- efficient real time multiprocessor scheduling

Director/es

Briz Velasco, José Luis
Ramírez Treviño, Antonio

<http://zaguan.unizar.es/collection/Tesis>



© Universidad de Zaragoza
Servicio de Publicaciones

ISSN 2254-7606



Universidad
Zaragoza

Tesis Doctoral

**CONTROL TECHNIQUES FOR THERMAL-AWARE
ENERGY-EFFICIENT REAL TIME
MULTIPROCESSOR SCHEDULING**

Autor

Laura Elena Rubio Anguiano

Director/es

Briz Velasco, José Luis
Ramírez Treviño,, Antonio

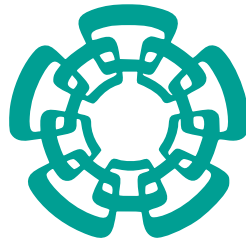
UNIVERSIDAD DE ZARAGOZA
Escuela de Doctorado

2022

Control techniques for thermal-aware energy-efficient real time multiprocessor scheduling

Laura Elena Rubio Anguiano

September, 2022



Cinvestav



Universidad
Zaragoza

A thesis submitted in total fulfillment for the degree of Doctor of Philosophy
in the Department of Systems and Computer Engineering

**Control techniques for thermal-aware
energy-efficient real time multiprocessor
scheduling**

Laura Elena Rubio Anguiano

Supervisors Antonio Ramírez Treviño
 José Luis Briz Velasco

September, 2022

Laura Elena Rubio Anguiano

Control techniques for thermal-aware energy-efficient real time multiprocessor scheduling

A thesis submitted in total fulfillment for the degree of Doctor of Philosophy in the Department of Systems and Computer Engineering, September, 2022

Scholarship granted by CONACYT, No. CVU 778073

Supervisors: Antonio Ramírez Treviño and José Luis Briz Velasco

Centro de Investigación y de Estudios Avanzados del I.P.N.

Unidad Guadalajara

Departamento de Control Automático

Avenida del Bosque 1145

45019

Zapopan, México

Universidad de Zaragoza

Grupo de Arquitectura de Computadores

Instituto de Investigación es Ingeniería de Aragón

Departamento de Informática e Ingeniería de sistemas

Calle Maria de Luna

50018

Zaragoza, España

Abstract

The use of multicore microprocessors is not only attractive to industry, but in many areas it is the only option. Real-time scheduling on these platforms is much more complex than on uniprocessors and generally worsens the over-design problem, leading to the use of more processors/cores than necessary. Algorithms based on fluid scheduling have been proposed to optimize the use of processors, but so far they have general drawbacks that distance them from their practical application, not the least of which is the high number of context switches and migrations.

This thesis is based on the hypothesis that it is possible to design algorithms inspired on fluid scheduling, which optimize the use of processors, complying with temporal, thermal and energy restrictions, with a low number of context switches and migrations, and compatible with both the off-line generation of cyclic executives attractive to the industry, as well as schedulers that integrate control techniques in execution time that allow the efficient management of both aperiodic tasks and parametric deviations or small disturbances.

In this regard, this thesis contributes with several solutions. First, it improves a model methodology that represents all the dimensions of the problem under a single formalism (Timed Continuous Petri Nets). Second, it proposes a method for generating a cyclic executive, calculated in processor cycles, for a set of hard real-time tasks on multiprocessors that optimizes the utilization of processing cores while also respecting thermal and energy constraints.

Considering the overhead due to the number of context switches and migrations in a cyclic executive calculated with a preemptive scheduler poses a causality dilemma: the number of context switches (and thus their overhead) is only known after the cyclic executive is calculated, but to properly calculate such cyclic executive, that number should be known. The thesis proposes a solution to this dilemma by means of an iterative method, with proven convergence, that manages to minimize the mentioned overload.

Specifically, the thesis manages to exploit the idea of fluid scheduling to maximize utilization (where maximizing utilization is a major issue in the industry) generating a simple cyclic executive with minimum overhead (overhead represents a big problem for schedules based on fluid scheduling).

Finally, a method is proposed to use the offline schedule as reference, i.e the cyclic executive, by an online frequency controller, so that small disturbances and parametric variations can be faced, integrating the management of aperiodic tasks (soft real time) while ensuring the integrity of the execution of the hard real time set.

These contributions constitute a novelty in the field, endorsed by the publications derived from this thesis work.

Resumen

La utilización de microprocesadores multinúcleo no sólo es atractiva para la industria sino que en muchos ámbitos es la única opción. La planificación tiempo real sobre estas plataformas es mucho más compleja que sobre monoprocesadores y en general empeoran el problema de sobre-diseño, llevando a la utilización de muchos más procesadores /núcleos de los necesarios. Se han propuesto algoritmos basados en planificación fluida que optimizan la utilización de los procesadores, pero hasta el momento presentan en general inconvenientes que los alejan de su aplicación práctica, no siendo el menor el elevado número de cambios de contexto y migraciones.

Esta tesis parte de la hipótesis de que es posible diseñar algoritmos basados en planificación fluida, que optimizan la utilización de los procesadores, cumpliendo restricciones temporales, térmicas y energéticas, con un bajo número de cambios de contexto y migraciones, y compatibles tanto con la generación fuera de línea de ejecutivos cíclicos atractivos para la industria, como de planificadores que integran técnicas de control en tiempo de ejecución que permiten la gestión eficiente tanto de tareas aperiódicas como de desviaciones paramétricas o pequeñas perturbaciones.

A este respecto, esta tesis contribuye con varias soluciones. En primer lugar mejora una metodología de modelo que representa todas las dimensiones del problema bajo un único formalismo (Redes de Petri Continuas Temporizadas). En segundo lugar, propone un método de generación de un ejecutivo cíclico, calculado en ciclos de procesador, para un conjunto de tareas tiempo real duro sobre multiprocesadores que optimiza la utilización de los núcleos de procesamiento respetando también restricciones térmicas y de energía, sobre la base de una planificación fluida.

Considerar la sobrecarga derivada del número de cambios de contexto y migraciones en un ejecutivo cíclico plantea un dilema de causalidad: el número de cambios de contexto (y en consecuencia su sobrecarga) no se conoce hasta generar el ejecutivo cíclico, pero dicho número no se puede minimizar hasta que se ha calculado. La tesis propone una solución a esta dilema mediante un método iterativo de convergencia demostrada que logra minimizar la sobrecarga mencionada.

En definitiva, la tesis consigue explotar la idea de planificación fluida para maximizar el utilización (donde maximizar la utilización es un gran problema en la industria)

generando un sencillo ejecutivo cíclico de mínima sobrecarga (ya que la sobrecarga implica un gran problema de los planificadores basados en planificación fluida).

Finalmente, se propone un método para utilizar las referencias de la planificación fuera de línea establecida en el ejecutivo cíclico para su seguimiento por parte de un controlador de frecuencia en línea, de modo que se pueden afrontar pequeñas perturbaciones y variaciones paramétricas, integrando la gestión de tareas aperiódicas (tiempo real blando) mientras se asegura la integridad de la ejecución del conjunto de tiempo real duro.

Estos aportaciones constituyen una novedad en el campo, refrendada por las publicaciones derivadas de este trabajo de tesis.

Acknowledgments

I would like to thank my advisors Antonio Ramírez Treviño and José Luis Briz Velasco for their patience, time, and guidance along this work.

This thesis was developed under a co-tutelage program between CINVESTAV and Universidad de Zaragoza, this could have not been possible without the aid of my advisor Dr. José Luis Briz, to whom I am deeply grateful.

In addition, thanks to CONACYT for the economic support provided during my PhD. To the GaZ (Group of Computer Architecture of the University of Zaragoza), which provided partial economic support through the Aragón Government T5820R research group funds, and grant PID2019-105660RB-C21, MCIN/ AEI /10.13039/501100011033 in which project I was invited to collaborate.

I also thank the Department of Computing and Systems Engineering of the University of Zaragoza for their support during my stay at the institution.

Contents

1. Introduction	1
1.1. Rationale	1
1.2. Thesis Objectives	3
1.3. Contributions	3
1.3.1. Summary of publications	4
1.4. Structure of this document	7
2. Background and prior work	9
2.1. Hardware Foundations	9
2.2. Real time systems	11
2.2.1. Task model	11
2.2.2. Scheduling concepts	13
2.2.3. Scheduling: from Uniprocessor to Multiprocessors	15
2.2.4. RT Multiprocessor scheduling	18
2.2.5. Control strategies in RT scheduling	25
2.2.6. Thermal and energy aware strategies in RT scheduling	26
2.3. System Model	27
2.3.1. Background on Petri Nets	28
2.3.2. TCPN global model	33
2.3.3. Global model	39
2.4. Tertimuss	40
3. Energy-Efficient Thermal-Aware RT Multiprocessor Scheduling	41
3.1. Problem definition	41
3.2. Off-line stage	43
3.2.1. Set of working frequencies	43
3.2.2. Workload computation	46
3.3. On-line stage	53
3.3.1. Scheduler	53
3.3.2. Aperiodic tasks and Adaptive Scheduler	55

3.4. Experimental Results	57
3.4.1. Temperature control and utilization	59
3.4.2. Handling aperiodic tasks	59
3.5. Conclusions	60
4. RT Multiprocessor Scheduling based on continuous control	65
4.1. Problem definition	66
4.2. Overview of the ALECS scheduling system	67
4.3. Task set conditioner	68
4.4. Pre-scheduler	68
4.4.1. Workload Assignment: ILP definition	68
4.5. Compute references	77
4.6. On-line controller ALECS	80
4.6.1. Allocation control	81
4.6.2. Execution control	83
4.6.3. Online Aperiodic Manager	85
4.7. Conclusions	87
5. Maximizing utilization and minimizing migration	89
5.1. Problem definition	90
5.2. Overview of the CALECS scheduling system	91
5.3. Off-line stage	92
5.3.1. Task set Conditioner and base example	93
5.3.2. Task clustering	94
5.3.3. Pre-schedule	99
5.3.4. Get-reference	100
5.4. On-line controller	101
5.4.1. TCPN equations in scalar form	102
5.4.2. Allocation control	103
5.4.3. Frequency control	104
5.4.4. Aperiodic Manager	107
5.5. Comparison with RUN	109
5.5.1. Simulation environment and setup	110
5.5.2. Migrations per job	111
5.5.3. Preemptions per job	114
5.6. Computational complexity	114
5.7. Conclusions	116

6. Accounting for preemption and migration costs on CE	119
6.1. The AdWECT algorithm	122
6.1.1. Convergence of AdWCET algorithm	126
6.2. DP-U, an approach based on the utilization	127
6.3. Experimental results	129
6.3.1. Experimental setup	130
6.3.2. Methodology for task generation	130
6.3.3. Results	132
6.4. Comparison between AdWCET and DP-U	135
6.5. Conclusion	136
7. Conclusion	139
7.1. Summary of contributions	139
7.2. Complexity	141
7.3. Conclusions	141
7.4. Future work	142
8. Conclusiones	143
Bibliography	149
A. Tertimuss	157
B. Modeling methodology using TCPNs	159
C. Notes on preliminary comparison between RUN and AIECS	175
D. Unimodularity	177
List of Figures	179
List of Tables	183
Acronyms	185

Introduction

1.1 Rationale

Multiprocessor architectures are becoming increasingly common in embedded systems. One reason of their spread usage is that such platforms now constitute a significant share of the cost-efficient components-of-the-shelf (COTS). Besides, the usage of multicore systems on chip (MPSoCs) helps to consolidate components into single miniaturized solutions. Also, they are allowing major improvements for some demanding applications such as engine controllers, RT image processing or advance driving assistance systems (ADAS).

Current automotive and aerospace systems encompass a large and increasing number of electronic control units (ECUs). For example, mid-size sedans can reach 125 ECUs as of this writing, a number which is expected to grow as more and more features are incorporated and the transition to hybrid and electric cars unfolds [82][43]. For this reason, *ECU consolidation* is a desirable objective to reduce size, weight, power and cost (SWaP-C factor). The cellphone market and edge computing designs subject to RT constraints in IoT (Internet of Things) ecosystems also benefit from powerful, compact multiprocessor systems on a chip (MPSoCs).

However, ensuring the accomplishment of Hard Real Time (HRT) and safety constraints on multicores can easily lead to overprovisioning, requiring more cores, and more powerful, than the ones actually demanded. This problem comes out from a variety of sources, with two of them standing out: worst-case execution time (WCET) estimation for each task, and task scheduling.

Algorithms for RT multiprocessor scheduling can be broadly classified into partitioned, global, and hybrid schemes (semipartitioned, clustered) [13, 67]. Partitioned schedulers [14, 54] statically allocate tasks to processors. They can leverage well-known RT uniprocessor schedulers such as RM, which limits CPU utilization to a 69.3% in uniprocessors [50], or EDF which is optimal on uniprocessors. Unfortunately, a partitioned approach on multiprocessors decreases utilization to 50% [59] under a sufficient schedulability condition.

Global schedulers allocate tasks to any CPU. They are mostly preemptive and allow task migration among CPUs. A well-known example is *gEDF*, which guarantees soft real-time (SRT) schedulability for implicit-deadline task sets [10] but is not HRT optimal [38, 11]. Aiming to achieve maximum CPU utilization, *fluid* global schedulers leverage the theoretical principle of instantly sharing all CPUs among all active jobs. In this vein, *pfair* algorithms [8] achieve HRT optimal schedulability for implicit-deadline tasks sets, but they incur in an unfeasible number of context switches. Approaches like *deadline partitioning* can notably reduce such overhead [36]. However, global schedulers are considered too complex to be practical, whereas hybrid approaches combined with ad-hoc heuristics can achieve similar optimality [15].

Meanwhile, the industry is reluctant to adopt these state-of-the-art algorithms in critical systems, where the use of static scheduling is pervasive [31, 2]. Particularly, a *cyclic executive* (CE) is defined as a deterministic scheme of repeated execution of a series of *minor frames*. Each minor frame defines a sequence of jobs, that execute within the frame. The collection of minor frames is referred as a *major frame* (MAF). CEs are usually implemented as tables with invocation times for each job. A CE offers two significant advantages: its *predictability* and *low run-time overhead*.

Automotive (AUTOSAR) and aerospace (ARINC) specifications are already contemplating multiprocessor systems, based on partitioned scheduling techniques. But the generation of CEs both on single-core processors and on MPSoCs with partition algorithms wastes a lot of the available cores: it implies low utilization and overprovisioning, i.e., require more elements than necessary (cores/processors/ECUs), the previous without even taking into account redundancy systems. Recent research shows that we can get best of two worlds by calculating a predictable, low-overhead CE while obtaining the optimal utilization provided by global or clustered schemes [31]. However, there are quite a few open issues to consider, such as accounting for the context switching and migration overhead when calculating the CE leveraging effective but complex preemptive multiprocessor schedulers. Finally, today's multiprocessors provide flexibility and scalability in many different ways. For example, they are amenable to implement dynamic control strategies that can cope with unexpected parametric variations, limited disturbances and the arrival of aperiodic events, while ensuring the correct execution of a HRT task set and the thermal integrity of the whole system and minimizing energy consumption. This

has also been an open venue for the past few decades, all the more promising now because of the existent technological opportunities.

1.2 Thesis Objectives

The avenue taken in the Thesis has been that of studying, proposing and leveraging state-of-the-art multiprocessor RT schedulers suitable for the industry, holding the following traits and benefits:

- Maximum processor utilization so as to minimize the number of required cores and help avoid overprovisioning.
- Low number of context switches and job migrations with respect to previous proposals, to help reduce the scheduling overhead.
- Ability to compute a multiprocessor CE or, alternatively, to provide runtime support to manage parametric variations, CPU failures and aperiodic task arrivals.
- Compliance with HRT, thermal and energy constraints.

1.3 Contributions

In pursuing the objectives exposed above, the work on this Thesis results in the following contributions.

- A modeling methodology that represents all the dimensions of the problem under a single formalism (TCPN): task arrival, their allocation to CPUs and subsequent execution, and the energy consumption and thermal behavior of the system. It improves previous work, which required different tools making integration harder. For example, in [77] they only model processor scheduling and power consumption, in [57] only temperature and frequency are modeled but not their relationship with tasks. The work [28], models all the above aspects in TCPN, but does not consider a power dissipation model but energy consumption parameters per task. This contribution has been reported in [63], [62], [64], and [67].

- A CE for multiprocessors that meets thermal, minimum power, and maximum utilization constraints. Its novelty lies in that it solves these problems all together. The performance of this CE schedule is compared against other schemes to see its performance in terms of context switches and migrations, [61] and [31], exhibiting promising results. This contribution has been reported in [62], [64], and [67].
- An iterative methodology to compute a CE on a multicore processor considering the overhead costs introduced by preemptions and migrations ensuring no deadline misses. Up to our knowledge, this is the first proposal to pose and solve this problem, accepted and published in [66]
- An online frequency controller, which ensures compliance with a CE, also allowing limited disturbances to be rejected, such as limited CPU failures, parameter variations or the arrival of aperiodic tasks. The latter constitutes a novel approach concerning previous proposals, which often relies on periodic servers as in [55], which entails a number of inconveniences. This contribution is reported in [64] and [67].

1.3.1 Summary of publications

Journals

A Flexible Framework for Real-Time Thermal-Aware Schedulers using Timed Continuous Petri Nets, published on journal *Computación y Sistemas*

It was developed in collaboration with Desirena-López, G., Ramírez-Treviño, A., and Briz, J. L. This work presents *TCPN-ThermalSim* the predecessor of *Tertimuss* [21], the current and improved simulation tool used in this thesis. *TCPN-ThermalSim* is a software tool for testing Real-Time Thermal-Aware Schedulers. This framework consists of four main modules. The first one helps the user to define the problem: task set with periods, deadlines and worst case execution times in CPU cycles, along with the CPU characteristics, temperature and energy consumption. The second module is the Kernel simulation, which builds up a global simulation model according to the configuration module. In the third module, the user selects the scheduler algorithm. Finally the last module allows the execution of the simulation and present the results. The framework encompasses two modes: manual and automatic. In manual mode the simulator uses the task set data provided in the first

section. In automatic mode the task set is generated by parameterizing the integrated UUniFast algorithm.

Energy efficient thermal-aware multiprocessor scheduling for real-time tasks using TCPN [62]. This work was developed in collaboration with Desirena-López, G., Ramírez-Treviño, A., and Briz, J. L., and published on Discrete Event Dynamic Systems Journal,

This work provides the baseline of the thesis, and is extended on chapter 3. Herein, we present an energy-efficient thermal-aware real-time global scheduler for a set of hard real-time (HRT) tasks running on a multiprocessor system. This global scheduler fulfills the thermal and temporal constraints by handling two independent variables, the task allocation time and the selection of clock frequency. To achieve its goal, the proposed scheduler is split into two stages. An off-line stage, based on a deadline partitioning scheme, computes the cycles that the HRT tasks must run per deadline interval at the minimum clock frequency to save energy while honoring the temporal and thermal constraints, and computes the maximum frequency at which the system can run below the maximum temperature. Then, an on-line, event-driven stage performs global task allocation applying a Fixed-Priority Zero-Laxity policy, reducing the overhead of quantum-based or interval-based global schedulers. The on-line stage embodies an adaptive scheduler that accepts or rejects soft RT aperiodic tasks throttling CPU frequency to the upper lowest available one to minimize power consumption while meeting time and thermal constraints. This approach leverages the best of two worlds: the off-line stage computes an ideal discrete HRT multiprocessor schedule, while the on-line stage manage soft real-time aperiodic tasks with minimum power consumption and maximum CPU utilization.

Maximizing utilization and minimizing migration in thermal-aware energy-efficient real-time multiprocessor scheduling [67]. This work was developed in collaboration with Ramírez-Treviño, A., and Briz, J. L., and L. E., Chils, and published on IEEE Access journal,

This articles builds over the previous results from the conference paper in WODES 2020, chapter 3. This work proposes CAIECs, a clustered scheduling system for MPSoCs subject to thermal and energy constraints. It calculates off-line a cyclic executive honoring temporal and thermal constraints, for a hard real-time (HRT) task set at minimum frequency to reduce consumed energy, minimizing context switches and migrations. It also provides an on-line controller able to manage system and task parametric variations and soft real-time (SRT) tasks, always meeting the HRT task

set constraints and the system thermal bound. CAIECS maximizes CPU utilization to help avoid overprovisioning contributing to a low SWaP factor. Its modular design allows the utilization of different modeling and scheduling approaches, and makes the off-line and on-line components independent from each other to better suit the requirements of a specific system. We experimentally show that the cyclic executive provided by CAIECS for HRT task sets outperforms RUN, a reference off-line algorithm in terms of optimal number of context switches.

Accounting for preemption and migration costs in the calculation of hard real-time cyclic executives for MPSoCs, published on IEEE Robotics and Automation Letters,

Based on our previous results on the computation of a compelling preemptive CE. This work introduces a methodology to consider preemption and migration overhead in hard real-time cyclic executives on multicore architectures. The approach performs two iterative stages. The first stage takes a cyclic executive, from which the number and timing of all preemptions and migrations for every task is known. Then, it includes this overhead by updating the worst-case execution time (WCET) of the tasks. The second stage calculates a new cyclic executive considering the new WCET of tasks. The stages iterate until the preemption and migration overhead keeps constant.

Conferences

Real time scheduler for multiprocessor systems based on continuous control using Timed Continuous Petri Nets, presented on the 15th IFAC Workshop on Discrete Event Systems **WODES 2020** — Rio de Janeiro, Brazil, 11-13 November 2020 [64]

This work exploits Timed Continuous Petri Nets (TCPN) to design and test a novel energy-efficient thermal-aware real-time global scheduler for a hard real-time (HRT) task set running on a multiprocessor system. The TCPN model encompasses both the system and task set, including thermal features. In previous work we calculated the share of each task that must be executed per time interval by solving off-line an Integer Programming Problem (ILP). A subsequent on-line stage allocated jobs to processors. We now perform the allocation off-line too, including an allocation controller and an execution controller in the on-line stage. This adds robustness by ensuring that both task allocation and runtime execution honor the safe schedule provided off-line. Last, the on-line controllers allow the design of an improved soft RT

aperiodic task manager. Also, we experimentally prove that our scheduler yields fewer context switches and migrations on the HRT task set than RUN, a reference algorithm. *Accounting for preemption and migration costs in the calculation of hard real-time cyclic executives for MPSoCs*. presented on the 18th IEEE International Conference on Automation Science and Engineering IEEE **CASE 2022** [65]. This is the conference paper for the journal article that shares the same name [66].

1.4 Structure of this document

The remainder of this thesis is organized as follows. We review hardware foundations, an introduction on Real Time (RT) systems and prior work in Chapter 2. Also in that chapter we introduce the Timed Continuous Petri Net (TCPN) model that describes the interactions among task arrival, task execution in processors, energy consumption and heat generation.

Chapter 3 introduces a scheduling algorithm proposal that minimizes energy consumption and satisfies a thermal bound. In this chapter, we present the thermal analysis based on the TCPN model from Chapter 2.

Chapters 4-5 present our control strategies in RT scheduling. Chapter 4, details our first control contribution which is an scheduler named Allocation and Execution Control Scheduler (AIECS), it applies a control strategy to honor an offline scheduler, in the presence of runtime overheads and aperiodic task arrivals. Chapter 5 improves several aspects of its predecessor, it implements a clustering stage to minimize migrations, furthermore we propose a small, but key, change on the model in order to simplify the execution controller and the aperiodic task management. The scheme from this chapter is called Clustered Allocation and Execution Control Scheduler (CAIECS).

Chapter 6 takes the cyclic executive computed from the offline stages of CAIECS and re-computes a cyclic executive that considers the overhead due to preemptions and migrations.

Finally, Chapter 7 summarizes our results, raises open questions, and discusses future work.

Background and prior work

In this chapter we briefly review key concepts on RT, the underlying hardware architecture and relevant scheduling algorithms which are assumed to be known throughout this dissertation.

2.1 Hardware Foundations

The RT multiprocessor scheduling problem addressed in this work assumes the baseline hardware described in this section. Also, we refer the reader to [14] for a good complementary introduction to the subject.

We use the terms *processor*, *CPU* and (processor) *core* interchangeably to denote an entity capable of executing instructions according to a given Instruction Set Architecture (ISA), specifically one that executes the scheduled tasks.

The term *multiprocessor* encompasses a wide range of system architectures. This work assumes homogeneous Symmetric Multiprocessor (SMP)s with variable frequency, with a simple memory hierarchy with no cache memories.

A *multiprocessor* consists of multiple, interconnected CPUs. They can be arranged as multiple processor cores integrated on a single (multicore) chip and interconnected through a bus or Network on Chip (NoC), or as multiple uni-core processor chips interconnected through an on-board bus. A Multiprocessor System on a Chip (MPSoC) is a particular but widespread case of chip multicore, common in embedded systems. MPSoCs integrate different sets of processor cores, such as general-purpose cores, specific cores for RT systems — the ones, if present, to be managed by the scheduling algorithms exposed in this work— and other specialized computational units often known as *accelerators* such as cryptographic units, Graphic Processing Unit (GPU) or units supporting algorithms to solve neural networks for AI applications. We refer to all these systems indistinctly, always considering the specific set of cores devoted to the execution of RT tasks.

There are two classes of multiprocessors according to the nature of their interconnections: SMPs and Distributed Shared Memory (DSM) multiprocessors. SMPs share a centralized (commonly banked) memory connected through a bus or crossbar. In the absence of caches (per-core or per cluster of cores), all processors access such shared memory in similar conditions (Uniform Memory Access (UMA)). In contrast, DSM configurations physically distribute the memory banks on different nodes, so that accessing a memory address located at a memory bank in the local node is faster than reaching it when located in other nodes (Non-Uniform Memory Access (NUMA)).

As far as RT scheduling is concerned, those memory models have a dramatic impact on the migration of jobs. Thus, the tough chore of establishing reasonable safe upper bounds for memory access times and inter-core job migration in SMPs, becomes unrealistic in the case of DSM machines. Current RT multiprocessor systems are most implemented on SMP organizations, either on general-purpose multicore chips, or leveraging the general-purpose cores or the RT specific cores available in MPSoCs. Therefore, we assume this later approach as the underlying hardware organization throughout this Thesis.

Finally, multiprocessor systems are also classified according to the ISA of the processors and their computing capabilities. Thus, we talk of *identical* multiprocessors when there are no differences among the processors, and therefore the execution time of a given task does not depend on the processor. In contrast, in *uniform heterogeneous* multiprocessors the execution time of a task can be different upon the type of processor, as long as we can establish a speed ratio for each processor type with respect to a reference, e.g. the slower type of processor. All processors implement the same ISA, but they can differ in their Cycles per Instruction (CPI) or in the range of their operating frequencies. Finally, in the case of *unrelated heterogeneous* multiprocessors, processors can show important differences in their mechanisms to hide memory latency, or even implement different ISAs, making difficult well nigh impossible to establish fixed speed ratios among them. In this case, a task executing on a processor cannot migrate to another processor because of the difficulty to assess the actual progress of the task in the current processor and the point to resume its execution in the destination processor.

In this work we focus on identical multiprocessors, yet the TCPN model (Sec. 2.3) is able to represent all of this cases with minimum changes.

Summarizing, in this dissertation we consider a platform of SMP identical multiprocessors with Dinamyc Voltage and Frequency Scaling (DVFS), defined as follows:

Definition 2.1 Let $\mathcal{P} = \{cpu_1, \dots, cpu_m\}$ be a the set of m constituent processors of an identical SMP multiprocessor, where all processor cores support DVFS and run at the same given clock frequency, which can dynamically vary among a discrete set of values $f \in F = \{f_1, \dots, f_{max}\}$.

2.2 Real time systems

A RT system is often defined as a computer system whose outcome is only correct if logically (mathematically) correct, and temporally correct (produced in due time). According with the consequences of violating the temporal constraint, RT systems are broadly classified into Hard Real Time (HRT)), Firm Real Time (FRT) and Soft Real Time (SRT). A deadline violation leads to a complete system failure in a HRT systems, with potentially catastrophic consequences. FRT systems tolerate a number of deadline misses, and the value of results after the missed deadline is zero. Deadline misses in SRT systems do not lead to a complete system failure, but degrade system performance or cause the system to run awkwardly. Life-support, control flight or assisted driving systems are examples of HRT systems. Conference video streaming is often cited as a FRT system although most audio and video systems are considered SRT. Core systems in network routers and switches usually fall into the SRT category.

2.2.1 Task model

Processes on a RT system are named tasks, and they are typically recurrent. The *periodic* task model of Liu and Layland [50] presents the classic notion of a recurrent task. Each task τ is characterized by its request period ω and an execution requirement c , such that the task must complete its execution before its next activation ($c \leq \omega$). Every instance of a task is referred as *job*, therefore a periodic task can produce an infinite number of jobs.

Based on [50], Al Monk [56] provides the notion of *sporadic task model*, also known as *the three-parameter model*. In this model, each task τ is defined by a deadline d , a period ω and an execution requirement c , such that task τ must complete its

execution before its deadline ($c \leq d$). If $d = \omega$, then the deadline is called *implicit*, if $d \leq \omega$ then the deadline is *constrained*. Otherwise, when there are not restrictions on the deadline with respect to the period it is called *arbitrary* deadline. The period of a task is commonly denoted p , however we use ω in this dissertation to avoid confusion when introducing the concept of *place* p on a Petri Net (Sec. 2.3.1).

An alternative way of understanding the difference between *periodic* and *sporadic* tasks is considering their *activation pattern* [42]. Thus, the period defines the fixed interval of time at which a periodic task activates, whereas it stands for the *minimum* (and different from zero) interarrival time of an sporadic task —the sporadic task can arrive later but not sooner than its period.

The execution requirement c of a task is selected as an upper bound on the greatest time that it will take to perform its execution when it has sole access to all computational resources and under the complete set of all possible environmental conditions. This upper bound is known as the Worst Case Execution Time (WCET), and this dissertation assumes it is given in CPU cycles, such that a task τ executed on a processor P at frequency f , requires $c = \frac{cc}{f}$ processor time at every ω interval.

The task $\tau_i = (c_i, \omega_i, d_i)$ is a process that invokes a sequence of jobs j_h . Each job arrives at time a_h , has its execution requirement c_i and a deadline at $a_h + d_i$. Thus, job j_h should be allocated to a processor during the time interval $[a_h, a_h + d_i)$. If τ_i is a *periodic* task, then it invokes its first job at time 0 and all its remaining jobs are invoked exactly ω_i time units apart, i.e., $a_h = (h - 1)\omega_i$.

The utilization of a task, u , is the ratio of its execution requirement to its period,

$$u = \frac{cc}{f\omega} \quad (2.1)$$

since $c \leq \omega$, u is always $0 \leq u \leq 1$. The utilization of a task determines the share of CPU that it requires. Since constrained-deadline tasks ($d < \omega$) need to execute at a higher rate than u , this urgency is best represented by the execution *density* δ , defined as the rate of the execution time to the deadline. However, if $d > \omega$, then the density should be computed using the period, and is defined as $\delta = \frac{c}{\min(d, \omega)}$.

Based on the previous concepts, a RT workload consists of the set of n sequential tasks $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$.

This work assumes an *implicit* deadline model, with a task set formally defined in Def. 2.2

Definition 2.2 Let $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$ be a set of n independent periodic tasks under hard real time (HRT) constraints. Each task is defined by the 3-tuple $\tau_i = (cc_i, d_i, \omega_i)$, where cc_i is the WCET in cycles which takes to complete any job, ω_i the period and d_i is the relative implicit deadline ($d_i = \omega_i$)

A task set with implicit deadlines is feasible if the following (sufficient) condition holds:

$$U = \sum_{i=1}^n \frac{c_i}{\omega_i} \leq m \quad (2.2)$$

where U is the system utilization and m is the number of processors. Condition 2.2 is also necessary in the case of HRT task sets.

Besides the recurrent tasks, *aperiodic* and *sporadic* tasks are defined by its WCET cc , a deadline d , and its arrival time r , which is unknown beforehand. The inter-arrival time can take any value in the case of aperiodic tasks, whereas a minimum interarrival time, different from zero is established in the case of sporadic tasks.

Definition 2.3 Let $\mathcal{T}_a = \{\tau_1^a, \dots, \tau_p^a\}$ be a set of p independent aperiodic tasks. Each task is identified by the 3-tuple $\tau_i^a = (cc_i^a, d_i^a, r_i^a)$, in which cc_i^a (WCET) and d_i^a (deadline) are known, but the arrival time r_i^a is unknown.

The hyperperiod (H) of a task set \mathcal{T} is denoted as the least common divisor (lcm) of the periods of all the tasks in \mathcal{T} and the quantum (q) as the greatest common divisor (gcd) of the periods of all tasks in \mathcal{T} .

2.2.2 Scheduling concepts

One of the most important aspects while designing a RT system is selecting appropriate methods for task *scheduling*, to ensure that timing constraints are met. The result of an scheduling method is a *sequence* or schedule of the assignment between processor time (resources) and tasks (activities).

Feasibility and schedulability A task set \mathcal{T} is said to be *feasible* upon an specified platform if there *exists* a correct schedule for which every job released by each task meets its deadline. Let A be an scheduling algorithm, then a task set \mathcal{T} is said to be schedulable by A if algorithm A guarantees the deadlines of all job of each task in \mathcal{T} .

Notation	Description	Definition
\mathcal{P}	Set of processor	$\mathcal{P} = \{CPU_1, \dots, CPU_m\}$
CPU_j	The j^{th} processor	$1 \leq j \leq m$
F	Set of discrete clock frequencies of \mathcal{P}	$F = \{f_1, \dots, f_{max}\}$
\mathcal{T}	Set of sporadic implicit-deadline tasks	$\mathcal{T} = \{\tau_1, \dots, \tau_n\}$
τ_i	The i^{th} sporadic implicit-deadline task	$1 \leq i \leq n$
$j_{i,j}$	The j^{th} job of τ_i	$j \geq 1$
cc_i	The worst-case execution in CPU cycles of τ_i	
c_i	The worst-case execution time of τ_i at f	$c_i = \frac{cc_i}{f}$
d_i	The relative deadline of τ_i	$d_i = \omega_i$
ω_i	The period of τ_i	$d_i = \omega_i$
u_i	The utilization of task τ_i	$u_i = \frac{cc_i}{f \omega_i}$
U	System utilization	$\sum_{\tau_i \in \mathcal{T}} u_i$
H	Hyperperiod of \mathcal{T}	$lcm(\omega_1, \dots, \omega_n)$

Table 2.1.: Summary of the notation used for tasks and processors

An A-schedulability test accepts as input the specifications of a sporadic task system and a platform, and determines whether the task system is A-schedulable. An A-schedulability test is said to be *exact* if it identifies all A-schedulable systems, and *sufficient* if it identifies only some A-schedulable systems [31].

Optimality A scheduling algorithm is defined as *optimal for a class of task systems* if its schedulability condition is identical to the feasibility condition for that class [76].

Capacity loss Ideally, any task set \mathcal{T} should be feasible as long as the utilization of the system never exceed the number of processors, i.e $U \leq m$. However, in practice it may not be possible to allocate all processor capacity to the RTtask set. Such *capacity loss* has primarily two causes: the scheduling policy and runtime overheads [14]. The first cause corresponds to an *algorithmic* capacity loss; if the scheduling policy is non-optimal then a feasible task set may not be schedulable. The second cause is due to hardware inefficiencies and system management activities, such as scheduling decisions and context switches.

Non-preemptive vs Preemptive A scheduler or system is considered *non-preemptive* if jobs are allocated on a processor up to completion. Alternatively, it is considered *preemptive* if a job's execution can be *preempted*, i.e. interrupted before completion such that a higher priority task can be allocated to the same processor instead.

Overheads

A *context switch* is the process of storing the state of a outgoing job (either finished or preempted), and loading the state of an incoming job (which either starts or resume execution). There is a compulsory context switch whenever a job is first released or terminates. The cost of these compulsory context switches is therefore independent of the scheduling algorithm, and can be easily added to the WCET, to analyze schedulability and for scheduling purposes.

Preemptive schedulers introduce additional context switches at preemption points, in a number that largely varies with the scheduling algorithm. Accounting for the overall cost of such context switches may not be trivial and is just the problem addressed in Chapter 6.

A *migration* is a particular case of context switch such that the context is saved in the current processor but restored in a different processor once the job is scheduled to resume execution.

2.2.3 Scheduling: from Uniprocessor to Multiprocessors

For many years, RT systems have only been deployed on uniprocessor systems. The theory for uniprocessor RT systems is mature and most results on scheduling algorithms are optimal for this type of hardware. Nevertheless, by the reasons discussed in Sec. 1.1, multiprocessor platforms are now becoming increasingly common in embedded RT systems, with such platforms often available even as COTS.

Uniprocessor RT scheduling

Fig. 2.1 shows an overarching classic classification of scheduling algorithms. On the one hand, there is *static scheduling*, which are schemes where the sequence of tasks to be executed is obtained offline. *Cyclic executives* (CEs) are a common example, on which we will extend later on. On the other hand, there is *dynamic scheduling*, in which scheduling decisions are made at run time, triggered upon specific events other than the clock interruption, and always selecting the highest priority task for execution.

Algorithms can be also classified according to their priority allocation policy:

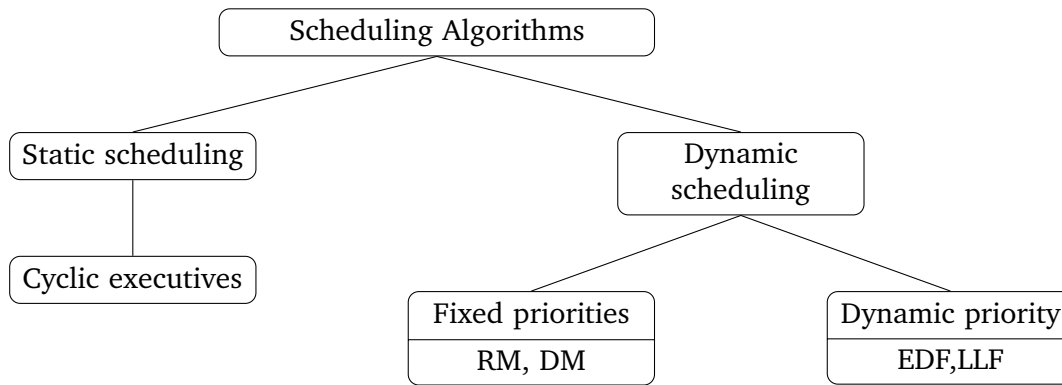


Figure 2.1.: Classification of scheduling algorithms

- Fixed task priority (FTP). Priority is statically assigned to tasks and is fixed for every job of a task.
- Fixed job priority (FJP). Different jobs of the same task may be assigned different priorities. However, the job priority, once assigned, may not change.
- Dynamic job priority (DJP). The priority of a job may change arbitrarily while it is alive.

Definition 2.4 (*Priority driven algorithms*) A scheduling algorithm is said to be a priority driven scheduling algorithm if and only if it satisfies that for every pair of jobs j_i and j_j , if j_i has higher priority than j_j at some instant in time, then j_i always has higher priority than j_j .

Static priorities are assigned before execution and are not changed at runtime. Tasks with higher priority are executed first. If the system allows preemption, a higher priority task will cause the preemption of a lower priority task, overtaking the CPU of the latter. Rate Monotonic (RM) and Deadline Monotonic(DM) are two well-known preemptive scheduling algorithms with static priorities.

Rate Monotonic (RM) It assigns priorities according to the task periods: the shorter the period of the task, the higher its priority. Deadlines are assumed to be *implicit* (Sec. 2.2.1). The schedulability condition was derived for the first time in [50], for n tasks $U(n) \leq \ln 2$ must be met, this is deduced from $U = \sum_{i=1}^n \frac{c_i}{p_i} \leq n(2^{1/n} - 1) = U(n)$, when n grows $\lim_{n \rightarrow \infty} U(n) = \ln 2$.

Deadline Monotonic (DM) It assigns priorities according to the relative deadlines: the closer to activation the deadline is, the higher the priority of the task. Deadlines are assumed to be *constrained* (Sec. 2.2.1). The schedulability condition was given

by [49]. It says that for any initial lag of the tasks all deadlines are met if and only if $\forall i, 1 \leq i \leq n, \min \left(\sum_{j=1}^i \frac{C_j}{t} \lceil \frac{t}{p_j} \rceil \right) \leq 1$.

Earliest Deadline First (EDF) In contrast with RM and DM, under EDF scheduling the task priorities are assigned at runtime. Jobs are scheduled in order of urgency (first the ones with the *earliest* deadline) i.e, different jobs of the same task may have different priorities, still once the job is assigned a priority it will not change until its completion. Its schedulability condition is $U \leq 1$ [50].

Note that the EDF schedulability condition is equal to the its feasibility condition, therefore this algorithm has not capacity loss. On the other hand, RM and DM are unable to schedule every feasible task set, therefore they suffer from algorithmic capacity loss.

Least Laxity First (LLF) Each job under LLF receives a priority according with their *laxity*, $Li(t) = d_i - t - E_i(t)$, where $E_i(t)$ is the pending execution of the i th job at time t . LLF recalculates laxity every time an instance finishes executing or on every activation. Therefore, a job can have different priorities while it is active. This algorithm schedules all sets of tasks with utilization $U \leq 1$ just like EDF on uniprocessors, but it is more computational expensive.

Cyclic executives

A Cyclic Executive (CE) is a schedule determined prior to run-time, which describes a sequence of jobs to be performed on a fixed period of time called *major cycle* or *major frame*. This approach to scheduling is applicable only when the system is highly deterministic, except for a few aperiodic tasks that can be scheduled under deterministic frameworks.

This structure implies that scheduling decisions are made periodically, at the beginning of each time interval, referred as a *minor frame*.

Generally, there are three restrictions in the definition of the minor frame:

1. Ideally, the minor frame size ϕ should be long enough to accommodate each job up to completion

$$\phi \geq \max_{1 \leq i \leq n} (c_i) \quad (2.3)$$

2. The minor frame should divide the hyperperiod H of the task set. This condition is met when ϕ divides the period ω_i of at least one task $\tau_i \in \mathcal{T}$

$$\lfloor \omega_i / \phi \rfloor - \omega_i / \phi = 0 \quad (2.4)$$

3. In order to ensure that deadlines are met, the frame size should be sufficiently small so that between the release time and deadline of every job, there is at least one frame:

$$2\phi - \text{gcd}(\omega_i, \phi) \leq d_i \quad (2.5)$$

2.2.4 RT Multiprocessor scheduling

The classification provided in Fig. 2.1 also applies for multiprocessor scheduling, yet the inclusion/increment on *system capacity* poses new challenges. Thus, scheduling algorithms in multiprocessor architectures are frequently distinguished according to their migration constraints:

- *No migration*. Tasks are statically allocated to processors and never migrate (e.g. *partitioned scheduling*).
- *Partial migration*. Tasks can only perform a limited number of migrations, only some tasks are allowed to migrate or tasks can only migrate on a subset of processors (e.g. *Semi-partitioned* and *clustered scheduling*)
- *Full migration*. Tasks are dynamically allocated to processors and can migrate at any time on any processor (*Global scheduling*).

Partitioned scheduling

Partitioned scheduling boils down to finding disjoint subsets Q_j in a task set, such that the utilization s_j of each subset is $s \leq 1$. It is widely known that such partition is equivalent to the bin-packing problem, hence it is highly intractable NP-hard in the strong sense. Once the partitions are computed, each subset of tasks can be scheduled with any well-known uniprocessor scheduling algorithm. Since migration is forbidden, processors may be underutilized.

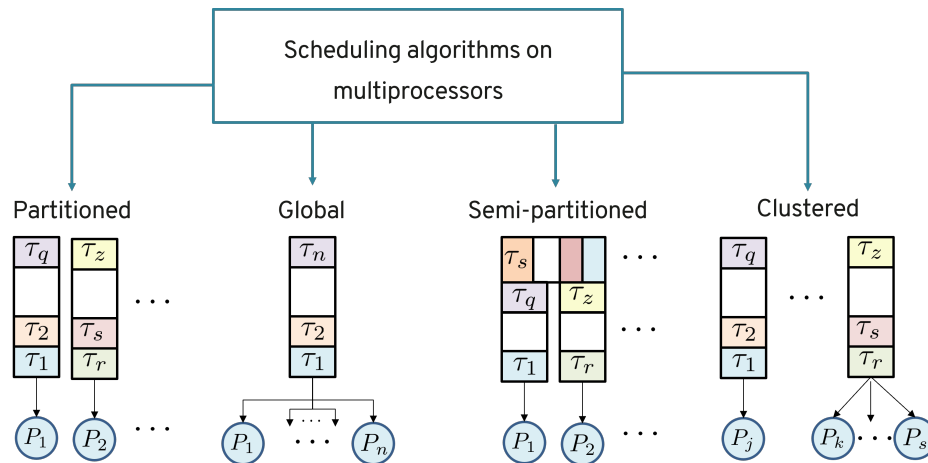


Figure 2.2.: Classification of RT multiprocessor scheduling

The bin-packing problem

The Bin packing problem (BPP) states the following: Pack n objects of different sizes s_1, s_2, \dots, s_n into the minimum number of bins (containers) of fixed capacity c . The total volume V of the items is then $V = \sum_{i=1}^n s_i$.

This problem appears in many practical cases, e.g. how to cut pieces of beams from beams of a given length to minimize waste, or how to fit luggage in a flight. Various heuristics exist to approximate solutions for the bin-packing problem:

- Next Fit (NF). Place each item in the same bin as the last item. If it does not fit, start a new bin.
- First Fit (FF). Place each item in the first bin that can contain it.
- Best Fit (BF). Place each item in the bin with the smallest empty space
- Worst Fit (WF). Places each item in the used bin with the largest empty space, otherwise start a new bin

BPP algorithms are said to be *online* if items arrive one at the time, and they must be served before the next one arrives. Alternatively, they are said to be *offline* when all items are known beforehand, such that they can be grouped into bins in any order.

The heuristics performance can be evaluated through a performance ratio $\rho = B_A/B_0$, where B_A stands for the number of bins used by algorithm A, and B_0 the minimum number of bins used by the optimal algorithm.

If all items are known, sorting them first usually improves performance on the heuristics. First Fit Descending (FFD) is the heuristic that employs a FF algorithm with items sorted in descending order.

The formulation of the task allocation problem as a BPP is straightforward. The tasks are considered as items with *size* equal to its utilization u_i , and each processor (container) has a capacity of 1, or a speedup factor. Instead of comparing heuristics upon a performance ratio, we define the *worst-case achievable utilization* $U_{wc,S-A}$ for a given scheduler S and a given algorithm A , such that any task set with utilization $U \leq U_{wc,S-A}$ is schedulable by S using A .

The worst-case achievable utilization on m processors with any of the past heuristics is at most $(m + 1)/2$.

Utilization bound for EDF Lopez et.al [51] showed that any task system with utilization at most $(m + 1)/2$ can be correctly scheduled on m processor with *EDF* as scheduling policy and FF or BF as heuristics algorithms.

EDF partitioned schedulability can be improved imposing restrictions on the maximum utilization per task.

Disadvantages of partitioning The major disadvantage of partitioned approaches is a low bound on utilization 50%, even when relying on optimal schedulers for uniprocessors, like EDF [59, 53].

Global scheduling

In contrast to partitioned scheduling, global scheduling allow task migration among all processors. From a queue theory point of view this is advantageous, because the response time of a task can be reduced. Nevertheless, Dhall and Liu [32] showed that classic algorithms, like RM and EDF, performed poorly for global scheduling.

Dhall's effect. The schedulability bound of global-EDF and global-RM is equal to 1, independently of the number m of available processors. This means that given a platform of m identical processors, there exist task sets with $U > 1$ that are not schedulable by global-EDF and global-RM. To prove this result it suffices to identify a task set \mathcal{T} with utilization $U = 1 + \epsilon$, where ϵ is an arbitrarily small constant, that is not schedulable by global-EDF and global-RM. This is exemplified on Fig. 2.3.

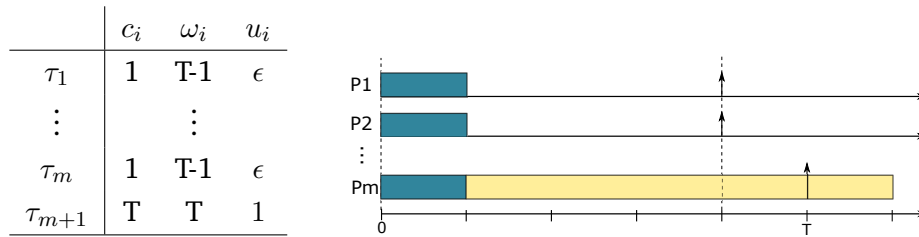


Figure 2.3.: Dhall's effect. Example with m processors and $m + 1$ tasks where EDF and RM produce an unfeasible schedule with a total utilization arbitrarily close to 1

Proportional fairness

The concept of *proportional fairness* applied to multiprocessor RT scheduling was first proposed to improve the under-utilization shown by partitioned schemes [8]. Conceptually, the execution of each task could be represented through a *fluid rate curve*, as in Fig. 2.4. If each task could be executed according to its fluid curve at every instance of time, then a feasible schedule should always exist for systems $U \leq m$. This implies sharing all processors among all tasks at each time instant, which is unrealistic.

Instant sharing is just a theoretical concept, so in practice we can allocate processor time in discrete time units, or *quanta*. The utilization u_i of a task defines the rate at which it has to be scheduled. A fair scheduler closely tracking the fluid curves would provide each task with a share of $u_i \cdot t$ during a time interval $[0, t)$. The closer the algorithm becomes to the fluid curves (the fairer it is), the more preemptions and context switches it triggers, getting the idea unpractical.

Pfair [8] was a seminal algorithm leveraging the principle of proportionate scheduling to design practical RT multiprocessor schedulers. It creates a scheduling event and recomputes the set of running tasks at every multiple of a discrete time quantum. The notion of proportional fairness used in *Pfair* is very strict, requiring that the actual work completed by every task is within 1 unit of its fluid rate curve at each time quantum. The result of this policy is a large number of scheduling calculations and context switches, with correspondingly high overhead.

Most variations on the *Pfair* algorithm aim to solve the problem of priority ties (*PD* [7] and *PD*² [4], [76]).

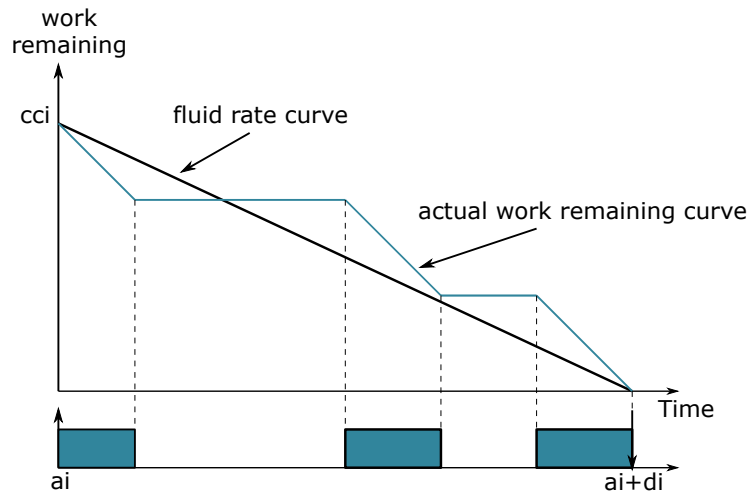


Figure 2.4.: Fluid versus practical schedules. The actual remaining work decreases at a rate of 1 or the processor frequency f when task is executing, and is horizontal when idle. The fluid rate curve decreases at a constant rate of u_i

Deadline partitioning

Deadline partitioning fairness (DP-fair) was proposed to avoid the overhead introduced by Pfair algorithms [36]. Intuitively, it seems unnecessary to adhere so closely to the fluid schedule: performance could be improved by a more judicious choice of scheduling events. Thus, any given job really only needs to match its fluid rate curve at its own deadline, which is the original problem to solve. A sufficient (although not necessary) compromise is to require all tasks to match their fluid rate curves at the deadline of each task. This new requirement over-constrains the system, but greatly simplifies the scheduling process.

DP-fair is a technique of partitioning time into slices/frames, demarcated by all the deadlines of all tasks in the system. All jobs are allocated a workload within each frame, and all workloads within each frame share the same deadline.

Deadline partitioning deals with two aspects: allocating the workloads for all tasks for each time slice, and scheduling within a time slice. We say that an algorithm using this approach is *DP-correct* if (i) The time slice scheduler executes the allocated workload of all jobs by the end of the time slice whenever it is possible to do so, and (ii) Jobs are allocated workloads at each slice so that such workloads complete within the slice, and the completion of these workloads makes the actual deadlines of all tasks to be eventually met. In other words, any DP-correct scheduler is optimal [36].

Dual scheduler

DP-fair scheduling was designed upon the observation that tracking the fluid curve as strictly as in P-fair methods is unnecessarily over-constraining, even though the fluid curve is the most precise way to indicate how a task must be executed in order to fulfil the timing constraints. This over-constriction not only happens when meeting the fluid constraint at each quantum or time-window as in P-fair, though. Ensuring the accomplishment at the deadline level as *deadline partitioning* approaches is also over-constraining. The clue lies on the following theorem in Funk's et al. seminal paper of DP-Fair [36]:

Theorem 2.1 *When the total utilization of a periodic task set with implicit deadlines is equal to the number of processors, then no feasible schedule can allow any processor to remain idle for any length of time. [36]*

This theorem entails that properly scheduling idle time is just as important as scheduling the workloads. In fact, these two problems are interchangeable (*dual*). Levering this dual approach can relax the aforementioned over-constriction, which hampers the practical value of the methods related to P-fair and DP-fair. The notion of a *dual schedule* is based on Theorem 2.2. It constitutes the basis of further scheduling algorithms such as RUN and QPS, as well as the algorithms contributed in this Thesis.

Theorem 2.2 *Any scheduling problem with m processors and $m + 1$ tasks, where the total rate of the tasks is m , may be scheduled by applying EDF or LLF to the uniprocessor dual. [36]*

RUN

Reduction to Uniprocessor (RUN) [61, 60] is a scheduling algorithm built upon the idea of dual scheduling, to overcome the shortcomings of previous proposals. RUN considers feasible systems composed of independent periodic (not sporadic) tasks with implicit deadlines running on identical processors. RUN transforms the multiprocessor RT scheduling problem into an equivalent set of uniprocessor problems. The key underlying tool is the utilization of a task and its *dual*, e.g. if 0.7 is the utilization of a task, its dual is 0.3. RUN first tries to find off-line *proper utilization subsets*, which are task groups (named *servers*) with an aggregated utilization equal

to 1 This is called a *pack* operation (bin-packing, actually). A *proper utilization subset* can be allocated to a virtual processor, configuring a *proper subsystem*. After a successful *pack*, a *dual* operation follows. The *dual* operation makes groups with an aggregated dual utilization of 1 groups considering the *dual* utilization. The *pack-dual* operations continue until a single utilization system is found. The algorithm guarantees convergence. Then, RUN uses EDF to schedule each task group on-line, reversing (*unpacking*) the *pack-dual* operations performed off-line. Under the DUAL and PACK operations, this algorithm yields a small number of preemptions and migrations.

QPS

Quasi-Partitioned Scheduling (QPS) [55] partitions the task set into subsets. There are two types of subsets, *minor* and *major* execution sets, depending on whether they require one or multiple processors. If all subsets are minor, QPS boils down to a partitioned EDF. Major execution sets are scheduled either by a set of QPS servers on multiple processors, or by local EDF on a single processor depending on their execution requirements.

Unlike RUN, QPS can adapt its scheduling strategy to the system load by monitoring major execution sets at run-time. Like RUN, QPS partition is performed off-line, then generating the schedule on-line. Upon the arrival of aperiodic tasks, QPS needs to recompute the servers, unlike fluid schedulers, which are more amenable to be empowered with adaptive runtime policies.

Hybrid approaches

The limitations of partitioned and global algorithms can be mollified or even overcome employing hybrid approaches [15]. On the one hand, *semi-partitioned* algorithms limit the number of *migratory* tasks. Usually, this method leverages a BPP heuristic to allocate tasks to a single processor, minimizing the number of bins; tasks that do not fit *exactly* are allowed to migrate among processors. Some EDF-based algorithms are EDF-fm [3], EDF-WM, and NPS-F [9].

On the other hand, *clustered* algorithms group processors into clusters, and tasks are pre-allocated to clusters [16]. Tasks can only migrate within the cluster they are allocated to. Clusters can be single- or multi-processor, such that the global

scheduling problem is split into *smaller* problems. This, in turn, improves the overall performance of a global scheduler [15].

2.2.5 Control strategies in RT scheduling

The aforementioned RT scheduling algorithms are *open-loop*, because the scheduling decisions are based on worst-case estimations of task parameters. They do not continuously observe the performance of the system, nor they adjust the decisions or task parameters dynamically to improve performance. In contrast, *close-loop* scheduling algorithms exploit the feedback information of the system to adjust task and/or scheduler parameters, thereby improving system's performance.

Feedback is a powerful concept that has played an important role in the development of many areas of engineering. Feedback can make a system robust towards external and internal disturbances and uncertainties [70]. An advantage of using feedback alongside RT scheduling is that precise schedulability models are no longer needed: the system can dynamically adapt its resource allocation policy or its load in a controlled way to ensure the desired temporal behaviour.

Control techniques rely on the use of actuators handled by a controller to act on the system and modify its output. In regular control systems, there is a desired value (*set-point*) that the system must achieve, such as a certain water level in a tank. A controller will act upon the valves of the tank either by modifying the input or output water flow. Scheduling systems leverage performance metrics, instead of defining a set-point. The most common metric is the deadline miss ratio. For this reason, these approaches have been limited to FRT or SRT systems, allowing for a certain percentage of missed deadlines.

These proposals appear in the literature under names as *feedback scheduling quality of service control* and *control scheduling-co-design*. These works, instead of assuming worst-case estimations of task parameters, they usually consider average-case workload parameters, and are ready to deal with bounded transient overloads. The concept of quality of service (QoS) is traditionally focused on performance metrics, such as throughput, delay and jitter. QoS software adjusts the system resource allocation on-line to maximize the performance in some respects.

Some works on feedback scheduling techniques in the line of [34, 35, 75], provide a firm performance guarantee in terms of deadline misses while achieving high

utilization and throughput at the same time. Sahoo et al. [68] present a closed loop approach for dynamically estimating the execution times of tasks based on both deadline miss ratio and task rejection ratio in the system.

Co-design approaches have also been proposed when the RT tasks constitute digital control loops. In control scheduling-co-design, the development of the control system is performed along with the scheduling, based on a trade-off consideration of scheduling constraints and control performance. For example, the task periods are seen as a variable and the control design should include variable sampling times, as in [5]. These varying times are considered in the controller gains within the control loop tasks.

The concept deadline miss ratio violates the main assumption in HRT systems, and therefore it cannot be used in feedback scheduling for HRT. To overcome this drawback, authors in [29, 28] leverage a fluid scheduling approach and impose a sliding-mode controller such that fluid-rate curves of tasks, Fig. 2.4, are tracked. An undesired side-effect of this technique is that it leads to a high number of context switches, in the line of P-fair schedulers. The authors address this inconvenient through a discretization step, following a deadline partitioning approach. This decreases the number of context switches significantly, but still leaves room for improvement.

Other contributions based on control theory are focused on smoothing the transition of high critical systems, formerly deployed on single processor architectures to multicores. In [33], a closed performance control loop enables a standalone WCET estimation of a HRT application and execution on a multicore system concurrently to other applications.

2.2.6 Thermal and energy aware strategies in RT scheduling

Thermal-aware scheduling has been largely studied for the past few decades. There are at least two elements to consider when tackling this problem: a model representation to study the thermal behaviour of the processors, and a scheduling policy.

As far model representation is concerned, most works rely on mathematical models based on RC circuits or a linearized model. Alternative approaches such as [28, 62]

provide a state model which encompass the thermal properties and dynamics of the system.

Thermal-aware scheduling on single core systems exploit Dinamyc Voltage and Frequency Scaling (DVFS) to reduce power consumption and temperature ([46, 40]). Authors in [18] study the temperature problem in uniprocessors and in homogeneous multiprocessor systems. They leverage a partitioned EDF-based algorithm that minimizes energy, and derive an approximation bound for the maximum temperature. In [69], they perform a worst-case temperature analysis for RT tasks with non-deterministic workloads running on multiprocessors systems. Authors in [1] tackle the problem of thermal-constrained scheduling of periodic tasks leveraging a fluid scheduling model. They assume a partitioned scheduling scheme, which limits the outreach of their results. While in [17], they use an equivalent circuit model to estimate the temperature for a given set of HRT tasks on a multicore system, also referred to a partitioned scheme.

Authors in [71] provide a HRT multiprocessor scheduler which accounts for energy and thermal constraints by maximizing workload-per-joule (WPJ) subject to a thermal constraint. The allocation of tasks to processors relies on an (enhanced) variable-size bin packing algorithm (En-VSBP): the size of a bin depends on the capability (frequency) of each core. The downside of the heuristic approach the authors take is that it cannot ensure schedulability, and it only explores a very small portion of the solution space, yielding sub-optimal results.

Feedback control has also been used to consider energy constraints, for example Soria-Lopez, et al. [75] presented an energy-based feedback control scheduling framework for power-aware soft real-time tasks executing in dynamic environments where real-time parameters are not previously known *a priori*. The framework contains an energy-based feedback scheduler and a power-aware optimization algorithm.

2.3 System Model

This section exposes the TCPN global model for tasks, CPUs and thermal behaviour, as well as a brief introduction to Tertimuss, the simulation tool used through the work.

2.3.1 Background on Petri Nets

Petri Nets (PNs) is a widely accepted paradigm for modeling, analyzing and control of discrete event systems, basically because of their powerful mathematical background and their nice graphical representation [58]. Petri nets present two interesting characteristics. Firstly, they make it possible to model and visualize behaviors with parallelism, concurrency, synchronization and resource sharing. Secondly, the theoretical results concerning them are plentiful; the properties of these nets have been and still are extensively studied. A Petri Net PN , has two classes of nodes: places and transitions. Places are represented as circles, while transitions with bars, even though some authors have used boxes. Edges can only connect places to transitions, and transitions to places. The following definitions will further describe a PN and its characteristics.

Basic definitions

Definition 2.5 A Petri net structure is a bipartite graph defined with a four tuple $N = \langle P, T, \mathbf{Pre}, \mathbf{Post} \rangle$, where P and T are finite non-empty disjoint sets of nodes named places and transitions, respectively. $\mathbf{Pre} : P \times T \rightarrow \mathbb{N} \cup \{0\}$ is the pre-incidence function that specifies the weighted arcs directed from places to transitions and $\mathbf{Post} : P \times T \rightarrow \mathbb{N} \cup \{0\}$ is the post-incidence function specifying the weighted arcs directed from transitions to places.

A subnet of N , $N' = \langle P', T', \mathbf{Pre}', \mathbf{Post}' \rangle$, is a Petri net structure where $P' \subseteq P$, $T' \subseteq T$ are subsets of places and transitions of N , respectively; and $\mathbf{Pre}' = \mathbf{Pre}[P', T']$ and $\mathbf{Post}' = \mathbf{Post}[P', T']$ are the pre- and post-incidence functions of N restricted to P' and T' . The preset and postset of a node $v \in P \cup T$ are denoted as $\bullet v$ (set of input nodes) and $v \bullet$ (set of output nodes), respectively. These definitions can be naturally extended: let $V \subseteq P \cup T$ be a set of nodes, $\bullet V$ (respectively, $V \bullet$) denotes the union of the preset (respectively, of the postset) of every node $v \in V$.

The incidence matrix of a petri net N is defined as $C = \mathbf{Post} - \mathbf{Pre}$. A column vector $y \neq 0$ is called a P -flow of N if $y^T C = 0$; if the non-null entries of such vector are positive, then it is called a P -semiflow. Similarly, a column vector $x \neq 0$ is termed a T -flow of N if $Cx = 0$; if the non-null entries of that vector are positive, then it is termed a T -semiflow. If there is a P -semiflow (T -semiflow) such that $y > 0$ ($x > 0$), the net is said to be conservative (consistent), respectively.

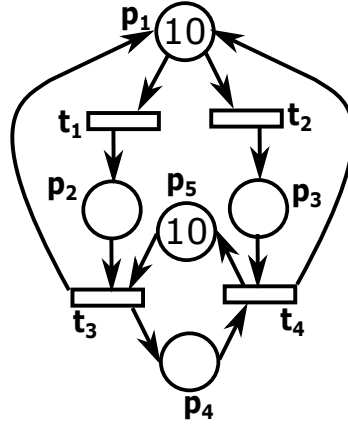


Figure 2.5.: Example of a Petri net structure

Definition 2.6 A marking is a function $m : P \rightarrow N \cup \{0\}$ that assigns to each place a non-negative integer number

Definition 2.7 A Petri net system (or Petri net) is the duple $PN = (N, \mathbf{m}_0)$ where N is a Petri net structure and \mathbf{m}_0 is an initial marking.

Graphically, the marking of a Petri net is given by $m(p)$ black dots or the number $m(p)$ at place p .

In a PN, a transition t_j is *enabled* at marking \mathbf{m}_k iff $\mathbf{m}_k[p_i] \geq \text{Pre}(p_i, t_j), \forall p_i \in P$. In a PN, an enabled transition t_j , at \mathbf{m}_k , can be fired. The firing of an enabled transition t_j produces a new marking that is computed with the fundamental PN equation:

$$\mathbf{m}_{k+1} = \mathbf{m}_k + \mathbf{C}\mathbf{v}_k \quad (2.6)$$

where $\mathbf{v}_k(j) = 1$ and $\mathbf{v}_k(i) = 0, \forall i \neq j$.

Example 2.1 Fig. (2.5) shows a Petri net with 4 places and 4 transitions, where $P = \{p_1, p_2, p_3, p_4, p_5\}$, $T = \{t_1, t_2, t_3, t_4\}$, and an initial marking:

$$\mathbf{m} = [m(p_1) \quad m(p_2) \quad m(p_3) \quad m(p_4) \quad m(p_5)]^T = [10 \quad 0 \quad 0 \quad 0 \quad 10]^T \quad (2.7)$$

The *Pre*, *Post* and *C* matrices are:

$$Pre = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad Post = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} -1 & -1 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \quad (2.8)$$

Firing an enabled transition consists of removing as many marks as indicated by the weight of the arc from the input place of the fired transition and placing the number of marks indicated by the weight of the arc towards the place in the post.

Example 2.2 In Petri net form Fig. (2.5), transitions t_1 and t_2 are enabled under the initial marking $m_0 = [10 \ 0 \ 0 \ 0 \ 10]^T$. If t_1 is fired, marking $m_1 = [9 \ 1 \ 0 \ 0 \ 10]^T$ is reached, i.e., $m_0 \xrightarrow{t_1} m_1$.

Continuous Petri net system

The marking of a place in a *PN* may correspond to the state of a device, a CPU is or is not idle. This marking can be compared to a Boolean variable. A marking can also be associated with an integer, e.g. the number of parts in the input buffer of a robot. In the latter, the number of tokens may be a large number. This may result in such a large number of reachable markings that a *PNs* would suffer from the so called problem of state explosion, and become useful. The continuous Petri net (CPN) is a model in which the number of marks in the places are real numbers instead of integers [22]. CPNs arise from a fluidization process, in which each mark is divided into k parts, such that when $k \rightarrow \infty$, the marking becomes a real number, and the firing can be made through real quantities.

The marking in a CPN is a mapping $m : P \rightarrow \mathbb{R}_{\geq 0}^{|P|}$ that assigns to each place of N a non negative real value.

Definition 2.8 A continuous Petri net (CPN) system is a net N with an initial marking $m_0 : P \rightarrow \mathbb{R}_{\geq 0}^{|P|}$, and it is denoted as $\langle N, m_0 \rangle$

Unlike the discrete case, in a CPN a transition t_i is enabled at a marking \mathbf{m} if and only if $\forall p \in \bullet t_i, m(p) > 0$. The *enabling degree* of a transition t at a marking \mathbf{m} is defined as

$$\text{enab}[t] = \min_{p \in \bullet t} \left\{ \frac{m[p]}{\text{Pre}[p, t]} \right\}. \quad (2.9)$$

An enabled transition t can be fired in any real amount between the interval $0 \leq \alpha \leq \text{enab}(t, \mathbf{m})$. Its firing leads to a new marking $\mathbf{m}' = \mathbf{m} + \alpha \mathbf{C}(P, t)$. If a marking \mathbf{m}' can be reached from \mathbf{m}_0 with a firing sequence σ . The set of reachable markings satisfy the following fundamental equation:

$$\mathbf{m} = \mathbf{m}_0 + \mathbf{C}\sigma \quad (2.10)$$

where $\sigma \in \{\mathbb{R} \cup 0\}^{|\mathcal{T}|}$ is the firing count vector.

Timed continuous Petri net systems

If the firing of transitions is timed, the marking of the continuous system evolves along a trajectory within the set of reachable markings in a deterministic way. In this sense, Eq. 2.10 will depend explicitly on time such that $\mathbf{m}(\zeta) = \mathbf{m}_0 + \mathbf{C}\sigma(\zeta)$, and its time derivative yields the *state equation*,

$$\dot{\mathbf{m}}(\zeta) = \mathbf{C}\dot{\sigma}(\zeta) \quad (2.11)$$

The derivative of the firing count vector $\dot{\sigma}(\zeta)$ is known as the firing flow or through-put vector of the timed model $\mathbf{f}(\zeta) = \dot{\sigma}(\zeta)$. The firing flow may be defined according with a semantic, in this work the *infinite servers semantic* is used, but other common semantics are *finite servers semantic* or *product semantic*, each one will be defined later.

Definition 2.9 A *timed continuous Petri net TCPN system* is a CPN along with a firing rate vector λ , and is denoted as $\langle N, \mathbf{m}_0 \rangle$, where $\lambda : T \rightarrow \mathbb{Q}_{>0}^{|\mathcal{T}|}$.

Definition 2.10 Under infinite servers semantics the firing flow $f(\zeta)$ of transition t_j is proportional its enabling degree, and is defined as

$$f[t_j] = \lambda[t_j] \cdot \min_{p \in \bullet t_j} \left\{ \frac{m[p]}{\mathbf{Pre}[p, t_j]} \right\} \quad (2.12)$$

Definition 2.11 Under finite server semantics, if the marking of the input places of transition t_i is strictly greater than zero, then its flow is constant and equal to λ_i , otherwise the flow is the minimum between the maximum firing speed and the total input flow of the empty places.

$$f[t_j] = \begin{cases} \lambda[t_j] & \text{if } \forall p \in \bullet t_j, m[p] > 0 \\ \min \left\{ \min_{p \in \bullet t_j | m[p]=0} \left\{ \sum_{t_q \in \bullet p} \frac{f_q \cdot \mathbf{Post}[t_q, p]}{\mathbf{Pre}[p, t_j]} \right\}, \lambda[t_j] \right\} & \text{otherwise} \end{cases} \quad (2.13)$$

Definition 2.12 Under product semantics the firing flow $f(\zeta)$ of transition t_j is defined as

$$f[t_j] = \lambda[t_j] \cdot \prod_{p \in \bullet t_j} \left\{ \frac{m[p]}{\mathbf{Pre}[p, t_j]} \right\} \quad (2.14)$$

A TCPN under the semantics of infinite servers is a switched linear system, given by the min operator that appears in the flow equation 2.12. And its marking evolution can be represented by an affine positive piecewise linear system [73]. Throughout this work, the semantics of infinite servers will be used for the modeling of systems.

A *configuration* \mathcal{C} of a net N is a set of arcs (p, t) , one for each transition of the net, such that $p \in \bullet t$. A configuration \mathcal{C} has associated a configuration matrix $\mathbf{\Pi}_{\mathcal{C}} \in \mathbb{Q}_{\geq 0}^{|\mathcal{T}| \times |\mathcal{P}|}$,

$$\mathbf{\Pi}_{\mathcal{C}}[p_i, t_j] = \begin{cases} \frac{1}{\mathbf{Pre}[p_i, t_j]} & \text{if } (p_i, t_j) \in \mathcal{C} \\ 0, & \text{otherwise} \end{cases} \quad (2.15)$$

A configuration \mathcal{C} is active at marking m if $\mathbf{\Pi}_{\mathcal{C}} m = \text{enab}(m)$.

Then, the flow through the transitions in a given configuration \mathcal{C} can be written as the vector $\mathbf{f}(\mathbf{m}) = \mathbf{\Lambda}\mathbf{\Pi}_{\mathcal{C}}\mathbf{m}$, where $\mathbf{\Lambda}$ is the diagonal matrix whose elements $\Lambda_{i,i}$ are the firing transition rates λ_i .

In the systems modeled by TCPNs, control actions may be applied to transitions as means to decrease their speed, i.e their maximum transition flow $f[t]$. A transition t is *controllable* if its flow can be reduced or stopped, the set of all controllable transitions is denoted by T_c . And the set of uncontrollable transitions is denoted by $T_{nc} = T \setminus T_c$.

The control vector $\mathbf{u} \in \mathbb{R}^{|T|}$ is defined such that u_i represents the control action over $t_i \in T$. Then the *effective flow* through a controlled transition is given by:

$$\begin{aligned} w_i(m) &= f[t_i](m) - u_i, \\ 0 &\leq u_i \leq \lambda_i \cdot \text{enab}[t_i](m) \end{aligned} \quad (2.16)$$

where $w_i(m) \geq 0$.

Thus, the behaviour of a TCPN system, in any configuration \mathcal{C} is described by the state equation:

$$\begin{aligned} \dot{\mathbf{m}} &= C\mathbf{\Lambda}\mathbf{\Pi}_i\mathbf{m} - C\mathbf{u}, \\ &= C\mathbf{\Lambda}\mathbf{\Pi}_i\mathbf{m} - C_c\mathbf{u}_c \end{aligned} \quad \text{with } \mathbf{0} \leq \mathbf{u} \leq \mathbf{\Lambda}\mathbf{\Pi}_i\mathbf{m} \quad (2.17)$$

where $C\mathbf{\Lambda}\mathbf{\Pi}_i$ is the *dynamic matrix* for configuration \mathcal{C} ; the *input matrix* $C_c = C(P, T_c)$ the incidence matrix C constrained to the columns of controllable transitions, and $\mathbf{u}_c = \mathbf{u}(T_c)$ is \mathbf{u} restricted to the controllable transitions T_c .

2.3.2 TCPN global model

According to the objectives of this Thesis, we attempt to solve the problem of scheduling HRT tasks on a multicore system while complying with thermal and energy constraints. Therefore, it is of great interest to obtain a model encompassing all of these phenomena.

We have chosen a system model based on Timed Continuous Petri Nets (TCPN) because this formalism is amenable to describe discrete event systems such as the arrival of tasks and allocation of resources, while also being capable to describe

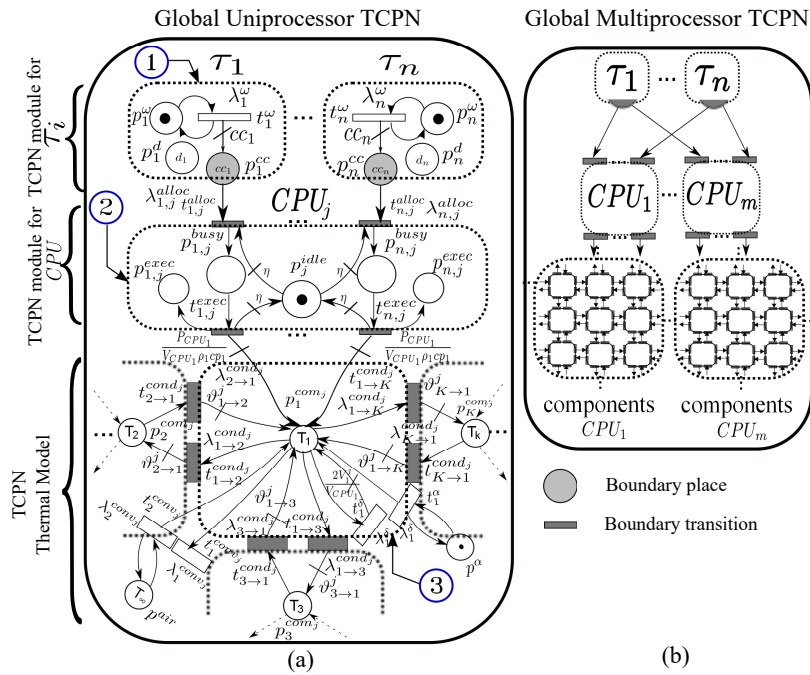


Figure 2.6.: Schema of the TCPN global model, on the right is an abstract representation of the different modules and how they interconnect. On the left, is a more detail representation of a TCPN model for n tasks assigned to one CPU, and one component for the thermal model.

continuous behaviours, such as the heat transfer in a multicore system due to task execution.

The TCPN global model is built by merging three main modules: a *task arrival module*, a *CPU module* and a *thermal model*. Figure 2.6 shows the schema of the model composition. The following sections will extend on how to obtain the model. This methodology is not a contribution of this thesis; we provide an explanation here to offer a better understanding of the contributions, and because we perform changes on the CPU module required for the execution controller in Ch. 5. The task arrival and CPU modules were first presented in [29], then improved in [28]. The heat transfer model methodology using TCPN was first introduced on [78] for a greenhouse climate control, then it was adapted to a multicore system in [30]. Finally, the global TCPN model for tasks, CPUs and temperature were combined into a global single model in [28] and [63].

Remark 2.1 A TCPN is an approximation of the discrete PN, therefore the model shows the *throughput* or *average* value.

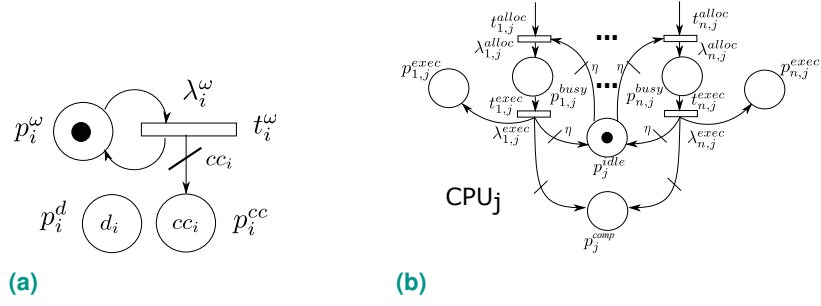


Figure 2.7.: (a) TCPN module for task arrival τ_i . (b) TCPN module for CPU_j .

Task module

The task module represents the arrival of a periodic task into the system, and is displayed on Fig. 2.7a. Each periodic task τ_i is defined by the 3 – tuple $\tau_i = (cc_i, d_i, \omega_i)$, where cc_i is the worst-case execution time in processor cycles (WCET), ω_i the period and d_i is the relative implicit deadline ($d_i = \omega_i$) [6].

The period ω_i implies that $\frac{1}{\omega_i}$ jobs arrive per second in average. This is captured as the firing rate $\lambda_i^\omega = \frac{1}{\omega_i}$ of transition t_i^ω in the TCPN module. The duration of the task is represented by the arc going from transition t_i^ω to place p_i^{cc} . This arc models job arrival. Accordingly, the marking of place p_i^{cc} stands for the CPU cycles that remain to be executed, its initial state is cc_i because each task activates at time zero.

CPU module

The CPU module, shown on Fig. 2.7b, models two important behaviours: task allocation to a processor and task execution on a processor.

As with the task module, there is one module per CPU on the system. It is composed of n transitions $t_{i,j}^{alloc}$, n transitions $t_{i,j}^{exec}$, n places $p_{i,j}^{busy}$, n places $p_{i,j}^{exec}$ and a place p_j^{idle} (Fig. 2.7b).

The initial marking of place p_j^{idle} is set to 1 to indicate that CPU_j is available or idle.

Task allocation The firing of transition $t_{i,j}^{alloc}$ means that a job from τ_i has been allocated on CPU_j and will be executed. The transition rate $\lambda_{i,j}^{alloc}$ should be close

to infinity, since this is assumed as an instantaneous event¹. Places $p_{i,j}^{busy}$ represent the busy state of processor CPU_j due to the allocation of a job from τ_i .

Task execution Transitions $t_{i,j}^{exec}$ are fired when τ_i is executing on processor CPU_j , and its transition rate $\lambda_{i,j}^{exec}$ models the execution rate of the task. Places $p_{i,j}^{exec}$ hold the accumulated execution of τ_i over time.

The arcs going from transitions $t_{i,j}^{exec}$ to place p_j^{idle} and from place p_j^{idle} to transitions $t_{i,j}^{alloc}$ are weighted by a constant value η , to ensure that the flow through transitions $t_{i,j}^{alloc}$ is limited by the throughput capacity of the CPU, and the TCPN evolves in only one configuration.

Thermal model

Each processor generates heat due to task execution. This heat propagates across the system by conduction from more energetic particles to the adjacent less energetic ones [37]. Finally, the surfaces in contact with air experience convection, given the exchange of heat between air particles and the boundary surface.

Heat transfer can be modeled by a partial differential equation (PDE) and a set of defined boundary conditions. Based on the methodology presented on [63] and [28] it is possible to build the model using elementary TCPN modules that represent the convection, conduction and heat generation. Thus, avoiding to work with a model formulated with PDEs. Appendix B provides a very detailed explanation on how to produce TCPN elementary modules from a PDE.

The first step to use the elementary TCPN modules is to perform an spatial discretization of the multi-core platform, such that the geometry is divided into smaller components. Then each component is associated to an elementary TCPN module according to the phenomena present on that element. For example, on the components corresponding to a CPU, there is heat generation and heat conduction with the neighbour elements.

Following that methodology, the thermal model is built for the layout of the multicore system and its physical properties. Fig. 2.8 shows the resulting TCPN for a component subjected to conduction, convection and heat generation; place $p_{1,com}$ represents the average temperature of component one.

¹For simulations purposes it is enough to set $\lambda_{i,j}^{alloc}$ one order of magnitude greater than $\lambda_{i,j}^{exec}$

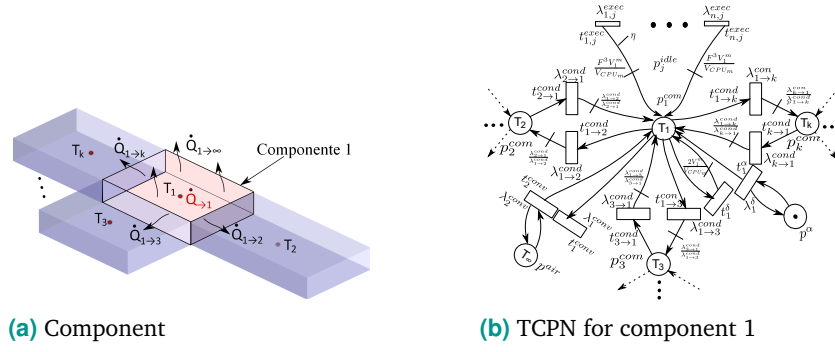


Figure 2.8.: TCPN for a component subjected to conduction, convection and heat generation; place p_{1com} represents the average temperature the component.

In the methodology exposed in Appendix B, the heat generation modules provide a constant heat generation rate. However, in the multi-core system the heat generation might not be constant, and it rather depends on its power dissipation. For this reason the heat generation modules differ and are shown on Table. 2.2. The power model used in this Thesis appears as a weight on the arc that connects the CPU module with the thermal model. More precisely, there is an arc connecting every transition $t_{i,j}^{exec}$ to each component (from the spatial discretization) on CPU_j , relating task execution (i.e frequency) and heat generation.

Power Dissipation and Energy consumption The power dissipated by a processor has two main components: the dynamic power due to operation and the static power. It is computed as:

$$P_{CPU_j} = P_{dyn_j} + P_{leak_j} \quad (2.18)$$

on the expression, P_{dyn_j} corresponds to the dynamic power on CPU_j , and P_{leak_j} refers to the power dissipated due to leakage currents, which is the main component for the static power.

The dominant component of power consumption in CMOS technology is the dynamic power P_{dyn_j} given by $P_{dyn_j} = C_{eff} V_{dd}^2 F$, where C_{eff} is the effective switching capacitance, V_{dd} the supply voltage and F is the frequency of the clock. Given that $V_{dd} \propto F$ and k is a modeling constant, P_{dyn_j} can be stated as:

$$P_{dyn_j} = kF^3. \quad (2.19)$$

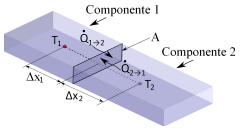
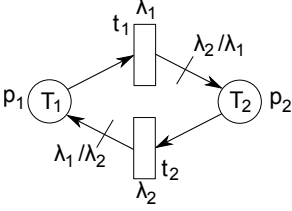
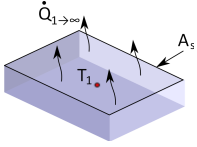
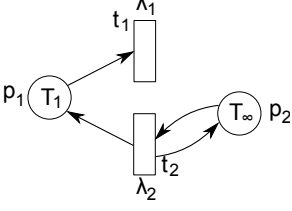
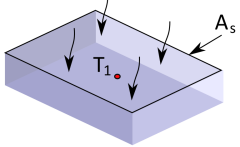
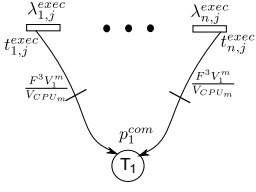
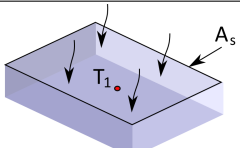
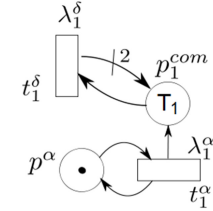
	Component	TCPN module	Parameter
Conduction			$\lambda = \begin{bmatrix} \frac{1}{V_1 \rho_1 c p_1} \cdot \frac{k_1 k_2 A}{k_2 \Delta x_1 + k_1 \Delta x_2} \\ \frac{1}{V_2 \rho_2 c p_2} \cdot \frac{k_1 k_2 A}{k_2 \Delta x_1 + k_1 \Delta x_2} \end{bmatrix}$ $Pre = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad Post = \begin{bmatrix} 0 & \frac{\lambda_1}{\lambda_2} \\ \frac{\lambda_2}{\lambda_1} & 0 \end{bmatrix}$
Convection			$\lambda = \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} \frac{h A_s}{V_1 \rho_1 c p_1} \\ \frac{h A_s}{V_1 \rho_1 c p_1} \end{bmatrix}$ $Pre = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad Post = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$
Heat generation due to task execution			$Pre = [0 \dots 0]$ $Post = \begin{bmatrix} \frac{F^3 V_1^1}{V_{CPU_1}} & \dots & \frac{F^3 V_m^k}{V_{CPU_m}} \end{bmatrix}$
Heat generation due to P_{leak}			$\lambda^{leak} = \begin{bmatrix} \lambda_1^\delta \\ \lambda_1^\alpha \end{bmatrix} = \begin{bmatrix} \delta \\ \alpha \end{bmatrix}$ $Pre = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad Post = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$

Table 2.2.: TPCN modules for element subjected to conduction, convection and heat generation

On the other hand, P_{leak_j} can be modeled as a linear function of temperature ([1]),

$$P_{leak} = \delta T + \rho, \quad (2.20)$$

where T is the CPUs temperature and δ and ρ are modeling constants.

Based on the previous statements, the average energy E_j consumed during a time interval interval $(\zeta_1, \zeta_2]$ by the tasks running on CPU_j is defined as:

$$E_j = \int_{\zeta_1}^{\zeta_2} P_{CPU_j}(F) d\zeta \quad (2.21)$$

2.3.3 Global model

The global model is obtained by merging the tasks modules, the CPU modules and the thermal model together. This is achieved connecting *boundary places and transitions* with arcs.

Tasks are linked to CPUs by connecting places p_i^{cc} to transitions $t_{i,j}^{alloc}$ with an arc. This arc implies that jobs from task τ_i can be allocated on processor CPU_j , therefore by selecting which arcs to include it is possible to model partitioned, clustered or global schemes.

The CPU and thermal model are connected by means of the the power dissipation/heat generation with the weighted arcs mentioned on the previous section.

Fundamental equation

The dynamic behavior of the global model (Fig. 2.6) is provided by the following equations:

$$\dot{m}_{\mathcal{T}} = C_{\mathcal{T}}\Lambda_{\mathcal{T}}\Pi_{\mathcal{T}}(m)m_{\mathcal{T}} + C_{\mathcal{T}}^{alloc}w^{alloc} \quad (2.22a)$$

$$\dot{m}_{\mathcal{P}} = C_{\mathcal{P}}\Lambda_{\mathcal{P}}\Pi_{\mathcal{P}}(m)m_{\mathcal{P}} + C_{\mathcal{P}}^{alloc}w^{alloc} \quad (2.22b)$$

$$\dot{m}_{exec} = f^{exec} \quad (2.22c)$$

$$\dot{m}_T = C_T\Lambda_T\Pi_T(m)m_T + C_a\Lambda_a\Pi_a(m)m_a + C_{\mathcal{P}}^{exec}f^{exec} \quad (2.22d)$$

$$\dot{m}_a = 0 \quad (2.22e)$$

C_x , Λ_x , and $\Pi_x(m)$ are the incidence matrix, the firing rate transitions and the configuration matrix ($x = \{T, \mathcal{T}, \mathcal{P}\}$) of the thermal, task, and processor subnets respectively. Each equation from system (2.22) represents a module from the TCPN representation on Fig. 2.6. Eq. (2.22a) describes the periodic arrival of each task, Eq. (2.22b) the processors behavior, and Eq. (2.22c) the processors execution rate (i.e frequency). Finally, Eq. (2.22d) represents the thermal evolution of the system due to task execution, with Eq. (2.22e) indicating that the environmental temperature keeps constant during observation time (its derivative is neglected).

2.4 Tertimuss

Tertimuss is an open-source framework to model a RT multiprocessor system, simulate different RT schedulers, and process the results. It is publicly available at [21].

The development of Tertimuss begun with algorithms and tools first implemented on MatLab [27]. It was utterly refactored in [20], porting the MatLab algorithms to Python, adding new ones, refactoring the framework in a modular way, and improving the components in terms of performance, usability and maintainability. Tertimuss has been also extended as part of the work of this Thesis. All the proposed algorithms have been implemented in Tertimuss. The experimental artifacts are available in the framework.

Appendix A provides an overview of the architecture and characteristics of Tertimuss.

Energy-Efficient Thermal-Aware RT Multiprocessor Scheduling

This chapter presents an energy-efficient thermal-aware RT global scheduler for a set of HRT tasks on a multiprocessor system. This global scheduler fulfills thermal and temporal constraints by handling two independent variables: task allocation and clock frequency selection. The proposed scheduler is split into two stages: one offline and another online.

The off-line stage takes two steps. The first step computes a frequency range $[F^*, F^+]$ derived from the analysis of the TCPN global model from Section 2.3.2, such that the HRT task set can execute correctly. Using the minimum frequency F^* power consumption is minimized, whilst the system is at full capacity. Meanwhile, F^+ is the maximum frequency that still guarantees the fulfilment of the thermal bound. In other words, any frequency increase above F^+ would imply a thermal bound violation. Then, the off-line stage leverages a deadline partitioning approach to compute the CPU cycles that each task must execute per time interval. Finally, the on-line stage performs the task allocation employing a Fixed-Priority until Zero-Laxity policy (FPZL, [25]). It also includes an Adaptive Scheduler (AS) which adds robustness by throttling the CPU frequency to accept/reject SRT aperiodic tasks, thus ensuring the correct execution of the HRT task set minimizing energy consumption and keeping a controlled temperature.

3.1 Problem definition

This section defines the system model and design constraints for the scheduling problem herein addressed, which is formally stated in Problem 3.1.

Multiprocessor system The multiprocessor system is composed by m processors, where job migration is permitted. $\mathcal{P} = \{CPU_1, \dots, CPU_m\}$ is the set of the m

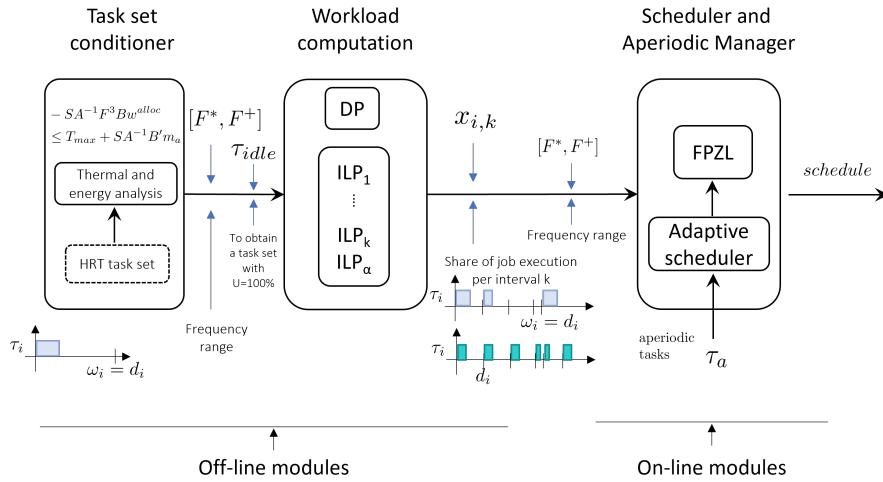


Figure 3.1.: Overview of the scheduling solution: the task set conditioner module is detailed on section 3.2.1 and it works upon the TCPN global model. The workload computation is presented on sec. 3.2.2, it leverages a DP-fair scheme and solves α integer linear programming problems (ILP). Finally, the workload is allocated to processors on the scheduler and aperiodic manager module which are on section 3.3

identical processors with an homogeneous clock frequency $f \in \mathcal{F} = \{f_1, \dots, f_{max}\}$, i.e a change on the operating frequency reaches all m processors.

Task model The set of HRT tasks is denoted by $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$. Each task is independent, fully preemptive and defined by the 3-tuple $\tau_i = (cc_i, d_i, \omega_i)$, where cc_i is the worst-case execution time (WCET) in cycles, ω_i is the task period, and d_i is the relative implicit deadline ($d_i = \omega_i$) ([6]).

The arrival of asynchronous, SRT aperiodic tasks is also considered. Each aperiodic task τ_i^a is defined as a 3-tuple (cc_i^a, d_i^a, r_i^a) in which cc_i^a (required CPU cycles) and d_i^a (deadline) are known at task arrival time, and the arrival time r_i^a is unknown.

Remark 3.1 It is assumed that all task parameters are integers and that every job can be preempted at any time. The activation of the HRT tasks occur at time zero.

Problem 3.1 *Minimum Energy Thermal Aware RT Scheduler (METARTS)*. Given the sets \mathcal{T} of tasks and \mathcal{P} of CPUs, the METARTS problem consists in designing an algorithm to allocate the tasks in \mathcal{T} to the m identical CPUs within the hyperperiod H such that deadlines for \mathcal{T} are always satisfied, the CPU temperatures are always kept below a given bound T_{max} and the consumed energy is minimum. Additionally, the scheduler must accept/reject aperiodic tasks upon arrival subject to HRT tasks and thermal constraints.

3.2 Off-line stage

This stage computes two main elements: a range of operating frequencies and the workload to execute. The range of *valid* operating frequencies $[F^*, F^+]$ is obtained from the analysis of the TCPN global model presented on section 2.3.2. The second part of the offline stage leverages a *deadline partitioned* (DP) scheme, that partitions time into slices. Within each slice, we solve an integer linear programming problem (ILP) to compute the workload.

3.2.1 Set of working frequencies

The TCPN global model is a mathematical representation of three phenomena present on our problem: task execution, energy consumption and heat generation (due to the execution of tasks on processors). Thus, the relationship between task execution and processors temperature is captured in a *thermal constraint*, obtained from an steady state analysis, which will be discussed on the following section. Therefore, the range of *valid* operating frequencies $[F^*, f^+]$ has to guarantee the *thermal constraint* even when working at full capacity, i.e $U = m$, besides ensuring the HRT temporal constraints. Both frequencies F^* , F^+ are computed using mathematical optimization.

Thermal constraint

As it was mentioned earlier, the TCPN thermal equation Eq. 2.22d is used to derive the thermal constraint through a steady state analysis. The schedule is periodic because it repeats every hyperperiod, therefore the same applies for thermal solution. In this case, the initial and final temperature in every hyperperiod must be equal.

$$\begin{aligned} \dot{m}_T &= A m_T + B' m_a + F^3 B f^{exec} \\ Y_T &= S \vec{m}_T \end{aligned} \quad (3.1)$$

where $A = C_T \Lambda_T \Pi_T(m)$, $B = C_P^{exec}$ and $B' = C_a \Lambda_a \Pi_a(m)$. In a steady state temperature, $m_{T_{ss}}$, when time tends to infinite $\dot{m}_T = 0$. Hence,

$$m_{T_{ss}} = -A^{-1}(F^3 B w^{alloc} + B' m_a).$$

The steady state temperature must be less than or equal to its maximum temperature level to comply with the thermal constraint of CPUs, i.e. $S\mathbf{m}_{T_{ss}} \leq T_{max}$ (thermal constraint) then:

$$-SA^{-1}F^3B\mathbf{w}^{alloc} \leq T_{max} + SA^{-1}B'\mathbf{m}_a \quad (3.2)$$

This equation provides the thermal constraints that the allocation of tasks to the processors (\mathbf{w}^{alloc}) must fulfill. It includes the the clock frequency, the temperature bounds and the allocated tasks, which in the steady state are equivalent to the executed tasks. This equation will be used to compute the range of feasible operation frequencies.

Minimum frequency

The proposed approach aims to minimize the system dynamic energy consumption under the HRT, and thermal constraints. The energy minimization is explored under DVFS, such that processor frequency can vary by selecting one from a finite set of a preset values, i.e. $F = \{f_{min}, \dots, f_{max}\}$.

Recall the energy and power dissipation equations from section 2.3.2, considering only the dynamic component of the dissipated power, the consumed energy is minimized Eq. 2.21 iff the clock frequency f is minimized. Nevertheless, f must be fast enough to ensure that the temporal constraints are met. The next proposition obtains the minimum clock frequency that fulfills the temporal constraints.

Proposition 3.1 *Assuming that the task utilization is less than the number of processors $U \leq m$ and $u_i \leq 1 \forall f \in \mathcal{F}$, the clock frequency that minimizes the total energy consumption while meeting temporal constraints is constant:*

$$F^{**} = \max\{f_{min}, \frac{1}{m} \sum_{i=1}^n \frac{cc_i}{\omega_i}\}. \quad (3.3)$$

Proof 3.1 *According to Eq. (2.21) with $P = kf^3$, the energy has a minimum iff the consumer power is minimum. This occurs when f^3 is minimum and fulfills that $\sum_{i=1}^n \frac{cc_i}{fw_i} = m$, and $f \geq f_{min}$. Using Lagrange multipliers, the Lagrangian function is $L = f^3 + \mu_1(\frac{1}{f} \sum_{i=1}^n \frac{cc_i}{w_i} - m) + \mu_2(f_{min} - f)$. The solution yields four cases: a) Both multipliers are inactive ($\mu_{1,2} = 0$); b) Both multipliers are active ($\mu_{1,2} \geq 0$); c) $\mu_1 = 0$ and $\mu_2 \geq 0$; and d) $\mu_1 \geq 0$ and $\mu_2 = 0$. The first case is unfeasible, because f cannot*

be zero. In the second case, the only solution is $f = f_{min} = \frac{1}{m} \sum_{i=1}^n \frac{cc_i}{w_i}$. Finally, if one multiplier is active while the other one is inactive there are two possible solutions: $f = f_{min}$ or $f = \frac{1}{m} \sum_{i=1}^n \frac{cc_i}{w_i}$. Consequently, in order to fulfill both constraints, the frequency that minimizes the total energy consumption becomes $F^{**} = \max\{f_{min}, \frac{1}{m} \sum_{i=1}^n \frac{cc_i}{w_i}\}$.

The frequency F^{**} meets the temporal constraints. We must compute w^{alloc} and solve Eq. (3.2) to guarantee the thermal constraint fulfilment. At frequency F^{**} the system utilization is $U = \sum_{i=1}^n \frac{cc_i}{\omega_i f} = m$ and the processor frequency is:

$$F^* = \min\{f \in F | f \geq F^{**}\} \quad (3.4)$$

given the nature of the discrete set of frequencies.

When computing F^{**} (Eq. 3.3) we assume a fully utilized system, but actual F^* in Eq. 3.4 can make the execution faster, causing the utilization to become below 100%. In those cases we introduce an idle task $u_{idle} = m - \sum_{i=1}^n \frac{cc_i}{F^* \omega_i}$ to ensure that system utilization is 100%. The task allocation ratio is calculated as follows, to make the distribution of the CPU cycles required to execute all tasks homogeneous:

$$w^{alloc} = \left[\frac{1}{m} \sum_{i=1}^n \frac{cc_i}{\omega_i F^*}, \dots, \frac{1}{m} \sum_{i=1}^n \frac{cc_i}{\omega_i F^*} \right]^T \quad (3.5)$$

w^{alloc} controls the flow of the *allocation transitions* in the TCPN ($t_{i,j}^{alloc}$ in Fig. 2.6), thus modeling the allocation rate of tasks to CPUs (Eq. 2.22). If w^{alloc} satisfies Eq. (3.2), then the thermal constraints are also satisfied. Otherwise, the *METARTS* problem does not have a solution. If it result in a feasible F^* , then we can compute the maximum CPU cycles available for aperiodic tasks, and the maximum clock frequency that can be used subject to thermal constraints.

Maximum frequency

The *maximum thermal frequency* $F^+ \in \mathcal{F}$ is the greatest frequency at which all CPUs can operate at 100% of utilization and still meet the thermal constraint. To compute

F^+ , first we solve the programming problem in Eq.(3.6) to find the frequency upper bound F_c that satisfies the constraints.

$$\begin{aligned}
& \max \quad F_c \\
& \text{s.t.} \\
& -\mathbf{SA}^{-1}\mathbf{F}_c^3\mathbf{B} \left[\frac{CC_1}{F_c H}, \dots, \frac{CC_m}{F_c H} \right]^T \leq \mathbf{T}_{max} + \mathbf{SA}^{-1}\mathbf{B}'\mathbf{m}_a \\
& \frac{CC_j}{F_c H} = 1 \quad \forall j = 1, \dots, m \\
& F^* \leq F_c \leq F_{max}
\end{aligned} \tag{3.6}$$

The first constraint establishes the thermal requirements. CC_j represents the cycles that CPU_j must execute per hyperperiod. Since all CPUs must work at their maximum capacity, the second constraint implies that the CPU utilization is 100%. The last constraint bounds f_c to the actual clock frequency range of CPUs. Finally, the solution for F^+ has to be in the set F of discrete frequencies, thus the processor frequency f^+ is calculated as,

$$F^+ = \max\{f \in F | f \leq f_c\}. \tag{3.7}$$

With the minimum frequency f^* and the maximum thermal frequency f^+ we define

$$\mathcal{F} = \{f \in F | f^* \leq f \leq f^+\}, \tag{3.8}$$

as the set of operating frequencies that meet the thermal constraint.

3.2.2 Workload computation

Given the periodic task system comprising n tasks to be executed on m -processors and frequency F^* , the second part of the offline stage leverages a *deadline partitioned* (DP) scheme 2.2.4. In this offline stage we will compute the workload per time interval, which will also be referred as frame. This is, the share of execution that each job should execute per frame to satisfy its temporal requirements.

The workload computation is performed by solving an ILP at each frame. Then, its solution is carried away as the input for the ILP on the following frame. This process

is repeated up to the last frame. The reason why we impose an *integral* solution is because the workload is expressed on cycles rather than time, and a cycle is an indivisible unit. Nevertheless, it will be proved that the ILP formulation meets the *unimodularity* property, therefore the solution of the optimization problem through a simplex method always ensures *integral* solutions.

ILP representation and unimodularity

A job is any instance of a periodic task τ_i . Therefore, a periodic task can produce an infinite number of jobs. Nevertheless, in the context of periodic tasks, the hyperperiod represents the smallest time window that would be repeated over time. For this reason, we can study the behaviour of the system by analysing the set of jobs within H .

Definition 3.1 Let SD be the ordered set of deadlines of all jobs $j \in \mathcal{J}$ within $H \cup \{0\}$. Such that $sd_0 = 0$, $sd_{k-1}, sd_k \in SD$, $sd_{k-1} < sd_k$, where k is an integer value greater than zero. Then, the frames (scheduling intervals) are defined as $\phi_k = [sd_{k-1}, sd_k)$ $\forall sd_k \in SD$.

Every $\tau_i \in \mathcal{T}$ spawns $\frac{H}{\omega_i}$ jobs within the hyperperiod H . And the number of scheduling intervals (α) is always one less than the cardinality of the set of deadlines, i.e.:

$$\alpha = |SD| - 1 \quad (3.9)$$

Thus, there will always be α linear programming problems to solve.

Definition 3.2 The absolute laxity cc'_i of τ_i is the maximum time (in cycles) that τ_i can remain idle before compromising its deadline, when executing at speed (frequency) f . This is, $cc'_i = (d_i f) - cc_i$, where d_i is the deadline and cc_i the WCET of τ_i .

Example 3.1 Consider a task set $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, with implicit deadlines $d_1 = 4, d_2 = 5, d_3 = 10$ and $d_4 = 20$. Therefore, the hyperperiod is $H = lcm(4, 5, 10, 20) = 20$, the task set spawns 12 jobs $\mathcal{J} = \{j_1, j_2, \dots, j_{12}\}$. Each task generates a different number of jobs (Fig. 3.2).

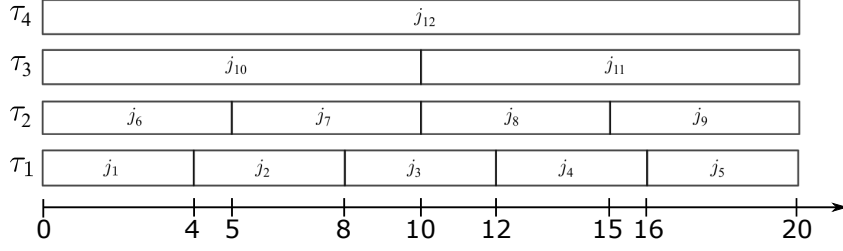


Figure 3.2.: Jobs generated by the task set of Example 1

The set of deadlines is:

$$SD = \{0, 4, 5, 8, 10, 12, 15, 16, 20\},$$

where $|SD| = 9$, such that it generates $\alpha = 8$ frames:

$$\begin{aligned} \phi_1 &= [0, 4) & \phi_2 &= [4, 5) & \phi_3 &= [5, 8) & \phi_4 &= [8, 10) \\ \phi_5 &= [10, 12) & \phi_6 &= [12, 15) & \phi_7 &= [15, 16) & \phi_8 &= [16, 20) \end{aligned}$$

Variables

The variable $x_{i,k}$ denotes the share of execution cycles from the task τ_i that is scheduled upon the k -th interval/frame ϕ_k . That is, variables $x_{i,k}$ take integer values such that $x_{i,k} \leq cc_i$, assuming that an *execution cycle* is an indivisible unit.

The index i takes each integer value in the range $[1, n]$, where n is the number of tasks in \mathcal{T} . For each i , the index k iterates on the number of frames ϕ_k , i.e. on range $[1, \alpha]$. In total, the number of $x_{i,k}$ variables per LPP is equal to n and there are α LPPs to solve. Furthermore, let $X_{HRT,k}$ be the set of variables in frame ϕ_k , and X_{HRT} the set of all variables $x_{i,k}$, $\forall i \forall k$.

Remark 3.2 When the task set utilization is $U < m$, its necessary to add an idle or *dummy task*, to ensure a system at maximum capacity. Such that,

$$u_{idle} = m - \sum_{i=1}^n \frac{cc_i}{f\omega_i}$$

where f is the system frequency and $\tau_{idle} = (u_{idle} \times H, H)$.

Constraints

The time is multiplied by the frequency F^* to represent the available execution capacity over a time window, since the $x_{i,k}$ variables represent execution cycles. This is, in frame $\phi_1 = [0, 4)$, from Example 3.1 there are $m \times |\phi_1| \times F^* = 4$ available cycles for $m = 1$ and $F^* = 1$.

The following constraints ensure the completion of a task before its deadline:

1. The *Maximum utilization constraint* (M.c) ensures that the system utilization per frame ϕ_k is 100%. That is, the summation of every share of execution per task on each frame has to add up to the available system capacity,

$$\sum_{i=1}^n x_{i,k} = m \times |\phi_k| \times F^* \quad (3.10)$$

There are 1 of such constraints per frame ϕ_k .

2. The following constraint ensures that each job completes before its deadline. This is expressed by means of the accumulative execution, because the solution of the previous LPP is carried on to the next LPP. It can take two forms, depending on whether the frame matches the task deadline or not. Since frames were defined from the set of deadlines, at least one task will have its own deadline on ϕ_k , but not necessarily all. To know if τ_i has its deadline on $\phi_k = [sd_{k-1}, sd_k)$, it suffices to write $sd_k = q_{i,k} \times d_i + r_{i,k}$ and check if the residual $r_{i,k}$ is zero. The quotient $q_{i,k}$ represents the number of past jobs that have completed, which is needed to compute the cumulative execution.

$$e = \begin{cases} \sum_{j=\gamma}^k x_{i,j} = q_{i,k} \times cc_i & \text{if } r_{i,k} = 0 \\ \sum_{\gamma=1}^k x_i^\gamma \geq q_{i,k} \times cc_i + \max\{0, F^* \sum_{\gamma=1}^k |\phi^\gamma| - F^* q_{i,k} d_i - cc_i'\} & \text{if } r_{i,k} \neq 0 \end{cases} \quad (3.11)$$

Given that $x_{i,j}$ variables are cycles from τ_i they must all be assigned values greater or equal to zero $x_{i,j} \geq 0 \forall i \forall k$. This is the rational underlying the max operator.

3. Task parallelism is not allowed, therefore each task cannot execute more cycles than those available in each frame,

$$x_{i,k} \leq |\phi_k| \times F^* \quad (3.12)$$

This constraint is referred as *Sequential constraint* (S.c), with as many constraints as tasks (n).

Thus, the final amount of constraints per LPP is $2n + 1$. Constraint (3.11) is improved on the following chapters.

Example 3.2 Let us consider the task set from Example 3.1. The constraints for the third LPP in $\phi_3 = [5, 8)$ yield

$$x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} = m \times 3 \times f$$

Task τ_1 has a deadline on ϕ_3 , therefore,

$$x_{1,1} + x_{1,2} + x_{1,3} = 2 \times cc_1$$

variables $x_{1,1}$ and $x_{1,2}$ have already been solved, so we replace them with their calculated values. The other three tasks do not have a deadline on ϕ_3 , their residuals $r_{i,3}$ are: $r_{2,3} = 3$, $r_{3,3} = 8$ and $r_{4,3} = 8$. Only τ_2 had released its second job, so the quotients are $q_{2,3} = 1$, $q_{3,3} = 0$ and $q_{4,3} = 0$. Thus,

$$x_{2,1} + x_{2,2} + x_{2,3} \geq cc_2 + \max \{0, (3 \times F^*) - cc'_2\}$$

$$x_{3,1} + x_{3,2} + x_{3,3} \geq \max \{0, (8 \times F^*) - cc'_3\}$$

$$x_{4,1} + x_{4,2} + x_{4,3} \geq \max \{0, (8 \times F^*) - cc'_4\}$$

as in the past case, variables $x_{2,1}, x_{2,2}, x_{3,1}, x_{3,2}, x_{4,1}$ and $x_{4,2}$ have already been solved so we replace them with their calculated values. Finally, the upper bounds for $x_{i,3}$ variables are:

$$x_{1,3} \leq 3 \times F^*, \quad x_{2,3} \leq 3 \times F^*, \quad x_{3,3} \leq 3 \times F^*, \quad x_{4,3} \leq 3 \times F^*.$$

Objective function

The objective function minimizes

$$\sum_{i=1}^n x_{i,k} \quad (3.13)$$

This function is only intended to ensure a solution, and other metrics could be used.

Proposition 3.2 *Given a task set \mathcal{T} , where the task utilization at F^* is equal to the number of CPUs, the solution of the LPPs in Eq. (3.14) is always integer. Moreover, if each task τ_i is executed exactly x_i^k cycles during the k – th interval, then a feasible schedule is obtained.*

$$\begin{aligned} \forall k = 1, \dots, \alpha, \text{ solve :} \\ \min \quad & \sum_{i=1}^n x_i^k \\ \text{s.t} \quad & \\ & \sum_{i=1}^n x_i^k = m * |I_{SD}^k| * F^* \\ & \begin{cases} \sum_{j=\gamma}^k x_{i,j} = q_{i,k} \times cc_i & \text{if } r_{i,k} = 0 \\ \sum_{\gamma=1}^k x_i^\gamma \geq -q_{i,k} \times cc_i + \max\{0, F^* \sum_{\gamma=1}^k |\phi^\gamma| - F^* q_{i,k} d_i - cc'_i\} & \text{if } r_{i,k} \neq 0 \end{cases} \\ & \forall i \quad x_i^k \leq |I_{SD}^\gamma| * F^* \end{aligned} \quad (3.14)$$

Proof 3.2 Let $\mathcal{T}^k = \mathcal{T}_1^k \cup \mathcal{T}_2^k$, where \mathcal{T}_1^k and \mathcal{T}_2^k partition the task set. $\mathcal{T}_1^k = \{\tau_1, \dots, \tau_v\}$ is the set of tasks that have their deadlines at sd_k and $\mathcal{T}_2^k = \{\tau_{v+1}, \dots, \tau_n\} = \mathcal{T} - \mathcal{T}_1^k$. In the LPP 3.14, the last two constraints must be converted into equality equations. This is solved by adding slack variables h_i to each constraint. Then all the constraints are represented as $My = b$ where the vector of variables is composed of the workload and slack variables, i.e. $y = [x \ h]^T$. Notice that vector b is always integer. By construction, the restriction matrix M has the form:

$$M = \left[\begin{array}{c|c} L_{(v+1) \times n} & \emptyset \\ \hline Q_{(2n-v) \times n} & I_{(2n-v)} \end{array} \right] \quad (3.15)$$

where L has the form:

$$L_{v+1 \times n} = \left[\begin{array}{cccc|cccc} 1 & 1 & \cdots & 1 & 1 & \cdots & 1 & \\ 1 & 0 & \cdots & 0 & 0 & \cdots & 0 & \\ 0 & 1 & \cdots & 0 & 0 & \cdots & 0 & \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & \cdots & 1 & 0 & \cdots & 0 & \end{array} \right] \quad (3.16)$$

It is easy to observe that the rank of L is $v + 1$. Hence, the rank of M is $\text{rank}(M) = \text{rank}(L) + \text{rank}(I) = 2n + 1$, i.e M is a full row rank matrix. In order to prove that the solution is always integer, we will demonstrate that the restriction matrix M is unimodular.

M is a full row rank matrix, therefore the determinant of every square submatrix (M_{si}) of order $2n + 1$, obtained by removing columns, must be equal to 1, 0 or -1 to prove that M is unimodular [72]. Once the columns are removed, the following three scenarios are possible.

1) If any of the first v columns is removed, M_{si} losses rank, because a row with only zero elements remains, hence the determinant is 0. 2) If the removed column $M(\bullet, j)$ contains a nonzero entry $M(i, j)$, where $j > v$, then the corresponding row $M(i, \bullet)$ has a nonzero element among the first v columns, then M_{si} losses rank since the resulting row is duplicated among the first v rows. Thus the determinant is 0. 3) When any other column not listed before is deleted, the resulting matrix always can be arranged as

$$M_{si} = \left[\begin{array}{c|c} A & \emptyset \\ \hline B & I \end{array} \right] \quad (3.17)$$

Then, according to Theorem 3.2 in [72], A is always a totally unimodular matrix, thus $\det(A) = 0, \pm 1$. Also, the determinant for the identity matrix is always 1; therefore, applying determinant per blocks, $\det(M_{si}) = \det(A) = 0, \pm 1$.

Output from the Offline stage Summarizing, this offline stage computes two components. First, the set of valid CPU clock frequencies, Eq. (3.8), $\mathcal{F} = \{f \in F | F^* \leq f \leq F^+\}$. Second, the set X_{HRT} with every variable $x_{i,k}$, that denotes the share of cycle execution from HRT task τ_i that has to be scheduled upon the k -th interval/frame ϕ_k .

Algorithm 1 Scheduler

- 1: **Input** \mathcal{T} : Task set; phi_k – Scheduling (deadline) intervals;
 X – Task cycles per deadline interval;
 $ex_{i,k}$ – Cycles actually executed for each task since the last scheduling event;
 F_n – CPU frequency
 - 2: **Output** $W_{i,j}^{alloc}$ or pair (τ_i, CPU_j)
 - 3: **Aux. functions**
 - 4: **if** reach a new ϕ_k **then**
 - 5: Update scheduling interval: $k = k + 1$
 - 6: Compute task priorities using *Priority Levels*
 - 7: Execute the m tasks with higher priority
 - 8: **else if** reach a zero laxity **then**
 - 9: Compute task priorities using *Priority Levels*
 - 10: Execute the m tasks with higher priority
 - 11: **else if** aperiodic task arrival **then**
 - 12: call the Adaptive Scheduler
 - 13: **end if**
-

3.3 On-line stage

The inputs for the on-line stage are: the ordered set of deadline intervals or frames ϕ_k , the cycles that each task must execute per deadline interval X_{HRT} , the set of feasible CPU frequencies \mathcal{F} , the cycles that each job have executed since the last scheduling event $ex_{i,k}$, and the aperiodic task parameters. The output signal $w_{i,j}^{alloc}$ (Eq. 3.5), is the vector that determines the task allocation ratio of the allocation transitions $t_{i,j}^{alloc}$ in the TCPN model (Eq. 2.22). If the scheduler is implemented on a real system instead of tested on the TCPN model of the system, it represents the vector of pairs (τ_i, CPU_j) determining the allocation of tasks to CPUs.

3.3.1 Scheduler

The on-line scheduler (Alg. 1) leverages a Fixed Priority until Zero-Laxity (FPZL) algorithm ([25]) to allocate tasks to processors until the next scheduling event. The inputs are the outputs of the off-line stage, and the accumulated runtime execution times of each task since the previous scheduling event. A scheduling event occurs whenever a job reaches its zero laxity, a job completes, or an aperiodic task arrives. In the latter event, if the aperiodic task is accepted, the adaptive scheduler (AS) provides the cycles that the aperiodic task must run during the deadline interval $(x_{k,\tau_{a_i}})$, along with the adjusted CPU frequency F_n .

Priority Levels

Task priorities are updated according to their laxity Whenever an event occurs. Per-job laxities are calculated and ordered in a set $SL = \{l_i | l_i = sd_{k+1} - (F_n \times x_{i,k} - ex_{i,k}) - \zeta\}$. ζ is used to denote time, t is avoided because it is already employed for transitions.

Jobs reaching their zero-laxity time are given the maximum priority (= 1). Jobs being executed and with laxity different from zero receive priority equal to 2. The remaining jobs receive priority level equal to 3 (the lowest one). Thus, zero laxity tasks have the highest priority and must be executed immediately.

Execution of m tasks with the highest priority

In Alg. 1, steps 7, 10, m tasks are dispatched to the m CPUs. To reduce the number of migrations, tasks that are executed during two consecutive events are allocated to the same CPU. In a system simulated by a TCPN, this step means to compute Eqs. 2.22 according to $w_{i,j}^{alloc}$ (Eq. 3.5), in order to advance the simulation. In a real system, the set of m tasks are just passed to the dispatcher of the operating system.

Preemptions and migrations bound

Job preemption is one of the causes of run time overhead and large memory requirements in RT scheduling. In this section we prove that the number of preemptions and migrations incurred by Alg. 1 is bounded.

Proposition 3.3 *Assuming that the conditions of Proposition 3.2 hold, then the context switches at each scheduling interval ϕ_k caused by Alg. 1 are upper-bounded by*

$$2m + na \tag{3.18}$$

where na represents the number of active tasks in each frame ϕ_k .

Proof 3.3 . *Let X be the solution of the LPPs in Eq.(3.14), such that $x_{i,k} \in X_{HRT}$. During interval ϕ_k , a job from task τ_i must run for $x_{i,k}$ cycles at a given F_n clock frequency, but it can be the case that $x_{i,k} = 0$, hence the number of active tasks na_k*

in frame ϕ_k can be less than n . Recall that a task can only leave a processor under two conditions: it has finished its execution or it is preempted by another task that has reached zero laxity (ZL). On the best-case scenario, all tasks assigned to the processors are already in ZL, therefore $na_k = m$. Under this condition, each task is allocated to a processor, generating m context switches (CS) and another m CS at the end of their execution, thus producing a total of $2m$ CS. Also, it is possible that the m tasks that were first allocated finished their execution before the end of ϕ_k . Therefore, at most $na_k - m$ tasks will be allocated and incurring in $(na_k - m) + 2m$ CS. Finally, there are cases when some tasks reach ZL and others just finished their execution. Since the schedule is feasible there could only be m tasks reaching ZL at most, lest they miss their deadline. Hence the upper bound of CS is $(na_k - m) + 2m + m$, which is the same as $2m + na_k$.

Proposition 3.4 *Assuming that Proposition 3.2 holds, Alg. 1 causes at most $m - 1$ migrations of tasks.*

Proof 3.4 *Recall that a task τ_i is preempted only when another task has reached zero laxity, hence when τ_i is preempted it will migrate because its original processor will become unavailable. Therefore at most there will be $m - 1$ migrations.*

3.3.2 Aperiodic tasks and Adaptive Scheduler

Aperiodic tasks arrive asynchronously to the system. An *adaptive scheduler* (AS) determines if these tasks can be executed without compromising the HRT constraints of the periodic task set. If so, a new CPU clock frequency is computed allowing the execution of the aperiodic task. The computed frequency must be in the set $\mathcal{F} = \{F^* \dots F^+\}$, because from the off-line stage we know that these frequencies meet the thermal constraints. Moreover, the frequency must also be as low as possible, in order to guarantee a minimum power consumption while meeting the temporal constraints.

Upon an aperiodic arrival, the AS determines the current frame ϕ_k and the scheduling interval Γ at which τ_i^a has its deadline ($r_i^a + d_i^a$). Recall that the scheduler is periodic on the hyperperiod, such that the scheduling interval ϕ_Γ is the one that contains the element $g = (r_i^a + d_i^a) \bmod H$.

Then the AS computes the required CPU cycles C_u for all active tasks from the current scheduling interval j to Γ as:

$$C_u = \sum_{i=1}^{|X_k|} (x_{i,k} - ex_{i,k}) + \sum_{\gamma=k+1}^{\Gamma+g} \sum_{i=1}^{|X_\gamma|} x_{i,\gamma} \quad (3.19)$$

where $ex_{i,k}$ is the execution of active task i in CPU cycles, since the last scheduling event and $X_k = X_{HRT,k} \cup X_{ap,k}$ represents the set of CPU cycles that every active task must execute during the k -th scheduling interval. Hence, the first sum in Eq. (3.19) stands for the CPU cycles that the system still need to execute, while the second sum does the same for the subsequent scheduling intervals up to ϕ_Γ . With this information, the algorithm computes the maximum amount of CPU cycles C_{free} that the processors can spare when running at maximum frequency F^+ ,

$$C_{free} = (m \times d_i^a \times F^+) - C_u \quad (3.20)$$

where m is the number of CPUs and d_i^a is the aperiodic task relative deadline. If C_{free} is greater than the cycles demanded by the aperiodic task cc_i^a , the system is capable of serving τ_i^a , hence the AS determines the new operating frequency F_n as:

$$F_n = \min \left\{ f \in \mathcal{F} \mid f \geq \frac{C_u + cc_i^a}{m \times d_i^a} \right\} \quad (3.21)$$

Once F_n has been determined, the algorithm calculates the number of cycles $x_{\tau_i^a}^k$ from τ_i^a that will be executed in each scheduling interval from ϕ_j to ϕ_Γ , which is an iterative procedure:

$$cc_r = cc_i^a \quad (3.22)$$

$$x_{\tau_i^a}^k = \min \left\{ m(|\phi_k| - r_i^a)F_n - \sum_{i=1}^{|X^k|} (x_i^k - ex_i^k), cc_r \right\}$$

For $\gamma = k + 1$ **to** $(\Gamma + g)$

$$cc_r = cc_r - x_{\tau_i^a}^{\gamma \bmod \alpha - 1}$$

$$x_{\tau_i^a}^{\gamma \bmod \alpha} = \min \left\{ m(|\phi_{\gamma \bmod \alpha}|)F_n - \sum_{i=1}^{|X_{\gamma \bmod \alpha}|} x_i^{\gamma \bmod \alpha}, cc_r \right\}$$

Algorithm 2 Adaptive Scheduler (AS)

1: **Input**
 X^k – Task cycles per deadline interval; I_{SD}^k – Scheduling intervals;
 cc_i^a, d_i^a – Aperiodic tasks parameters;
 $\mathcal{F} = \{F^*, \dots, F^+\}$ – Clock frequencies;
 F_n – CPUs operating frequency;
 ex_i^k – Cycles executed for each task since the last scheduling event;

2: **Output**
 F_n updated operating frequency,
 $x_{\tau_i^a}^k$ cycles of the aperiodic task per scheduling interval;

3: **if** periodic task arrives **then**
4: Determine interval ϕ_Γ , where τ_i^a has its deadline
5: Compute required CPU cycles for active tasks during interval {Eq. (3.19)}
6: Calculate free CPU cycles C_{free} {Eq.(3.20)}
7: **if** $C_{free} \geq cc_i^a$ **then**
8: Accept task τ_i^a
9: Determine new F_n {Eq.(3.21)}
10: Calculate $x_{\tau_i^a}^k$ from current ϕ_k to ϕ_Γ {Eq. (3.22)}
11: **else**
12: Reject task
13: **end if**
14: **end if**
15: **if** an aperiodic task finishes **then**
16: Discard the CPU cycles associated to the aperiodic task
17: Recalculate the new frequency
18: **end if**

Complexity

The complexity of the on-line stage depends on two algorithms. The priority level and the computation of laxity in Alg. 1 is linear in the number of tasks. At most $n = |\mathcal{T}|$ tasks will end its execution x_i^k in the $k - th$ interval (there are at most n tasks). Also, n tasks will reach their zero laxity at most. If q aperiodic tasks arrive in the $k - th$ interval, then the nested while loop ends in $(n + n + q) \times (n + n)$ (*number of events* \times *number of operations*). Considering that the outer loop runs $\alpha = |I_{SD}|$ times, then the number of steps of this algorithm is polynomial in the order of tasks. Alg. 2 runs on the arrival of an aperiodic task and is polynomial in the order of tasks and independent of the number of CPUs. Thus the proposed algorithm is polynomial in the order of tasks.

3.4 Experimental Results

In this Section we simulate the behavior of *EETAMS*, a scheduler implemented according to the on-line and off-line stages previously described. First, we present

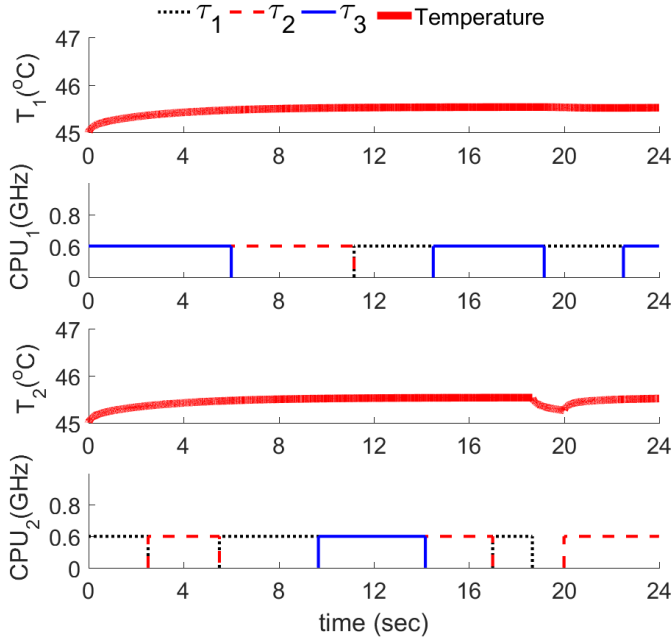


Figure 3.3.: Temperature evolution (upper plot) for the periodic schedule (lower plot) at CPU_1 (above) and CPU_2 (below). The maximum temperature produced by this schedule is $T_{CPU_{1,2}} = 45.3^\circ C$

an example to study the thermal behavior and real utilization considering the HRT task set. Then, we prove the ability of the scheduler to deal with the arrival of an aperiodic task while maximizing CPU utilization, controlling temperature and optimizing energy consumption.

Experimental environment

We assume a platform composed of two homogeneous Intel XScale silicon microprocessors mounted over a copper heat spreader for all the experiments. The isotropic thermal properties and dimensions of the materials are taken from [30]. The power model for the Intel XScale is based on [19]. The processor supports five operating frequency levels $F = \{0.15, 0.4, 0.6, 0.8, 1\}$ GHz, consuming $P_{CPU} = \{80, 170, 400, 900, 1600\}$ mWatt respectively. Thus, the power consumption function can be represented approximately as $P_{CPU} = 0.08 + 1.52 \cdot \phi^3$ Watt. The temperature of the surrounding air is constant and set to $45^\circ C$. In the experiments we assume cache memories and speculative mechanisms non-existent or turned off.

3.4.1 Temperature control and utilization

In this first experiment, we study the thermal control capabilities and real utilization achieved by *EETAMS*. We consider a set of sporadic tasks with implicit deadlines $\mathcal{T} = \{\tau_1, \tau_2, \tau_3\}$, where $\tau_1 = (1.5e9, 4)$, $\tau_2 = (3e9, 8)$, $\tau_3 = (5e9, 12)$, the hyperperiod is $H = 24$. The maximum operating temperature level is set to $T_{max_{1,2}} = 50^\circ$ C. First, the minimum frequency for the periodic task set is computed off-line according to Eq. (3.3), obtaining $F^{**} = 0.5833$ GHz. This frequency is raised to the nearest upper frequency actually available for the processor, which is $F^* = 0.6$ GHz. Eq. (3.6) provides the maximum clock frequency $f^+ = 1$ GHz, such that the *METARTS* problem has a solution. We assume that scheduling and context-switch overheads are included in task WCET. Then, solving the LPP from Eq. 3.14 for F^* yields the CPU cycles of each task to be executed per interval $(x_{i,k})$.

Fig. 3.3 provides the schedule and temperature evolution produced by the algorithm without considering aperiodic tasks. The off-line LPP and the FPZL on-line scheduler are work-conserving and yield a theoretical 100% CPU utilization. However, the fact that $F^* > F^{**}$ makes tasks allocated in CPU_2 to run faster in this simulation. This translates into the slack time that appears in the lowest plot of the figure (interval [18, 20]), which temporarily lowers the temperature as shown in the corresponding temperature graph, and decreases the theoretical utilization by about 8%.

3.4.2 Handling aperiodic tasks

We now show the behavior of the *EETAMS* scheduler upon the arrival of an aperiodic task while running the same HRT task set considered in the previous experiment. Fig. 3.4 depicts the outcome when an aperiodic task $\tau_1^a = (4000, 10)$ arrives at $\zeta = 2$, during the ϕ_1 time slice. τ_1^a has an absolute deadline at $\zeta = 12$. Since $C_{free} \geq cc_1^a$, the *AS* accepts the aperiodic task, and computes $F_n = 0.8$ GHz $\in \mathcal{F}$ as the frequency at which the processors must execute during interval [2, 12). The solid red line shows that temperature increases during this interval because of the execution of the tasks at frequency F_n , and then it decreases after $\zeta = 12$ because a new (lower) frequency has been calculated for the next interval. In both experiments, with and without the aperiodic tasks, CPU_1 achieves full utilization, whereas CPU_2 shows a slack (idle time) at about $\zeta = 18$, which translates into a temperature valley. As it happened in the previous experiment, this slack appears because the exact optimal frequency

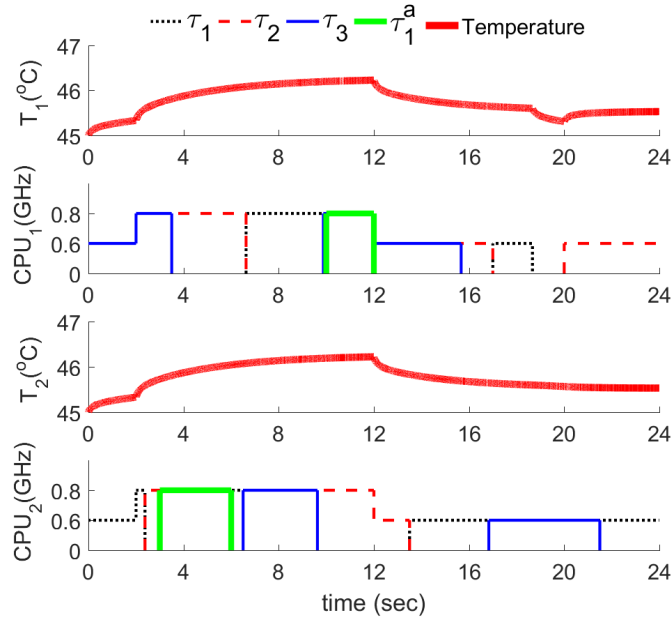


Figure 3.4.: Temperature evolution (upper plot) for the periodic schedule (lower plot) at CPU_1 (above) and CPU_2 (below) upon acceptance of the aperiodic task τ_1^a . The maximum temperature produced by this schedule is $T_{CPU_{1,2}} = 46.24^\circ C$

calculated in Eq. (3.3) is upper bounded by a frequency belonging to the discrete set of frequencies actually available in the microprocessor ($F_n \in \mathcal{F}$).

3.5 Conclusions

This work shows that the TCPN formalism is a suitable tool for the design of thermal-aware RT schedulers, particularly when complexity rises because of the confluence of thermal and time constraints plus aperiodic task management. Leveraging this formalism, we build a two-stage, energy-efficient thermal-aware scheduling system in which an HRT periodic task set executes at minimum clock frequency on a set of processors, with the ability to manage aperiodic tasks, optimizing power consumption, maximizing CPU utilization and honoring the HRT and thermal constraints in all cases. The TCPN models the activity of the tasks, their allocation to CPUs, heat generation and transfer, and how the latter affects to the overall system temperature. The thermal schedule feasibility is proved by an LPP that captures the RT and thermal restrictions as linear constraints. If there exists a feasible solution, then the LPP finds the maximum operating frequency f^+ to satisfy the thermal constraint.

The following chapter analyzes and improves a few points of the solution presented so far. Particularly, some feasible task sets are not schedulable under this algorithm, because no feasible solution exist for one of the ILPs (Prop. 3.2), on some cases. The reason behind this is that the relationship among different intervals is not considered, such that a past decision causes that the constraints of a following ILP are not satisfied. This phenomenon is exemplified below.

Example 3.3 Consider a task set $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, each task is defined as $\tau_1 = (2, 2)$, $\tau_2 = (5, 40)$, $\tau_3 = (3, 8)$ and $\tau_4 = (2, 4)$, and assume $F^* = 1$. To be schedule on $m = 2$ processors. The task set utilization is $U = 2$, therefore the task set is feasible. The hyperperiod is $H = lcm(2, 40, 8, 4) = 40$, and the set of deadlines is: $SD = \{0, 2, 4, 6, \dots, 40\}$, such that it generates $\alpha = 20$ intervals,

$$\phi_1 = [0, 2) \quad \phi_2 = [2, 4) \quad \phi_3 = [4, 6) \quad \phi_4 = [6, 8) \quad \dots \quad \phi_{20} = [38, 40)$$

all of them share the same duration $|\phi_k| = 2$.

Solving the ILP 3.14 for ϕ_1 , ϕ_2 and ϕ_3 , using the Simplex algorithm from MATLAB, yields the following solution:

		k=1	k=2	k=3
x_i^k	i=1	2	2	2
	i=2	0	2	0
	i=3	0	0	2
	i=4	2	0	0

Unfortunately, because of the selection of the previous solutions, which comply with the LPP 3.14, there is no feasible solution for the LPP of frame $\phi_4 = [6, 8)$. In which follows, we explain the problem generated for ignoring the relationship among consecutive frames. First, we present the constraints for the preceding frames, as long as the restrictions for ϕ_4 . Since every frame has the same duration, the upper bound for the variables is:

$$x_{i,k} \leq 2$$

and will be omitted in the following.

The restrictions for frame $\phi_1 = [0, 2)$ are:

$$\begin{aligned}
x_{1,1} &= 2 \\
x_{2,1} &\geq 0 \\
x_{3,1} &\geq 0 \\
x_{4,1} &\geq 0 \\
x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} &= 4
\end{aligned}$$

And the solution found was:

$$x_{i,1} = \{2, 0, 0, 2\} \forall i, i = 1, 2, 3, 4$$

For frame $\phi_2 = [2, 4)$, the constraints are as follows:

$$\begin{aligned}
x_{1,1} + x_{1,2} &= 4 \\
x_{2,1} + x_{2,2} &\geq 0 \\
x_{3,1} + x_{3,2} &\geq 0 \\
x_{4,1} + x_{4,2} &= 2 \\
x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} &= 4
\end{aligned}$$

Variables $x_{1,1}, x_{2,1}, x_{3,1}$ and $x_{4,1}$ have already been solved, so they are replaced with their calculated values, such that:

$$\begin{aligned}
x_{1,2} &= 2 \\
x_{2,2} &\geq 0 \\
x_{3,2} &\geq 0 \\
x_{4,2} &= 0 \\
x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} &= 4
\end{aligned}$$

Thus, a solution that complies with the former restrictions is:

$$x_{i,2} = \{2, 2, 0, 0\} \forall i, i = 1, 2, 3, 4$$

The constraints for frame $\phi_3 = [4, 6)$ are shown on the left, whilst the previous solutions are substituted on the right:

$$\begin{array}{rcl}
x_{1,1} + x_{1,2} + x_{1,3} & = & 6 \\
x_{2,1} + x_{2,2} + x_{2,3} & \geq & 0 \\
x_{3,1} + x_{3,2} + x_{3,3} & \geq & 1 \text{ substituting,} \\
x_{4,1} + x_{4,2} + x_{4,3} & \geq & 2 \\
x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} & = & 4
\end{array}
\qquad
\begin{array}{rcl}
x_{1,3} & = & 2 \\
x_{2,3} & \geq & 0 \\
x_{3,3} & \geq & 1 \\
x_{4,3} & \geq & 0 \\
x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} & = & 4
\end{array}$$

Solution:

$$x_{i,3} = \{2, 0, 2, 0\} \forall i, i = 1, 2, 3, 4$$

The constraints for frame $\phi_4 = [6, 8)$ are as follows:

$$\begin{array}{rcl}
x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} & = & 8 \\
x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} & \geq & 0 \\
x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} & = & 3 \text{ substituting,} \\
x_{4,1} + x_{4,2} + x_{4,3} + x_{4,4} & = & 4 \\
x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} & = & 4
\end{array}
\qquad
\begin{array}{rcl}
x_{1,4} & = & 2 \\
x_{2,4} & \geq & 0 \\
x_{3,4} & = & 1 \\
x_{4,4} & = & 2 \\
x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} & = & 4
\end{array}$$

The only solution that satisfies the first four constraints, such that the time requirements are met, is:

$$x_{i,4} = \{2, 0, 1, 2\} \forall i, i = 1, 2, 3, 4$$

However, the last constraint cannot be fulfilled ($x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} = 5 \neq 4$). This implies that the system capacity is over passed, i.e. the actual execution time on the available CPUs would be longer than the execution time required to meet all constraints.

This drawback is addressed on the following chapter. Results provided on this chapter were presented on [62].

RT Multiprocessor Scheduling based on continuous control

This chapter builds from the results obtained previously. The main premise is to endow the scheduler with a dynamical component such that it could react to unpredictable events such as CPU detentions or time drifting due to unexpected latencies while guaranteeing the HRT constraints.

The use of continuous control on RT scheduling was considered in [28]. Therein, authors tackled the thermal-aware RT global scheduling problem considering two stages, one stage to compute the workload, the other one to allocate tasks to CPUs. A fluid approach avoided the NP-completeness in the computation of the task workload per CPU, requiring the subsequent discretization algorithm. The fluid approach improved system's resilience to parameter variations, but led to a high number of context switches which hampered the feasibility of the scheduler. A different approach was used in [77], where authors proposed an energy-efficient scheduler for two heterogeneous processors. They formulate a general case as a non-linear integer programming problem to obtain a schedule, which yields algorithms with high computational complexity.

In Chapter 3 we proved that the task workload could be solved in polynomial time using a *DP-fair* ([36]) scheme, avoiding the discretization stage from [28] and reducing the number of context switches. Then, we resorted to a zero-laxity policy ([24]) to allocate tasks to CPUs on-line. Nevertheless, this approach solved the workload allocation through an Integer Linear Programming Problem (ILP) that ignored the relationship between consecutive *scheduling intervals* (frames), thus some feasible task sets were not schedulable under the algorithm, as shown on example 3.3.

Herein, two deficiencies of the previous solution are corrected: first the workload computation is performed up the hyperperiod, rather than per frame, thus enlarging the class of problems that could be solved, secondly the previous solution is endowed with a controller to add robustness to the scheduler by rejecting small disturbances

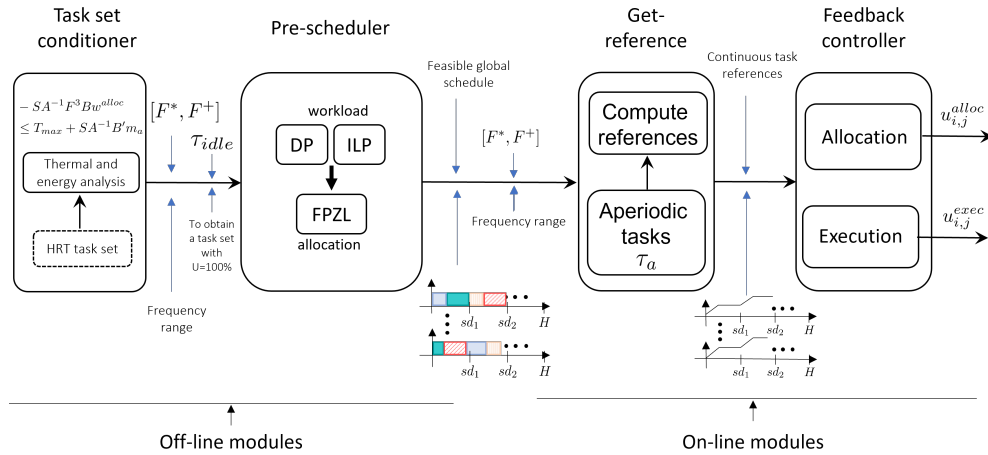


Figure 4.1.: Allocation and Execution Control Scheduler (AIECS) Overview: the task set conditioner module is borrowed from the previous chapter, but a brief summary is in section 4.3, the Pre-scheduler module is defined on section 4.4. In Get reference, a schedule is translated into continuous references, sec. 4.5, It also includes the aperiodic manager (sec. 4.6.3) because it alters the schedule. Finally, the Feedback controller is presented on section 4.6

such as CPU detentions, time drifting due to unexpected latencies or parametric issues obviated in the model.

The proposed scheme consists of three main components. The first component is an off-line scheduler, similar to the scheduler presented on Chapter 3, but entirely offline. The second component implements an on-line controller to add robustness to the offline scheduler. Finally, the third component adds the capability of scheduling SRT aperiodic tasks.

4.1 Problem definition

The same multiprocessor system and task model as in Chapter 3 is used herein, but the increment on frequency values is assumed continuous.

Multiprocessor system The multiprocessor system is composed by m processors, where migration is permitted. $\mathcal{P} = \{CPU_1, \dots, CPU_m\}$ is the set of the m identical processors with an homogeneous clock frequency $F \in \mathcal{F} = [f_1, \dots, f_{max}]$, i.e a change on the operating frequency is reflected on all m processors. Furthermore, each CPU is subjected to a thermal bound T_{max} .

Task model The set of HRT tasks is denoted by $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$. Each task is independent, fully preemptive and is identified by the 3-tuple $\tau_i = (cc_i, d_i, \omega_i)$, where cc_i is the worst-case execution time (WCET) in cycles, ω_i is the task period, and d_i is the relative implicit deadline ($d_i = \omega_i$) ([6]).

The arrival of asynchronous, SRT aperiodic tasks is also considered. Each aperiodic task τ_i^a is defined as a 3-tuple (cc_i^a, d_i^a, r_i^a) in which cc_i^a (required CPU cycles) and d_i^a (deadline) are known at task arrival time, and the arrival time r_i^a is unknown. Let $\mathcal{T}_a = \{\tau_1^a, \dots, \tau_p^a\}$ be a set of the p independent aperiodic tasks.

Problem 4.1 Control RT Scheduler (CRTS). *Given the system of multiprocessors in \mathcal{P} and the HRT tasks \mathcal{T} , the CRTS problem consists in designing a control law that tracks a set of feasible references. Additionally, the controller should execute or reject aperiodic tasks from \mathcal{T}_a upon arrival, subject to the temporal and thermal constraints from the HRT tasks.*

Such that the feasible references hold the characteristics from Definition 4.1

Definition 4.1 *A feasible reference is a function that represent a schedule that successfully allocate the tasks in \mathcal{T} to the m identical processors within the hyperperiod H , such that: the deadlines for \mathcal{T} are guaranteed, the temperatures of the processors are kept below a bound T_{max} and the consumed energy is minimum.*

4.2 Overview of the AIECS scheduling system

This section describes AIECS (*Allocation and Execution Control Scheduler*) as a solution to the CRTS problem. AIECS consists of three components. The first component yields, entirely off-line, a correct schedule for a HRT task set that minimizes energy and ensures a processor temperature below a thermal bound T_{max} , similar to the one presented in the previous chapter. The second component is an on-line stage focused on adding robustness by rejecting small disturbances such as CPU detentions, time drifting due to unexpected latencies or parametric issues obviated in the model. The first controller acts upon the flow of transition $t_{i,j}^{alloc}$ in the TCPN model, to ensure the correct allocation of tasks to processors according to the task execution paths calculated off-line. A second controller modifies the flow of transition $t_{i,j}^{exec}$ to ensure the timed execution of the task execution paths by

adjusting the CPU frequency. This controller warrants that the off-line schedule is met despite errors on the modeling section or unexpected (but bounded) overheads at run time. Finally, the third component provides the ability of managing SRT aperiodic tasks by means of the Online Aperiodic Manager (OnAM). The following sections extend on each component of *AIECS*.

4.3 Task set conditioner

This module is borrowed from Chapter 3. It yields a set of operating frequencies computed from the analysis performed on the TCPN global model, from sec. 2.3.2, and the energy equation 2.21.

$$\mathcal{F} = \{f \in \mathcal{F} | F^* \leq f \leq F^+\}, \quad (4.1)$$

We leverage DVFS to vary processor frequency by selecting one from the finite set of preset values \mathcal{F} . F^* minimizes the energy consumption and ensures the temporal requirements of the HRT tasks, and F^+ is the maximum permissible frequency at maximum utilization. This frequency range ensures the accomplishment of the thermal bound T_{max} .

4.4 Pre-scheduler

This section details scheduler obtained off-line, it resembles the scheduler from chapter 3, but it solves the drawbacks of the previous scheme.

First, it will be described the workload assignment of tasks per time interval, then the allocation of this workload to the processors. The output from this stage is an static offline schedule for the HRT tasks \mathcal{T} , i.e a cyclic executive.

4.4.1 Workload Assignment: ILP definition

The workload computation is performed by solving an ILP defined for the task set execution within the hyperperiod, in contrast with last chapter, where such calculations were done per time interval (frame) and the solution was carried on to

the next interval. The reason for imposing an *integral* solution is the same as before, the workload is expressed on cycles rather than time, and a cycle is an indivisible unit. It will be proved that the ILP formulation still meets the *unimodularity* property, regardless of the changes introduced. Therefore, the solution of the optimization problem through a simplex method always ensures *integral* solutions.

Given a periodic task system comprising n tasks to be executed on m -processors, the following linear programming problem poses the necessary constraints to build a feasible schedule such that no deadline is missed.

The ILP solves the share of cycle execution from task τ_i , such that no deadlines are missed, as is described for the hyperperiod H , which represents the smallest time window where the schedule would be repeated over time, for this reason by analysing the set of jobs within H , we can study the behaviour of the system.

Definition 4.2 Let $\mathcal{J} = \{j_1, \dots, j_x\}$ denote all the jobs generated by task set \mathcal{T} that have their arrival times and deadlines within H , such that each τ_i generates $\frac{H}{\omega_i}$ jobs.

Therefore, the number of jobs generated by task set \mathcal{T} within the hyperperiod H , i.e. the cardinality of \mathcal{J} , is:

$$H \sum_{i=1}^n \frac{1}{\omega_i} \quad (4.2)$$

Definition 4.3 Let SD be the ordered set of deadlines of all jobs $j \in \mathcal{J}$ within H , union $\{0\}$. Such that $sd_0 = 0$, $sd_{k-1}, sd_k \in SD$, $sd_{k-1} < sd_k$, where k is an integer value greater than zero. Then, the frames (scheduling intervals) are defined as $\phi_k = [sd_{k-1}, sd_k) \forall sd_k \in SD$.

Within the hyperperiod H , every $\tau_i \in \mathcal{T}$ spans $\frac{H}{\omega_i}$ jobs, therefore $|SD| < |\mathcal{J}|$. And, the number of scheduling intervals (α), is always one less than the cardinality of the set of deadlines, i.e.,

$$\alpha = |SD| - 1 \quad (4.3)$$

Definition 4.4 The absolute laxity cc'_i of τ_i in cycles is the maximum time (in cycles) that τ_i can remain idle before compromising its deadline, when executing at speed (frequency) f . This is, $cc'_i = (d_i \cdot f) - cc_i$, where d_i is the deadline/period and cc_i the WCET of τ_i .

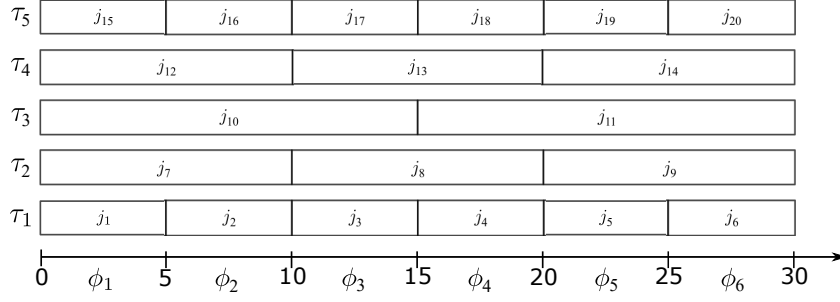


Figure 4.2.: Jobs generated by the task set of Example 4.1

Example 4.1 Consider a task set $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$, each task is defined as $\tau_1 = (3, 5)$, $\tau_2 = (6, 10)$, $\tau_3 = (9, 15)$, $\tau_4 = (6, 10)$ and $\tau_5 = (3, 5)$. Therefore, the hyperperiod is $H = lcm(5, 10, 15) = 30$, the task set spans 20 jobs $\mathcal{J} = \{j_1, j_2, \dots, j_{20}\}$. Each task generates a different number of jobs (Fig. 4.2).

The set of deadlines is:

$$SD = \{0, 5, 10, 15, 20, 25, 30\},$$

where $|SD| = 7$, such that it generates $\alpha = 6$ frames:

$$\phi_1 = [0, 5) \quad \phi_2 = [5, 10) \quad \phi_3 = [10, 15) \quad \phi_4 = [15, 20) \quad \phi_5 = [20, 25) \quad \phi_6 = [25, 30)$$

Variables

The variable $x_{i,k}$ denotes the share of cycle execution from task τ_i that is scheduled upon the k -th interval/frame ϕ_k . That is, variables $x_{i,k}$ take integer values such that $x_{i,k} \leq cc_i$. The former under the assumption that an *execution cycle* is an indivisible unit.

The index i takes on each integer value in the range $[1, n]$, where n is the number of tasks in \mathcal{T} . For each i , the index k iterates on the number of frames ϕ_k , i.e., on range $[1, \alpha]$. In total, the number of $x_{i,k}$ variables is equal to $(n \cdot \alpha)$.

Remark 4.1 When the task set utilization is $U < m$, it is necessary to add an idle or *dummy task*, to ensure a system at maximum capacity. Such that,

$$u_{idle} = m - \sum_{i=1}^n \frac{cc_i}{f \cdot \omega_i}$$

where f is the system frequency and $\tau_{idle} = (u_{idle} \cdot H, H)$.

Remark 4.2 Despite the use of *task* to refer to the piece of execution upon a processor, recall that it is the job who actually is executed (an instance of the task), nevertheless using the term *task* and its numbering helps to simplify the definition of the linear programming variables and constraints.

Example 4.2 For the task set from Example 3.1, this example shows the variables for each task and its correspondence between its jobs.

The variables for task τ_1 are:

$$\underbrace{x_{1,1}}^{j_1} \quad \underbrace{x_{1,2}}^{j_2} \quad \underbrace{x_{1,3}}^{j_3} \quad \underbrace{x_{1,4}}^{j_4} \quad \underbrace{x_{1,5}}^{j_5} \quad \underbrace{x_{1,6}}^{j_6}$$

Those corresponding to τ_2 are:

$$\underbrace{x_{2,1}, x_{2,2}}^{j_7} \quad \underbrace{x_{2,3}, x_{2,4}}^{j_8} \quad \underbrace{x_{2,5}, x_{2,6}}^{j_9}$$

The ones for τ_3 are:

$$\underbrace{x_{3,1}, x_{3,2}, x_{3,3}}^{j_{10}} \quad \underbrace{x_{3,4}, x_{3,5}, x_{3,6}}^{j_{11}}$$

Those spanned for τ_4 are:

$$\underbrace{x_{4,1}, x_{4,2}}^{j_{12}} \quad \underbrace{x_{4,3}, x_{4,4}}^{j_{13}} \quad \underbrace{x_{4,5}, x_{4,6}}^{j_{14}}$$

Finally, the variables for τ_5 are:

$$\underbrace{x_{5,1}}^{j_{15}} \quad \underbrace{x_{5,2}}^{j_{16}} \quad \underbrace{x_{5,3}}^{j_{17}} \quad \underbrace{x_{5,4}}^{j_{18}} \quad \underbrace{x_{5,5}}^{j_{19}} \quad \underbrace{x_{5,6}}^{j_{20}}$$

Constraints

The time is multiplied by the frequency to represent the available execution capacity over a time window, since variables $x_{i,j}$ represent execution cycles.

The following constraints ensure the completion of task execution before its deadline:

1. The *Maximum utilization constraint* (M.c) ensures that the system utilization per frame ϕ_k is 100%. That is the summation of every share of execution per task on each frame has to add up to the available system capacity,

$$\sum_{i=1}^n x_{i,k} = m \cdot |\phi_k| \cdot f \quad (4.4)$$

There are $\alpha - 1$ of such constraints, one per frame ϕ_k but the last. Because the restriction on the frame where the hyperperiod occurs is linearly dependant, because every task will have a deadline on the last frame, this constraint will impose the completion of every job, along with the following execution constraint.

2. The following constraint, *Execution constraint* (E.c), ensures that each job completes its execution before its deadline. This is defined per each frame and task. It can take two forms, depending on whether the frame is a deadline for the task or not. Since the frames were defined by the set of deadlines, at least one task will have its own deadline on ϕ_k , but not all. To know if τ_i has its deadline on $\phi_k = [sd_{k-1}, sd_k)$, it suffices to write $sd_k = qd_i + r_{i,k}$ and check if the residual $r_{i,k}$ is zero.

$$e = \begin{cases} \sum_{j=\gamma}^k x_{i,j} = cc_i & \text{if } r_{i,k} = 0 \\ \sum_{j=\gamma}^k x_{i,j} \geq (r_{i,k} \cdot f) - cc'_i & \text{if } r_{i,k} \neq 0 \end{cases} \quad (4.5)$$

where γ is the current job activation's frame

the counter γ helps to keep track of the frame where the current job had its activation, thus avoiding to compute the cumulative execution as in the LPP (3.11) in previous Chapter (Sec. 3.2.2). The residual $r_{i,k}$ is given in time units, but cc'_i in cycles. For this reason, the residual is multiplied by the frequency f .

3. Task parallelism is not allowed, therefore each task cannot execute more cycles than those available in one frame:

$$x_{i,k} \leq |\phi_k| \cdot f \quad (4.6)$$

This constraint is referred as *Sequential constraint* (S.c), and there are $(n \cdot \alpha)$.

4. Given that $x_{i,j}$ variables are cycles from τ_i they must all be assigned values greater or equal to zero $x_{i,j} \geq 0 \forall i \forall k$, but also comply with the execution constraint (4.6) for $r_{i,k} \neq 0$, therefore the second case of Eq. 4.6 results in:

$$\sum_{j=\gamma}^k x_{i,j} \geq \max \{0, (r_{i,k} \cdot f) - cc'_i\} \quad \text{if } r_{i,k} \neq 0 \quad (4.7)$$

Thus, the total amount of constraints in this LPP is $\alpha(2n + 1) - 1$.

Example 4.3 For the task set from Example 4.1. The constraints for $\phi_5 = [20, 25)$ yield,

$$x_{1,5} + x_{2,5} + x_{3,5} + x_{4,5} + x_{5,5} = m \times 5 \cdot f$$

Task τ_1 and τ_5 have a deadline on ϕ_5 , therefore,

$$x_{1,5} = cc_1 \quad x_{5,5} = cc_5$$

The other three tasks do not have a deadline on ϕ_5 , their residuals $r_{i,5}$ are: $r_{2,5} = 5$, $r_{3,5} = 10$ and $r_{4,5} = 5$. Thus,

$$\begin{aligned} x_{2,5} &\geq \max \{0, (5 \cdot f) - cc'_2\} \\ x_{3,4} + x_{3,5} &\geq \max \{0, (10 \cdot f) - cc'_3\} \\ x_{4,5} &\geq \max \{0, (5 \cdot f) - cc'_4\} \end{aligned}$$

Finally, the upper bound for $x_{i,5}$ variables:

$$x_{1,5} \leq 5 \cdot f, \quad x_{2,5} \leq 5 \cdot f, \quad x_{3,5} \leq 5 \cdot f, \quad x_{4,5} \leq 5 \cdot f, \quad x_{5,5} \leq 5 \cdot f.$$

Objective function

The objective function minimizes,

$$\sum_{i=1}^n \sum_{k=1}^{\alpha} x_{i,k} \quad (4.8)$$

which is the summation of all the variables. Even though every solution holds the same value for the objective function, it was chosen to ensure a solution. A different objective function can be used, for instance to maximize only one $x_{i,k}$ per job and reduce task partition, the downside is that it will vary according to the problem.

Proposition 4.1 *Given a task set \mathcal{T} , where task utilization at frequency f is equal to the number of processors, the solution X of the LPP (4.9) is always a vector of integer numbers $x_{i,k}$, and if each task τ_i is executed for exactly $x_{i,k}$ cycles during the k -th interval, then an optimal schedule is obtained.*

$$\begin{aligned}
& \max \quad \sum_{i=1}^n \sum_{k=1}^{\alpha} x_{i,k} \\
& s.t \quad \forall k \quad \sum_{i=1}^n x_{i,k} = m \cdot |\phi_k| \cdot f \quad \text{M.c} \\
& \quad \forall i, k \quad \begin{cases} \sum_{j=\gamma}^k x_{i,j} = cc_i & \text{if } r_{i,k} = 0 \\ \sum_{j=\gamma}^k x_{i,j} \geq \max\{0, (r_{i,k} \cdot f) - cc'_i\} & \text{if } r_{i,k} \neq 0 \end{cases} \quad \text{E.c} \quad (4.9) \\
& \quad \text{where } \gamma \text{ is the current job activation's frame, and } r_{i,k} \text{ is the residual from } sd_k = q\omega_i + r_{i,k} \\
& \quad \forall i, k \quad x_{i,k} \leq |\phi_k| \cdot f \quad \text{S.c}
\end{aligned}$$

Proof 4.1 *The execution constraints (eq. 4.6) from LPP (4.9) ensure that every task meets its deadline. Hence, if it the LPP has a solution, then the workload allocation leads to a feasible schedule. Furthermore, by Theorem 4 from [36], the schedule is also optimal.*

To prove that the solution for the LPP (4.9) is always integer, we will show that the restriction matrix is unimodular. Let $M \cdot y = b$ be the constraints from LPP (4.9), where $y = [x \ h]$, x is the solution vector from the LPP, h represents the vector of slack variables and M is the restriction matrix. By construction, M has the form:

$$M = \left[\begin{array}{c|c} A & \emptyset \\ \hline B & I_h \\ I_{sc} & \end{array} \right] \quad (4.10)$$

$$\begin{bmatrix} A \\ B \end{bmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \hline 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 \end{pmatrix}$$

Figure 4.3.: Form of submatrix $[A^T \ B^T]^T$

A represents the equality constraints, B the execution constraints (Eq. 4.6) that resulted on inequalities, I_{sc} the sequential constraints (one for each $x_{i,k}$) and I_h corresponds to the slack variables. All constraints are linearly independent among them, by construction, hence M is full row rank with $rank(M) = \alpha(2n + 1) - 1$, where α is the number of scheduling intervals.

It is well known, from the properties of totally unimodular matrices (TUM), that the property of total unimodularity (TU) holds under the adjoining of unit vectors [80]. Thus, M is TUM if submatrix $[A^T \ B^T]^T$ is TUM.

Submatrix $[A^T \ B^T]^T$ has the particular form showed in Fig. 4.3. It contains a special structure of -1 s in stair. If we remove all unit vectors from $[A^T \ B^T]^T$ but those in the stairs the TU property still holds. Let this new matrix be M' . The TU property is also preserved under elementary row operations with no scaling, then each row in the stair structures from M' can be transformed to unit vectors. Thus it is sufficient to prove that A is TUM.

We claim that A is TUM because it satisfies Theorem 3 from [41], with a row partition (T_1 and T_2) such that rows associated with the Maximum utilization constraints (M.c) are elements of T_1 and the restrictions corresponding to the Execution constraints (E.c) are in T_2 .

Zero Laxity policy

In this section we discuss how to construct a preemptive schedule based on the solution of the linear programming problem 4.9. The workload X previously computed determines that task τ_i must be allocated $x_{i,k}$ cycles at frequency F^*

during the interval ϕ_k to satisfy the HRT and thermal constraints. This implies that the frequency can be throttled up to F^+ without violating the thermal restriction. However, the actual allocation of tasks to processors requires a scheduling algorithm. In this work, we leverage a FPZL policy as posed in Algorithm 3, following the results from Prop. 4.1.

Algorithm 3 ZLH policy

```

1: Input  $I_{SD}^k$  – Scheduling intervals;  $X^k$  – CPU cycles per interval of each task;  $ex_i^k$  – Current
   execution  $P$  cycles in interval  $t_0$  – Initial time  $t_f$  – Final time
2: Output A feasible schedule;
    $k = 0$ ,
3: for  $t = t_0$  to  $t_f$  do
4:   Compute the laxity of every active task
5:   if reach  $I_{SD}^{k+1}$  then
6:      $k=k+1$ ;
7:     Compute task priorities as: Jobs with Zero laxity get higher priority, followed by jobs that are
       being executed
8:     Execute the  $m$  tasks with higher priority
9:   else if reach a zero laxity then
10:    Compute task priorities
11:    Execute the  $m$  tasks with higher priority
12:   end if
13: end for

```

Example 4.4 Consider the same task set from Example 4.1, $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$, each task was defined as $\tau_1 = (3, 5)$, $\tau_2 = (6, 10)$, $\tau_3 = (9, 15)$, $\tau_4 = (6, 10)$ and $\tau_5 = (3, 5)$. Assume the frequency is $F^* = 1$, therefore, $U = 3$, so it is feasible on $m = 3$ processors. The solution from LPP 4.9 (per task τ_i and per ϕ_k) is

		k=1	k=2	k=3	k=4	k=5	k=6
$x_{i,k}$	i=1	3	3	3	3	3	3
	i=2	3	3	5	1	5	1
	i=3	5	1	3	3	1	5
	i=4	1	5	1	5	3	3
	i=5	3	3	3	3	3	3

Applying the ZL policy (Alg. 3) up to the hyperperiod, we find the target schedule in Fig. 4.4.

So far, we have found an off-line schedule that can be implemented on a real system, in the absence of disturbances.

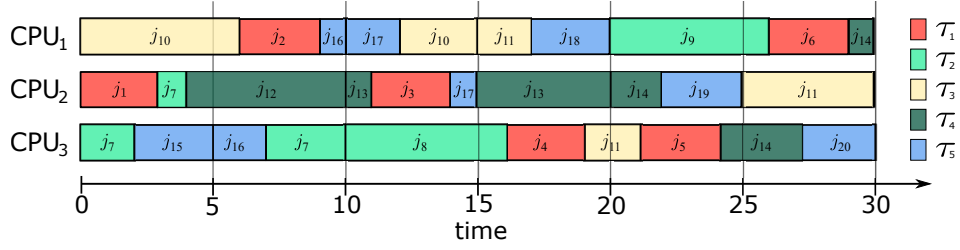


Figure 4.4.: Schedule of task set $\mathcal{T} = \{(3, 5), (6, 10), (9, 15), (6, 10), (3, 5)\}$ on 3 processors. Task τ_1 (red blocks) generates jobs: j_1 to j_6 , τ_2 (light green) produces jobs j_7 to j_9 , τ_3 (yellow) creates jobs j_{10} and j_7 , task τ_4 (dark green) spawns jobs j_{12} to j_{14} , and finally, τ_5 (blue) generates jobs j_{15} to j_{19}

4.5 Compute references

The off-line schedule obtained from the previous module is now translated into a set of accumulative execution functions. These functions represent the accumulated execution of every task in each processor in cycles, i.e there are $n \cdot m$ accumulative functions. These functions could be used as reference for the TCPN model to track a feasible schedule.

In order to define these accumulative execution functions, we first need to specify when a task is being executed on a specific processor. Despite the use of *task* to refer to the piece of execution upon a processor, recall that it is the *job* (an instance of the task) which is actually executed. Using the term *task* and its numbering helps simplify the notation.

Definition 4.5 Let $W_{i,j}(\zeta)$ be the activation function of τ_i in CPU_j , such that

$$W_{i,j}(\zeta) = \begin{cases} 1 & \text{if } \tau_i \text{ is executed on } CPU_j \\ 0 & \text{otherwise} \end{cases} \quad (4.11)$$

where ζ represents time.

Remark 4.3 Note that whenever a job from τ_i in CPU_j is preempted (allocated), function $W_{i,j}$ will show a break point, i.e a drastic change from 1 to zero (0 to 1). Therefore, when a *context switch* occurs on any of the processors, at least one function $W_{i,j}$ has a break point. Thus, every $W_{i,j}$ is constant in between every pair of context switches.

Definition 4.6 Let $S = \{\delta_0, \delta_1, \dots, \delta_k, \dots, \delta_h\}$ be the set with all the time stamps δ_k from a given cyclic executive when a context switch occurred, where $\delta_0 = 0$ and δ_h is the last context switch.

Definition 4.7 The execution interval

$$\Delta_k = (\delta_{k-1}, \delta_k], \quad (4.12)$$

is the time interval between consecutive context switches δ_{k-1} and δ_k , where $k = 1, \dots, h$.

Now the accumulative execution functions are formally defined on Def. 4.8.

Definition 4.8 Let $R_{i,j}(\zeta)$ be the accumulative execution function of task τ_i in processor j , at time ζ . $R_{i,j}(\zeta)$ is a continuous function such that

$$\dot{R}_{i,j}(\zeta) = f \cdot W_{i,j}(\zeta) \quad (4.13)$$

where ζ is the time, f the operating frequency of CPU_j , $i = 1, 2, \dots, |\mathcal{T}|$, $j = 1, 2, \dots, |\mathcal{P}|$ and $W_{i,j}(\zeta)$ is the activation function. Thus, integrating Eq. 4.13 for $\zeta \in \Delta_k$, such that $W_{i,j}(\zeta)$ is constant, yields

$$\begin{aligned} \int_{R_{i,j}(\delta_{k-1})}^{R_{i,j}(\zeta)} dQ &= \int_{\delta_{k-1}}^{\zeta} f \cdot W_{i,j}(\tau) d\tau \\ R_{i,j}(\zeta) - R_{i,j}(\delta_{k-1}) &= f \cdot W_{i,j}(\zeta) \int_{\delta_{k-1}}^{\zeta} d\tau \end{aligned}$$

Thus,

$$R_{i,j}(\zeta) = R_{i,j}(\delta_{k-1}) + f \cdot W_{i,j}(\zeta) \cdot (\zeta - \delta_{k-1}) \quad \forall \zeta \in \Delta_k \quad (4.14)$$

Example 4.5 Consider the task set from example 4.1, and its schedule from Fig. 4.4. Therein, task τ_1 executes in CPU_1 during the time intervals $[6, 9)$ and $[26, 29)$, in CPU_2 at $[0, 3)$ and $[11, 14)$, and in CPU_3 during $[16, 19)$ and $[21, 24)$. The execution of task τ_1 is divided into the three processors, resulting in one accumulative execution function per processor, this is $R_{1,1}$, $R_{1,2}$ and $R_{1,3}$.

$$R_{1,1}(\zeta) = \begin{cases} 0 & 0 \leq \zeta \leq 6 \\ (\zeta - 6) & 6 < \zeta \leq 9 \\ 3 & 9 < \zeta \leq 26 \\ 3 + (\zeta - 26) & 26 < \zeta \leq 29 \\ 6 & 29 < \zeta \leq 30 \end{cases} \quad R_{1,2}(\zeta) = \begin{cases} \zeta & 0 \leq \zeta \leq 3 \\ 3 & 3 < \zeta \leq 11 \\ 3 + (\zeta - 11) & 11 < \zeta \leq 14 \\ 6 & 14 < \zeta \leq 30 \end{cases} \quad (4.15)$$

$$R_{1,3}(\zeta) = \begin{cases} 0 & 0 < \zeta \leq 16 \\ (\zeta - 16) & 16 < \zeta \leq 19 \\ 3 & 19 < \zeta \leq 21 \\ 3 + (\zeta - 21) & 21 \leq \zeta \leq 24 \\ 6 & 24 < \zeta \leq 30 \end{cases} \quad (4.16)$$

Functions $R_{1,1}$, $R_{1,2}$, $R_{1,3}$ are plotted in Fig. 4.5.

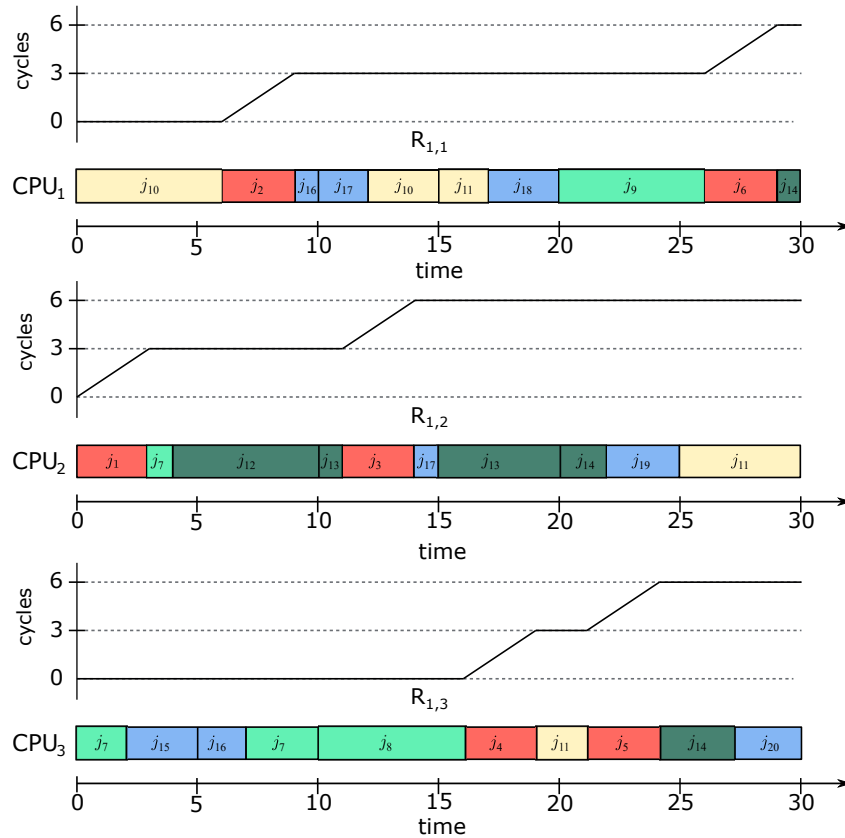


Figure 4.5.: Execution accumulative functions for τ_1 on each processor, $R_{1,1}$ and $R_{1,2}$ are defined on Eq. (4.15) and $R_{1,3}$ on Eq. (4.16)

Solving Eq. (4.13) at the end of interval Δ_k provides the amount of cycles that τ_i must complete within Δ_k ,

$$R_{i,j}(\delta_k) = R_{i,j}(\delta_{k-1}) + f \cdot [W_{i,j}(\delta_{k-1})](\delta_k - \delta_{k-1}) \quad (4.17)$$

Consecutively, $R_{i,k}(\delta_k) - R_{i,k}(\delta_{k-1})$ represents the number of CPU cycles that τ_i must execute in CPU_j within Δ_k before being preempted.

4.6 On-line controller ALECS

The allocation and execution of the HRT tasks is modeled in Eq. (2.22a)-(2.22c). Specifically, it is determined by the flow of transitions $t_{i,j}^{alloc}$ for the allocation and $t_{i,j}^{exec}$ for the execution.

The firing of transition $t_{i,j}^{alloc}$ represents that task τ_i is allocated to CPU_j , and the marking of place $m_{i,j}^{busy}$ holds the number of cycles from τ_i allocated to CPU_j . Accordingly, the marking $m_{i,j}^{exec}$ at $p_{i,j}^{exec}$ represents the accumulated execution of task τ_i in CPU_j . The controllers for allocation and execution read these measures either from the TCPN equations, when simulating a TCPN model of the system, or from actual counters, when running in a real system.

Figure 4.6 shows the Timed Continuous Petri Net (TCPN) for the CPU module, each transition $t_{i,j}^{alloc}$ has two places on its preset: p_j^{idle} and p_i^{cc} . The former represents the idle state of a processor, while the latter relates to the amount of cycles from τ_i ready to be allocated, but is omitted from this figure. As stated previously, the model uses an infinite server semantics and its transition firing policy,

$$f[t_j] = \lambda[t_j] \cdot \min_{p \in \bullet t_j} \left\{ \frac{m[p]}{Pre[p, t_j]} \right\},$$

has the min operator. In this work we assume that p_j^{idle} will always constraint the flow of transitions $t_{i,j}^{alloc}$ by selecting an adequate value for η , such that the TCPN model evolves in only one configuration.

Also, from Figure 4.6 it becomes apparent that transitions $t_{i,j}^{alloc}$ can be fired simultaneously, nevertheless this is an undesired behaviour that disappears when the following allocation control is imposed.

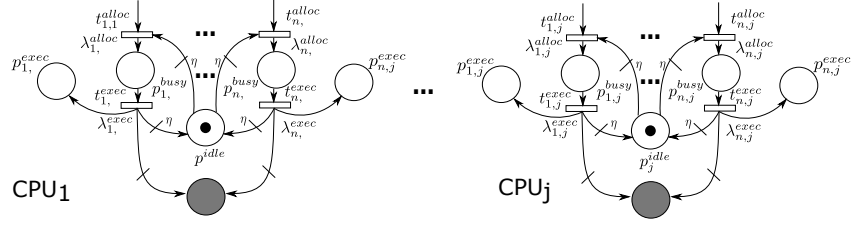


Figure 4.6.: TCPN module for CPU_1 to CPU_j . The gray places

The allocation control can be thought of as the scheduler, it takes the decision of which task should be allocated on each processor. On the other hand, the execution control is a flow controller that can increase or decrease the speed of execution. Thus, leading to the accomplishment of the offline schedule.

4.6.1 Allocation control

At each *executing interval* $\Delta_k = (\delta_{k-1}, \delta_k]$ places $p_{i,j}^{busy}$ and $p_{i,j}^{exec}$ should hold the required marking $R_{i,j}(\delta_k)$ (Eq. 4.17).

Let define the *allocation error vector* $\mathcal{E}^{alloc}(\zeta) = [\mathcal{E}_{1,1}^{alloc}, \dots, \mathcal{E}_{n,1}^{alloc}, \dots, \mathcal{E}_{n,m}^{alloc}]^T$, where each $\mathcal{E}_{i,j}^{alloc}(\zeta)$ is computed as:

$$\mathcal{E}_{i,j}^{alloc}(\zeta) = m_{i,j}^{exec}(\zeta) + m_{i,j}^{busy}(\zeta) - R_{i,j}(\delta_k) \quad (4.18)$$

where $\delta_{k-1} \leq \zeta < \delta_k$. The change in time of the marking of a place in a TCPN, is computed as the input flow minus its output flow. Then, taking the time derivative of Eq. 4.18, the dynamics of the error is given by:

$$\dot{\mathcal{E}}_{i,j}^{alloc}(\zeta) = \dot{m}_{i,j}^{exec} + \dot{m}_{i,j}^{busy} \quad (4.19)$$

$$= \lambda_{i,j}^{exec} m_{i,j}^{busy} + \frac{\lambda_{i,j}^{alloc}}{\eta} m_{i,j}^{idle} - \lambda_{i,j}^{exec} m_{i,j}^{busy} - u_{i,j}^{alloc} \quad (4.20)$$

$$= f_{i,j}^{alloc} - u_{i,j}^{alloc} = w_{i,j}^{alloc} \quad (4.21)$$

where $w_{i,j}^{alloc}$ is the controlled flow through transition $t_{i,j}^{alloc}$ and $f_{i,j}^{alloc}$ is

$$f_{i,j}^{alloc} = \frac{\lambda_{i,j}^{alloc}}{\eta} m_{i,j}^{idle}. \quad (4.22)$$

In a TCPN, the flow of transitions can only be decreased. For this reason, an ON/OFF control is proposed, such that control $u_{i,j}^{alloc}$ cancels $f_{i,j}^{alloc}$ when necessary. Whenever

the initial condition of $\mathcal{E}_{i,j}^{alloc}$ is zero the error itself can always be driven back to zero. This idea is formalized on the following proposition.

Proposition 4.2 Let $u_{i,j}^{alloc}$ be an ON/OFF control for system (4.21), such that

$$u_{i,j}^{alloc} = \begin{cases} \frac{\lambda_{i,j}^{alloc}}{\eta} m_{i,j}^{idle} & \text{if } \mathcal{E}_{i,j}^{alloc}(\zeta) \geq 0 \\ 0 & \text{if } \mathcal{E}_{i,j}^{alloc}(\zeta) < 0 \end{cases} \quad (4.23)$$

Then system (4.21) is stable and each $\mathcal{E}_{i,j}^{alloc}$ remains bounded for all $\delta_{k-1} < \zeta \leq \delta_k$.

Proof 4.2 To prove the stability of all allocation errors, we assume that place m_j^{idle} constrains transitions $t_{i,j}^{alloc}$, hence we write the dynamic of each allocation error as:

$$\dot{\mathcal{E}}_{i,j}^{alloc} = \frac{\lambda_{i,j}^{alloc}}{\eta} m_j^{idle} - u_{i,j}^{alloc}. \quad (4.24)$$

Then, a Lyapunov candidate function V ([45]) can be defined, satisfying $V(\mathcal{E}^{alloc}) > 0$, $\forall \mathcal{E}^{alloc} \neq 0$ and $V(\mathcal{E}^{alloc}) = 0$ for $\mathcal{E}^{alloc} = 0$, as

$$V = \frac{1}{2} \mathcal{E}^{allocT} \mathcal{E}^{alloc}$$

Taking its time derivative yields:

$$\dot{V} = \mathcal{E}^{allocT} \dot{\mathcal{E}}^{alloc}$$

For each $\mathcal{E}_{i,j}^{alloc}$ there are two possible scenarios:

1. $\mathcal{E}_{i,j}^{alloc} \geq 0$. Then, $u_{i,j}^{alloc} = \frac{\lambda_{i,j}^{alloc}}{\eta} m_{i,j}^{idle}$, therefore

$$\mathcal{E}_{i,j}^{alloc} \dot{\mathcal{E}}_{i,j}^{alloc} = 0.$$

2. $\mathcal{E}_{i,j}^{alloc} < 0$. Since $\mathcal{E}_{i,j}^{alloc} = -|\mathcal{E}_{i,j}^{alloc}|$ and $u_{i,j}^{alloc} = 0$ we can state that

$$\mathcal{E}_{i,j}^{alloc} \dot{\mathcal{E}}_{i,j}^{alloc} = -\frac{\lambda_{i,j}^{alloc}}{\eta} |\mathcal{E}_{i,j}^{alloc}| m_j^{idle} \leq 0.$$

because $m_j^{idle} \geq 0$ for all t . Consequently we can conclude that $\dot{V} \leq 0$ which implies that $V(\zeta) \leq V(\delta_{k-1})$, $\forall \delta_{k-1} \leq \zeta < \delta_k$. Therefore each $\mathcal{E}_{i,j}^{alloc}$ remains bounded for all $\delta_{k-1} \leq \zeta < \delta_k$.

4.6.2 Execution control

The execution control relies on the proper performance of the allocation control, since it is only concerned with the rate of execution, i.e. the frequency at which the processor should operate to satisfy the accumulative execution functions.

The frequency is a parameter on the execution module of the TCPN model in Sec. 2.22c, specifically, it appears on the firing rate $\lambda_{i,j}^{exec}$ of transitions $t_{i,j}^{exec}$. The flow through those transitions represents the maximum rate at which the processor CPU_j can execute cycles, namely the maximum frequency CPU_j . Said maximum frequency F^+ has already been calculated in Sec. 4.4, it is the maximum frequency at which the processors may operate while complying with the thermal constraint and guarantee the HRT requirements of set \mathcal{T} . On the other hand, F^* is the minimum frequency for the processor, otherwise HRT requirements could not be satisfied. Therefore, the flow through transitions $t_{i,j}^{exec}$ should vary along a range defined by F^* and F^+ . Furthermore, from the problem definition all the processors execute at the same frequency.

The purpose of this controller is to keep the *execution error* $\mathcal{E}_{i,j}^{exec}(\zeta)$ equal to zero, that is, the marking $m_{i,j}^{exec}$ at places $p_{i,j}^{exec}$ has to track its appropriate accumulative execution function $R_{i,j}$.

Now, let define the *execution error vector* $\vec{\mathcal{E}}^{exec}(\zeta) = [\mathcal{E}_{1,1}^{exec}, \dots, \mathcal{E}_{n,1}^{exec}, \dots, \mathcal{E}_{n,m}^{exec}]^T$, where each $\mathcal{E}_{i,j}^{exec}(\zeta)$ is computed as:

$$\mathcal{E}_{i,j}^{exec}(\zeta) = m_{i,j}^{exec} - R_{i,j}(\zeta). \quad (4.25)$$

Then the dynamic system of the *execution error* is:

$$\dot{\mathcal{E}}_{i,j}^{exec}(\zeta) = \dot{m}_{i,j}^{exec} - \dot{R}_{i,j}(\zeta) \quad (4.26)$$

$$\begin{aligned} &= \lambda^{exec} m_{i,j}^{busy} - u^{exec} - \dot{R}_{i,j}(\zeta) \\ &= \eta F^+ m_{i,j}^{busy} - u^{exec} - \dot{R}_{i,j}(\zeta) \quad (4.27) \\ &= w_{i,j}^{exec} - \dot{R}_{i,j}(\zeta) \end{aligned}$$

where $w_{i,j}^{exec}$ is the controlled flow through transition $t_{i,j}^{exec}$:

$$w_{i,j}^{exec} = f_{i,j}^{exec} - u^{exec}$$

Proposition (4.3) shows how to select the control flow $u_{i,j}^{exec}$ for each transition $t_{i,j}^{exec}$. But since the multiprocessor platform is identical all processors work at the same frequency, for this reason the control action is referred as u^{exec} , to denote that it is the same for every $u_{i,j}^{exec}$.

Proposition 4.3 Let u^{exec} be a control law for system (4.27), such that

$$u^{exec} = \begin{cases} 0 & \text{if } \gamma \geq \eta F^+ m_E^{busy} \\ \eta(F^+ - F^*)m_E^{busy} & \text{if } \gamma \leq \eta F^* m_E^{busy} \\ \eta F^+ m_E^{busy} - \dot{R}_E + \alpha E & \text{otherwise} \end{cases} \quad (4.28)$$

$$\gamma = \dot{R}_E - \alpha E$$

where α is a positive constant, E is the element $\mathcal{E}_{i,j}^{exec}$ such that $|\mathcal{E}_{i,j}^{exec}| = \|\mathcal{E}^{exec}\|_\infty^1$, and m_E^{busy} , \dot{R}_E are the elements $m_{i,j}^{busy}$ and $\dot{R}_{i,j}$ respectively associated to E . Then the execution error is locally exponentially stable and the controlled flow is bounded $\eta F^* m_E^{busy} \leq w_E^{exec} \leq \eta F^+ m_E^{busy}$ for all $\delta_{k-1} \leq \zeta < \delta_k$.

Proof 4.3 Let V be a Lyapunov candidate function, satisfying $V(\mathcal{E}^{exec}) > 0, \forall \mathcal{E}^{exec} \neq 0$ and $V(\mathcal{E}^{exec}) = 0$ for $\mathcal{E}^{exec} = 0$

$$\begin{aligned} V &= \frac{1}{2 \cdot m \cdot n} \mathcal{E}^{execT} \mathcal{E}^{exec} \\ &= \frac{1}{2 \cdot m \cdot n} \sum_j^m \sum_i^n \mathcal{E}_{i,j}^{exec2} \leq \frac{1}{2 \cdot m \cdot n} m \cdot n \cdot E^2 \end{aligned}$$

Let,

$$Q = \frac{1}{2} E^2,$$

if Q is driven to zero, then V too. Taking the time derivative of Q :

$$\dot{Q} = E \dot{E},$$

¹ $\|x\|_\infty = \max_i |x_i|$

using Eq. (4.27)

$$\begin{aligned}\dot{Q} &= E\dot{E} \\ \dot{Q} &= E \left(\eta F^+ m_E^{busy} - u^{exec} - \dot{R}_E \right)\end{aligned}\quad (4.29)$$

with

$$u^{exec} = \eta F^+ m_E^{busy} - \dot{R}_E + \alpha E \quad (4.30)$$

then Eq.(4.29) yields:

$$\dot{Q} = -\alpha E^2 = -2\alpha Q \quad \forall \delta_{k-1} \leq \zeta < \delta_k. \quad (4.31)$$

Therefore the system is globally and exponentially stable for all $\delta_{k-1} \leq \zeta < \delta_k$. Nevertheless, the processor frequency must operate between F^* and F^+ , therefore the controlled flow associated with \dot{E} should operate in the interval $\eta F^* m_E^{busy} \leq w_E^{exec} \leq \eta F^+ m_E^{busy}$. Hence, u^{exec} cannot always take the value from Eq. (4.30). Then u^{exec} is selected as Eq. (4.28) and the exponential stability condition no longer holds globally, but in a region

$$\Omega = \left\{ E \in \mathbb{R} \mid \frac{\dot{R}_E - \eta F^+ m_E^{busy}}{\alpha} \leq E \leq \frac{\dot{R}_E - \eta F^* m_E^{busy}}{\alpha} \right\}.$$

Example 4.6 Consider the same task set from example 4.1. To prove the effectiveness of the control law, suppose there is an overhead in the system at time $[15, 16]$, such that the execution of all active tasks was interrupted, see Fig. 4.7b. Then, the system must increase the frequency to compensate for the delay.

Fig. 4.7a shows the target functions and Fig. 4.7b shows the output of the system and how the execution accelerates to reach the execution paths. This is achieved thanks to the execution control law which ensures that the current allocation of task is met, despite the time overhead (system perturbation).

4.6.3 Online Aperiodic Manager

Now, we endow our scheduler with a third component, the Online Aperiodic Manager (OnAM), which manages SRT aperiodic tasks. Upon arrival of a SRT aperiodic task, the OnAM determines if such task can be executed without compromising the

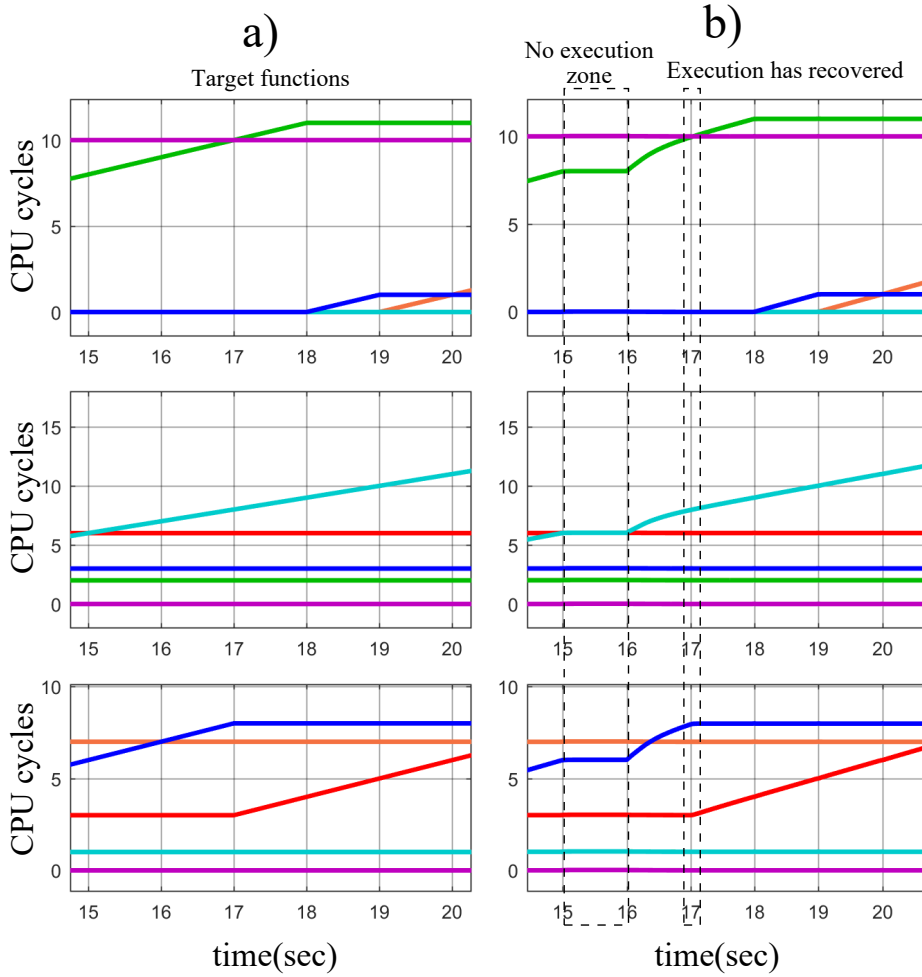


Figure 4.7.: a) Task execution paths and b) System output recovering from a system overhead in the interval [15, 16]

constraints of the HRT periodic task set. If so, it re-computes the task execution paths $R_{i,j}$ in order to include the execution of τ_i^a .

First, the OnAM determines the *scheduling intervals* where τ_i^a will be active (ϕ_r to ϕ_f). Second, it computes the processor cycles C_u pending to satisfy the rest of the active tasks until the *deadline* of τ_i^a . Third, with this information, the algorithm computes the frequency that would allow the execution of the current active jobs and the incoming aperiodic task, $F_n = \max \left\{ \frac{C_u + cc_i^a}{m \cdot d_i^a}, \frac{cc_i^a}{d_i^a} \right\}$. Last, if $F_n \leq F^+$, then OnAM accepts the incoming task into the system and assigns the workload xa_k (per frame) for τ_i^a proportionally to the *scheduling interval* duration: $xa_{i,k} = \frac{|\phi_k|}{d_i^a} cc_i^a$. If the arrival or deadline ($r_i^a + d_i^a$) of the aperiodic task does not match any current deadline, the current interval duration will differ from ϕ_k , thus xa_k is slightly modified. If the mismatch occurred at arrival time, instead of ϕ_k , the remaining time $sd_k - r_i^a$ is

Algorithm 4 OnAM Online aperiodic manager

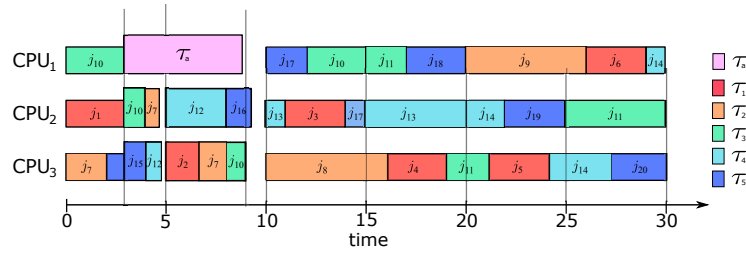
```
1: Input  $I_{SD}^k$  – Scheduling intervals;  $X^k$  – tasks CPU cycles per interval;  $ex_i^r$  – current execution  $P$ 
   cycles in interval  $cc_i^a, d_i^a$  – Aperiodic tasks parameters;
2: Output  $_j R_i$ 
3: if periodic task arrives then
4:   Determine intervals  $I_{SD}^r$  to  $I_{SD}^f$ , where  $\tau_i^a$  is active
5:   Compute required CPU cycles for active tasks
6:   if  $F_n \leq F^+$  then
7:     Accept task  $\tau_i^a$ 
8:     Assign workload  $xa_i^k$  from  $I_{SD}^r$  to  $I_{SD}^f$ 
9:     Execute Alg.1 from  $[ra, sd_f)$ 
10:    Compute task execution paths  $R_{i,j}$ 
11:   else
12:     Reject task
13:   end if
14: end if
15: if aperiodic task deadline then
16:   Execute Alg.1 from  $[sd_f, H)$ 
17:   Compute task execution paths  $R_{i,j}$ 
18: end if
```

used. However, if it occurred at the deadline, then we use $(r_i^a + d_i^a) - sd_{k-1}$. The frequency F_n along with the new workload that accounts for τ_i^a serves as input for Alg. 3, which now computes $W_{i,j}$ and $R_{i,j}$ on-line, for the interval $[ra, sd_f)$, as in Eq. (4.11) and Eq. (4.14). When task τ_i^a reaches its deadline, the execution paths are computed for the rest of active tasks under the previous frequency. Alg. 4 details the workflow of OnAM.

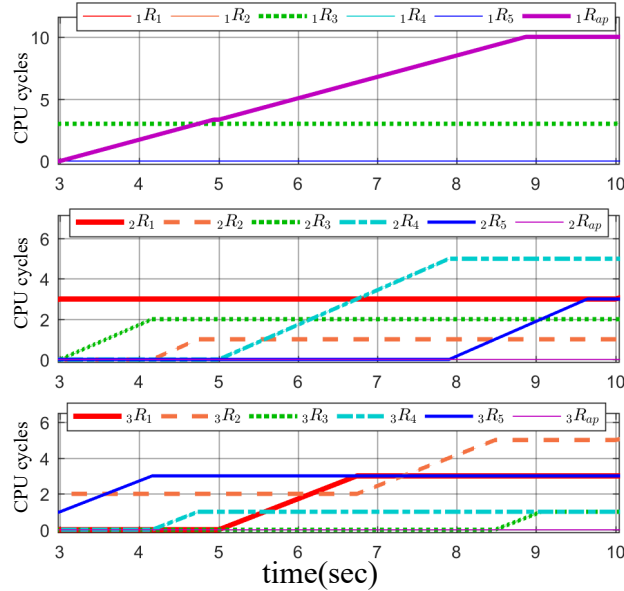
Example 4.7 Recall example 4.1. An aperiodic task arrives at $t=3$, i.e. $\tau_1^a = (10, 6, 3)$. The interval of admissible frequencies is $F = [1, 3]$. Fig. 4.8a shows the schedule for \mathcal{T} and τ_1^a , the change in height of the boxes is intended to represent the increment in the processors frequency. Fig. 4.8b shows the new accumulative execution functions computed to accommodate the aperiodic task. On CPU 1, $R_{ap,1}$ shows a ramp until $t = 9$ whereas $R_{ap,2}$ and $R_{ap,3}$ are flat for the whole interval in CPU 2 and 3, meaning that the incoming aperiodic task can be entirely accommodated in CPU 1 without disturbing the correct execution of tasks 2 and 3 now scheduled on CPU 2 and 3.

4.7 Conclusions

We have introduced a scheduling scheme which provides entirely off-line a schedule that meets the HRT, thermal and power constraints of a periodic task set. We



(a) Schedule



(b) Accumulative execution functions in time interval

Figure 4.8.: Aperiodic task management, serving an aperiodic task $\tau_{ap} = (10, 6, 3)$ that arrives at time 3 and has its deadline at time 9, for example 4.7

reformulate the ILP solved in chapter 3.14 to calculate the share of each task job per time interval. This greatly improves the schedulability of the former approach, by posing and solving the ILP taking into account the whole hyperperiod, furthermore the unimodularity property is conserved. Besides, we apply a ZL task policy off-line to allocate jobs to processors, in contrast with chapter 3 where the allocation is performed on-line. We add robustness to the system by designing a second on-line component where an allocation controller and an execution controller respectively compare actual or simulated allocation and execution data with the values (execution paths) provided by the previous off-line stage, taking action to bring the error to zero. As a bonus, the execution controller allows the design of an aperiodic task manager (onAM) simpler and lighter than the one presented in section 3.3.2.

Maximizing utilization and minimizing migration

We introduce in this chapter a scheduling framework named Clustered Allocation and Execution Control Scheduler (CAECS) which involves two strands coming out from the work exposed in the previous chapter. One of them led to improving the computation of the CE. The other one led to the redesign of the *online execution controller*.

The first strand came out from a preliminary comparison of context switch (CS) and migration overhead between the cyclic executive obtained from Sec. 4.4 and RUN ([61]), Fig. 5.1. This comparison resulted in that whereas the CE from ALECS performed better than RUN concerning the CS overhead, the results concerning the migration overhead were a bit different. The number of migrations was lower in ALECS than in RUN, but in task set 7 RUN performed an exact *partition* on the task set, thus avoiding any migration. Therefore, in a few favourable cases RUN performs better because it finds *partitions* on the task set while ALECS does not. The workload distribution in ALECS only depends on the results of the ILP, and trying to minimize the number of migrations from the ILP definition would inevitably take us back to a Bin packing problem. The solution proposed is a clustering algorithm summarized in Sec. 5.3.2 which was conceived and developed in tight collaboration with Abel Chil's MSc Thesis [20].

The second strand, the online execution controller, improves the translation from the control law in Sec. 4.6.2, to the frequency that will actually be deployed on the multicore platform. It also improves the performance of the TCPN model, and more importantly it detaches the controller implementation from the TCPN parameters, facilitating both the usage of the TCPN model for simulation and the computation of CPU frequencies for deployment. The aperiodic manager is also re-designed to exploit the capabilities of the new *frequency controller*, and avoid the laborious stage of re-computing the execution references every time an aperiodic task was released.

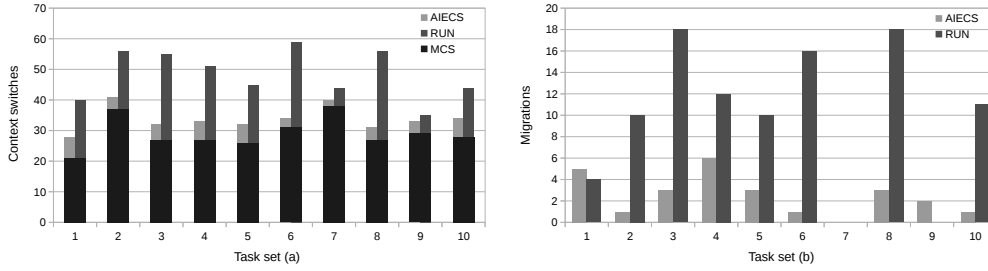


Figure 5.1.: (a) Breakdown of context switches generated by our proposal vs RUN along the ten task sets. The bottom black bars stand for the Mandatory Context Switches MCS and the upper stacked bars represent the Coerced Context Switches CCS of the proposal (grey) and RUN (dark grey). MCS are given by job activation and termination, and therefore are independent from the scheduler, unlike CCS. (b) Number of job migrations produced by our proposal (grey) vs. RUN (dark grey)

1

5.1 Problem definition

This section defines the system model and design constraint for the scheduling problem herein addressed. The main difference between the definitions from last chapter is that here the frequency of the processor is restricted to a set of discrete values.

Multiprocessor system The multiprocessor system is composed by m processors, where migration is permitted. $\mathcal{P} = \{CPU_1, \dots, CPU_m\}$ is the set of the m identical processors with an homogeneous clock frequency $F \in \mathcal{F} = \{f_1, \dots, f_{max}\}$. Furthermore, each CPU is subjected to a thermal bound T_{max} .

Task model The set of HRT tasks is denoted by $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$. Each task is independent, fully preemptive and is identified by the 3-tuple $\tau_i = (cc_i, d_i, \omega_i)$, where cc_i is the WCET in cycles, ω_i is the task period, and d_i is the relative implicit deadline ($d_i = \omega_i$) ([6]).

The arrival of asynchronous, SRT aperiodic tasks is also considered. Each aperiodic task τ_i^a is defined as a 3-tuple (cc_i^a, d_i^a, r_i^a) in which cc_i^a (required CPU cycles) and d_i^a (deadline) are known at task arrival time, and the arrival time r_i^a is unknown. Let $\mathcal{T}_a = \{\tau_1^a, \dots, \tau_p^a\}$ be a set of the p independent aperiodic tasks.

Remark 5.1 It is assumed that all task parameters, are integers and that every job can be preempted at any time. The activation of the HRT tasks occur at time zero.

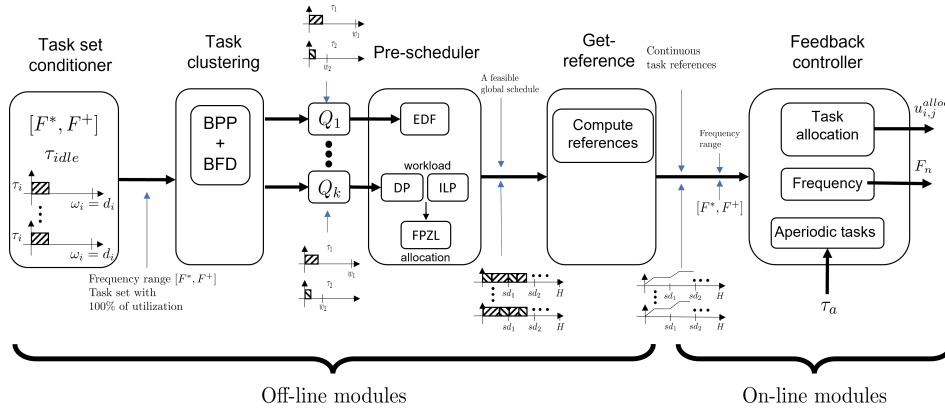


Figure 5.2.: Clustered Allocation and Execution Control Scheduler (CAIECS) Overview: the novelty of this chapter resides on the task clustering (Sec. 5.3.2) and feedback controller (Sec. 5.4) modules, the rest of modules remain practically the same under AIECS

5.2 Overview of the CAIECS scheduling system

CAIECS constitutes a good starting point to design either a standalone cyclic executive for a HRT task set or a *Minor Frame* in a collective hyperperiod.

As with the two previous chapter, we leverage a *divide and conquer* strategy. This section will help the reader to identify the elements of CAIECS as they are extended on in further sections. The CAIECS architecture appears in Fig. 5.2. The modules *task set conditioner*, *clustering*, and *pre-schedule* process the HRT task set, to yield an optimal schedule entirely off-line, ensuring minimum energy consumption and honoring a thermal bound. The module *get-reference* translates the off-line schedule into a set of references for the feedback controller. The module *feedback controller* acts on-line at runtime, it guarantees the accomplishment of the HRT and thermal constraints upon the arrival of SRT aperiodic tasks or in the presence of disturbances.

The *task set conditioner* determines the schedulability of the HRT task set on the available processors. Also, as in previous chapter it solves the minimum frequency F^{*} and the maximum frequency F^{+} , but the execution controller presented in this chapter is capable of working at discrete values of the frequency, thus instead of defining a range of operation, it defines a set of operating frequencies.

Then, the *clustering* module partitions the task set into $k \leq m$ subsets, if the number of subsets is equal to m (number of processors) then a perfect partition was achieved,

these subsets are named *clusters*. Therefore, each Q_j , where $j = 1, \dots, k$, is a task set whose utilization s_j is always integer and,

$$\sum_{j=1}^k s_j = m$$

This module uses a bin packing problem (BPP) algorithm based on the Best Fit Descending (BFD) heuristic. The novelty of this approach is that it does not enumerate all the possible solutions, and it always favours the computation of smaller size clusters.

The *pre-schedule* module finds a feasible schedule per cluster Q_j . When the utilization of a cluster is unitary (*minor cluster*), Q_j is assigned to a single *CPU* and we resort to an EDF scheduling policy to find a feasible schedule. If the cluster utilization is greater than one (*major cluster*), then Q_j is assigned to s_j processors and the pre-schedule module from *AIECS* is used, please refer to Sec. 4.4 for further details. Finally, the individual schedules are combined into a single schedule, that could be implemented straightforward, as a simple cyclic executive with minimum overhead, in the absence of additional SRT sporadic tasks or system disturbances.

The module *get-reference* uses this feasible off-line schedule to generate task-allocation references for the next module. This module is described on Sec. 4.5 and will not be repeated here.

At runtime, the *feedback controller* allocates and executes tasks. The allocation control remains the same as in *AIECS*, but the execution control is re-designed. A significant but minor change is implemented on the TCPN mode to allow an easier control law computation, it is further described under Sec. 5.4.3. The *Aperiodic manager*, Sec. 5.4.4, is capable of accepting or rejecting SRT aperiodic tasks by implementing a smart ZL policy that relies on the frequency control and, so that the temporal and thermal constraints of the system are always met. The main difference of this aperiodic manager with respect from that of *AIECS* is that this strategy avoids re-computing an schedule and the change in control references.

5.3 Off-line stage

The scheduling scheme presented in this chapter (*CAIECS*) starts with the off-line computation of the scheduling references that are later tracked by the on-line

controller at runtime. This off-line computation is broke down in four stages, which correspond to the modules *Task-set conditioner*, *Task-clustering*, *Pre-scheduler* and *Get-reference*.

The first module, *Task-set conditioner*, was introduced in Sec. 3.8 and employed in the scheduling methods of chapters 3 and 4. It is also used *as is* in CAIECS. In this Section, we first set a base example (Example 5.1) that will referenced in further explanations. Therefore, we starts by applying the *Task-set conditioner* to this base example (subsection 5.3.1).

A key point in CAIECS is an off-line clustering stage which was absent in the schemes of previous chapters. This stage is carried out by the new module *Task-clustering*, introduced in subsection 5.3.2.

The remaining modules (*Pre-scheduler* and *Get-reference*) were introduced and used in AIECS (Sec. 4.2). The last two subsections here explain the modifications required to adapt these modules to CAIECS.

5.3.1 Task set Conditioner and base example

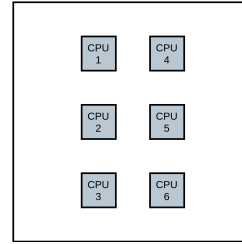
As explained in Sec. 3.8, this module checks that the HRT task set \mathcal{T} is schedulable on the available cores. Also, it yields the set of operating frequencies $\mathcal{F} = \{F^*, \dots, F^+\} \subset F$, where F^* minimizes the energy consumption, and F^+ is the maximum permissible frequency at maximum utilization.

Example 5.1 The task set conditioner stage requires knowledge of the processors layout, materials, thermal properties and the CPU operating frequencies. In this example, the MPSoC considered is composed of six $1\text{ cm} \times 1\text{ cm}$ silicon microprocessor mounted over a $7\text{ cm} \times 7\text{ cm}$ copper board, as shown in Fig. 5.3b. The thermal properties of the materials appear on Table 5.3a [30], where cp , ρ and k stands for the specific heat capacity, density and thermal conductivity coefficient, respectively.

The equation that describes the power dissipation for each core is assumed to be $P_{dyn} = C_1 * F^3 + C_2$, where $C_1 = 0.8421$ and $C_2 = 9.1579$. Table 5.1 shows the corresponding power frequency pairs. Let us consider the task set $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7\}$, where each task is defined as $\tau_1 = (10, 20)$, $\tau_2 = (5, 10)$, $\tau_3 = (7, 10)$, $\tau_4 = (7, 10)$, $\tau_5 = (7, 10)$, $\tau_6 = (14, 20)$ and $\tau_7 = (3, 5)$ Solving the minimum frequency yields $F^* = 1\text{ Hz}$, as described in Sec. 3.2.1. Since the utilization of every task under $F^* = 1\text{ Hz}$ is less than 1 and the total utilization $U_{tot} = 4.4$ is less than the number of

	Silicon	Copper
c_p	712 J/Kg K	385 J/Kg K
ρ	2330 Kg/m ³	8933 Kg/m ³
k	148 W/m °C	400 W/m °C

(a) Material properties



(b) Layout

Figure 5.3.: Material properties and layout of the MPSoC, needed to characterize the TCPN thermal model

Frequency	Power consumption
1.0 Hz	10 Watts
1.5 Hz	12 Watts
2.0 Hz	15.895 Watts
2.5 Hz	22.316 Watts
3.0 Hz	31.895 Watts

Table 5.1.: Example 5.1. Power frequency pairs

processors ($m = 6$), we only need five out of the six processors to correctly run the HRT task set. Therefore, CPU_6 will be off for the remaining of this example. Then, the optimization problem from Eq. (3.6) yields $F^+ = 2.5$ Hz as the the maximum frequency that the system stands with five cores running at maximum load and the sixth core off, while keeping the MPSoC temperature under $T_{max} = 110^\circ C$.

5.3.2 Task clustering

This clustering module was conceived and developed in tight collaboration with Abel Chills during his MSc Thesis [20]. The algorithm aims to reduce the number of job migrations by constraining such migrations within clusters of processors. Clustering also downsizes the global scheduling problem, now reduced to each cluster. A desirable effect of the algorithm in this module is that it usually reduces the number of preemptions as well. The worst case appears when only one cluster, containing all the processors, can be obtained. In such case the global scheduling algorithm must deal with the whole set of processors and tasks, boiling down to the AIECS algorithm. The clustering problem can be formally stated as follows.

Algorithm 5 Clustering algorithm

```
1: Input  $\mathcal{T}$ : Task set;  $m$ : Number of CPUs
2: Output A set of clusters;
3: Aux. functions
  ·  $\text{SolveBPP}(\text{task set}, \text{binVolume})$  – Solves BPP for bins with volume  $\text{binVolume}$ ;
  ·  $\text{utilization}(\text{bin})$  – Returns the sum of the utilizations of the tasks in the bin
4:  $Q = \emptyset$ ,
5:  $\text{binVolume} = 1$ ,
6:  $\text{cpusToAssign} = m$ ,
7:  $\text{tasksWithoutCluster} = \mathcal{T}$ 
8:  $\mathcal{T} = \{\mathcal{T} \cup \tau_{\text{idle}}\}$ 
9: while  $\text{binVolume} \leq \text{cpusToAssign}$  do
10:    $\text{SolveBPP}(\text{tasksWithoutCluster}, \text{binVolume})$ 
11:   for all  $\text{bin} \in \text{bins}$  do
12:     if  $\text{utilization}(\text{bin}) == \text{binVolume}$  then
13:       // Each data structure cluster has (number of CPUs in cluster, tasks in the cluster)
14:        $Q = Q \cup (\text{binVolume}, \text{bin})$ 
15:        $\text{tasksWithoutCluster} = \text{tasksWithoutCluster} \setminus \text{bin}$ 
16:        $\text{cpusToAssign} = \text{cpusToAssign} - \text{binVolume}$ 
17:     end if
18:   end for
19:    $\text{binVolume} = \text{binVolume} + 1$ 
20: end while
21: // Ensure that all tasks and CPUs are in a cluster
22: if  $\text{cpusToAssign} \neq 0$  then
23:    $Q = Q \cup (\text{cpusToAssign}, \text{tasksWithoutCluster})$ 
24: end if
```

Problem 5.1 *Task Clustering.* Given a task set \mathcal{T} with n tasks HRT-schedulable on m processors, find a set partition $Q = \{Q_1, \dots, Q_k\}$ of \mathcal{T} into Q_1, \dots, Q_k subsets (clusters) such that they are pairwise disjoint and $\bigcup_{j=1}^k Q_j = \mathcal{T}$, the utilization U_j of tasks in cluster Q_j . We call this utilization the size of Q_j , and is always an integer, such that the summation of all sizes equal the number of processors m , i.e. $\sum_{j=1}^k U_j = m$.

A cluster Q_j is a *minor cluster* if it has an unitary utilization $U_j = 1$, and a *major cluster* if $U_j \geq 2$. Tasks from a minor cluster are scheduled on a single processor.

A task set \mathcal{T} which is HRT feasible before the partition, will also be so after the partition, furthermore the algorithm will never increase the system capacity. Problem 5.1 implies systems with full utilization, for which we resort to add an idle task when required; details are discussed below.

Approximate solution

The clustering problem 5.1 is a form of Bin packing problem (BPP) [44], which is an NP-hard problem. Nevertheless, we can approximate a solution through the

use of heuristics. This approximate solution is designed to maximize the number of clusters of the smallest possible size upon the intuition that the number of migrations decreases with the size of a cluster. This intuition proved to be coherent with the experimental results presented in Sec. 5.5.

Definition 5.1 *The BPP states that given n items and m bins, with w_i as the weight of item i and c as capacity of each bin, assign each item to one bin so that the total weight of the items in each bin does not exceed capacity s and the number of bins used is minimum.[52]*

In order to pose problem 5.1 as a BPP, we define bins of capacity $s \leq m$ and tasks as items with weight equal to their utilization u_i . The utilization U_b of bin b is equal to the sum of the utilization of the tasks it contains. Then, a bin of size s is said to be *fully utilized* if its utilization is exactly equal to its size, $U_b = s$.

Notice that in a BPP every bin has the same capacity, therefore we will solve a BPP for each possible size of bin $s = 1, 2, \dots, m$ until a solution for problem 5.1 is found.

The process is as follows. We first solve a BPP for bins of size 1. Then, each fully utilized bin constitutes a valid cluster Q_j with utilization $U_j = 1$, and the group of tasks in each bin is assigned to a cluster. The rest of unassigned tasks are used to pose a BPP for bins of size 2, where we look again for fully utilized bins and assign their tasks to a cluster. This process continues until no unallocated tasks are left.

In the worst case, only one bin of size $s = m$ is found, this solution is always assured because the task set is already feasible $U \leq m$. In the best case scenario, there are m unit-size bins.

We resort to a Best Fit Descending (BFD) heuristic [44] to solve the BPP but other solutions are possible. Actually, the problem can also be undertaken by leveraging heuristics that solve the Knapsack Problem or the Cutting Stock problem. We made some preliminary calculations and found that the BPP formulation with the BFD heuristics provided the most promising results for us.

Alg. 5d describes the algorithm for the approximate solution. It starts by adding an idle task τ_{idle} to \mathcal{T} , for cases where $\sum u_i < m$ (line 8). We make the deadline of this task be equal to the hyperperiod, so that it has always the lowest priority. Actually, it will never be scheduled.

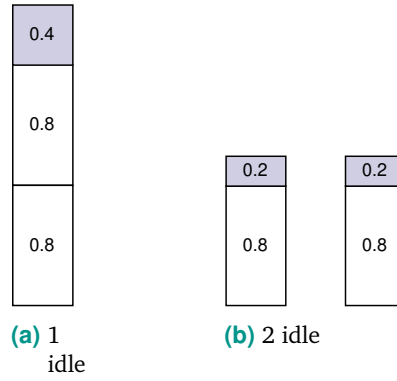


Figure 5.4.: Differences in clustering between the usage of one idle (forces one cluster) and two idle tasks (idle tasks are in gray)

Then, we solve the BPP for bins of size 1 (minor clusters). If there are bins with a utilization lower than 1, we solve the BPP again considering bins of size 2 (i.e. clusters with $s_j = 2$), then 3 and so on until no unallocated tasks are left.

A group of tasks is assigned to a cluster if the the bin containing them is fully utilized. The variables *cpusToAssign* and *tasksWithoutCluster* represent, respectively, the number of CPUs and tasks unassigned to any cluster yet. Since this heuristic does not guarantee an optimal solution of the BPP, the condition in line 22 from Alg. 5 ensures that the last cluster contains any possible group of tasks with total utilization $y < m$ that failed to conform a bin when $binVolume = y$.

Idle task: other options

Adding a single idle task τ_{idle} in Alg. 5 line 8 is just a simple solution, well suited to our scheduling scheme, which can deal with aperiodic tasks and disturbances in later stages. However, it does not always help minimize migrations. Let us consider two tasks such that $u_1 = u_2 = 0.8$, to be scheduled on two processors. Adding an idle task with $u_{idle} = 0.4$ leads to the solution in Fig. 5.4 (a), with a single cluster holding all the three tasks τ_1, τ_2 and τ_{idle} . A global scheduler could yield migrations over the two processors. Alternatively, if we add two idle tasks with utilization equal to 0.2, we reach solution (b) in Fig. 5.4, with two clusters of size 1, that will be scheduled separately using EDF, with no migrations.

When possible, the idle task is split to favour the composition of unit size bins.

Example 5.2 Consider the same task set as in example 5.1:

$$\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7\}, \text{ each task is defined as } \tau_1 = (10, 20), \tau_2 = (5, 10),$$

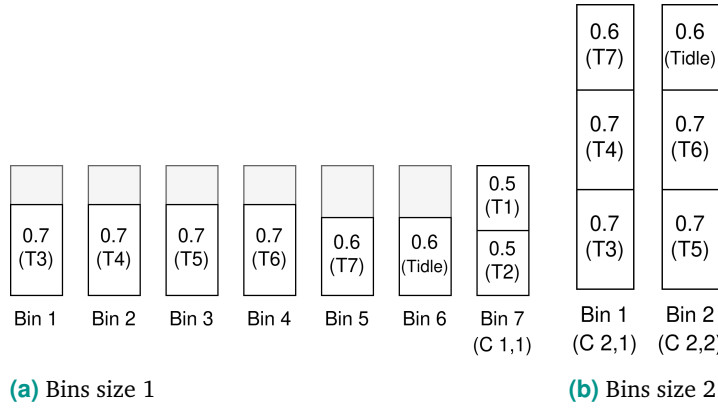


Figure 5.5.: BBP in Alg. 5: a) Iteration 1; b) Iteration 2

$\tau_3 = (7, 10)$, $\tau_4 = (7, 10)$, $\tau_5 = (7, 10)$, $\tau_6 = (14, 20)$ and $\tau_7 = (3, 5)$. The utilization per task is yields, $u_1 = 0.5$, $u_2 = 0.5$, $u_3 = 0.7$, $u_4 = 0.7$, $u_5 = 0.7$, $u_6 = 0.7$ and $u_7 = 0.6$. Thus, $U = 4.4$ and the task set is feasible on $m = 5$ processors.

We now apply Alg. 5 to allocate the tasks to the five processors maximizing utilization. Line 8 adds the idle task τ_{idle} such that $d_{idle} = 20$. Since $U = 4.4$ and $m = 5$, the utilization of $\tau_{idle} = 0.6$ and $cc_{idle} = 12$.

The first iteration of Alg. 5 solves the BPP with bins of size $binVolume = 1$ using a BFD heuristic. The *size* of each task constitutes its utilization. This yields seven bins (Fig. 5.5 a), where only one of them (Bin 7 (C 1,1)) is fully utilized. The heuristics ensures that no bin will be filled above its capacity. This bin (Bin 7 (C 1,1)) can be allocated to a single CPU (CPU₁), and accordingly removed from the pool of available cores, conforming a *minor cluster*.

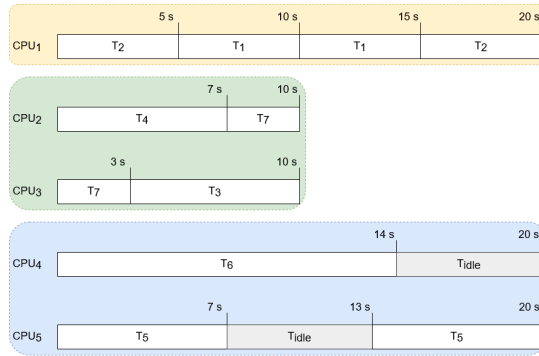
Since the iteration number (1) is less than the number of cores still available (4), the algorithm performs a second iteration using bins of volume $binVolume = 2$. This yields two major bins, (Bin 1, (C 2,1)) and (Bin 2, (C 2,2)) (Fig. 5.5 b) each of which requires two processors, conforming two *major clusters*. The current number of available cores (0) is less than the iteration number (3), so the algorithm ends.

Consequently, the task set partition yields three clusters,

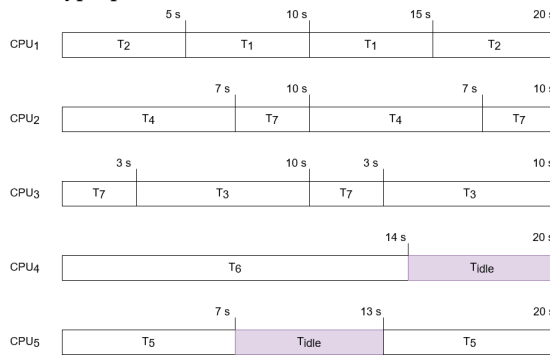
$$Q_1 = \{\tau_1, \tau_2\}$$

$$Q_2 = \{\tau_3, \tau_4, \tau_7\}$$

$$Q_3 = \{\tau_5, \tau_6, \tau_{idle}\}$$



(a) Schedule of each clusters over their respective hyperperiod



(b) CE for the HRT task set \mathcal{T}

Figure 5.6.: CAIECS pre-schedduler for task set

We allocate CPU₁ to cluster Q_1 , CPU₂ and CPU₃ to Q_2 , and CPU₄ and CPU₅ to Q_3 . Each cluster can be scheduled using any optimal global scheduler.

5.3.3 Pre-schedule

The *pre-scheduler* computes a valid off-line schedule per cluster $Q_j \in Q$. If the cluster has size 1, then it is scheduled using EDF [26]. Otherwise, the clusters is scheduled using the off-line stage of AIECS described in Sec. 4.4.

The individual schedules per each cluster are then put together into a single Cyclic Executive (CE) within the same major frame. For every CE under CAIECS, the *major cycle* is selected as the hyperperiod H of the task set. The *minor cycles* are selected as the time intervals defined from the set SD of deadlines of the task set.

Example 5.3 Consider the task clustering in Example 5.6. We apply the proper scheduler to each cluster. Fig. 5.6a (CPU₁) shows the result of applying EDF to $Q_1 = \{\tau_1, \tau_2\}$, hyperperiod equal to 20s, resulting in zero migrations, since it is

a single-core cluster. For the dual-core clusters Q_2 and Q_3 we apply the AIECS global scheduler, with an hyperperiod equal to $10s$ in both cases. Upon the resulting per-cluster scheduling, we can now generate a CE for the HRT task set \mathcal{T} over its hyperperiod $H = 20s$, replicating the scheduling in the case of clusters whose hyperperiod is less than the hyperperiod of \mathcal{T} (Fig. 5.6b). The hyperperiod of each cluster is always a divisor of the hyperperiod of the initial HRT task set. In a system with no (on-line) aperiodic tasks or disturbance management, this is the resulting thermal-safe HRT schedule, at minimum frequency, with low context switches and migrations, It can be implemented as a CE in a straightforward manner.

5.3.4 Get-reference

The off-line schedule obtained from the previous module is now translated into accumulative execution functions, such that they express the accumulated execution of every task in each processor in cycles, i.e there are $n \cdot m$ accumulative functions. The CAIECS module *get-reference* remains as the same module from AIECS, mainly for compatibility, because the frequency controller does not need the knowledge of the accumulative execution function $R_{i,j}$ at all time, only at the end of the execution interval Δ_k .

For readability, we remember the definition of such intervals Δ_k .

Let $S = \{\delta_0, \delta_1, \dots, \delta_k, \dots, \delta_h\}$ be the set with all the time stamps δ_k from a given cyclic executive when a context switch occurred, where $\delta_0 = 0$ and δ_h is the last context switch. Then, the execution interval

$$\Delta_k = (\delta_{k-1}, \delta_k], \quad (5.1)$$

is the time interval between consecutive context switches δ_{k-1} and δ_k , where $k = 1, \dots, h$.

Therefore, instead of fully define the accumulative execution functions $R_{i,j}$ as we did on Eq. 4.14, herein we only need to solve each $R_{i,j}$ at the end of each interval Δ_k and store those values, since they will be needed for the allocation and frequency controller.

Example 5.4 Consider the CE from example 5.3, shown on Fig. 5.6b). First, the set S of context switches, up to the hyperperiod, is obtained from the CE,

$$S = \{0, 3, 5, 7, 10, 13, 14, 15, 17, 20\}$$

It yields $h = 9$ execution intervals $\Delta_k = (\delta_{k-1}, \delta_k]$, for $k = 1, \dots, h$

$$\begin{aligned} \Delta_1 = (0, 3], \quad \Delta_2 = (3, 5], \quad \Delta_3 = (5, 7], \quad \Delta_4 = (7, 10], \quad \Delta_5 = (10, 13], \\ \Delta_6 = (13, 14], \quad \Delta_7 = (14, 15], \quad \Delta_8 = (15, 17], \quad \Delta_9 = (17, 20] \end{aligned}$$

Considering these intervals and the CE, we easily obtain the cycles each task must execute on each CPU at each interval Δ_k , i.e solve for $R_{i,j}(\delta_k)$ as in Eq. 4.17.

Table 5.2 shows $R_{i,j}(\delta_k) \forall i, j, k$. However, given that the task set was partitioned, some tasks never execute on some cores, such that those accumulative execution function will always be zero, those combination are excluded from Table 5.2. For instance, functions

$$R_{i,1}(\delta_k) = 0, \quad \forall i = \{3, 4, 5, 6, 7\}, \forall k$$

because only tasks τ_1 and τ_2 are allocated on CPU₁, thus they are excluded from the table.

The references in this table are the input for the on-line feedback controller at the next stage.

5.4 On-line controller

The allocation and execution of the HRT tasks at runtime is determined by the flow of transitions $t_{i,j}^{alloc}$ and $t_{i,j}^{exec}$ respectively. The markings $m_{i,j}^{busy}$ and $m_{i,j}^{exec}$ represent the tasks allocated and executed, both constitute the variables under control. The allocation and execution control will perform per δ_k interval.

We proved in Sec. 4.6.2 that the CE was guaranteed even in the presence of disturbances that could divert the task execution from the predefined schedule, thanks to the proper selection of $u_{i,j}^{exec}$. Still, there persisted a drawback on how to translate $u_{i,j}^{exec}$ to the actual selection of a frequency, which is eventually the controlled variable in the multiprocessor platform.

		k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9
$R_{i,1}(\delta_k)$	i=1	0	0	2	3	3	1	1	0	0
	i=2	3	2	0	0	0	0	0	2	3
$R_{i,2}(\delta_k)$	i=3	0	0	0	0	0	0	0	0	0
	i=4	3	2	2	0	3	1	1	2	0
	i=7	0	0	0	3	0	0	0	0	3
$R_{i,3}(\delta_k)$	i=3	0	2	2	3	0	1	1	2	3
	i=4	0	0	0	0	0	0	0	0	0
	i=7	3	0	0	0	3	0	0	0	0
$R_{i,4}(\delta_k)$	i=5	0	0	0	0	0	0	0	0	0
	i=6	3	2	2	3	3	1	0	0	0
$R_{i,5}(\delta_k)$	i=5	3	2	2	0	0	1	1	2	3
	i=6	0	0	0	0	0	0	0	0	0

Table 5.2.: $R_{i,j}(\delta_k)$, cycles that each task must execute on each CPU at every interval Δ_k

In order to address this inconvenient, the flow of transitions $t_{i,j}^{exec}$ is expressed differently. The TCPN, as a formalism, is an continuous approximation of a discrete PN that works very good when the *marking* is *large* and behaves poorly or not as close to the discrete system when the marking is small. This phenomenon induces a *delay* time before the throughput reaches a steady rate, in our systems it relates to the frequency of the CPUs, whenever the marking is sparse. Such behaviour is welcome on the allocation transitions $t_{i,j}^{alloc}$, because it allows to model latency times, such as context switch delay. Nevertheless, on the execution transitions $t_{i,j}^{exec}$ we expect to have a ramp response were the slope correspond to the operating frequency and the linear/slow delays appear both at the beginning and near the end of the execution of a task, such that the model behaves different than the system and the translations of the previous $u_{i,j}^{exec}$ depended on several model parameters.

5.4.1 TCPN equations in scalar form

The dynamics of the marking of each place of the TCPN model can be represented as the difference between the input and output flows. Specifically the dynamics of markings $m_{i,j}^{busy}$, from places $p_{i,j}^{busy}$, are given by (Sec. 2.3.2):

$$\dot{m}_{i,j}^{busy} = \frac{\lambda_{i,j}^{alloc}}{\eta} m_{i,j}^{idle} - u_{i,j}^{alloc} - f_{i,j}^{exec}, \quad (5.2)$$

where $0 \leq u_{i,j}^{alloc} \leq \frac{\lambda_{i,j}^{alloc}}{\eta} m_{i,j}^{idle}$.

Recall that flow $f_{i,j}^{exec}$ represents the rate at which CPU_j executes cycles, which holds the same interpretation as the operating frequency of the processor. This analogy was also presented on Sec. 4.6.2, where the firing execution rate λ^{exec} incorporated the frequency value F such that $f_{i,j}^{exec} = \eta F m_{i,j}^{busy}$.

The considerations described in the introduction regarding the inconvenient of the previous execution control Sec. 4.6.2 are translate into a restriction upon the dynamics of markings $m_{i,j}^{exec}$, such that it should have a constant flow equal to the processors frequency F when task τ_i is active on processor CPU_j, otherwise there is no flow. Thus:

$$\dot{m}_{i,j}^{exec} = \lambda_{i,j}^{exec} m_{i,j}^{busy} - v_{i,j}^{exec} = \sigma_{i,j} F, \quad (5.3)$$

where $v_{i,j}^{exec}$ stands for a corrective parameter that can take any real value and:

$$\sigma_{i,j} = \begin{cases} 1 & \text{if } m_{i,j}^{busy} > 0 \\ 0 & \text{if } m_{i,j}^{busy} = 0. \end{cases} \quad (5.4)$$

The marking $m_{i,j}^{busy}$ is controlled in the allocation control, therefore that marking is different from zero only when task τ_i is active on CPU_j. Thus the flow $f_{i,j}^{exec}$ in transition $t_{i,j}^{exec}$ is

$$f_{i,j}^{exec} = \sigma_{i,j} F. \quad (5.5)$$

5.4.2 Allocation control

The allocation control will result on the same equations as its homologous from Sec. 4.6.1, but first we will show that the changes introduced on the dynamic definition of $m_{i,j}^{exec}$ do not alter previous results.

First, define the vector allocation error $\mathcal{E}^{alloc}(\zeta)$ as,

$$\mathcal{E}^{alloc}(\zeta) := [\mathcal{E}_{1,1}^{alloc}, \dots, \mathcal{E}_{n,1}^{alloc}, \dots, \mathcal{E}_{n,m}^{alloc}]^T \quad (5.6)$$

where each $\mathcal{E}_{i,j}^{alloc}(\zeta)$ is computed as,

$$\mathcal{E}_{i,j}^{alloc}(\zeta) = m_{i,j}^{exec}(\zeta) + m_{i,j}^{busy}(\zeta) - R_{i,j}(d_k) \quad (5.7)$$

where $d_{k-1} \leq \zeta < d_k$. Taking the time derivative of Eq. 5.7 and using Eqs. (5.2)-(5.3), the dynamics of each error is given by :

$$\begin{aligned} \dot{\mathcal{E}}_{i,j}^{alloc}(\zeta) &= \dot{m}_{i,j}^{exec} + \dot{m}_{i,j}^{busy} \\ &= \sigma_{i,j}F + \frac{\lambda_{i,j}^{alloc}}{\eta} m_{i,j}^{idle} - u_{i,j}^{alloc} - \sigma_{i,j}F \\ &= \frac{\lambda_{i,j}^{alloc}}{\eta} m_{i,j}^{idle} - u_{i,j}^{alloc} \\ &= f_{i,j}^{alloc} - u_{i,j}^{alloc} = w_{i,j}^{alloc} \end{aligned} \quad (5.8)$$

Eq. (5.8) is equal to Eq. (4.21), therefore results from Proposition (4.2) hold.

5.4.3 Frequency control

In accordance with the off-line calculations, we assume that the CPUs are identical, they work at the same frequency and the operating frequency f is in the set $\mathcal{F} = \{F^*, \dots, F^+\}$. Below we offer a discussion, to accommodate different assumptions.

Each processor can only attend one task at a time and parallelism is not allowed, consequently, the number of active tasks is at most $|\mathcal{P}|$. Therefore there must be, at most, $|\mathcal{P}|$ transitions $t_{i,j}^{exec}$ enabled. Accordingly, we define the vector of active tasks $\mathbf{m}_{active}^{exec}$ as,

$$\mathbf{m}_{active}^{exec} = A(\Delta_k) \mathbf{m}^{exec} \quad (5.9)$$

where \mathbf{m}^{exec} is a vector that holds every element $m_{i,j}^{exec}$, and $A(\Delta_k)$ is a matrix with $\{0, 1\}$ entries that depends on the tasks which are active during the interval Δ_k , defined in Eq. (4.7). Thus, $\mathbf{m}_{active}^{exec}$ holds only $m_{i,j}^{exec}$ values corresponding to the active tasks in the Δ_k . Therefore, $A(\Delta_k)$ has only one nonzero element per row, and no more than one nonzero element per column.

Then, the execution error of the active tasks \mathcal{E}_{exec} will be defined as:

$$\mathcal{E}_{exec}(\zeta) := A(\Delta_k)\mathbf{m}^{exec} - \mathbf{R}(\delta_k) \quad (5.10)$$

for all $\zeta \in \delta_k$, and $\mathbf{R}(\delta_k)$ is the vector of elements $R_{i,j}(\delta_k)$ computed from Eq. (4.14) corresponding to every active tasks in Δ_k .

Using Eq. (5.10), the dynamics of the execution error is given by

$$\dot{\mathcal{E}}_{exec}(\zeta) = A(\Delta_k)\dot{\mathbf{m}}^{exec}(\zeta) \quad (5.11)$$

where $A(\Delta_k)$ remains constant during each interval Δ_k .

Proposition 5.1 *Let F be the frequency at which all CPUs work during the execution interval $\Delta_k = (\delta_{k-1}, \delta_k]$, such that*

$$F \geq \frac{\|\mathcal{E}_{exec}(\delta_{k-1})\|}{\lambda|\Delta_k|} \quad (5.12)$$

where $|\Delta_k|$ is the duration of interval Δ_k , and $\mathcal{E}_{exec}(\delta_{k-1})$ is the execution error at the beginning of the interval.

Then $\mathcal{E}_{exec}(\zeta)$ reaches zero before the end of the interval.

Proof 5.1 *Let V be the candidate Lyapunov function:*

$$V = \frac{1}{2}\mathcal{E}_{exec}^T \mathcal{E}_{exec}. \quad (5.13)$$

Deriving Eq. (5.13) and using Eq. (5.3),

$$\begin{aligned} \dot{V} &= \mathcal{E}_{exec}^T \dot{\mathcal{E}}_{exec} \\ &= \mathcal{E}_{exec}^T A(\Delta_k)\dot{\mathbf{m}}^{exec}(\zeta) \\ &= \mathcal{E}_{exec}^T A(\Delta_k)\boldsymbol{\sigma}F = \mathcal{E}_{exec}^T \Phi F \end{aligned} \quad (5.14)$$

where $\boldsymbol{\sigma}$ is a vector containing all $\sigma_{i,j}$ and Φ is a vector with all $|\mathcal{P}|$ entries equal to 1. Then Eq. 5.14 can be rewritten as

$$\dot{V} = \sum_{j=1}^{|\mathcal{P}|} \mathcal{E}_{exec,j} F \quad (5.15)$$

From proposition 4.2, u_{alloc} restricts transitions t_{alloc} such that only $R_{i,j}(\delta_k)$ tokens are available during Δ_k . Hence, $\mathcal{E}_{exec_j} \leq 0$, for $j = 1, \dots, |\mathcal{P}|$, therefore:

$$\dot{V} = -\|\mathcal{E}_{exec}\|_1 F. \quad (5.16)$$

From Eq. (5.13),

$$V = \frac{1}{2} \mathcal{E}_{exec}^T \mathcal{E}_{exec} = \frac{1}{2} \|\mathcal{E}_{exec}\|_2^2$$

Solving for $\|\mathcal{E}_{exec}\|$,

$$\|\mathcal{E}_{exec}\|_2 = (2V)^{1/2}, \quad (5.17)$$

Using Eq. (5.16),

$$\begin{aligned} \dot{V} &= -\|\mathcal{E}_{exec}\|_1 F \leq -\lambda \|\mathcal{E}_{exec}\|_2 F \\ \dot{V} &\leq -\lambda (2V)^{1/2} F \end{aligned} \quad (5.18)$$

By the comparison lemma [45], where $\zeta \in \Delta_k$,

$$\begin{aligned} \frac{dV}{d\zeta} &\leq -\lambda (2V)^{1/2} F \\ \int_{V(d_{k-1})}^{V(\zeta)} x^{-1/2} dx &\leq -\lambda \sqrt{2} F \int_{d_{k-1}}^{\zeta} d\tau \\ 2V(\zeta)^{1/2} - 2V(d_{k-1})^{1/2} &\leq -\lambda \sqrt{2} F (\zeta - d_{k-1}) \end{aligned} \quad (5.19)$$

Substituting $V(\zeta)$ from Eq. (5.17) and solving for $\|\mathcal{E}_{exec}(\zeta)\|$ we obtain:

$$\begin{aligned} \frac{2}{\sqrt{2}} \|\mathcal{E}_{exec}(\zeta)\| - \frac{2}{\sqrt{2}} \|\mathcal{E}_{exec}(d_{k-1})\| &\leq -\lambda \sqrt{2} F (\zeta - d_{k-1}) \\ \|\mathcal{E}_{exec}(\zeta)\| &\leq \|\mathcal{E}_{exec}(d_{k-1})\| - \lambda F (\zeta - d_{k-1}) \end{aligned}$$

Solving for ζ when $\|\mathcal{E}_{exec}(\zeta)\| = 0$, we get

$$\zeta \leq \frac{\|\mathcal{E}_{exec}(d_{k-1})\|}{\lambda F} + d_{k-1}$$

To ensure that task deadlines are met, the execution error should be zero before the interval Δ_k is over, thus

$$\zeta \leq \frac{\|\mathcal{E}_{exec}(\delta_{k-1})\|}{\lambda F} + \delta_{k-1} \leq \delta_k$$

Hence,

$$\begin{aligned} \frac{\|\mathcal{E}_{exec}(\delta_{k-1})\|}{\lambda F} &\leq \delta_k - \delta_{k-1} \\ \frac{\|\mathcal{E}_{exec}(\delta_{k-1})\|}{\lambda|\Delta_k|} &\leq F \end{aligned} \quad (5.20)$$

Any frequency that satisfies Eq. (5.20) is suitable to drive \mathcal{E}_{exec} to zero before interval Δ_k is over. Therefore, F is chosen as the smallest $F \in \mathcal{F}$ (from Ec. (4.1)) such that Eq.(5.20) is satisfied.

It is straightforward to prove by induction the stability of \mathcal{E}_{exec} up to the hyperperiod, as this Proposition holds for the interval Δ_1 and therefore no error is carried over into the next execution interval. Therefore,

$$\|\mathcal{E}_{exec}(\zeta)\|_\infty \leq \max\{|\Delta_1|, \dots, |\Delta_h|\} F^*.$$

Proposition 5.1 shows that when selecting the operating frequency as in Eq.(5.12) the HRT tasks will not miss their deadlines even when a bounded overload is present.

5.4.4 Aperiodic Manager

As stated before, the on-line control is capable to reject disturbances, and thus the execution controller treats aperiodic tasks as a type disturbances. Namely, when an aperiodic task τ_a arrives to the system and is accepted, a disturbance is simulated on processor CPU_{*j*} assigned to the τ_a . In this way, the controller will automatically update the CPU frequency. When the HRT task that was originally assigned to CPU_{*j*} reaches its zero laxity, τ_a will be assigned to the next available CPU. For this purpose, we first calculate the minimum frequency F_n required to serve both the HRT tasks and the aperiodic task τ_a ,

$$F_n = F + \frac{cc_a}{d_a U}$$

Algorithm 6 Aperiodic Manager

```
1: Input  $U$ : System utilization;  $\mathcal{P}$ : CPUs
2: Output  $F_n$ : frequency for aperiodic tasks;  $D$ : disturbance
3: Aux. functions
   · LaxityInCPU( $CPU_j, t$ ) – Returns laxity for  $\tau_{ap}$  at time  $t$  ;
4:  $F_n = F + \frac{cc_a}{d_a U}$  ,
5: if  $F_n \leq F^+$  then
6:   accept=1; // Accept  $\tau_{ap}$ 
7: end if
8: while accept == 1 do
9:   if LaxityInCPU( $CPU_j, t$ ) > 0 then
10:    Assign  $\tau_{ap}$  to  $CPU_j$ 
11:    Simulate disturbance on  $CPU_j$ 
12:   else
13:     $CPU_j$  = next available CPU;
14:   end if
15:   if  $\tau_a$  finishes execution then
16:    accept=0;  $F_n = F^*$ ;
17:    Eliminate disturbance from  $CPU_j$ 
18:   end if
19: end while
```

where F is obtained from Eq. (5.12), U is the system utilization and (cc_a, d_a) are the parameters of the aperiodic task τ_a . If $F_n \leq F^+$, the aperiodic task is accepted, otherwise it is rejected. F_n is now the minimum frequency acceptable in order to reach every task deadline. Therefore the set \mathcal{F} is restricted to those $F \geq F_n$. The aperiodic task manager proceeds according to Alg. 6.

Example 5.5 Consider the same system from previous examples 5.3-5.4. To demonstrate the behavior of the feedback controller and the aperiodic manager, assume that an aperiodic task with a relative deadline of 9 s and WCET of 7 s arrives at $t = 1$ s. The multiprocessor are assume to be executing at $F = 1$ Hz. In Fig.5.7 we see that the feedback controller increases the frequency of the processors to 1.5 Hz at $t = 1$ s, holding it up to $t = 7$ s, in order to allow the execution of the aperiodic task. The feedback controller will be able to execute the aperiodic task during this interval meeting the timing constraints, then restoring the previous frequency and the execution references at $t=7s$.

Thermal management To demonstrate the proper thermal behavior the system, we have performed a simulation with the processor distribution shown in Fig. 5.3b. The MPSoC is cooled down by forced air, with a heat transfer coefficient of $500 W/m^2 * K$ [47]. Fig. 5.8a displays the temperature evolution at the center of the cores when the system is running the cyclic executive. Aligned side-by-side, Fig. 5.8b shows the temperature evolution at the same places when the system manages an aperiodic

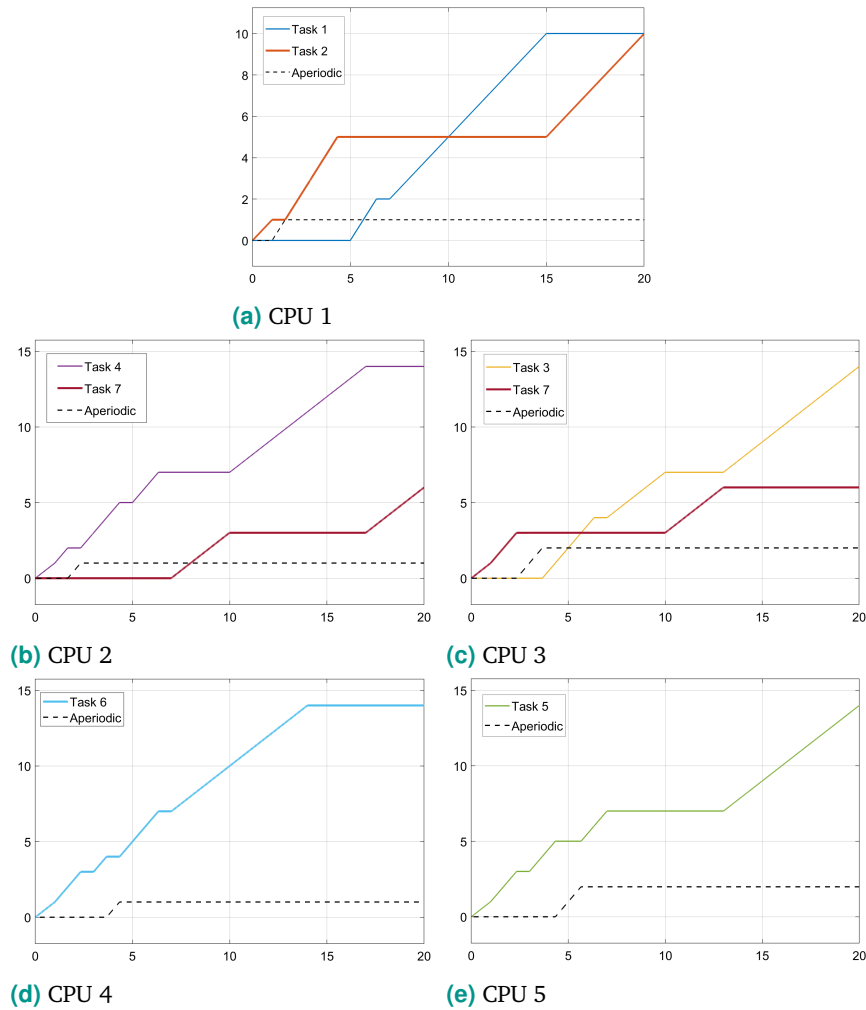
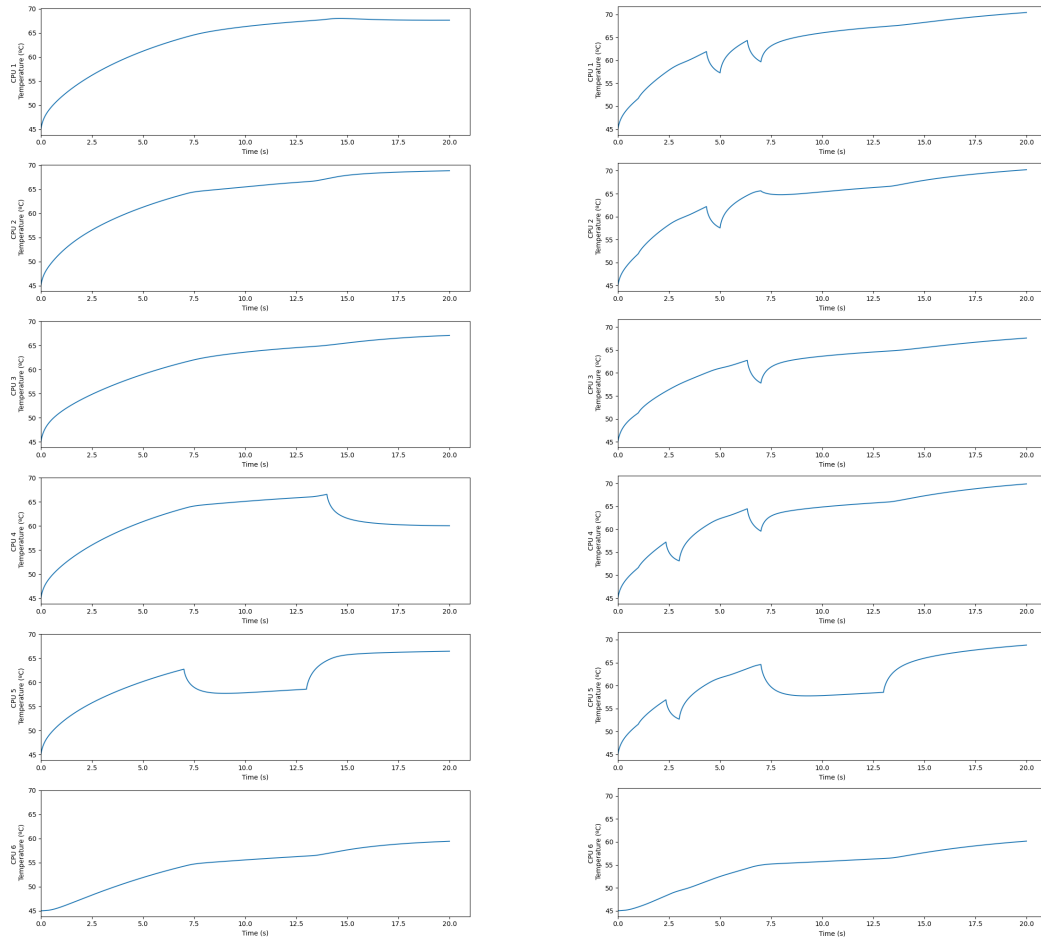


Figure 5.7.: Execution of tasks with the admission of an aperiodic task

task throttling the frequency during that interval. The core temperature increases along the interval $[1, 7]$, then decreases when the frequency is set back to F^* , once normal execution resumes.

5.5 Comparison with RUN

CAIECS specifically deals with the challenge of maximizing processor utilization while minimizing job migration and context switching. In order to evaluate this capability, in this Section we compare the number of context switches and migrations entailed by CAIECS, AIECS and RUN.



(a) CE

(b) CE and aperiodic task admission

Figure 5.8.: Processors temperature evolution, on the left the nominal behaviour and on the right the temperature evolution with the admission of an aperiodic tasks

We consider RUN because it is considered an optimal reference (Sec. 2.2.4) as far as context switches and migration is concerned. Also, we include AIECS because it is just the worst-case clustering solution of CAIECS —the situation in which there is a single (major) cluster encompassing the whole HRT task set and all the CPUs.

5.5.1 Simulation environment and setup

We carry out the comparison using Tertimuss [21]. We only consider job preemptions as context switches. We rule out job start and job termination events, since their number is common to every scheduling algorithm. A migration is computed when a

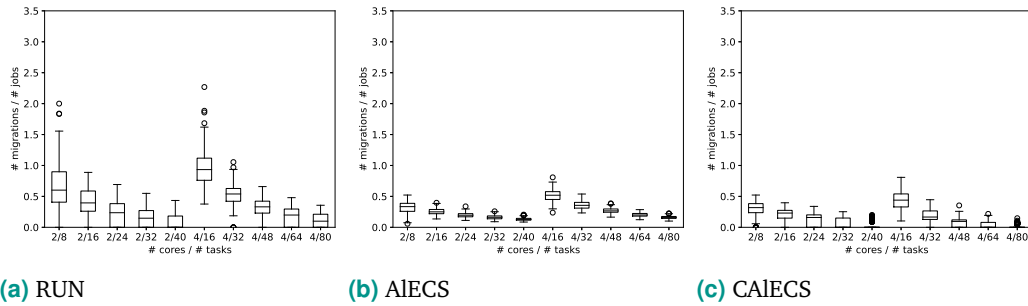


Figure 5.9.: Simulation results for the number of migrations/number of jobs.

job resumes execution on a CPU different from the one on which it was previously running.

The task sets for the comparison are generated using the UUniFast-discard algorithm [23]. The total utilization of each task set is equal to the number of processors in the experiment. Task periods are randomly selected between the divisors of 60, to obtain a major cycle of at most 60 s. The task sets are executed on systems with 2 and 4 cores, with task-to-core ratios of 4, 8, 12, 16, and 20. This amounts to 200 experiments per combination, totalling 2000 experiments in all.

Tertimuss takes the WCET in cycles, and performs the simulation on a cycle-by-cycle basis. Since RUN defines its tasks as *fixed rate tasks* and can yield fractional execution times, we apply some adjustments in the task set creation to avoid floating point errors.

Following the original description of the algorithm in [61], we implemented RUN using a Worst-Fit policy for the packing operation, the EDF policy for the scheduling of the servers, and the original task-to-processor allocation policy. When applying EDF, deadline ties in servers are either solved at random if no server has run yet, or by choosing the server that was last executed, otherwise. Since all the three schedulers under comparison are optimal and the selected task sets are feasible, all the three schedulers obtain a feasible schedule in all experiments.

5.5.2 Migrations per job

The boxplots in Fig. 5.9 summarize the number of migrations per job (Y-axis) as the number of tasks per processor (TPP) varies (X-axis). Tab. 5.3 details the mean and the standard deviation. A common trend to all the three schedulers under comparison is that both the mean and the standard deviation of migrations per job

m	n	RUN		AIECS		CAIECS	
		Mean	SD	Mean	SD	Mean	SD
2	8	0,676	0,375	0,320	0,096	0,298	0,122
	16	0,407	0,254	0,254	0,054	0,193	0,119
	24	0,224	0,192	0,196	0,039	0,113	0,101
	32	0,145	0,151	0,162	0,033	0,059	0,083
	40	0,087	0,117	0,130	0,024	0,032	0,058
4	16	0,957	0,283	0,516	0,096	0,431	0,140
	32	0,526	0,165	0,359	0,061	0,192	0,099
	48	0,316	0,153	0,270	0,041	0,090	0,070
	64	0,182	0,135	0,203	0,031	0,041	0,048
	80	0,108	0,112	0,159	0,024	0,014	0,028

Table 5.3.: Number of migrations per job depending on the tasks-per-processor ratio (TPP)

decrease as the number of processors decreases and the TPP increases. These results support the intuition discussed in Sec. 5.3.2, upon the rationale that a minor cluster yields zero migrations.

AIECS achieves fewer migrations than RUN with low TPP ratios, but RUN outperforms AIECS with high TPP ratios (16, 20). Also, the mean of migrations decreases faster in RUN than in AIECS as the TPP increases. Last, AIECS never reaches zero migrations, while RUN yields a Q1 of zero with high TPP ratios, also in the case of the 24/2 TPP, and reaches a median of zero in the case 40 tasks executed in two CPUs. However, the standard deviation is outstandingly lower in AIECS than in RUN in all cases. The analysis of the behavior of RUN with high TPP ratios shows that RUN manages to find a full partition of the task-set (minor cluster) during its packing operation in a considerable amount of experiments, leading to a zero migrations figure in those cases.

These results prove that CAIECS outperforms AIECS and RUN in all configurations because it is able to find as many partitionable task sets as RUN, while retaining the properties of AIECS when scheduling major clusters. Also, it can reach a median of zero migrations when the TPP ratio is high, even yielding a Q3 of zero in the case of 80 tasks executed in four CPUs. The standard deviation is lower in CAIECS than in RUN but higher than in AIECS.

RUN is not specifically designed to find clusters, particularly minor clusters (achieving zero migrations), but its ability to find them often, when possible, makes RUN reach the zero migrations notch in cases where AIECS fails to do this. The *task clustering* stage in CAIECS aims to find low-size clusters, reaching zero migrations

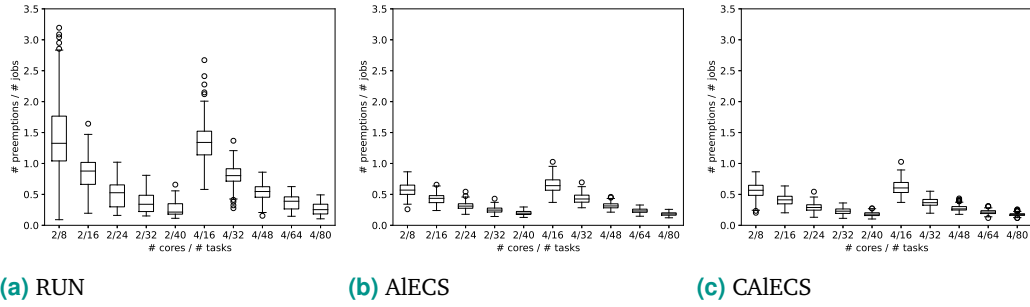


Figure 5.10.: Simulation results for the number of preemption / number of jobs.

m	n	RUN		AIECS		CAIECS	
		Mean	SD	Mean	SD	Mean	SD
2	8	1,449	0,608	0,575	0,117	0,561	0,132
	16	0,832	0,291	0,431	0,082	0,410	0,090
	24	0,495	0,199	0,313	0,058	0,288	0,063
	32	0,367	0,157	0,249	0,047	0,228	0,049
	40	0,267	0,114	0,199	0,034	0,183	0,035
4	16	1,362	0,307	0,653	0,116	0,614	0,120
	32	0,816	0,172	0,433	0,074	0,371	0,068
	48	0,534	0,137	0,317	0,049	0,273	0,047
	64	0,370	0,116	0,236	0,035	0,214	0,034
	80	0,266	0,090	0,184	0,026	0,174	0,025

Table 5.4.: Number of preemptions per job

in more experiments than RUN (Fig. 5.9). We can examine Tables 5.5 5.6 to better understand the clustering behavior of RUN and CAIECS (AIECS does not apply clustering) and to assess the consistency of our experimental results. The third column in this table shows the number of clusters of each size (from 1 to 4 CPUs) obtained for each TPP. For example, with 64 tasks on four processors, Table 5.6, CAIECS finds a full partition (four minor clusters) in 108 experiments, whereas RUN does it only 56 times. In 85 experiments, CAIECS fits the HRT task set into two minor clusters and a major cluster with two CPUs, whereas RUN only finds that configuration in 20 experiments. This trend holds for all design points, with CAIECS always finding substantially more minor clusters (entailing zero migrations) than RUN.

m	n	Distribution of CPUs in clusters	CAIECS	RUN
2	8	1,1	13	6
		2	187	194
	16	1,1	49	33
		2	151	167
	24	1,1	85	72
		2	115	128
	32	1,1	132	92
		2	68	108
	40	1,1	152	120
		2	48	80

Table 5.5.: Clustering performance

5.5.3 Preemptions per job

The boxplots in Fig. 5.10 and Tab. 5.4 help to analyze the mean of preemptions per job by experiment for each TPP configuration. The results indicates that AIECS performs better than RUN on the average, yielding a lower standard deviation and fewer outliers. However, there are some specific cases where RUN outperform AIECS, such as when scheduling 8 tasks over two CPUs, with RUN providing a minimum value of 0.09091 while AIECS reaches 0.2593. As for CAIECS, it outperforms RUN and AIECS on the average, with a slightly greater standard deviation than AIECS nonetheless.

As we observed with migrations, the mean of preemptions and its standard deviation decreases in all the three schedulers as the TPP ratio increases. Also, for each TPP ratio, the mean of preemptions decreases with the number of CPUs.

5.6 Computational complexity

This section gathers the computational complexity of the various parts of the proposed system, Fig. 5.2. The off-line stages calculate the minimum and maximum frequencies, resolve the task clustering, and apply a pre-scheduling which, depending on the container size of each cluster, follows the EDF or ZL dispatching rules. The on-line stage consists on a continuous controller and the aperiodic manager.

m	n	Distribution of CPUs in clusters	CAIECS	RUN
4	16	1, 1, 1, 1	0	0
		1, 1, 2	7	0
		2, 2	15	0
		1, 3	62	17
		4	116	183
	32	1, 1, 1, 1	7	4
		1, 1, 2	97	9
		2, 2	11	0
		1, 3	60	40
		4	25	147
	48	1, 1, 1, 1	53	20
		1, 1, 2	113	21
		2, 2	9	0
		1, 3	22	59
		4	3	100
	64	1, 1, 1, 1	108	56
		1, 1, 2	85	20
		2, 2	2	0
		1, 3	4	56
		4	1	68
80	1, 1, 1, 1	160	89	
	1, 1, 2	38	26	
	2, 2	1	0	
	1, 3	1	32	
	4	0	53	

Table 5.6.: Clustering performance

Off-line calculations

The Task Set Conditioner calculates the minimum and maximum frequencies (F^* and F^+). Computing F^* is linear in the number of tasks $\mathcal{O}(n)$. The calculation of F^+ requires solving a non-linear optimization problem, whose number of iterations to find a solution depends on some parameters such as the required convergence error and the gradient weighting. To this end, we employ an interior point algorithm. Tests show that it provides a good performance and converges to the optimum in a very short time.

The computational complexity of our task clustering (Sec. 5.3.2) depends on Alg. 5. The outer `while` loop (line 9) iterates m times. At each iteration it first executes `SolveBPP` (line 10), whose complexity is $n \times \log(n)$. Then, it runs the inner `for`

loop. All instructions inside the inner for loop are executed $\alpha \times m$ times. Thus, the computational complexity of Alg. 5 is of order $\mathcal{O}(m^2 \times n \times \log(n))$.

Next, the complexity of solving Eq. (4.9) to compute the workload per scheduling point is linear in x_i^k , i.e. in $n^2 \times \beta$, where $\beta = \max_{\tau_i} \frac{H}{w_i}$. Thus, it is in the order of $\mathcal{O}(n^2)$. Finally, *EDF* or *ZL* (Alg.3) policies run in polynomial time.

Hence, all the algorithms in the off-line stage run in polynomial time except , but the non-linear optimization. As mentioned before, in our tests, the non-linear optimization algorithm runs in a very short time.

On-line computations

References $R_{i,j}(\delta_k)$ (Eq. 4.17) are pre-computed, and only checked when a context switch occurs, which results on gathering the appropriate execution requirement .

The feedback controller (Sec. 5.4) runs at every sampling period, which depends on the minimum difference between two consecutive $\delta_i - \delta_{i-1}$ from set S . Since the RT clock routine is executed with a fixed period and more frequently than any other RT task, we propose to implement the feedback controller routine as a call back (deferred function, softirq or tasklet depending on the RT operating system), activated at each RT clock routine execution. In this approach we include a feedback controller routine per *CPU*. Thus, their complexity is $\mathcal{O}(1)$, because it requires to solve Eqs. (4.23) and (5.12), that have a fixed number of operations. A different implementation approach may use a devoted *CPU* for the feedback controller routine.

5.7 Conclusions

RT scheduling on multicore processors remains a challenge in many ways, all the more when temperature and energy counts. One of the points is maximizing processor utilization to avoid overprovisioning. Partitioned scheduling approaches fall short in this aspect, whereas optimal global schedulers are too complex to be effective. Near-optimal solutions are possible by leveraging a mix of ad-hoc techniques, but there is still a shortage of practical methods and tools interesting enough for the industry to adopt them.

CAIECS leverages the ability of known scheduling algorithms to simplify the global scheduling problem, and the power of a fluid model and feasible mathematical optimization to account for thermal constraints and to maximize processor utilization. It provides an off-line cyclic executive for a HRT task set which is thermal-safe, energy efficient, easy to implement and more effective than RUN. Our comparison with RUN reveals that CAIECS is able to find more minor clusters and more major clusters of smaller size, along with the fact that the global scheduler AIECS performs particularly well in lowering migrations in major clusters.

As a cyclic executive, the number of preemptions and migrations is known for the hyperperiod, and therefore the WCET, which usually includes scheduling costs, can be fine-tuned in future work to obtain a schedule with more realistic bounds. This cyclic executive can feed an on-line controller which manages the SRT aperiodic tasks, deals with small disturbances, due to parameter variations for example. It could easily fit a slack reclamation scheme. The modular design of CAIECS permits designers to employ other thermal models or schedulers.

Accounting for preemption and migration costs on CE

The problem we are addressing comes from a causality dilemma which appears when leveraging preemptive multicore RT scheduling algorithms to calculate a CE. On the one hand, the final number of preemptions (some of which entail a migration) is only known after calculating the CE. On the other hand, to calculate a CE we must account for the overheads of all context switches and migrations. Since we do not know the final number of context switches and their type (with or without migration) until we have calculated the CE, the only possible solutions are either considering an upper bound for preemptions and migrations, or to proceed iteratively, ensuring that the RT constraints are always met. Neither solution is trivial, as we analyze in which follows.

There is a context switch whenever a job is released or terminates. We name these cases *compulsory context switches*. Its number is just the number of jobs, considering all tasks in the set, and is independent on the scheduler, although different schedulers can allocate tasks to CPUs with different outcomes. Thus, it is trivial to account for the cost of such context switches either with non-preemptive or preemptive schedulers, because we can add it beforehand to the WCET. In contrast, considering the cost of the new context switches /migrations introduced by preemptive schedulers is far more challenging.

In the case of priority-driven scheduling algorithms, such as RM or EDF, the upper bound for the number of preemptions is strictly less than the number of jobs that are being scheduled. This result also holds for the number of job migrations among CPUs [6]. In global DP-FAIR scheduling policies, the upper bound for preemptions and for job migrations per minor frame are, respectively, the number of tasks minus one, and the number of CPU minus one [36]. Nevertheless, these bounds lead to very pessimistic calculations, worsening the WCET overestimation problem.

A straightforward alternative approach consists on calculating the CE considering the WCET of tasks and obviating any overhead, then compute the overhead, add

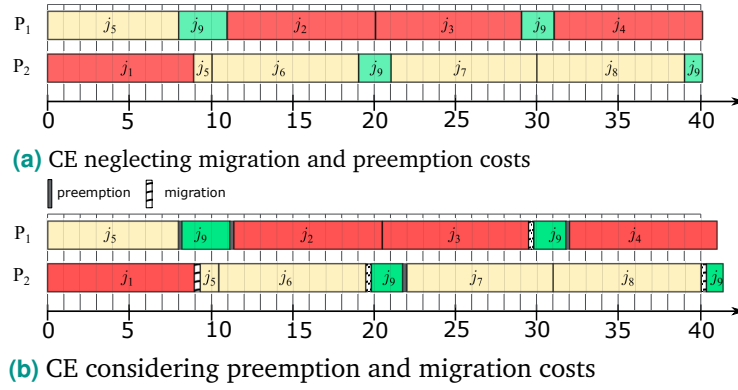


Figure 6.1.: Motivational example. CE of task set on 6.1. j_1 to j_4 are τ_1 jobs, j_5 to j_8 are τ_2 jobs, and j_9 is the only job of τ_3 . The black thin line represents a job boundary, the black bar indicates a preemption and the hatched bar indicates a migration.

it to the CE, and increase the frequency to augment system capacity accordingly. However, the following simple example (Fig. 6.1) proves that this naive approach leads to missing task deadlines.

Example 6.1 Consider a task set $\mathcal{T} = \{\tau_1, \tau_2, \tau_3\}$ with implicit deadlines, where $\tau_1 = (9000, 10)$, $\tau_2 = (9000, 10)$ and $\tau_3 = (8000, 40)$. WCET (cc_i) is given in cycles and relative deadlines (d_i) in time units. Utilization value u_i is computed assuming a frequency $f = 1000$, such that $u_1 = 0.9$, $u_2 = 0.9$ and $u_3 = 0.8$. The task set \mathcal{T} is scheduled on $m = 2$ processors. Fig. 6.1a shows a CE for this set of tasks, where preemption and migration costs are neglected. Illustratively, Fig. 6.1b highlights the four preemptions and four migrations, adding their computing time to the CE such that the overall execution overflows the hyperperiod. Now, assume a worst-case preemption (migration) cost p_{cost} (m_{cost}) of 10 and 20 cycles, respectively. Let $U_{\mathcal{T}}$ represent the task set utilization,

$$U_{\mathcal{T}} = \sum_i^3 \frac{cc_i}{f \cdot d_i} = 2 \leq m,$$

since $U_{\mathcal{T}} = m$, we said that the system is at full capacity. Now, let first compute the associated utilization overhead on the system $u_{overhead}$, given that the overhead is computed as the number of preemptions pre times the preemption cost p_{cost} plus the number of migrations mig times its cost m_{cost} ,

$$u_{overhead} = \frac{4p_{cost} + 4m_{cost}}{f \cdot H}$$

where H is the hyperperiod of \mathcal{T} . Then we *update* the frequency accordingly, from the fact that the maximum system utilization is equal to the number of cores, i.e $U \leq m = 2$,

$$U_{\mathcal{T}} + u_{overhead} \leq m$$

$$\frac{1}{f} \left(\sum_i^3 \frac{cc_i}{d_i} + \frac{4p_{cost} + 4m_{cost}}{H} \right) \leq m$$

Thus, new frequency is $f = 1001.5$, such that the system increased its capacity 1.5%. However, this approach leads to deadline misses. Both τ_1 and τ_2 should complete their first job by time=10. On processor P_2 , the first job of τ_1 (j_1) completes its WCET (9000 cycles), afterwards the first job of τ_2 (j_5), previously allocated on P_1 , resumes and completes its remaining 1000 cycles. Considering the corresponding migration cost, the execution ends at $\frac{9000+20+1000}{1001.5} = 10.005$, missing τ_2 deadline at 10.

In this thesis we propose two different solutions. Both of them consider the WCET obtained from static code an. They first compute a CE. The resulting CE provides the number and temporal localization of all preemptions and migrations for every job and core. In the first solution (named AdWCET, Sec. 6.1), the WCET of each task is adjusted by adding up the corresponding overhead. The stages iterate until the preemption and migration overhead keeps constant, thus concluding the adjustment.

The second solution, named DP-U (Sec 6.2, follows closer the idea presented in the motivational example (6.1). It also starts from a CE which provides the temporal location of every preemption and migration. Then, it finds the time interval between consecutive deadlines that holds the maximum overhead. Finally, it computes the increment in frequency required to accommodate such overhead.

Both AdWCET and DP-U satisfy the following assumptions:

Assumption 6.1 *The WCET of each task used to calculate the first CE is obtained from static tools and accounts for memory conflicts but not for scheduling overheads*

Assumption 6.2 *Both, the preemption overhead cost p_{cost} and migration cost m_{cost} are parameterized and given in cycles.*

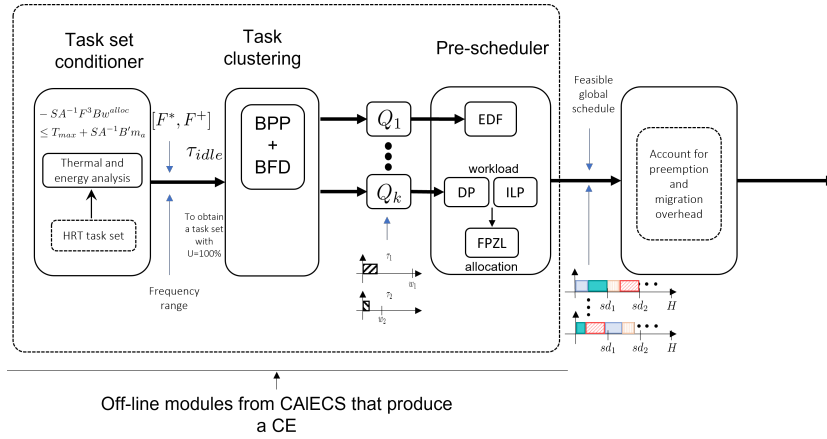


Figure 6.2.: Position of the algorithm that account for preemption and migration overhead, with regards of the CAIECS scheme which is defined inside the dotted rectangle

At the end of the chapter, we present a comparison between both strategies based on experimental results.

6.1 The AdWECT algorithm

AdWCET takes a CE as the input at every iteration, and updates the WCET according to this CE. When the WCET stabilizes, the algorithm ends. Otherwise a new iteration starts. The CE at each iteration can be computed using any appropriate scheduler. In this thesis we leverage CAIECS because it constitutes a reference in yielding a very low number of preemptions and migrations as we explained in Chapter 5.

Definition 6.1 *The overhead of a job j_x from task τ_i that incurs in pre_{j_x} preemptions and mig_{j_x} migrations is computed as,*

$$overhead_{j_x} = (p_{cost} \cdot pre_{j_x}) + (m_{cost} \cdot mig_{j_x}) \quad (6.1)$$

where p_{cost} (m_{cost}) is the associated cost in cycles of a preemption (migration).

The algorithm can be outlined as follows:

1. Compute a preemptive cyclic executive.
2. Count the number of preemptions and migrations per job on the CE.

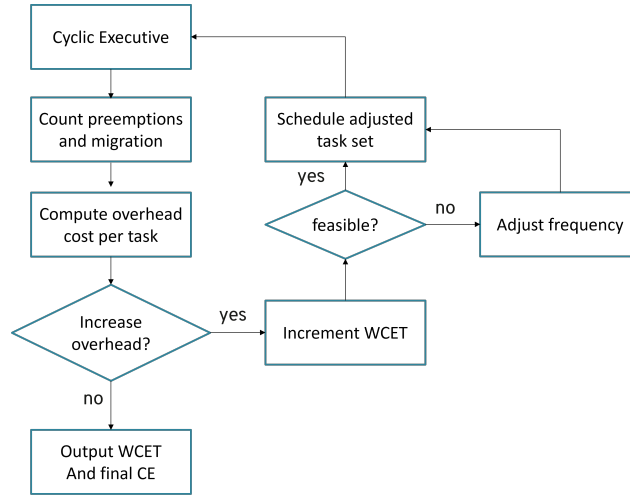


Figure 6.3.: Flow chart of AdWCET, the algorithm to adjust the WCET of a task set on a CE

3. Compute the overhead in cycles for each task. This steps requires two intermediate steps:
 - a) Compute the overhead of each of its jobs $overhead_{j_x}$, as in Eq. 6.1.
 - b) Select the task's overhead $overhead_{\tau_i}$ as the maximum value from the previous computation.
4. For each task, check if its new overhead is greater than the previous overhead. If there was no change, then the algorithm has finished. Otherwise, continue
5. Adjust the task's WCET. Add the task overhead from step 3 to its original WCET, this is $ad_cc_i = cc_i + overhead_{\tau_i}$.
6. Check if the *adjusted* task set is still schedulable with the current frequency, if true go to step 1. Otherwise, continue.
7. Increase the frequency f to make the task set schedulable again, for CAIECS this is true if $U \leq m$, then

$$f = \frac{1}{m} \sum_{i=1}^n \frac{ad_cc_i}{\omega_i}$$

where ad_cc_i is the adjusted WCET of the tasks. Go to step 1.

AdWCET requires a minimum of two iterations. The first iteration calculates a first schedule with its corresponding overheads; the second iteration adjusts the WCET based on this overhead.

Example 6.2 Consider the task set $\mathcal{T} = \{\tau_1 = (3000, 4), \tau_2 = (4000, 6), \tau_3 = (5000, 12), \tau_4 = (4000, 24)\}$, to schedule on $m = 2$ processors with the available discrete frequencies $f \in F = \{1000, 1020, 1040, \dots, 2000\}$ (Hz). The cost in cycles per preemption is $p_{cost} = 10$ and per migration is $m_{cost} = 20$ (cycles). This task set spawns 13 jobs, the task to job relation is given bellow,

$$\tau_1 : j_1, j_2, j_3, j_4, j_5, j_6$$

$$\tau_2 : j_7, j_8, j_9, j_{10}$$

$$\tau_3 : j_{11}, j_{12}$$

$$\tau_4 : j_{13}$$

This example converges in three iterations. The iterations proceed as follows:

Iteration 1

Step 1 Compute a cyclic executive, (Fig. 6.4a)

Step 2 Count migrations and preemptions per job:

Migrations per job $[(j_8, 1), (j_9, 1), (j_{10}, 1)]$.

Preemptions per job: $[(j_8, 1), (j_9, 1), (j_{10}, 1), (j_{11}, 1), (j_{12}, 1)]$

Step 3 Compute task's overhead

1. Calculate overhead per job: zero for every job but

$[(j_8, 30), (j_9, 30), (j_{10}, 30), (j_{11}, 10), (j_{12}, 10)]$

2. Select new overhead per task: $[(\tau_1, 0), (\tau_2, 30), (\tau_3, 10), (\tau_4, 10)]$

Step 4 Check if the new task overhead is greater than previous task overhead: this is true because this is the first iteration.

Step 5 Add overhead to original task set:

Adjusted WCET ad_cc_i by task: $[(\tau_1, 3000), (\tau_2, 4030), (\tau_3, 5010), (\tau_4, 4010)]$

Step 6 Check schedulability. System utilization is $U = 2.00625$, the system is no longer schedulable.

Step 7 Increase frequency, $f = \frac{1}{m} \sum_{i=1}^n \frac{adjustedWCET_i}{\omega_i} = 1003.12$, but because of the discrete values $f = 1020$. Return to step 1.

End of iteration 1

Iteration 2

Step 1 Compute a cyclic executive, Fig. 6.4b.

Step 2 Count migrations and preemptions per job:

Migrations by job: $[(j_8, 1), (j_{10}, 1), (j_{11}, 1), (j_{12}, 1), (j_{13}, 1)]$.

Preemption by job: $[(j_8, 1), (j_9, 1), (j_{10}, 1), (j_{11}, 2), (j_{12}, 2), (j_{13}, 2)]$

Step 3 Compute task's overhead

1. Calculate overhead per job: zero for every job but

$[(j_8, 30), (j_9, 10), (j_{10}, 30), (j_{11}, 40), (j_{12}, 40), (j_{13}, 40)]$

2. Select new overhead per task: $[(\tau_1, 0), (\tau_2, 30), (\tau_3, 40), (\tau_4, 40)]$

Step 4 Check if the new task overhead is greater than previous task overhead: this is true because this is the first iteration, $0 = 0, 30 = 30, 40 > 10, 40 > 10$

Step 5 Add overhead to original task set:

Adjusted WCET ad_cc_i by task: $[(\tau_1, 3000), (\tau_2, 4030), (\tau_3, 5040), (\tau_4, 4040)]$

Step 6 Check schedulability. System utilization is $U = 1.97$, the system is schedulable.

End of iteration 2 Go back to step 1.

Iteration 3

Step 1 Compute a cyclic executive, Fig. 6.4c.

Step 2 Count migrations and preemptions per job:

Migrations by job: $[(j_8, 1), (j_{10}, 1), (j_{11}, 1), (j_{12}, 1), (j_{13}, 1)]$.

Preemption by job: $[(j_8, 1), (j_9, 1), (j_{10}, 1), (j_{11}, 2), (j_{12}, 2), (j_{13}, 2)]$

Step 3 Compute task's overhead

1. Calculate overhead per job: zero for every job but

$[(j_8, 30), (j_9, 10), (j_{10}, 30), (j_{11}, 40), (j_{12}, 40), (j_{13}, 40)]$

2. Select new overhead per task: $[(\tau_1, 0), (\tau_2, 30), (\tau_3, 40), (\tau_4, 40)]$

Step 4 Check if the new task overhead is greater than previous task overhead: this is true because this is the first iteration, $0 = 0, 30 = 30, 40 = 40, 40 = 40$

The end

The overhead of the tasks for the second iteration increases from $[0, 30, 10, 10]$ to $[0, 30, 40, 40]$, and therefore the adjustment requires a third iteration. We can see that the overhead does not increase for any task anymore, therefore the algorithm converges. The resulting CE appears in Fig. 6.4c, which is the same as in a previous iteration Fig. 6.4b. The adjusted WCET of the tasks is: $[3000, 4030, 5040, 4040]$,

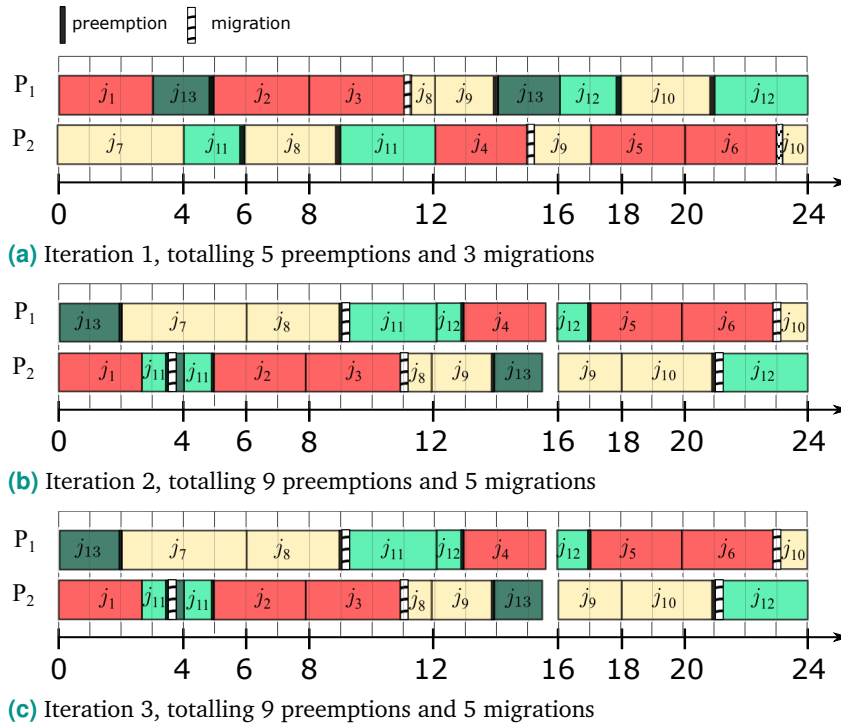


Figure 6.4.: Iterations for AdWCET to adjust the WCET for example, using CAIECS as scheduling policy.

with frequency $f = 1020$. The adjusted task set increments system utilization by 2% and ensures the accomplishment of temporal constraints.

6.1.1 Convergence of AdWCET algorithm

AdWCET either increases or holds the overhead of tasks, even if the current iteration yields a smaller overhead. Therefore, it will always converge. Besides, the number of iterations is bounded because the same scheduling policy holds throughout the adjustment. Also, the maximum overhead is bounded by the maximum number of preemptions (ρ) and migrations (μ) of the chosen scheduling policy, i.e. $n(\rho + \mu)$ iterations in the worst case, where n is the cardinality of \mathcal{T} . Actually, this bound is unlikely reached. The following section evaluates AdWCET performance choosing CAIECS [67] as the scheduling policy.

6.2 DP-U, an approach based on the utilization

In contrast with AdWCET, DP-U accounts for the overhead due to preemptions focusing on the utilization per interval, cautiously leveraging the motivational example exposed in 6.1. In such example, the overhead on system utilization $u_{overhead}$ is used to compute the new frequency f that should accommodate the preemptions and migrations, but fails to achieve the expected outcome.

The overhead on system utilization $u_{overhead}$ is computed as a task utilization, but instead of the execution time cc_i it uses the overhead computed in cycles, and for ω_i it takes the hyperperiod,

$$u_{overhead} = \frac{overhead}{f \cdot H} \quad (6.2)$$

The main issue of leveraging $u_{overhead}$ (6.2) to compute the new frequency f is that this computation does not take into account the exact temporal location of the added workload, and only computes its *weight* on the processor. Therefore, to effectively use this approach, the calculation of $u_{overhead}$ has to be performed differently.

Definition 6.2 Let SD be the ordered set of deadlines of all jobs $j \in \mathcal{J}$ within $H \cup \{0\}$. Such that $sd_0 = 0$, $sd_{k-1}, sd_k \in SD$, $sd_{k-1} < sd_k$, where k is an integer value greater than zero.

Definition 6.3 Let a frame ϕ_k be the time interval between two consecutive deadlines, such that $\phi_k = [sd_{k-1}, sd_k) \forall sd_k \in SD$. And the duration of the frame is $|\phi_k| = sd_k - sd_{k-1}$.

Instead of computing the overhead per job, as in adWECT (Eq. (6.1)), DP-U computes the overhead per frame k at each processor j .

Definition 6.4 The overhead in a frame ϕ_k , wherein $pre_{k,j}$ preemptions and $mig_{k,j}$ take place, is defined as

$$overhead_{k,j} = (p_{cost} \cdot pre_{k,j}) + (m_{cost} \cdot mig_{k,j}) \quad (6.3)$$

where p_{cost} (m_{cost}) is the associated cost in cycles of a preemption (migration).

Remark 6.1 To avoid confusion while referring to a processor's frequency f , let f_b be the baseline frequency, or the frequency for the first computed CE. And f_n denotes the new calculated frequency that accounts for the overhead.

A better selection of the frequency

The idea behind this strategy is to accommodate the overhead inside the CE, this is, how to increase the system capacity such that the preemption times fit inside the hyperperiod, and at the same time ensure HRT constraints.

If we enforce a DP-like constraint on the CE, such that every overhead $overhead_{\phi_k,j}$ fits in its frame ϕ_k along with the current job execution, then the HRT constraints will be satisfied. This idea is written as,

$$\frac{ex_{k,j} + overhead_{k,j}}{f_n} \leq |\phi_k| \quad (6.4)$$

$$\frac{ex_{k,j} + overhead_{k,j}}{|\phi_k|} \leq f_n \quad (6.5)$$

where $ex_{k,j}$ represents the execution in cycles during frame ϕ_k in processor CPU_j . Thus, the HRT constraints will be satisfied if we compute the required f_n per frame and processor, and we select the maximum value.

Observe that the maximum $ex_{k,j}$ appears when there is no spare time at ϕ_k in CPU_j . This yields:

$$ex_{k,j} \leq |\phi_k| \cdot f_b \quad (6.6)$$

Therefore, substituting in Eq. (6.5):

$$\begin{aligned} \frac{|\phi_k| \cdot f_b}{|\phi_k|} + \frac{overhead_{k,j}}{|\phi_k|} &\leq f_n \\ f_b + \frac{overhead_{k,j}}{|\phi_k|} &\leq f_n \end{aligned} \quad (6.7)$$

Defining,

$$u'_{k,j} = \frac{overhead_{k,j}}{|\phi_k|} \quad (6.8)$$

eq. (6.7) is written as,

$$f_b + u'_{k,j} \leq f_n \quad (6.9)$$

In this case, calculating f_n will suffice to select the maximum $u'_{k,j}$, and then compute Eq. (6.9) with this value.

Upon this analysis, the outline of the algorithm is as follows:

1. Compute a preemptive cyclic executive.
2. Count the number of preemptions $pre_{k,j}$ and migrations $mig_{k,j}$, per frame ϕ_k and per CPU_j .
3. Compute the overhead in cycles per frame ϕ_k and per CPU_j , as in Eq. (6.3)
4. Obtain the quotient of overhead per frame ϕ_k and the duration of the frame,

$$u'_{k,j} = \frac{overhead_{\phi_k,j}}{|\phi_k|}$$

5. Select the the maximum value $u'_{max} = \max\{u'_{k,j} | \forall k, j\}$ from the previous computation.
6. Select the new frequency f ,

$$f_n = (f_b + u'_{max})$$

7. Update CE with the new frequency f

6.3 Experimental results

This section compares empirically the performance of the two previous strategies. The main focus will be on the increment in system capacity demanded by each method. First, we explain the experimental set up. Then, we conduct some experiments per method. Finally, we perform a comparison to asses which method achieves the smallest increment in system capacity.

6.3.1 Experimental setup

We carry out the experiments using Tertimuss [21], an open-source framework to model a RT multiprocessor system, simulate different RT schedulers, and process the results.

Task sets are generated using the Dirichlet Rescale (DRS) algorithm [39]. The total utilization of each task set is equal to the number of processors in the experiment, with available frequencies $f \in F = \{1000, \dots, 1500\}$ (Hz). Task periods are randomly selected between the divisors of 60, to obtain a major cycle of at most 60 s. The task sets are executed on systems with 2 and 4 cores, with task-to-core ratios of 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44 and 48. This amounts to 100 experiments per combination, totalling 2400 experiments in all.

6.3.2 Methodology for task generation

The task sets for the experiments are generated such that they are valid and feasible under a m -processor system, i.e. $U \leq m$ and $u_i \leq 1 \forall \tau_i \in \mathcal{T}$.

We first calculate task utilization leveraging the DRS algorithm, which produces n -dimensional vectors uniformly distributed over the valid region, such that the components sum to a specified value (*total task set utilization*). Each component accepts lower and upper bounds. The algorithm has the signature $\mathbf{u} = \text{DRS}(n, U, \mathbf{u}^{\max}, \mathbf{u}^{\min})$, where $\mathbf{u} = (U_1, U_2, \dots, U_n)$ is the output vector of task utilization values, n is the cardinality of the task set, U is the specified total utilization and $\mathbf{u}^{\max} = (U_1^{\max}, U_2^{\max}, \dots, U_n^{\max})$ and $\mathbf{u}^{\min} = (U_1^{\min}, U_2^{\min}, \dots, U_n^{\min})$ are optional vectors to constrain the utilization values for the task set, by default $\mathbf{u}^{\max} = (1, 1, \dots)$ and $\mathbf{u}^{\min} = (0, 0, \dots)$.

To mimic a realistic setup, the minimum WCET and the preemption/migration costs are of different orders of magnitude. This follows the commonly accepted notion that a context switch is negligible with respect to the execution time of a task.

To this purpose, we define wcet_{\min} as the minimum WCET that any generated task can receive. The DRS algorithm allows imposing a lower bound for the computed utilization of every task in the generated task set, that we calculate as:

$$u_{\min} = \frac{\text{wcet}_{\min}}{fH} \quad (6.10)$$

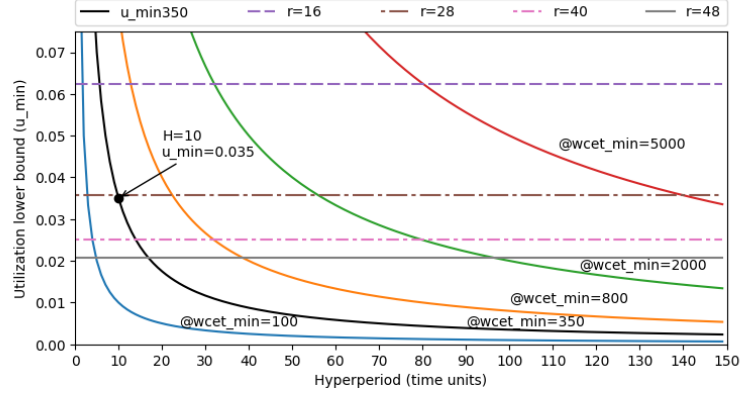


Figure 6.5.: Utilization lower bound u_{min} for different hyperperiods. The horizontal lines represent the constraint $1/r$, where r is the task-to-core ratio. $f = 1000$, fixed for every curve

where f is the frequency and H the hyperperiod. This expression is deduced from the fact that the utilization u_i of any task τ_i is inversely proportional to its period ω_i (Eq. (2.2)). Hence, the minimum utilization is achieved with the largest possible period, which is the hyperperiod H .

Nevertheless, when constraining the minimum value of every task we must ensure that the utilization constraint allows for a feasible task set. Therefore, the utilization lower bound should not be greater than the total of processors, i.e.:

$$\sum_{i=1}^n \frac{wcet_{min}}{fH} \leq m \quad (6.11)$$

Hence, the task-to-core ratio ($r = n/m$) cannot be selected arbitrarily. There exists a trade-off between hyperperiod, frequency and task-to-core ratio. From Eq. 6.11 it follows that:

$$\frac{wcet_{min}}{fH} \leq \frac{m}{n} \quad (6.12)$$

Therefore, u_{min} should always be less or equal than $1/r$ in order to generate a valid task set:

$$u_{min} \leq \frac{1}{r} \quad (6.13)$$

To illustrate the importance of Eq. (6.13) on the utilization lower bound, we plot in Fig. 6.5 the relation of u_{min} to different values of H . The frequency value

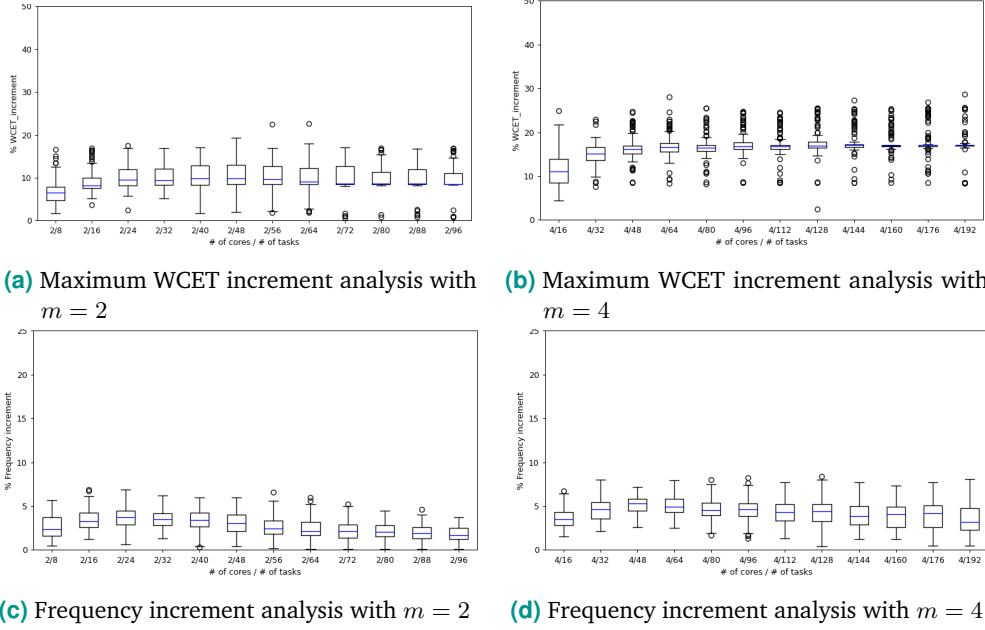


Figure 6.6.: Percentage of WCET maximum increase (a and b) and percentage of frequency increase (c and d)

is fixed at $f = 1000$, and every u_{min} curve is computed for each $wcet_{min} \in [100, 350, 800, 2000, 5000]$. The flat colored lines represent the constraint from Eq. (6.13), for four task-to-core ratios r . Therefore, to generate valid task sets, u_{min} must be below the $1/r$ desired line.

Note from Fig. 6.5 that it is easier to satisfy Eq. (6.13) for smaller values of $wcet_{min}$. Also, as $wcet_{min}$ increases the number of feasible task-to-core ratios decrease.

For example, with $wcet_{min} = 350$ and $H = 10$, we obtain $u_{min} = wcet_{min}/(f \times 10) = 0.035$, and therefore a task set with task-to-core ratio $r = 40$ will be unfeasible. However, this task-to-core ratio becomes feasible for $H \geq 14$. On the other hand, valid task sets for $H = 10$ can be generated with any $r \leq 28$. In our setup, we chose $wcet_{min} = 350$, and $H = 60$, therefore every task set on the experiments is valid.

6.3.3 Results

AdWCET

AdWCET iterates to adjust the WCET of a task set according to the overhead introduced by migrations and preemptions. The number of iterations is highly dependent on the scheduling algorithm, but this number is bounded.

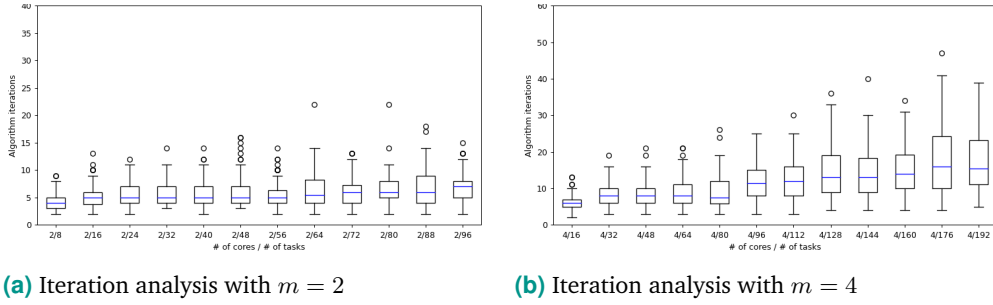


Figure 6.7.: Iteration analysis of AdWCET under CAIECS

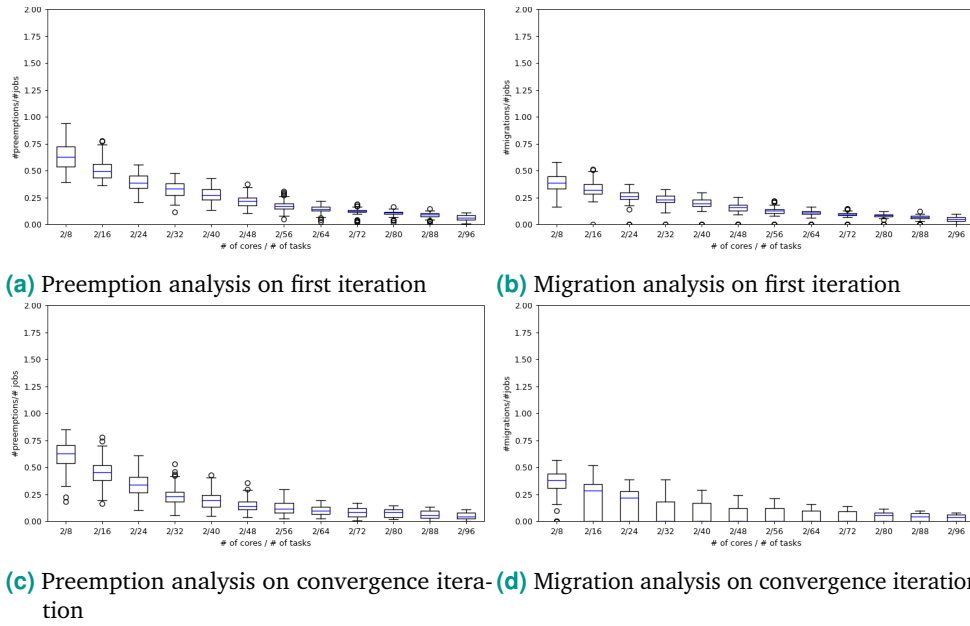


Figure 6.8.: Analysis of preemption and migrations per job on first and convergence iteration

We generate task sets as explained in Sec. 6.3.1 and focus on WCET and frequency increases after adjusting the CE, and on the number of iterations to reach convergence. The following figures share all the same scheme. The configurations, i.e. number of processors over the number of tasks, appear on the x-axis, whereas the y-axis holds the studied feature.

The results on WCET and frequency increments are both normalized and plotted as percentage increase. In the case of WCET Fig. 6.6a-6.6b, we selected the maximum percentage of increase among the WCETs on each task set, i.e. it shows the WCET increment from the task that had the maximum overhead, and does not represent the behaviour of the whole task set.

In the case of frequency, the percentage increase was computed as the increment between the frequency on the first iteration and the frequency at the convergence iteration. The latter indicates the required increment in system capacity. The results in Fig. 6.6c-6.6d show that the maximum WCET increase is bounded by 30% and is more variable than the frequency increase, which lies below 6% for most cases. The increment in frequency is more stable than the maximum increment in WCET as the number of tasks per core increase. This is consistent with the fact that the latter represents the maximum overhead of a single task while the former measures the overhead on the task set.

Fig. 6.7 shows the number of iterations required by AdWCET to converge to a solution for each task set in every configuration. Fig. 6.7a shows the iteration analysis for different task set sizes on 2-processors. The number of iterations is always below 15 except for an spurious case. Fig. 6.7b shows that it takes more iterations to reach convergence on 4-processors than on 2-processors. This is the logical result because the number of possible schedules increases with the number of tasks and processors.

Fig. 6.8 shows the analysis of preemptions and migrations per job in the experiments on 2-processors. We only take into account the data from the first and last iteration. The total number of preemptions and migrations never exceeds 1, because of the chosen scheduler (CAIECS). The number of preemptions and migrations will be consistent in both iterations for any other scheduler, as long as the number of tasks and processors holds.

The number of iterations increases with the ratio of tasks per processor, whereas preemptions and migrations decrease (Fig. 6.8, Fig. 6.7). This is because the theoretical bound for the iteration is $n(\rho + \mu)$, and therefore the number of iterations will always be proportional to the task set cardinality, and to the number of preemptions and migrations. Fortunately, on CAIECS both migrations and preemptions decrease as the number of tasks increase, thus keeping the number of AdWCET iterations low.

DP-U

In the following experimental analysis we now focus on the increment of system capacity, with is the relevant aspect to compare with AdWCET. We use the same task set in the experiments as in AdWCET. Also, the figures show again the configurations,

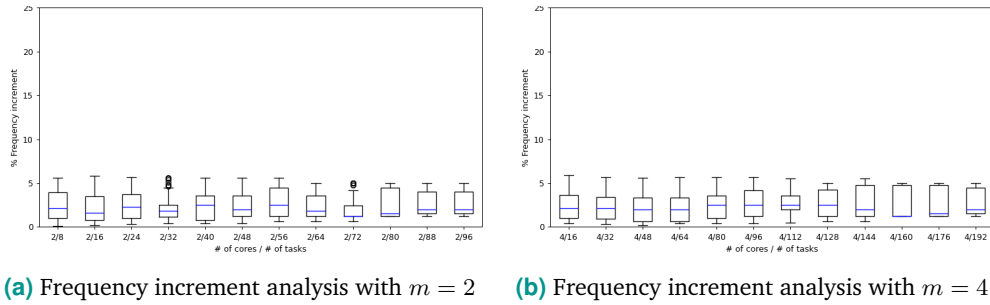


Figure 6.9.: Percentage of frequency increase in the utilization approach

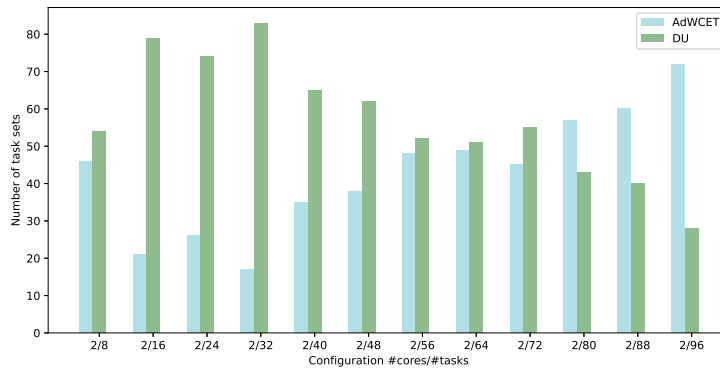
i.e. number of processors over the number of tasks, on the x-axis, whereas the y-axis holds the studied feature. The results on system capacity increment is plotted as percentage increase.

6.4 Comparison between AdWCET and DP-U

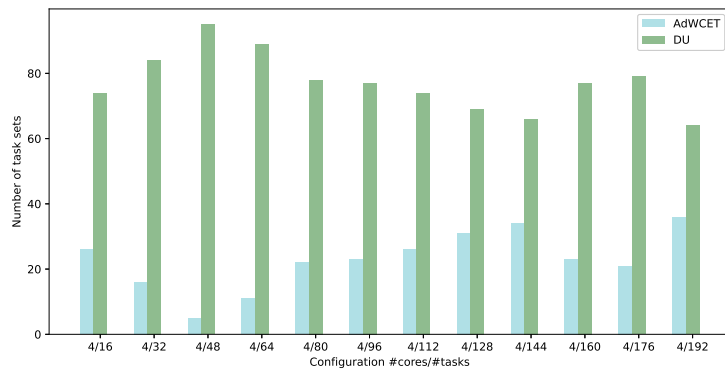
In this section, we compare the increment in system capacity under each method to analyze which one yields the smallest increment while honoring HRT constraints.

Each bar in the plots of Fig. 6.10 represents the number of experiments in which each algorithm yields the lowest frequency of the two. Applying a lower frequency results in a lower capacity increase. In the experiments with under $m = 2$ processors (Fig. 6.10a) both strategies improve CE calculation accounting for preemption overheads. However, with higher numbers of tasks (last three configurations) AdWCET performs notably better than the alternate approach. Whilst, DP-U clearly outperforms AdWCET on five configurations. On four configurations ('2/8', '2/56', '2/64', and '2/72') it cannot be clearly stated which strategy is better. Thus, according to the experimental results both strategies are suitable when working on two processor, but AdWCET is more advisable when having task sets with a task-to-core ratio greater than or equal 28, and the utilization approach for task-to-core ratio $r \leq 24$.

In contrast, DP-U outperforms AdWCET in every configuration under $m = 4$ processors (Fig. 6.10b).



(a) Task set scheduled on 2 processors



(b) Task set scheduled on 4 processors

Figure 6.10.: Comparison of frequency increment in AdWCET and DP-U. Each bar represents the number of experiments in which that algorithm obtained the lower frequency of the two.

6.5 Conclusion

In this chapter we presented two safe methodologies to account for the preemption and migration overheads when producing a preemptive Cyclic Executive (CE) schedule.

Usually, a CE enforces that a task cannot be preempted inside a minor frame, the consequence of such constraint is low system utilization. However, if we construct a preemptive CE then it is possible to achieve fully utilized systems. But, since this is an offline-solution the scheduling overhead has to be integrated. As mentioned on the introduction, it is a common assumption that the time cost of both migration and preemption is either neglected or considered part of the WCET. For the latter case, upper bounds on the number of migrations and preemptions should be

considered but this entails a highly pessimistic value and increases the problem of overprovisioning.

AdWCET is born from the idea of adding the overhead in the WCET, but instead of using upper bounds only use the values from current schedules. But since the WCET parameter is changed then the schedule is not longer valid, thus the reason for AdWCET to be an iterative process.

On the other hand, the utilization approach only increases the frequency to accommodate the overhead in between two consecutive deadlines, therefore it is not possible to schedule again the task set and the first activation time inside each frame has to remain the same as in the original schedule. Future work includes taking into account the possible memory conflicts while migrating a job on specific memory hierarchies.

Conclusion

7.1 Summary of contributions

The contributions of this Thesis can be gathered under two relevant achievements:

- A method to calculate a CE for a HRT task set on a multicore architecture which manages to maximize utilization leveraging fluid scheduling while minimizing preemptions (i.e. context switching and migration). Besides, the resulting CE accounts for the overhead introduced by preemptions, and is thermal-compliant at minimum energy.
- An runtime close loop control which uses the references provided by the CE to ensure the accomplishment of the time constraints of the HRT task set in the presence of small disturbances and parametric variations, managing the arrival of SRT aperiodic tasks.

The work under these two contributions is based on the following hints.

Instead of a co-design approach, as proposed by Desirena [28], the proposed methodology follows a modular design. This decision allows us to detach the different analysis, and to work from an input-output perspective.

First, the *open loop* scheduling algorithm considers a thermal and energy management. Again, we assume a periodic scheduler to perform a steady state thermal analysis, and on the TCPN model the frequency of the CPUs is stated explicitly as a variable. The thesis formulates two optimization problems upon this trick, alongside the assumption that each CPU execution cycle consumes the same power. The solution to these two problems entails a frequency range of operation, which is thermal compliant. Thus, working at those frequencies while reaching maximum CPU utilization will not overpass a temperature bound. Moreover, the lower bound of the frequency range F^* minimizes the energy consumption. This analysis is derived on Chapter 3, Sec. 3.8.

Upon this offline module, named *Task set conditioner* we can propose a number of scheduling schemes. The *open loop* scheduling algorithm (Chapter 3) is based

on the DP-fair approach, in the sense that the workload distribution analysis is performed on a deadline basis. Then, per each time interval (time between two consecutive deadlines) we pose a ILP, to solve the amount of cycles that each task should execute in order to fulfill their execution requirements. This ILP has the unimodularity property, and therefore it always provides integer solutions when solved with a simplex algorithm. Nevertheless, that ILP formulation was unable to provide solutions for every feasible task set. This drawback is addressed on Chapter 4.

Then, we proposed AIECS a *close loop* scheduling scheme, compatible with the *Task set conditioner*. That also follows a modular design, it computes an off-line schedule which then is used as reference for a continuous controller. In contrast with classic feedback schedulers, the error is computed with respect to the accumulated execution curves, rather than a miss deadline ratio, therefore we can prove HRT constraints. Still this controller acts directly upon on the TCPN model, and its translation to plant parameters was not straightforward. For the off-line schedule we performed a similar approach as in Chapter 3, but we reformulate the ILP to improve schedulability and preserve the unimodularity property.

In a preliminary comparison between the CE, obtained from the off-line stages of AIECS, and RUN it became apparent that AIECS produced less preemptions and migrations than RUN, however the latter achieved zero migrations in many cases. Inspired by this insight we propose a clustering stage on the task set, incorporated on the CAIECS scheduling scheme, from Chapter 5. Later, on the chapter we showed that CAIECS cyclic executive outperformed RUN and AIECS in terms of migrations, whilst preserving the good metrics in number of preemptions from AIECS.

The control stage from CAIECS presents a mayor improvements with respect to AIECS, in terms of simplicity to implement. Furthermore, important adjustment to the TCPN model provided better simulation of the plant.

Finally, we propose two methodologies for the preemptive CE to account for the overhead introduced by preemptions and migrations. This is a novel contributions and aids to avoid over provisioning. Even though, the RT community has move their research towards on line schedules, the industry still relies heavily on off-line ad hoc schedules specially for highly critical systems, as shown in a recent survey to practitioners [2]. Therefore, our scheme provides a solid ground to produce reliable preemptive CE that also accounts for the preemption overheads whilst achieving maximum utilization. In contrast with non-preemptive schedulers.

7.2 Complexity

The computational complexity of the various parts of the proposed system, is briefly review here. The *Task set conditioner* calculates the minimum and maximum frequencies (F^* and F^+). Computing F^* is linear in the number of tasks $\mathcal{O}(n)$. The calculation of F^+ requires solving a non-linear optimization problem, we employ an interior point algorithm. Tests show that it provides a good performance and converges to the optimum in a very short time. The computational complexity of the *Task clustering* (Sec. 5.3.2) is of order $\mathcal{O}(m^2 \times n \times \log(n))$.

Next, the complexity of solving the ILP Eq. (4.9) to compute the workload per frame is linear in $x_{i,k}$, i.e. in $n^2 \times \beta$, where $\beta = \max_{\tau_i} \frac{H}{w_i}$. Thus, it is in the order of $\mathcal{O}(n^2)$. Finally, *EDF* or *ZL* policies run in polynomial time. Hence, all the algorithms in the off-line stage run in polynomial time, but the non-linear optimization.

References $R_{i,j}(\delta_k)$ are pre-computed in the *Get reference* module, and only checked when a context switch occurs, which results on gathering the appropriate execution requirement.

The feedback controller (Sec. 5.4) runs at every sampling period, which depends on the minimum difference between two consecutive $\delta_i - \delta_{i-1}$ from set S . Since the RT clock routine is executed with a fixed period and more frequently than any other RT task, we propose to implement the feedback controller routine as a call back (deferred function, softirq or tasklet depending on the RT operating system), activated at each RT clock routine execution. In this approach we include a feedback controller routine per *CPU*. Thus, their complexity is $\mathcal{O}(1)$, because it requires to solve equations that have a fixed number of operations. A different implementation approach may use a devoted *CPU* for the feedback controller routine.

7.3 Conclusions

The results obtained prove that it is worth designing HRT schedulers based on the principles of fluid scheduling, which maximize utilization, while getting around the traditional overhead and complexity which hampered the practical application of many previous proposals.

The CE calculation and fine-tuning proposed in Chapter 6 is compatible with the application of close-loop controllers, which can increase the resilience of the system and open up the new opportunities commented in the next, final section.

7.4 Future work

Unrealistic worst-case bounds constitute a lingering problem in RT systems, and exacerbates the problem of overprovisioning. This problem keeps lying on the horizon of any research on the field including the two contributions of the Thesis as summarized in Sec. 7.1.

- The first contribution helps reducing overprovisioning by maximizing the utilization while minimizing the preemption activity and, therefore, its overhead. Working on better time bounds related to memory access conflicts besides a better WCET calculation is mandatory, and is orthogonal to this contribution, which is helpful and can still be exploited in any scenario.
- The second contribution leverages close-loop control techniques to ensure the runtime integrity of the execution of the HRT task set upon the occurrence of unexpected events. This approach has traditionally raised concern in the field, as the certifications of RT systems rely on precise worst-case assumptions. However, it represent now a whole new venue to explore in the line of recent proposals encompassing runtime additional hardware to monitor the execution of RT systems so as to find better time bounds on a probabilistic basis, which can be amenable for certification (see e.g. [48, 74, 79])

Conclusiones

Resumen de las contribuciones

Las contribuciones de esta Tesis se pueden agrupar bajo dos logros relevantes:

- Un método para calcular un CE para un conjunto de tareas HRT sobre una arquitectura multinúcleo que logra maximizar la utilización aprovechando la planificación fluida y minimizando las expulsiones (es decir, cambio de contexto y migración). Además, el CE resultante toma en cuenta la sobrecarga introducida por las expulsiones y cumple con cotas térmicas a energía mínima.
- Un control en lazo cerrado en tiempo de ejecución que utiliza las referencias proporcionadas por el CE para garantizar el cumplimiento de las restricciones temporales del conjunto de tareas HRT en presencia de pequeñas perturbaciones y variaciones paramétricas, gestionando la llegada de tareas aperiódicas SRT.

El trabajo detrás de estas dos contribuciones se basa en las siguientes sugerencias.

En lugar de un enfoque de codiseño, como lo propone Desirena [28], la metodología propuesta propone un diseño modular. Esta decisión nos permite separar los diferentes análisis, y trabajar desde una perspectiva entrada-salida.

Primero, el algoritmo de planificación en *lazo abierto* considera una gestión térmica y energética. Nuevamente, asumimos una planificación periódica para realizar un análisis térmico en estado estable, y en el modelo TCPN la frecuencia de las CPUs se establece explícitamente como una variable. La tesis formula dos problemas de optimización basados en este truco, junto con la suposición de que cada ciclo de ejecución de CPU consume la misma potencia. La solución a estos dos problemas proporciona un rango de frecuencias de operación, que cumple con la cota térmica. Por lo tanto, trabajar a esas frecuencias, mientras se está a utilización máxima en las CPUs, no superará una cota de temperatura. Además, el límite inferior del rango de frecuencias, F^* , minimiza el consumo de energía. Este análisis se deriva en el Capítulo 3, Sec. 3.8.

Sobre este módulo fuera de línea, denominado *Condicionador de conjunto de tareas*, se pueden proponer una serie de esquemas de planificación.

Nevertheless, that ILP formulation was unable to provide solutions for every feasible task set. This drawback is addressed on Chapter 4.

El algoritmo de planificación en *lazo abierto* (Capítulo 3) se basa en el enfoque DP-fair, en el sentido de que el análisis de distribución de la carga de trabajo se basa en los plazos de las tareas. Luego, por cada intervalo de tiempo (entre dos plazos consecutivos) planteamos un ILP, para resolver la cantidad de ciclos que debe ejecutar cada tarea para cumplir con sus requisitos de ejecución. Este ILP tiene la propiedad de unimodularidad y, por lo tanto, siempre proporciona soluciones enteras cuando se resuelve con un algoritmo simplex. Sin embargo, esa formulación del ILP no tenía solución para todo conjunto de tareas factibles. Este inconveniente se aborda en el Capítulo 4.

Luego, se propuso AIECS un esquema de planificación en *lazo cerrado*, compatible con el *Task set conditioner*. Este también sigue un diseño modular, calcula una planificación fuera de línea que luego se usa como referencia para un controlador continuo. A diferencia de los planificadores de retroalimentación clásicos, el error se calcula con respecto a las curvas de ejecución acumulada, en lugar de una tasa de incumplimiento de plazos, por lo tanto, se pueden asegurar las restricciones HRT. Aun así, este controlador actúa directamente sobre el modelo TCPN, y su traducción a los parámetros de la planta es sencilla.

Para la planificación fuera de línea, se siguió un enfoque similar al del Capítulo 3, pero se reformuló el ILP para mejorar planificabilidad y preservar la propiedad de unimodularidad.

En una comparación preliminar entre el CE de AIECS, obtenido de sus etapas fuera de línea, y RUN se hizo evidente que AIECS producía menos expulsiones y migraciones que RUN, sin embargo, este último generaba cero migraciones en muchos casos. Inspirándonos en esta comparativa, se propone una etapa de agrupamiento del conjunto de tareas, incorporado en el esquema de planificación CAIECS, Capítulo 5. Más adelante en el capítulo, se muestra que el ejecutivo cíclico de CAIECS superó a RUN y AIECS en términos de migraciones, al mismo tiempo que conservó las buenas métricas de número de expulsiones de AIECS.

La etapa de control de CAIECS presenta una gran mejora con respecto a AIECS, en términos de simplicidad de implementación. Además, el ajuste al modelo TCPN proporciona una mejor simulación del problema.

Finalmente, se proponen dos metodologías para considerar la sobrecarga introducida por las expulsiones y migraciones en el CE apropiativo. Se trata de una aportación novedosa que ayuda a evitar el sobreaprovisionamiento.

A pesar de que la comunidad RT ha movido su investigación hacia los planificadores en línea, la industria todavía depende en gran medida de planificación *ad hoc* fuera de línea, especialmente para sistemas altamente críticos, como se muestra en una encuesta reciente a profesionales [industry survey 2021]. Por lo tanto, nuestro esquema proporciona una base sólida para producir CE apropiativos, que son confiables y que además consideran los costos de sobrecarga por las expulsiones mientras logra utilización máxima, a diferencia de los planificadores no apropiativos.

Complejidad

La complejidad computacional de las diversas partes del esquema propuesto se resumen aquí. El *Acondicionador del conjunto de tareas* calcula las frecuencias mínima y máxima (F^* y F^+). Calcular F^* es lineal en el número de tareas $\mathcal{O}(n)$. El cálculo de F^+ requiere resolver un problema de optimización no lineal, para esto se emplea un algoritmo de punto interior. Las pruebas muestran que proporciona un buen rendimiento y converge al óptimo en muy poco tiempo. La complejidad computacional del *Agrupamiento de las tareas* (Sec. 5.3.2) es del orden de $\mathcal{O}(m^2 \times n \times \log(n))$.

Por otra parte, la complejidad de resolver el ILP Eq. (4.9) para calcular la carga de trabajo por intervalo es lineal en el número de variables $x_{i,k}$, es decir, en $n^2 \times \beta$, donde $\beta = \max_{\tau_i} \frac{H}{w_i}$. Por lo tanto, está en el orden de $\mathcal{O}(n^2)$. Finalmente, las políticas *EDF* o *ZL* se ejecutan en tiempo polinomial. Por lo tanto, todos los algoritmos en la etapa fuera de línea se ejecutan en tiempo polinomial, excepto la optimización no lineal.

Las referencias $R_{i,j}(\delta_k)$ se calculan previamente en el módulo *Obtener referencias* y solo se verifican cuando ocurre un cambio de contexto, lo que sólo implica la recopilación del dato de ejecución adecuado.

El controlador por retroalimentación (Sec. 5.4) se ejecuta en cada período de muestreo, que depende de la diferencia mínima entre dos $\delta_i - \delta_{i-1}$ consecutivos del conjunto S . Dado que la rutina de reloj RT se ejecuta con un período fijo y con más frecuencia que cualquier otra tarea RT, se propone implementar la rutina del controlador como una devolución de llamada (función diferida, `softirq` o `tasklet` dependiendo del sistema operativo RT), activada en cada ejecución de la rutina de reloj RT. En este enfoque, se puede incluir una rutina para el controlador por retroalimentación por *CPU*. Por lo tanto, su complejidad es $\mathcal{O}(1)$, porque requiere resolver ecuaciones que tienen un número fijo de operaciones. Otra implementación diferente puede usar una *CPU* dedicada para la rutina del controlador por retroalimentación.

Conclusiones

Los resultados obtenidos demuestran que vale la pena diseñar planificadores HRT basados en los principios de planificación fluida, que maximizan la utilización, al tiempo que evitan la tradicional sobrecarga y complejidad que han dificultado la aplicación práctica de propuestas anteriores.

El cálculo y ajuste fino del CE propuesto en el Capítulo 6 es compatible con la aplicación de controladores en lazo cerrado, que pueden aumentar la resiliencia del sistema y abrir nuevas oportunidades, que serán comentadas en la siguiente sección.

Trabajo futuro

Las cotas poco realistas del peor-caso de ejecución (WCET) constituyen un problema persistente en los sistemas RT y exacerban el problema del sobreaprovisionamiento. Este problema permanece en el horizonte de cualquier investigación en el campo, incluyendo las dos contribuciones de la Tesis como se resumen en Sec. 7.1.

- La primera contribución ayuda a reducir el aprovisionamiento excesivo al maximizar la utilización mientras se minimizan las expulsiones y, por lo tanto, su sobrecarga. Trabajar en obtener mejores cotas para los tiempos asociados con los conflictos de acceso a memoria, además de un mejor cálculo del WCET

es necesario, y ortogonal a esta contribución, que es útil y aún puede explotarse en cualquier escenario.

- La segunda contribución aprovecha técnicas de control en lazo cerrado para garantizar la íntegra ejecución de las tareas HRT ante la ocurrencia de eventos inesperados. Este enfoque tradicionalmente ha generado preocupación en el campo, ya que las certificaciones de los sistemas RT se basan en suposiciones precisas del peor caso. Sin embargo, ahora se abre un horizonte nuevo para explorar, en conjunto con recientes líneas propuestas que introducen hardware adicional, que en tiempo de ejecución monitorean la ejecución de los sistemas RT a fin de encontrar mejores cotas de tiempo sobre una base probabilística, que se pueden someter a certificación. (ver, por ejemplo, [48, 74, 79])

Bibliography

- [1] Rehan Ahmed, Parameswaran Ramanathan, and Kewal K Saluja. “Necessary and Sufficient Conditions for Thermal Schedulability of Periodic Real-Time Tasks Under Fluid Scheduling Model”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 15.3 (2016), p. 49.
- [2] Benny Akesson et al. “A comprehensive survey of industry practice in real-time systems”. In: *Real-Time Systems* (2021), pp. 1–41.
- [3] J.H. Anderson, V. Bud, and U.C. Devi. “An EDF-based scheduling algorithm for multiprocessor soft real-time systems”. In: *17th Euromicro Conference on Real-Time Systems (ECRTS’05)*. 2005, pp. 199–208. DOI: 10.1109/ECRTS.2005.6.
- [4] James H Anderson and Anand Srinivasan. “Mixed Pfair/ERfair scheduling of asynchronous periodic tasks”. In: *Real-Time Systems, 13th Euromicro Conference on, 2001*. IEEE. 2001, pp. 76–85.
- [5] K-E Arzén et al. “An introduction to control and scheduling co-design”. In: *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No. 00CH37187)*. Vol. 5. IEEE. 2000, pp. 4865–4870.
- [6] S. Baruah, M. Bertogna, and G. Butazzo. *Multiprocessor Scheduling for Real-Time Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2015. ISBN: 978-3-319-08695-8.
- [7] Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. “Fast scheduling of periodic tasks on multiple resources”. In: *IPPS 95*. IEEE. 1995, p. 280.
- [8] Sanjoy K Baruah et al. “Proportionate progress: A notion of fairness in resource allocation”. In: *Algorithmica* 15.6 (1996), pp. 600–625.
- [9] Andrea Bastoni, Bjorn B Brandenburg, and James H Anderson. “Is semi-partitioned scheduling practical?” In: *2011 23rd Euromicro Conference on Real-Time Systems*. IEEE. 2011, pp. 125–135.
- [10] Marko Bertogna and Michele 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.
- [11] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. “Improved Schedulability Analysis of EDF on Multiprocessor Platforms”. In: *Proceedings of the 17th Euromicro Conference on Real-Time Systems*. ECRTS ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 209–218. ISBN: 0-7695-2400-1.

- [12] Enrico Bini and Giorgio C Buttazzo. “Measuring the performance of schedulability tests”. In: *Real-Time Systems* 30.1-2 (2005), pp. 129–154.
- [13] Björn B Brandenburg and James H Anderson. “On the implementation of global real-time schedulers”. In: *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*. IEEE. 2009, pp. 214–224.
- [14] Bjorn B. Brandenburg. “Scheduling and Locking in Multiprocessor Real-time Operating Systems”. PhD thesis. Chapel Hill, NC, USA: University of North Carolina at Chapel Hill, 2011. ISBN: 978-1-267-25618-8.
- [15] Björn B. Brandenburg and Mahircan Gül. “Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi - Partitioned Reservation”. In: *IEEE Real-Time Systems Symposium (RTSS 2016)*. 2016, pp. 99–110.
- [16] John M. Calandrino, James H. Anderson, and Dan P. Baumberger. “A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms”. In: *Proceedings of the 19th Euromicro Conference on Real-Time Systems*. ECRTS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 247–258.
- [17] T. Chantem, X.S. Hu, and R.P. Dick. “Temperature-Aware Scheduling and Assignment for Hard Real-Time Applications on MPSOCs”. In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 19.10 (Oct. 2011), pp. 1884–1897. ISSN: 1063-8210.
- [18] Jian-Jia Chen, Chia-Mei Hung, and Tei-Wei Kuo. “On the minimization of the instantaneous temperature for periodic real-time tasks”. In: *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. IEEE. 2007, pp. 236–248.
- [19] Jian-Jia Chen and Tei-Wei Kuo. “Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems.” In: *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*. Nov. 2007, pp. 289–294.
- [20] Abel Chils Trabanco. “Planificación Tiempo Real en multiprocesadores: reducción de cambios de contexto y migraciones en ALECS”. <https://zaguan.unizar.es/record/106782#>. MA thesis. Universidad de Zaragoza, 2021.
- [21] Abel Chils Trabanco et al. *Tertimuss: Simulation environment for Real-Time Multiprocessor Schedulers*. <https://gaz.i3a.es/tertimuss-simulation-environment-for-thermal-aware-real-time-scheduling/>. 2019-2022. (Visited on 09/01/2022).
- [22] R. David and H. Alla. “Discrete, Continuous and Hybrid Petri Nets”. In: *Control Systems, IEEE* 28.3 (June 2008), pp. 81–84. ISSN: 1066-033X.
- [23] R. I. Davis and A. Burns. “Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems”. In: *2009 30th IEEE Real-Time Systems Symposium*. 2009, pp. 398–409.

- [24] Robert I Davis and Alan Burns. “A survey of hard real-time scheduling for multiprocessor systems”. In: *ACM computing surveys (CSUR)* 43.4 (2011), p. 35.
- [25] Robert I. Davis and Alan Burns. “FPZL Schedulability Analysis”. In: *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. RTAS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 245–256. ISBN: 978-0-7695-4344-4.
- [26] M. L. Dertouzos. “Control Robotics: The Procedural Control of Physical Processes”. In: *Proceedings of IFIP Congress (IFIP'74)*). 1974, pp. 807–813.
- [27] Gaddiel Desirena López et al. “A Flexible Framework for Real-Time Thermal-Aware Schedulers using Timed Continuous Petri Nets”. In: *Computación y Sistemas* 23.2 (2019), pp. 417–433.
- [28] G Desirena-Lopez et al. “Thermal-aware real-time scheduling using Timed Continuous Petri nets”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 18.4 (2019b), p. 36.
- [29] G. Desirena-Lopez et al. “On-line Scheduling in Multiprocessor Systems based on continuous control using Timed Continuous Petri Nets”. In: *13th International Workshop on Discrete Event Systems*. 2016, pp. 278–283.
- [30] G. Desirena-Lopez et al. “Thermal modelling for Temperature Control in MPSoC's Using Fluid Petri Nets”. In: *IEEE Conference on Control Applications part of Multi-conference on Systems and Control*. 2014.
- [31] Calvin Deutschbein et al. “Multi-core cyclic executives for safety-critical systems”. In: *Science of Computer Programming* 172 (2019), pp. 102–116.
- [32] Sudarshan K Dhall and CL Liu. “On a real-time scheduling problem”. In: *Operations research* 26.1 (1978), pp. 127–140.
- [33] Johannes Freitag and Sascha Uhrig. “Closed loop controller for multicore real-time systems”. In: *International Conference on Architecture of Computing Systems*. Springer. 2018, pp. 45–56.
- [34] Xing Fu, Xiaorui Wang, and Eric Puster. “Dynamic thermal and timeliness guarantees for distributed real-time embedded systems”. In: *2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2009, pp. 403–412.
- [35] Yong Fu et al. “Feedback thermal control of real-time systems on multicore processors”. In: *Proceedings of the tenth ACM international conference on Embedded software*. ACM. 2012, pp. 113–122.
- [36] Shelby Funk et al. “DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling”. In: *Real-Time Systems* 47.5 (2011), pp. 389–429.

- [37] A.J. Ghajar and D. Yunus A. Cengel. *Heat and Mass Transfer: Fundamentals and Applications*. McGraw-Hill Education, 2014. ISBN: 9780073398181. URL: <https://books.google.com.mx/books?id=B89MnwEACAAJ>.
- [38] Joeë L Goossens and Shelby Funk. “Priority-driven scheduling of periodic task systems on multiprocessors”. In: *Real-Time Systems* (2003), pp. 2–3.
- [39] David Griffin, Iain Bate, and Robert I Davis. “Generating utilization vectors for the systematic evaluation of schedulability tests”. In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2020, pp. 76–88.
- [40] Pradeep M. Hettiarachchi et al. “A design and analysis framework for thermal-resilient hard real-time systems”. In: *ACM Transactions on Embedded Computing Systems* 13.5s (2014), 146:1–146:25.
- [41] Alan J Hoffman and Joseph B Kruskal. “Integral boundary points of convex polyhedra”. In: *50 Years of integer programming 1958-2008*. Springer, 2010, pp. 49–76.
- [42] IEEE Technical Community on Real-Time Systems. *Terminology and notation: Definitions related to a computation*. <https://cmte.ieee.org/tcrts/education/terminology-and-notation/> - Last visited 2022-08-22.
- [43] Intel. *ECU consolidation reduces vehicle cost, weight and testing*. URL: <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ecu-consolidation-white-paper.pdf> (visited on 05/01/2021).
- [44] David Johnson. “Fast Algorithms for Bin Packing”. In: *Journal of Computer and System Sciences* 8 (June 1974), pp. 272–314.
- [45] Hassan K Khalil. *Nonlinear systems*. Upper Saddle River, 2002.
- [46] Joonho Kong, Sung Woo Chung, and Kevin Skadron. “Recent thermal management techniques for microprocessors”. In: *ACM Computing Surveys* 44.3 (2014), 13:1–13:42.
- [47] Philip Kosky et al. “Chapter 14 - Mechanical Engineering”. In: *Exploring Engineering (Fifth Edition)*. Ed. by Philip Kosky et al. Fifth Edition. Academic Press, 2021, pp. 317–340. ISBN: 978-0-12-815073-3. DOI: <https://doi.org/10.1016/B978-0-12-815073-3.00014-4>. URL: <https://www.sciencedirect.com/science/article/pii/B9780128150733000144>.
- [48] Leonidas Kosmidis et al. “Fitting processor architectures for measurement-based probabilistic timing analysis”. In: *Microprocessors and Microsystems* 47 (2016), pp. 287–302. ISSN: 0141-9331. DOI: <https://doi.org/10.1016/j.micpro.2016.07.014>. URL: <https://www.sciencedirect.com/science/article/pii/S0141933116300977>.
- [49] John Lehoczky, Lui Sha, and Yuqin Ding. “The rate monotonic scheduling algorithm: Exact characterization and average case behavior”. In: *RTSS*. Vol. 89. 1989, pp. 166–171.

- [50] Chung Laung Liu and James W Layland. “Scheduling algorithms for multiprogramming in a hard-real-time environment”. In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.
- [51] J.M. López, J.L. Díaz, and D.F. García. “Utilization bounds for EDF scheduling on real-time multiprocessor systems”. In: *Journal of Real Time Systems* 28.1 (2004), pp. 39–68.
- [52] Silvano Martello and Paolo Toth. “Bin-packing problem”. In: *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990, pp. 221–245.
- [53] A. Mascitti, T. Cucinotta, and L. Abeni. “Heuristic partitioning of real-time tasks on multi-processors”. In: *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*. 2020, pp. 36–42. DOI: 10.1109/ISORC49007.2020.00015.
- [54] E. Massa, G. Lima, and P. Regnier. “Revealing the Secrets of RUN and QPS: New Trends for Optimal Real-Time Multiprocessor Scheduling”. In: *2014 Brazilian Symposium on Computing Systems Engineering*. 2014, pp. 150–155.
- [55] Ernesto Massa et al. “Quasi-partitioned Scheduling: Optimality and Adaptation in Multiprocessor Real-time Systems”. In: *Real-Time Syst.* 52.5 (Sept. 2016), pp. 566–597. ISSN: 0922-6443.
- [56] Aloysius Ka-Lau Mok. “Fundamental design problems of distributed systems for the hard-real-time environment”. PhD thesis. Massachusetts Institute of Technology, 1983.
- [57] S. Murali et al. “Temperature-aware processor frequency assignment for MP-SoCs using convex optimization”. In: *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2007 5th IEEE/ACM/IFIP International Conference on*. 2007, pp. 111–116.
- [58] Tadao Murata. “Petri nets: Properties, analysis and applications”. In: *Proceedings of the IEEE* 77.4 (1989), pp. 541–580.
- [59] Dong-Ik Oh and T. P. Bakker. “Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignments”. In: *Real-Time Systems* 15.2 (1998), pp. 183–192.
- [60] Paul Regnier et al. “Multiprocessor scheduling by reduction to uniprocessor: an original optimal approach”. In: *Real-Time Systems* 49.4 (July 2013), pp. 436–474.
- [61] Paul Regnier et al. “RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor”. In: *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*. RTSS ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 104–115. ISBN: 978-0-7695-4591-2.

- [62] L Rubio-Anguiano et al. “Energy-efficient thermal-aware multiprocessor scheduling for real-time tasks using TCPN”. In: *Discrete Event Dynamic Systems* (2019), pp. 1–28.
- [63] L Rubio-Anguiano et al. “Energy-Efficient Thermal-Aware Scheduling for RT Tasks Using TCPN”. In: *IFAC-PapersOnLine* 51.7 (2018), pp. 236–242.
- [64] L. Rubio-Anguiano et al. “Real time scheduler for multiprocessor systems based on continuous control using Timed Continuous Petri Nets”. In: *IFAC-PapersOnLine* 53.4 (2020). 15th IFAC Workshop on Discrete Event Systems WODES 2020 — Rio de Janeiro, Brazil, 11-13 November 2020, pp. 371–377. ISSN: 2405-8963. DOI: <https://doi.org/10.1016/j.ifacol.2021.04.036>. URL: <https://www.sciencedirect.com/science/article/pii/S240589632100077X>.
- [65] Laura E. Rubio-Anguiano, José Luis Briz, and Antonio Ramírez-Treviño. “Accounting for Preemption and Migration Costs in the Calculation of Hard Real-Time Cyclic Executives for MPSoCs”. In: *2022 IEEE 18th International Conference on Automation Science and Engineering (CASE)*. 2022.
- [66] Laura E. Rubio-Anguiano, José Luis Briz, and Antonio Ramírez-Treviño. “Accounting for Preemption and Migration Costs in the Calculation of Hard Real-Time Cyclic Executives for MPSoCs”. In: *IEEE Robotics and Automation Letters* 7.3 (2022), pp. 7990–7997. DOI: 10.1109/LRA.2022.3186489.
- [67] Laura Elena Rubio-Anguiano et al. “Maximizing utilization and minimizing migration in thermal-aware energy-efficient real-time multiprocessor scheduling”. In: *IEEE Access* 9 (2021), pp. 83309–83328.
- [68] Deepak R Sahoo et al. “Feedback control for real-time scheduling”. In: *Proceedings of the 2002 American Control Conference (IEEE Cat. No. CH37301)*. Vol. 2. IEEE. 2002, pp. 1254–1259.
- [69] Lars Schor et al. “Worst-case temperature guarantees for real-time applications on multi-core systems”. In: *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*. IEEE. 2012, pp. 87–96.
- [70] Lui Sha et al. “Real time scheduling theory: A historical perspective”. In: *Real-time systems* 28.2-3 (2004), pp. 101–155.
- [71] Shi Sha et al. “Thermal-constrained energy efficient real-time scheduling on multi-core platforms”. In: *Parallel Computing* 85 (2019), pp. 231–242. ISSN: 0167-8191. DOI: <https://doi.org/10.1016/j.parco.2019.01.003>. URL: <https://www.sciencedirect.com/science/article/pii/S0167819118300280>.
- [72] Gerard Sierksma. *Linear and integer programming: theory and practice*. CRC Press, 2001.

- [73] M. Silva et al. “On fluidization of discrete event models: observation and control of continuous Petri nets”. In: *Discrete Event Dynamic Systems* 21(4).3 (Dec. 2011), pp. 427–497.
- [74] Mladen Slijepcevic et al. “pTNoC: Probabilistically Time-Analyzable Tree-Based NoC for Mixed-Criticality Systems”. In: *2016 Euromicro Conference on Digital System Design (DSD)*. 2016, pp. 404–412. DOI: 10.1109/DSD.2016.23.
- [75] Alberto Soria-Lopez, Pedro Mejia-Alvarez, and Julio Cornejo. “Feedback scheduling of power-aware soft real-time tasks”. In: *Sixth Mexican International Conference on Computer Science (ENC’05)*. IEEE. 2005, pp. 266–273.
- [76] Anand Srinivasan. *Efficient and flexible fair scheduling of real-time tasks on multiprocessors*. The University of North Carolina at Chapel Hill, 2003.
- [77] Mason Thammawichai and Eric C. Kerrigan. “Energy-efficient real-time scheduling for two-type heterogeneous multiprocessors”. In: *Real-Time Systems* 54 (2018).
- [78] José Luis Tovany et al. “Greenhouse modeling using continuous timed Petri nets”. In: *Mathematical problems in engineering* 2013 (2013).
- [79] David Trilla et al. “Randomization for Safer, more Reliable and Secure, High-Performance Automotive Processors”. In: *IEEE Design & Test* 36.6 (2019), pp. 39–47. DOI: 10.1109/MDAT.2019.2927373.
- [80] Klaus Truemper. *Matroid decomposition*. Vol. 6. Boston: Academic Press, 1992.
- [81] Wickström Ulf. “Heat Transfer by Radiation”. In: *Temperature Calculation in Fire Safety Engineering*. Springer, 2016. Chap. 5. DOI: 10.1007/978-3-319-30172-3_5.
- [82] K. Vipin. “CANNoC: An open-source NoC architecture for ECU consolidation”. In: *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*. Aug. 2018, pp. 940–943.

Tertimuss

Tertimuss is an open-source framework to model a RT multiprocessor system, simulate different RT schedulers, and process the results. It is publicly available at [21].

Architecture

In its first version, Tertimuss was designed to be used by commands through the command line, or a graphical interface. These methods in practice were of little use, rather Tertimuss is used as a set of libraries within Python scripts in which the simulation to be performed is defined. This new orientation does not exclude the future possibility of adding interfaces to the simulation environment. Organized as a set of libraries, Tertimuss allows the simulations to execute faster. Environment development is also simplified, since both the graphical interface and the command line interface require maintenance whenever new features are added. As drawback, this new design requires users familiar with Python syntax.

Components

Tertimuss components are detailed in figure A.1, and explained below.

- Task generator. It generates synthetic periodic tasks from a specification of the system configuration. Allows you to choose different algorithms to perform this operation.
- Scheduler simulation. It computes a schedule from a description of the system and of a set of tasks, may also provide the temperature evolution in the system. It relies on the scheduler and the thermal simulator.
- Scheduler. It reacts to different simulation events and responds accordingly with the assignment of tasks to CPUs, the change of frequency in the processing cores or the event generation request at an agreed time in the future.

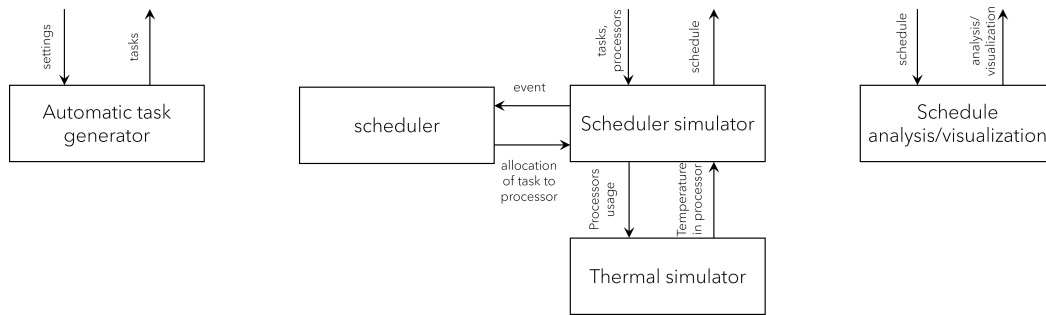


Figure A.1.: Main components of Tertimuss

- Thermal simulator. It calculates the temperature reached by a system A after executing a set of tasks \mathcal{T} during a given time interval.
- Schedule analysis. It extracts information from a schedule, and might generate visualizations of it. This includes different analysis techniques as well as different types of visualizations.

Modeling methodology using TCPNs

In this appendix we present a methodology based on TCPNs to model physical systems described by Parabolic partial differential equation (PDE).

It is intended as a tutorial, and a bridge between Timed Continuous Petri Net (TCPN) and PDE to obtain a mathematical model that has both a representation in ordinary differential equations and in TCPN, and perform control with well known techniques.

First, we briefly introduce a PDE, then a spatial discretization to derive a TCPN elementary module, B.2-B.3. The same procedure is performed for the boundary conditions, B.4. Then in Sec. B.5, we describe how to build the model, from the elementary modules without describing nor solving the PDE, that also describes the physical system. Finally in Sec. B.6, a heat transfer problem is shown to exemplify the methodology.

B.1 Parabolic partial differential equations

Recall that a parabolic PDE describes the change of the physical quantity with respect of time and position variables *inside the medium*. The the boundary conditions relate the interactions with the environment at the *surfaces* of the system.

A parabolic PDE can be expressed as,

$$\frac{1}{\alpha} \frac{\partial g(x, \tau)}{\partial \tau} = \frac{\partial^2 g(x, \tau)}{\partial x^2} + f(x, \tau), \quad (\text{B.1})$$

subject to appropriate initial and boundary conditions. To avoid confusions with the notation t for transitions, the time will be expressed by symbol τ . Then α is a constant, $\tau \in [0, \infty)$, $x \in \mathbb{R}^n$ and $g(x, \tau) : \mathbb{R}^n \times [0, \infty) \rightarrow \mathbb{R}$. If Eq.(B.1) is in its homogeneous form $f(x, \tau) = 0$, else $f(x, \tau) : \mathbb{R}^n \times [0, \infty) \rightarrow \mathbb{R}$.

Therefore, we first explain how to analyze the system *inside the medium* by approximating the parabolic PDE (B.1) to a set of linear first order differential equations. We later derive the TCPN elementary modules from this set.

B.2 Spatial discretization

In order to accomplish this aforementioned approximation, we resort to the traditional finite difference method on the right hand side of Eq. (B.1). To this purpose, the derivatives are replaced by differences. Now suppose that Eq. (B.1) is a one-dimensional PDE, in which x is discretized into $N + 1$ sections $0, \dots, n - 1, n, n + 1, \dots, N$, with only one point of interest into each section, as shown on Figure B.1.

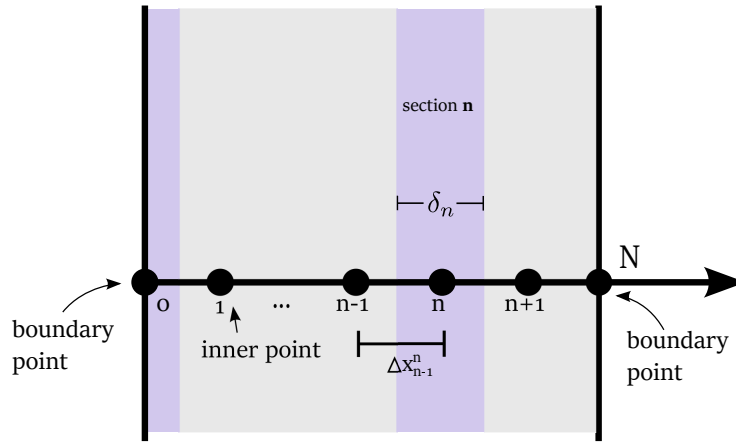


Figure B.1.: Points and sections for the finite difference formulation

Assume that the thickness of each section n is equal to δ_n for all $n \in \{0, 1, \dots, N\}$, and the separation between each pair of points $n - 1$ and n is equal to $\Delta x_{n-1}^n = \Delta x_n^{n-1}$. Then, defining $g_n(\tau) := g(n, \tau)$ for a fixed $n \in \{0, \dots, N\}$ and using a difference approximation on the partial derivatives of $g(x, \tau)$ along x , we can analyze $g(x, \tau)$ by means of $N + 1$ functions $g_n(\tau)$.

To approximate the element $\alpha \frac{\partial^2 g}{\partial x^2}$ at section n (see Fig. B.1), we first compute the approximations to the first order derivative at the mid-points $n - 1/2$ and $n + 1/2$ as follows:

$$\frac{\partial g(x, \tau)}{\partial x} \Big|_{n-1/2} \approx \frac{g_n - g_{n-1}}{\Delta x_{n-1}^n} \quad \text{and} \quad \frac{\partial g(x, \tau)}{\partial x} \Big|_{n+1/2} \approx \frac{g_{n+1} - g_n}{\Delta x_{n+1}^n} \quad (\text{B.2})$$

Then, the finite difference approximation of the second derivative of $g(x, \tau)$ at section n can be computed using Eq. (B.2) as:

$$\begin{aligned} \left. \frac{\partial^2 g(x, \tau)}{\partial x^2} \right|_n &\approx \frac{\frac{g_{n+1} - g_n}{\Delta x_{n+1}^n} - \frac{g_n - g_{n-1}}{\Delta x_{n-1}^n}}{\delta_n} \\ &= \frac{g_{n+1} - g_n}{\delta_n \Delta x_{n+1}^n} - \frac{g_n - g_{n-1}}{\delta_n \Delta x_{n-1}^n}. \end{aligned} \quad (\text{B.3})$$

Hence, the time derivative of $g_n(\tau)$ is:

$$\begin{aligned} \dot{g}_n &= \alpha \left\{ \frac{g_{n-1}}{\delta_n \Delta x_{n-1}^n} - \frac{g_n}{\delta_n \Delta x_{n-1}^n} - \frac{g_n}{\delta_n \Delta x_{n+1}^n} + \frac{g_{n+1}}{\delta_n \Delta x_{n+1}^n} \right\} + f_n(\tau), \\ \forall n &\in \{1, \dots, N-1\}. \end{aligned} \quad (\text{B.4})$$

Eq. (B.4) implies that the rate of change with respect to time of some inner point $g_n(\tau)$, depends on itself and on the two neighboring places plus some function of time $f_n(\tau)$.

For simplicity, assume that $f_n(\tau) = 0 \forall n, \tau$ for now. Fig. B.2 shows the TCPN for three interior places. The marking in each place p_n corresponds to the value of $g_n(\tau)$.

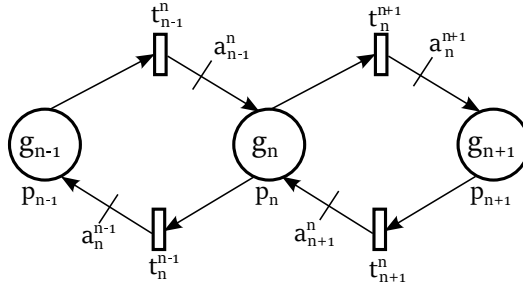


Figure B.2.: TCPN representation for interior places

Therefore the dynamics of g_n is given by,

$$\dot{g}_n = a_{n-1}^n \lambda_{n-1}^n g_{n-1} - (\lambda_n^{n+1} + \lambda_n^{n-1}) g_n + a_{n+1}^n \lambda_{n+1}^n g_{n+1} \quad (\text{B.5})$$

The values for the arc's weights and the firing rates are chosen such that Eq.(B.5) resembles Eq.(B.4). Therefore,

$$a_{n-1}^n = \frac{\lambda_n^{n-1}}{\lambda_{n-1}^n}, \quad \lambda_{n-1}^n = \frac{\alpha}{\delta_{n-1} \Delta x_{n-1}^n}, \quad \lambda_n^{n+1} = \frac{\alpha}{\delta_n \Delta x_{n+1}^n},$$

$$a_{n+1}^n = \frac{\lambda_n^{n+1}}{\lambda_{n+1}^n}, \quad \lambda_n^{n-1} = \frac{\alpha}{\delta_n \Delta x_{n-1}^n}, \quad \lambda_{n+1}^n = \frac{\alpha}{\delta_{n+1} \Delta x_{n+1}^n}.$$

Now, assume that $f_n(\tau) = \text{constant} \forall n, \tau$, then we can model this behavior as the TCPN shown in Fig. B.3, such that the firing rate λ_f of transition t_f is equal to f_n . i.e, $\lambda_f = \alpha f_n(\tau)$.

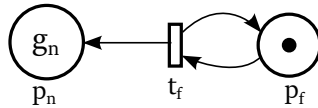


Figure B.3.: TCPN representation for $f_n(t)$ constant

Due to the superposition characteristic of TCPNs, by merging the TCPN module from Fig. B.3 to the TCPN in Fig. B.2 at place p_n , we obtain the same dynamics as in Eq. (B.4). In a heat transfer problem $f(\tau)$ usually corresponds to a heat generation factor. With these insights we can define a TCPN elementary module for the components of the PDE.

B.3 Elementary TCPN module

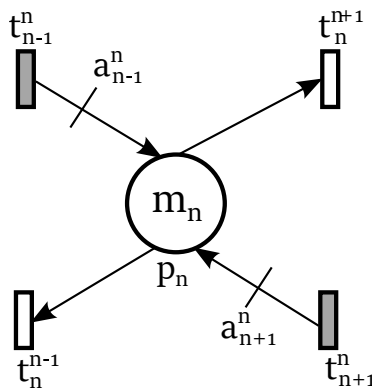


Figure B.4.: TCPN elementary module

The elementary TCPN module in Fig. B.4 is defined for every inner point inside a section, where sections are defined by domain discretization, as explained before.

Each inner TCPN module is composed by a place p_n , whose marking m_n relates to the quantity g_n ; two transitions t_n^{n-1} and t_n^{n+1} , which link p_n with its neighbors p_{n-1} and p_{n+1} , respectively, and other two transitions t_{n-1}^n and t_{n+1}^n . The former connects p_{n-1} to p_n , while the latter connects p_{n+1} to p_{n-1} , such that there exists a marking exchange.

The firing rates for transitions with p_n as pre-place are:

$$\lambda_n^{n+1} = \frac{\alpha}{\delta_n \Delta x_{n+1}^n}, \quad \lambda_n^{n-1} = \frac{\alpha}{\delta_n \Delta x_{n-1}^n}$$

The shadowed transitions in Fig. B.4 are defined by characteristics of the neighboring elementary modules from p_{n-1} and p_{n+1} . The arc's weights a_{n-1}^n and a_{n+1}^n are computed as:

$$a_{n-1}^n = \frac{\lambda_n^{n-1}}{\lambda_{n-1}^n}, \quad a_{n+1}^n = \frac{\lambda_n^{n+1}}{\lambda_{n+1}^n} \quad (\text{B.6})$$

In order to interconnect several inner elements we should merge the transitions.

The discretization discussed on Sec. B.2 was performed on one dimension only. The three-dimensional discretization is carried out following the same procedure, such that the dynamics per each point on the grid is the sum of the discretization performed in each dimension, just as in a typical finite-difference application. Therefore, the inner TCPN module will have an extra couple of transitions per dimension, in order to connect to the neighboring places.

Up to this point, we have showed how to represent elements of the PDE as a TCPN module. Next, we will cover how to write the corresponding boundary conditions.

B.4 Boundary conditions

In this section, we derive elementary TCPN modules for the Dirichlet, Neumann, Mixed and Robin boundary conditions (BC) for PDE (B.1). For example, common boundary conditions in heat transfer problems are convection and a specified surface

temperature. Each boundary condition can be translated into a TCPN representation.

B.4.1 Dirichlet boundary conditions

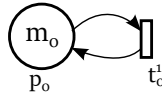


Figure B.5.: TCPN elementary module for Dirichlet BC

This type of boundary conditions specify a value at a surface point for every time (Eq. (B.1)). Considering PDE (B.1) in the domain $[0, 1]$, a Dirichlet boundary condition is,

$$g(0, \tau) = a, \quad g(1, \tau) = b \quad (\text{B.7})$$

where a and b are constant values. In a context of heat transfer, a Dirichlet boundary condition corresponds to an specified temperature at the surface.

The Dirichlet BC is represented by the TCPN in figure B.5, the marking m_0 at place p_0 corresponds to the constant boundary value a as in eq.(B.7) The TCPN state equation for place p_0 is,

$$\dot{m}_0 = \lambda_0^1 m_0 - \lambda_0^1 m_0 = 0 \quad (\text{B.8})$$

λ_0^1 is the firing rate of transition t_0^1 :

$$\lambda_0^1 = \frac{\alpha}{\delta_0 \Delta_0^1} \quad (\text{B.9})$$

where α is the same constant as in Eq. (B.1), and δ_0 and Δ_0^1 are the discretization lengths we defined in Sec. B.2 (Fig. B.1). Transition t_0^1 directly connects to the elementary TCPN for inner points by merging it with the transition defined on the inner element.

B.4.2 Neumann boundary condition

The Neumann boundary condition provides the value of the gradient of the dependent variable normal to the boundary, $\partial u/\partial n = q$. In a one-dimensional case for PDE (B.1), the Neumann boundary condition is expressed as,

$$\frac{\partial g(0, \tau)}{\partial x} = q_0, \quad \frac{\partial g(1, \tau)}{\partial x} = q_1. \quad (\text{B.10})$$

under the assumption that $x \in [0, 1]$, where q_0 and q_1 are given constants. The Neumann boundary condition is often viewed as a flux boundary condition.

The derivation of the TCPN module for this boundary condition comes from the finite difference approximation of Eq. (B.1), such that the dynamics of point g_0 yields:

$$\dot{g}_0 = \alpha \left(\frac{\frac{\partial g_1}{\partial x} - \frac{\partial g_0}{\partial x}}{\delta_0} \right) + \alpha f$$

Without loss of generality, assume $f = 0$. We have already shown how to add this term on the analysis. From the BC in Eq.(B.10) we know that $\frac{\partial g_0}{\partial x} = q_0$, therefore the dynamics of marking m_0 that represents the quantity g_0 is:

$$\dot{m}_0 = \alpha \left(\frac{m_1 - m_0}{\delta_0 \Delta_0^1} \right) + \alpha \frac{q_0}{\delta_0} \quad (\text{B.11})$$

The first term in Eq. (B.11) reflects the conduction-diffusion due to the second order derivatives, where m_1 is the marking of a neighbor place p_1 , and the second term is the imposition of the Neumann boundary condition. The later Neumann term can be modeled as we did for $f_n(\tau)$ on Fig. B.3.

Then, the TCPN representation for Eq. (B.11) is shown in Fig. B.6a, and the elementary TCPN module for the Neumann boundary condition is illustrated on Fig. B.6b. The firing rate for transition t_{Ne} is:

$$\lambda_{Ne} = \frac{\alpha}{\delta_0} q_0 \quad (\text{B.12})$$

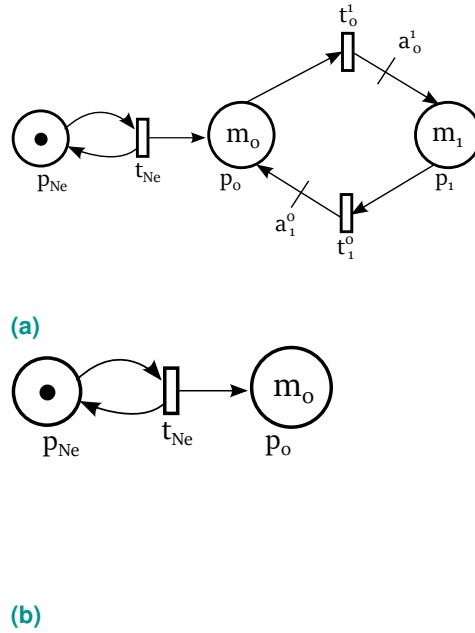


Figure B.6.: TCPN representation for the boundary point dynamics \dot{g}_0 , on the left , and on the right the TCPN elementary module for Neumann boundary conditions

B.4.3 Robin boundary condition

The Robin boundary condition is a linear combination of the two previous boundary conditions. For example in PDE (B.1), for g on a domain G with boundary ∂G , the Robin condition states: $ag + b\partial g/\partial n = q$ on ∂G , for non-zero constants (a, b) and a defined function q . For the one-dimensional case of PDE (B.1) where $x \in [0, 1]$, a Robin boundary condition yields,

$$ag(0, \tau) + b\frac{\partial g(0, \tau)}{\partial x} = q_0, \quad ag(1, \tau) + b\frac{\partial g(1, \tau)}{\partial x} = q_1. \quad (\text{B.13})$$

for given a, b, q_0 and q_1 constants. The Mixed boundary condition considers that the solution of the PDE must satisfy different boundary conditions on disjoint parts of the boundary domain. For the one-dimensional case of PDE (B.1) with $x \in [0, 1]$, it will entail that $g(0, \tau)$ is subjected to a boundary condition different to the one at $g(1, \tau)$. For example,

$$g(0, \tau) = a, \quad \frac{\partial g(1, \tau)}{\partial x} = q. \quad (\text{B.14})$$

The analysis to derive its TCPN representation is the same as in the Neumann boundary condition case that we conclude in Eq. (B.11). However, in this case $\partial g_0 / \partial x = \frac{1}{b}(q_0 - ag(0, \tau))$, therefore:

$$\dot{g}_0 = \alpha \left(\frac{g_1 - g_0}{\delta_0 \Delta_0^1} \right) + \frac{\alpha}{\delta_0 b} (g_0 - ag_0) \quad (\text{B.15})$$

Then, the elementary TCPN module for the Robin boundary condition is illustrated on Fig. B.7, and the firing rates for transitions t_2^R and t_R^2 are:

$$\lambda_2^R = \frac{\alpha a}{\delta_0 b}, \quad \lambda_R^2 = \frac{\alpha}{\delta_0 b} \quad (\text{B.16})$$

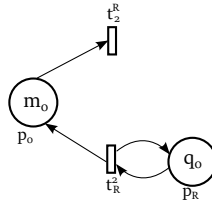


Figure B.7.: TCPN elementary module for Robin boundary condition

B.4.4 Mixed boundary condition

The case of mixed boundary conditions is straightforward. One should use the defined TCPN elementary modules accordingly.

B.5 Building the model

Once the elementary modules have been defined, we can build the model to represent the system. The following steps describe this process:

1. Define a spatial discretization, as described on Fig. B.1

2. Identify the boundary and inner points
3. Create a module for each point in the spatial discretization
 - a) Associate an inner module to each inner point
 - b) Associate a Dirichlet BC module to the boundary points subject to constant defined values
 - c) Associate a Neumann BC module to the points under input or output fluxes
 - d) Associate a Robin BC module to the boundary point usually related to convection-diffusion phenomena
4. Merge the constructed modules on the common transitions
5. Identify the unknown parameters

Following the previous procedure we obtain the global model as:

$$\dot{m} = C\Lambda\Pi_T(m)m \quad (\text{B.17})$$

where m is the value for the quantity under study on each grid point, C , Λ , and $\Pi(m)$ are the incidence matrix, the firing rate transitions and the configuration matrix of the grid elements. The TCPN system has only one configuration matrix because every transition has at most one input place. If we define some places as inputs, for example those related to input fluxes or to actuators, the model can be expressed as:

$$\dot{m} = C\Lambda\Pi_T(m)m + C_{in}f_{in} \quad (\text{B.18})$$

where C_{in} is the incidence matrix for the input transitions and f_{in} is the input flow. The model can also be written in state space form as:

$$\begin{aligned} \dot{m}_T &= Am + Bf_{in} \\ Y_T &= Sm_T \end{aligned} \quad (\text{B.19})$$

where $A = C\Lambda\Pi(m)$ and $B = C_{in}$. The output of the system is Y_T , such that S holds the measurable states.

B.6 Heat transfer application

In this section we provide some examples of the proposed methodology. We represent a variety of heat transfer problems with TCPN modules. This section generalizes the results from [78] and [30].

There are three physical phenomena in a typical heat transfer problem: heat conduction, convection and radiation. Heat conduction occurs in a medium and it is described as in Eq. (B.4), whereas convection and radiation are representatives of boundary conditions. The convection boundary condition is a type of Robin BC, while the radiation will be analysed as a type of Neumann BC. Also, there can be heat generation due to no modeled dynamics, but the heat generation factor is known.

B.6.1 Heat conduction

Conduction is the transfer of energy from more energetic particles of substance to the adjacent less energetic ones [37]. It is described by Eq. (B.1), where the constant α is referred as the *thermal diffusivity* of the material and represents how fast heat propagates through the material. The *thermal diffusivity* relates the thermal conductivity k , the density ρ and the specific heat c of the material, such that:

$$\alpha = \frac{k}{\rho c} \quad (\text{B.20})$$

Therefore, after performing a domain discretization as proposed in Sec. B.2, an inner point has the TCPN module shown in Fig. B.8.

The values for the firing rates are:

$$\lambda_n^{n+1} = \frac{\alpha}{\delta_n \Delta x_{n+1}^n}, \quad \lambda_n^{n-1} = \frac{\alpha}{\delta_n \Delta x_{n-1}^n}$$

where α is the thermal diffusivity, δ_n is the thickness of element n and Δx_{n+1}^n is the distance between element n and element $n + 1$. The firing rates for the elements k and $k + 1$ are derived similarly. Arc weights are define as in Eq. (B.6):

$$a_{n-1}^n = \frac{\lambda_n^{n-1}}{\lambda_{n-1}^n}, \quad a_{n+1}^n = \frac{\lambda_n^{n+1}}{\lambda_{n+1}^n} \quad (\text{B.21})$$

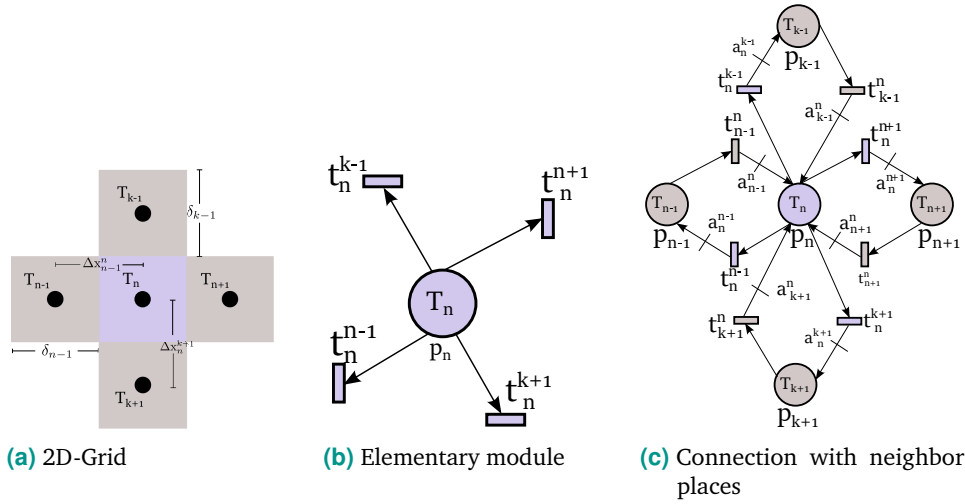


Figure B.8.: Heat conduction TCPN representation

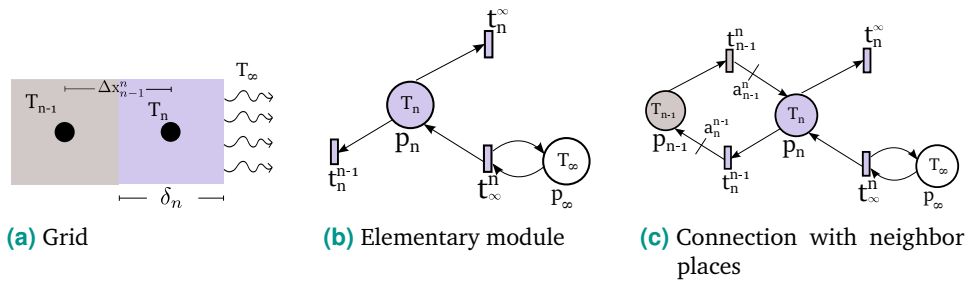


Figure B.9.: TCPN module for a convection boundary condition

B.6.2 Convection

A convection boundary condition is a type of Robin BC, and it is described by Newton's cooling law, such that the heat flux at the boundary point n is:

$$\frac{\partial T(n, \tau)}{\partial x} = -\frac{h}{k}[T_\infty - T(n, \tau)], \quad (\text{B.22})$$

where h is the convection heat transfer coefficient, T_∞ is the ambient temperature and k the thermal conductivity. Therefore, any element boundary element subject to convection can be modeled as in Fig. B.9, with the corresponding firing rates:

$$\lambda_n^\infty = \lambda_n^n = \frac{\alpha h}{\delta_n k}, \quad \lambda_n^{n-1} = \frac{\alpha}{\delta_n \Delta x_{n-1}^n} \quad (\text{B.23})$$

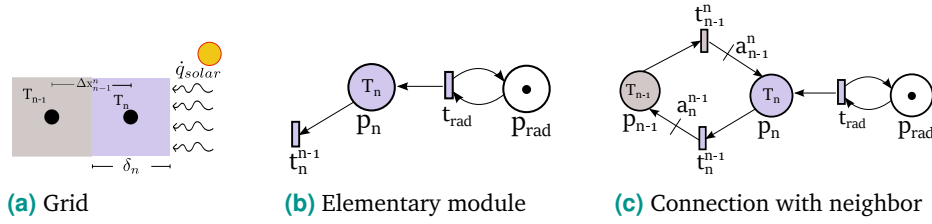


Figure B.10.: TCPN module for a radiation boundary condition

B.6.3 Radiation

In this example, radiation is always considered as an input heat flux, and it is described as a Neumann BC. This just constitutes a convenient simplification here; for a deeper understanding please refer to [37, 81]. In the case of *solar radiation* incident at a boundary surface, we have that:

$$\frac{\partial T(n, \tau)}{\partial x} = \frac{\epsilon}{k\alpha_s} \dot{q}_{solar} \quad (\text{B.24})$$

where α_s is the absorptivity ($0 \leq \alpha_s \leq 1$), ϵ is the emmissivity and \dot{q}_{solar} is the incident solar heat flux. Fig. B.10 shows a boundary element subject to solar radiation, with firing rates:

$$\lambda_{rad} = \frac{\alpha\epsilon}{\delta_n k\alpha_s} \dot{q}_{solar}, \quad \lambda_n^{n-1} = \frac{\alpha}{\delta_n \Delta x_{n-1}^n} \quad (\text{B.25})$$

B.6.4 Heat generation

Heat generation is represented as an energy generation rate inside an element, expressed in $[W/m^3]$. From Eq.(B.1), it implies that $f(\tau) \neq 0$. Therefore, it can be modeled as the TCPN from Fig. B.3, such that

$$\lambda_f = \lambda_{gen} = \frac{\alpha}{k} \dot{e}_{gen} \quad (\text{B.26})$$

where \dot{e}_{gen} is the heat generation rate.

B.6.5 Building the model

Once the previous elements have been set, the global model can be formulated as:

$$\dot{m}_T = C_T \Lambda_T \Pi_T(m) m_T + C_a \Lambda_a \Pi_a(m) m_a + C_{gen} f_{gen} \quad (\text{B.27})$$

where m_T is the temperature for the grid points, m_a is the ambient temperature T_∞ and f_{gen} is the input heat flux due to heat generation. C_x , Λ_x , and $\Pi_x(m)$ are the incidence matrix, the firing rate transitions and the configuration matrix ($x = \{T, a, gen\}$) of the grid elements, ambient temperature, and heat generation subnets respectively.

The model can be also written in state space form as:

$$\begin{aligned} \dot{m}_T &= A m_T + B' m_a + B f_{gen} \\ Y_T &= S m_T \end{aligned} \quad (\text{B.28})$$

where $A = C_T \Lambda_T \Pi_T(m)$, $B = C_{gen}$ and $B' = C_a \Lambda_a \Pi_a(m)$. The system output is Y_T , and S holds the measurable states.

Example

To better understand the previous methodology to model a heat transfer problem, we present the following simple example. Consider the transient heat conduction problem in a large Uranium Plate, as in [37]. The plate thickness is $L = 4$ with *thermal conductivity* $k = 28 \text{ W/mK}$ and *thermal diffusivity* $\alpha = 12.5 \times 10^{-6} \text{ m}^2/\text{s}$. Heat is uniformly generated at a constant rate of $\dot{e} = 5 \times 10^6 \text{ W/m}^3$. One side of the plate is maintained at 0°C at all times, while the other side is subject to convection to an environment at $T_\infty = 30^\circ\text{C}$ with a heat convection coefficient of $h = 45 \text{ W/m}^2\text{K}$, as shown in Fig. B.11.

Fig. B.11 shows that there is only one interior point of interest $T_1(\tau)$ and two boundary points T_0 and T_2 . The boundary condition for $T_0(\tau)$ is a specific temperature boundary condition, i.e $\dot{T}_0 = 0$, $T_0(0) = 0$. And the second boundary condition is that $T_2(\tau)$ is subjected to convection. Therefore, the TCPN model for this problem can be formed as the union of 1) an interior point element, 2) a specified tempera-

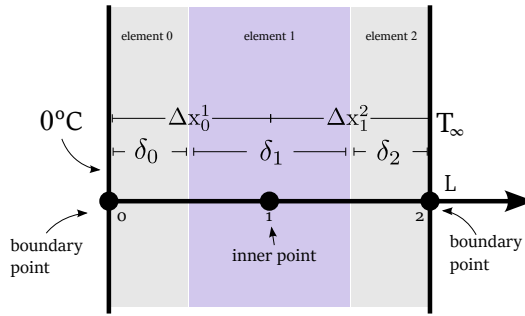


Figure B.11.: Schematic for Example B.6.5

ture element and 3) a convection module, plus the heat generator modules, as show in Figure B.12.

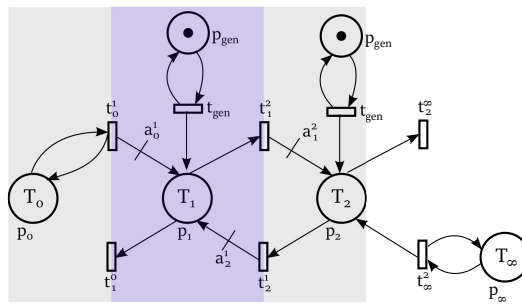


Figure B.12.: TCPN model for Example B.6.5 with three element discretization

The resulting model is:

$$\begin{aligned} \dot{T}_0 &= 0 \\ \dot{T}_1 &= \frac{\alpha T_0}{\delta_1 \Delta x_0^1} - \frac{\alpha T_1}{\delta_1 \Delta x_0^1} - \frac{\alpha T_1}{\delta_1 \Delta x_1^2} + \frac{\alpha T_2}{\delta_1 \Delta x_1^2} + \frac{\alpha}{k} \dot{e}_{gen} \\ \dot{T}_2 &= \frac{\alpha T_1}{\delta_2 \Delta x_1^2} - \frac{\alpha T_2}{\delta_2 \Delta x_1^2} - \frac{\alpha h T_2}{k \delta_2} + \frac{\alpha h T_\infty}{k \delta_2} + \frac{\alpha}{k} \dot{e}_{gen} \end{aligned}$$

where the firing rates of each transition are given by:

$$\begin{aligned} \lambda_0^1 &= \frac{\alpha}{\delta_0 \Delta_1^1}, & \lambda_1^0 &= \frac{\alpha}{\delta_1 \Delta_1^1}, & \lambda_1^2 &= \frac{\alpha}{\delta_1 \Delta_1^2} \\ \lambda_2^1 &= \frac{\alpha}{\delta_2 \Delta_1^2}, & \lambda_2^\infty &= \frac{\alpha h}{k}, & \lambda_2^\infty &= \lambda_\infty^2, & \lambda_{gen} &= \frac{\alpha \dot{e}_{gen}}{k} \end{aligned}$$

and the arc weights:

$$a_0^1 = \frac{\lambda_1^0}{\lambda_0^1} = \frac{\delta_0}{\delta_1}, \quad a_1^2 = \frac{\lambda_2^1}{\lambda_1^2} = \frac{\delta_1}{\delta_2}, \quad a_1^0 = \frac{\lambda_0^1}{\lambda_1^0} = \frac{\delta_1}{\delta_0}.$$

Notes on preliminary comparison between RUN and AIECS

We considered two processor cores with cache memories and speculative mechanisms non-existent or turned off. We produced task sets with 10 tasks per set and total utilization $U = 2$. Each task utilization was randomly generated under a uniform distribution by using UUnifast ([12]), integrated in Tertimuss. Deadlines were selected as 2, 5 and 10 s.

Unlike our proposal, RUN uses time, not cycles, to calculate a schedule. This requires rounding to cycles the time share of jobs as provided by RUN, which makes some task sets not schedulable in practice, despite the fact that they are theoretically schedulable according to RUN. For the comparison to be fair, we have only selected task sets which are schedulable under both schedulers.

We measured separately the number of *Mandatory Context Switches* (MCS) and *Coerced Context Switches* (CCS). MCS are given by job activation and termination, and therefore are independent from the scheduler, unlike CCS. Fig. 5.1 (a) displays two stacked bars per experimental task set, the bottom black bar of which represents the MCS, amounting to the same value in both schedulers as expected. RUN triggers between 6% (set 7) and up to 80% (set 6) more CCS than our proposal, averaging 44%. Fig. 5.1 (b) shows that our proposal cuts by almost a quarter the number of job migrations yielded by RUN, which reaches zero migrations against the two produced by our proposal in set 9 nonetheless.

Unimodularity

An integer programming problem (ILP) is defined as a LPP, however the solution space is restricted to integer values, because in practice it is often necessary to work with integer quantities. However, solving them can be a great challenge and it is usually highly computational demanding.

In particular, there are linear programming problems that can have integer solutions if their constraint matrix has the **Unimodular** characteristic.

A matrix A of rank r is *unimodular* if all its elements are integers and the determinant of each square submatrix of order r is either $0, +1$ or -1 . A matrix A is *totally unimodular* (TUM) if the determinant of each square submatrix of A is $0, \pm 1$. Total unimodularity implies unimodularity. [72]. Below are two classic results that allow evaluating the unimodularity of a matrix.

Theorem D.1 [72] Unimodularity and integer vertex. *Let A be an integer matrix of dimension (m,n) , with full rank by rows m and let $F = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ not empty. Then A is unimodular if and only if for every vector $b \in \mathbb{R}^m$ the set F has integer vertices.*

Theorem D.2 Total unimodularity and integer vertex. *Let A be an integer matrix and let $F = \{x \mid Ax \leq b, x \geq 0\}$ not empty. Then A is totally unimodular if and only if for every integer vector b the region F has integer vertices.*

If a LPP, with a unimodular restriction matrix A , is solved by the simplex method, the solution is integral, according to theorems (D.1) and (D.2). However, testing whether a matrix is unimodular can be very difficult. Theorem (D.3) establishes sufficient TUM conditions for a special type of matrix.

Theorem D.3 Sufficiency conditions for total unimodularity. *Any matrix A with elements $(-1, 0, 1)$ is totally unimodular if:*

1. each column of A contains no more than two nonzero elements, and

2. *the rows of A can be separated into two subsets such that:*

a) if a column contains two elements with the same sign, then the corresponding rows belong to different subsets, and,

b) if a column contains two elements with opposite signs, then the corresponding rows belong to the same subset.

Proofs for theorems (D.1),(D.2) and (D.3) can be consulted on [72].

List of Figures

2.1.	Classification of scheduling algorithms	16
2.2.	Classification of RT multiprocessor scheduling	19
2.3.	Dhall's effect. Example with m processors and $m + 1$ tasks for EDF and RM	21
2.4.	Fluid versus practical schedules	22
2.5.	Example of a Petri net structure	29
2.6.	TCPN global model	34
2.7.	TCPN module for task and CPU	35
2.8.	TCPN module for task and CPU	37
3.1.	EETAMS schema overview	42
3.2.	Jobs generated by the task set of Example 1	48
3.3.	Temperature evolution (upper plot) for the periodic schedule (lower plot) at CPU_1 (above) and CPU_2 (below). The maximum temperature produced by this schedule is $T_{CPU_{1,2}} = 45.3^\circ C$	58
3.4.	Temperature evolution (upper plot) for the periodic schedule (lower plot) at CPU_1 (above) and CPU_2 (below) upon acceptance of the aperiodic task τ_1^a . The maximum temperature produced by this schedule is $T_{CPU_{1,2}} = 46.24^\circ C$	60
4.1.	ALECS Overview	66
4.2.	Jobs generated by the task set of Example 4.1	70
4.3.	Form of submatrix $[A^T \ B^T]^T$	75
4.4.	Schedule for $\mathcal{T} = \{(3, 5), (6, 10), (9, 15), (6, 10), (3, 5)\}$	77
4.5.	Execution accumulative functions $R_{i,j}$	79
4.6.	TCPN module for CPU_1 to CPU_j . The gray places	81
4.7.	a) Task execution paths and b) System output recovering from a system overhead in the interval $[15, 16]$	86
4.8.	ALECS aperiodic task management	88
5.1.	Preliminary comparison for context switches between ALECS and RUN	90

5.2.	CAIECS scheduler scheme	91
5.3.	Example 5.1. Materials and Layout	94
5.4.	Differences in clustering between the usage of one idle (forces one cluster) and two idle tasks (idle tasks are in gray)	97
5.5.	BBP in Alg. 5: a) Iteration 1; b) Iteration 2	98
5.6.	CAIECS pre-schedule	99
5.7.	Execution of tasks with the admission of an aperiodic task	109
5.8.	Processors temperature evolution, on the left the nominal behaviour and on the right the temperature evolution with the admission of an aperiodic tasks	110
5.9.	Simulation results for the number of migrations/number of jobs.	111
5.10.	Simulation results for the number of preemption / number of jobs.	113
6.1.	Motivational example to account for preemption and migration overheads	120
6.2.	Position of the algorithm that account for preemption and migration overhead, with regards of the CAIECS scheme which is defined inside the dotted rectangle	122
6.3.	Flow chart of AdWCET	123
6.4.	AdWCET iterative process	126
6.5.	Utilization lower bound u_{min} for different hyperperiods. The horizontal lines represent the constraint $1/r$, where r is the task-to-core ratio. $f = 1000$, fixed for every curve	131
6.6.	Percentage of WCET maximum increase (a and b) and percentage of frequency increase (c and d)	132
6.7.	Iteration analysis of AdWCET under CAIECS	133
6.8.	Analysis of preemption and migrations per job on first and convergence iteration	133
6.9.	Percentage of frequency increase in the utilization approach	135
6.10.	Comparison of frequency increment in AdWCET and DP-U. Each bar represents the number of experiments in which that algorithm obtained the lower frequency of the two.	136
A.1.	Main components of Tertimuss	158
B.1.	Points and sections for the finite difference formulation	160
B.2.	TCPN representation for interior places	161
B.3.	TCPN representation for $f_n(t)$ constant	162

B.4.	TCPN elementary module	162
B.5.	TCPN elementary module for Dirichlet BC	164
B.6.	TCPN representation for the boundary point dynamics \dot{g}_0 , on the left , and on the right the TCPN elementary module for Neumann boundary conditions	166
B.7.	TCPN elementary module for Robin boundary condition	167
B.8.	Heat conduction TCPN representation	170
B.9.	TCPN module for a convection boundary condition	170
B.10.	TCPN module for a radiation boundary condition	171
B.11.	Schematic for Example B.6.5	173
B.12.	TCPN model for Example B.6.5 with three element discretization	173

List of Tables

2.1. Summary of the notation used for tasks and processors	14
2.2. TCPN modules for element subjected to conduction, convection and heat generation	38
5.1. Example 5.1. Power frequency pairs	94
5.2. $R_{i,j}(\delta_k)$, cycles that each task must execute on each CPU at every interval Δ_k	102
5.3. Number of migrations per job depending on the tasks-per-processor ratio (TPP)	112
5.4. Number of preemptions per job	113
5.5. Clustering performance	114
5.6. Clustering performance	115

Acronyms

AI Artificial Intelligence. 9

AIECS Allocation and Execution Control Scheduler. 7, 66, 67, 91–94, 99, 100, 109, 110, 112, 114, 117, 140, 144, 145, 179

BF Best Fit. 19, 20

BFD Best Fit Descending. 96, 98

BPP Bin packing problem. 19, 20, 24, 95–98

CAIECS Clustered Allocation and Execution Control Scheduler. 7, 89, 91–93, 99, 100, 109, 110, 112–114, 117, 122, 123, 126, 134, 140, 144, 145, 180

CE Cyclic Executive. 2–4, 15, 17, 99, 100, 119–122, 128, 133, 135, 136, 139, 140, 142–146

COTS Components-off-the-shelf. 1, 15

CPI Cycles per Instruction. 10

CPN Continuous Petri Nets. 30, 31

CPU Central Processing Unit. 3, 9, 12, 16, 139, 143

DJP Dynamic job priority. 16

DM Deadline Monotonic. 16, 17

DP Deadline partitioning. 43, 140, 144

DSM Distributed Shared Memory. 10

DVFS Dynamyc Voltage and Frequency Scaling. 11, 27

ECU Electronic Control Unit. 1

EDF Earliest Deadline First. 1, 17, 20, 23, 24, 27, 97, 99, 111, 119

FF First Fit. 19, 20

FFD First Fit Descending. 20

FJP Fixed job priority. 16

FPZL Fixed Priority until Zero Laxity. 41, 53, 59

FRT Firm Real Time. 11, 25

FTP Fixed task priority. 16

GPU Graphic Processing Unit. 9

HRT Hard Real Time. 1–3, 11, 13, 26, 27, 41–44, 52, 55, 58, 60, 65, 67, 68, 76, 80, 83, 86, 87, 93–95, 99, 100, 128, 135, 139–144, 146, 147

ILP Integer Linear Programming Problem. 43, 61, 65, 69, 140, 141, 144, 145, 177

ISA Instruction Set Architecture. 9, 10

LLF Least Laxity First. 17, 23

LPP Linear Programming Problem. 59, 177

MPSoC Multiprocessor System on a Chip. 1, 2, 9, 10, 93

NF Next Fit. 19

NoC Network on Chip. 9

NUMA Non-Uniform Memory Access. 10

PDE Parabolic partial differential equation. 159

PN Petri Net. 28, 102

QPS Quasi-Partitioned Scheduling. 23, 24

RM Rate Monotonic. 1, 16, 17, 20, 119

RT Real Time. 1, 7, 11–15, 21, 23, 25–27, 40–42, 54, 60, 119, 130, 140–142, 145–147, 157

RUN Reduction to Uniprocessor. 23, 24, 109–114, 140, 144

SMP Symmetric Multiprocessor. 9–11

SRT Soft Real Time. 2, 11, 25, 42, 139, 143

SWaP-C Space, Weight, Power and Cost. 1

TCPN Timed Continuous Petri Net. 3, 7, 31, 32, 60, 68, 77, 80, 81, 89, 92, 102, 139, 140, 143, 144, 159

TUM Total Unimodularity. 177

UMA Uniform Memory Access. 10

WCET Worst Case Execution Time. 1, 12, 13, 15, 26, 59, 111, 119–126, 130, 132–134, 142, 146, 180

WF Worst Fit. 19

