A Time-Predictable Parallel Programming Model for Real-Time Systems

Maria A. Serrano

Universitat Politècnica de Catalunya

Computer Architecture Department

PhD thesis

Doctoral programme on Computer Architecture

1st of February, 2019

A Time-Predictable Parallel Programming Model for Real-Time Systems

Maria Astón Serrano Gracia

February 2019

Universitat Politècnica de Catalunya
Computer Architecture Department

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

Advisor: Eduardo Quiñones, PhD, Barcelona Supercomputing Center
Tutor: Xavier Martorell, PhD, Universitat Politècnica de Catalunya

To Victor

Acknowledgements

Probably another thesis would have been possible, but I truly believe that it wouldn't be the same without my advisor, Eduardo Quiñones. Edu's infinite patience, constructive attitude and constant support have made this thesis one of the most positive and rewarding experience in my life. *Merci Edu, em sento afortunada de treballar amb tu.*

My gratitude is also to my cheerful colleague and friend, Sara Royuela, who always has a wise piece of advice for me. I've missed her a lot for the last months of this thesis, but she has a much more beautiful project to attend. It was also a pleasure to share my workspace with the rest of the people at CAOS group. Special thanks to Carles Hernandez, Enrico Mezzetti and Mikel Fernandez, for the great after-work moments that we shared. Also, thanks Enrico for your valuable work and recommendations for this thesis.

In these years, I've had the opportunity to collaborate with many people, specially in the context of the P-SOCRATES project. I really enjoyed working in such a great environment, so thanks to all the people that made it possible. A special mention to Marko Bertogna, whose knowledge about real-time scheduling was essential for this thesis.

I would like to thank Andrei Terechko for being my advisor during my internship at NXP Semiconductors in Eindhoven. It was a very productive period, both professionally and personally, thanks to Andrei's attitude. He was always available to teach me. My gratitude is also to Dirk Ziegenbein and Eric Jenn that reviewed this thesis and kindly provided helpful comments to improve it.

Finally, I would like to thank all my friends and family, *simply* for being there to contribute to the best of my memories. Gracias a Antonio, Bea, Carlos, Jenni, Jorge y Mary porque, pese a la distancia, ya sois y seréis *los de siempre*. Especialmente me gustaría darle las gracias a mi abuela Blanca, que tanto sufre con mis viajes por este "mundo loco", y a ti yayo, que con pocas palabras y una mirada me lo dices todo. Gracias a mi madre, Maria Aston, a mi padre, Javier, a mi hermano, Luis Manuel, y a mi hermana, Sofia, por su apoyo constante, sacrificio y confianza. Soy la persona que soy gracias, en gran parte, a mi familia. Estoy convencida de que Ana hubiera disfrutado mucho de este momento, asi que ésto también es para ella.

Y por último, muchas gracias Victor, compañero de vida, risas, emociones, sueños y aventuras, el dia a dia es mejor contigo.

The work reported in this thesis has been conducted at the Computer Architecture and Operating Systems (CAOS) group of the Barcelona Supercomputing Center (BSC), and has been supported by the European projects P-SOCRATES (FP7-ICT-2013-10) and CLASS (grant agreement No. 780622), and by Spanish Ministry of Science and Innovation under contracts TIN2012-34557 and TIN2015-65316-P.

Abstract

The recent technological advancements and market trends are causing an interesting phenomenon towards the convergence of the high-performance and the embedded computing domains. Critical real-time embedded systems are increasingly concerned with providing higher performance to implement advanced functionalities in a predictable way.

OpenMP, the de-facto parallel programming model for shared memory architectures in the high-performance computing domain, is gaining the attention to be used in embedded platforms. The reason is that OpenMP is a mature language that allows to efficiently exploit the huge computational capabilities of parallel embedded architectures. Moreover, OpenMP allows to express parallelism on top of the current technologies used in embedded designs (e.g., C/C++ applications). At a lower level, OpenMP provides a powerful task-centric model that allows to define very sophisticated types of regular and irregular parallelism. While OpenMP provides relevant features for embedded systems, both the programming interface and the execution model are completely agnostic to the timing requirements of real-time systems.

This thesis evaluates the use of OpenMP to develop future critical real-time embedded systems. The first contribution analyzes the OpenMP specification from a timing perspective. It proposes new features to be incorporated in the OpenMP standard and a set of guidelines to implement critical real-time systems with OpenMP. The second contribution develops new methods to analyze and predict the timing behavior of parallel applications, so that the notion of parallelism can be safely incorporated into critical real-time systems. Finally, the proposed techniques are evaluated with both synthetic applications and real use cases parallelized with OpenMP.

With the above contributions, this thesis pushes the limits of the use of task-based parallel programming models in general, and OpenMP in particular, in critical real-time embedded domains.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

List of Abbreviations

ADAS	Advanced Driver-Assistance Systems
API	Application Programming Interface
ARB	Architecture Review Board
AUTOSAR	AUTomotive Open System ARchitecture
BFS	Breadth-First Scheduling
CPU	Central Processing Unit
CRTES	Critical Real-Time Embedded Systems
DAG	Directed Acyclic Graph
DP	Dynamic Priority
DSP	Digital Signal Processor
ECU	Electronic Control Unit
EDF	Earliest Deadline First
EMS	Engine Management System
ESA	European Space Agency
FIFO	First-In, First-Out
FJP	Fixed Job Priority
FP	Fully Preemptive
FPGA	Field Programmable Gate Array

FTP	Fixed Task Priority
GCC	GNU Compiler Collection
GPU	Graphics Processing Unit
HOG	Histograms of Oriented Gradients
HPC	High-Performance Computing
ILP	Integer Linear Programming
IoT	Internet of Things
LIFO	Last-In, First-Out
LL	Least Laxity
LP	Limited Preemptive
MPPA	Massively Parallel processor Array
NoC	Network on Chip
NP	Non Preemptive
NPR	Non Preemptive Region
OpenMP	Open Multi-Processing
PC	Personal Computer
RAM	Random-Access Memory
RM	Rate-Monotonic
RTA	Response Time Analysis
RTOS	Real-Time Operating Systems
SPMD	Single Program, Multiple Data
SVM	Support Vector Machine
TDG	Task Dependency Graph

TSC	Task Scheduling Constraint
TSP	Task Scheduling Point
UoS	Unit of Scheduling
WCET	Worst-Case Execution Time
WFS	Work-First Scheduling

Chapter 1

Introduction

"*Be less curious about people and more curious about ideas.*"

— Marie Skłodowska-Curie

The possible applications of embedded systems have increased drastically over the past years. Nowadays, embedded computing systems are ubiquitous in our daily life, becoming mainstream in mobile phones, medical devices, automobiles, airplanes, satellites, etc. According to Crystal Market Research, the global embedded systems market was worth $132.50 billion in 2012 and is expected to reach approximately $254.87 billion by 2022 [1]. This increase is due to the fact that embedded systems are evolving to include general purpose and high-performance computing techniques in their designs. The intention is to cope with the performance requirements of modern systems, hence converging the high-performance and the embedded computing domains. As a result, embedded systems are able to implement more complex functionalities to support advanced features.

In the automotive industry, for instance, 90% of new components are driven by electronics [2], used for infotainment, safety, and engine control among others. Advanced Driver Assistance Systems (ADAS), such as automatic lane keeping and smart cruise control, are the standard on a number of current vehicles. And there are still plenty of opportunities to consider, from new in-vehicle infotainment software to autonomous driving. Consequently, the trend in future automotive architectures designs is to merge several single functionality microcontrollers, based on electronic control units (ECUs), into a few more powerful parallel platforms, reducing cabling and cooling provision, mass and space requirements, etc.

This trend is already observed in top providers of automotive chips across the globe, including NXP Semiconductors, Infineon Technologies, Renesas Electronics Corporation, STMicroelectronics and Texas Instruments. But also many other big industries like Google, NVIDIA or Intel® have recently invested significantly in future autonomous driving systems. As an example, Intel® acquired Mobileye, a leader company in computer vision-based autonomous driving technology, for $15.3 billion [3]. Overall, this trend is moving the traditional high-performance computing (HPC) and personal computer (PC) markets to a second position, being embedded systems the main contributor to the semiconductor industry revenue.

1.1 Critical Real-Time Embedded Systems

Critical Real-Time Embedded Systems (CRTES) are in charge of controlling fundamental parts of a device, e.g., the engine management system in a car, or the flight control system in an airplane. CRTES are used for a wide range of purposes where a failure to meet a requirement may lead to a catastrophic effect. In this context, the criticality of the system may relate to safety, security, mission or business aspects of the system. For instance, a failure may cause someone to get injured, an unacceptable loss of sensitive data or money, or a reduction on the quality of the service provided by the system.

History has shown examples where errors in critical systems caused the loss of a huge amount of money, and even lives, as well as the embarrassment of famous organizations such as the European Space Agency (ESA). As an example, one of the most famous mission failures, which resulted in a loss of more than $370 million, was the Ariane 5 rocket launch in 1996 [4]. An integer overflow caused a deviation from the rocket flight path, which ended up in an explosion, only about 40 seconds after being launched.

In order to avoid these type of situations (and also those less dramatic), CRTES must provide strong evidence on its correct *functional* and *timing* behavior, meaning that the system must guarantee that it operates correctly in response to its inputs, and that operations are performed within a predefined amount of time. To do so, the development of CRTES has to fulfill requirements given by safety standards, like the ISO26262 [5] in automotive, the DO-178C [6] in avionics or the IEC61511 [7] in the process industry. A number of software and hardware design principles and

requirements are listed in these standards to guarantee the successful behavior of the system. This thesis focuses on guaranteeing the correct timing behavior of CRTES.

1.1.1 The use of parallel architectures to implement CRTES

Like any other embedded system, CRTES are increasingly concerned with providing more functionalities that require higher performance, challenging the capabilities of current embedded platforms. As an example, NVIDIA or ARM predict that advanced driver assistance systems will require at least 100x more compute performance by 2024 compared to 2016 systems [8] [9].

In order to satisfy the increasing computational power requirements of those new functionalities, CRTES industries introduced a higher number of processing units in their products. However, this rapidly increased the computational elements used, with a clear impact on power consumption, size, weight and cost. As an example, the number of ECUs in a car increased to as much as 150 ECUs in 2015 [10]. For this reason, nowadays there is a clear evolution from architectures dedicated to a single functionality (like the ECUs) to parallel and heterogeneous architecture technologies, that may deal with this need for higher performance, while maintaining competitive costs. In addition, the trend is to integrate multiple functionalities into one domain or vehicle control unit in order to reduce the number of ECUs. Some examples include multi-core and many-core fabrics, GPUs, FPGA, etc. These parallel architectures are being appreciated, and now in high demand across various industry verticals.

1.2 Timing Analysis of Parallel CRTES

Timing guarantees are crucial in CRTES because it must be guaranteed that all operations finish within a predefined amount of time, i.e., in real-time terminology, all *real-time tasks* must meet their *deadlines*. To do so, a two-step verification procedure is needed to check whether the timing requirements of a system are satisfied. Firstly, it is of great importance to derive trustworthy and tight Worst-Case Execution Time (WCET) estimates for each real-time task [11]. The WCET represents an upper bound on the execution time of a task. Secondly, these WCET estimates are used for *real-time scheduling* which, based on the urgency of tasks (e.g., fixed priority, deadline), prioritizes their execution, and considering their interaction within the platform, determines if, in the worst possible scenario, the system meets the timing

(a) Single-core platform. **(b)** Multi-core platform.

Figure 1.1: Scheduling problem of a real-time system, sequential real-time tasks.

constraints or not. This thesis focuses on the latter step, i.e., on real-time scheduling, that must provide two features: (1) an algorithm for ordering the use of the available resources (mainly the computing units or CPUs); and (2) a method to predict the worst-case behavior of the system when the scheduling algorithm is applied. This thesis aims to provide *Response Time Analysis (RTA)* [12] techniques to compute the worst-case response time of each task which, if compared to the task's deadline, confirms if the timing requirements of the system are met.

While the scheduling problem for single-core platforms has been widely investigated for decades, producing a considerable variety of publications and applications, there are still many open problems regarding the scheduling analysis of systems running on a parallel platform. Certainly, the scheduling analysis becomes drastically more complex for multi-core platforms. The sequential execution implies that the access to physical resources is implicitly serialized, so, for instance, two tasks can never cause a contention for a simultaneous memory access. However, this is not the case in parallel architectures. Predicting the behavior of a real-time system running on such architectures involves considering the worst-case execution time of tasks, also analyzing the interference when accessing shared resources. This complicates the analysis when considering the more complex hardware of a multi-core platform, compared to a single-core.

Figure 1.2 illustrates the evolution of the real-time scheduling problem. Traditional real-time systems (Figure 1.1a) consider a set of concurrent tasks running sequentially on a single-core platform. With the incorporation of multi-core architectures, these concurrent tasks run simultaneously (i.e., in parallel) in the same platform (Figure 1.1b).

1.2.1 Parallel Programming in CRTES

The complexity of the scheduling problem increases even more when considering parallel programming models and heterogeneous architectures. The *sporadic directed Acyclic Graph (DAG) scheduling model* [13] has been recently introduced in the real-time literature to address the problem of modeling parallel work and applying scheduling techniques to verify its timing constraints. This thesis tackles the challenge of combine the use of HPC parallel programming models and the DAG scheduling model, to predict the timing behavior of parallel computation in real-time systems.

In the context of parallel architectures for CRTES, parallel programming models are of paramount importance for exploiting the computation capabilities of such architectures, while providing better programmability. In other words, parallel programming models may offer developers the abstraction level required to program parallel applications, while hiding the platform complexities. Besides performance and programmability, portability is also an essential property that parallel programming models can offer, not only across platforms but also across different inputs and calling contexts.

Overall, parallel computing is fundamental to enhance the efficiency of parallel architectures. Several approaches coexist with such a goal, and these can be grouped as follows [14]:

1. *Hardware-centric models* aim to replace the native platform programming with higher-level, user-friendly solutions, but still attached to a given hardware technology, e.g., Intel® TBB [15] and NVIDIA® CUDA [16]. These models focus on tuning an application to match a chosen platform, which makes their use neither a scalable nor a portable solution.
2. *Application-centric models* deal with the application parallelization from design to implementation, e.g., OpenCL [17]. Although portable, these models may require a full rewriting process to accomplish productivity.
3. *Parallelism-centric models* allow users to express typical parallel constructs in a simple and effective way, and at various levels of abstraction, e.g., POSIX threads [18] and OpenMP [19]. This approach allows flexibility and expressiveness, while decoupling design from implementation.

Among the vast amount of parallel programming models available, OpenMP has proved to be advantageous for many reasons. In the next section, we describe the

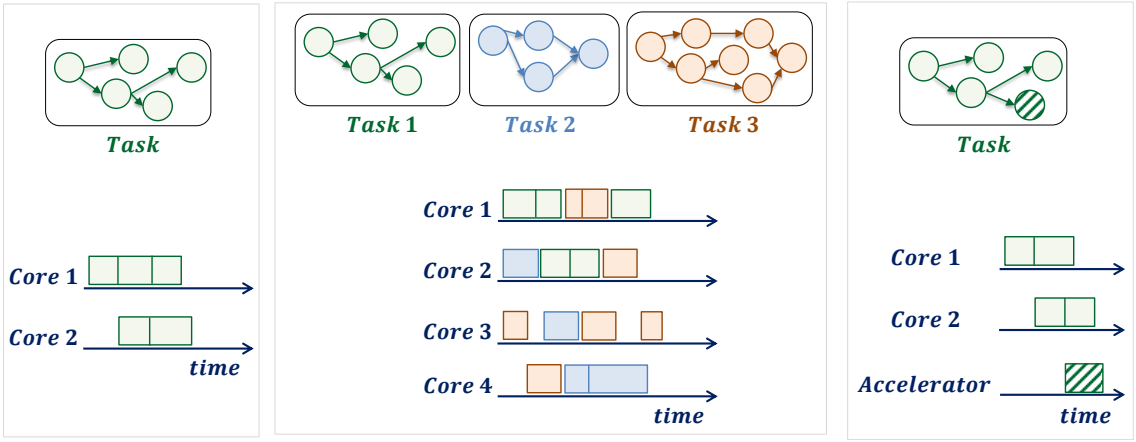
most relevant benefits of OpenMP in general, and also the particular reasons for considering OpenMP to implement CRTES.

1.2.2 Why OpenMP?

OpenMP [20], the de-facto standard for shared memory parallel programming in HPC, has been already considered as a candidate to parallelize real-time embedded systems [21]. As an example, OpenMP is supported in embedded parallel and heterogeneous platforms, for instance, the Texas Instruments Keystone II [22] [23] [24] and the Kalray Massively Parallel Processor Array (MPPA) [25], processors that target the automotive and the avionics industries. Also the timing predictability properties of OpenMP have been exposed in different research works [26] [27]. Originally focused on a thread-centric model to exploit massively data-parallel and loop-intensive applications, the latest specifications of OpenMP have evolved to a task-centric model that enables very sophisticated types of fine-grain and irregular parallelism, as well as support for heterogeneous architectures.

When comparing OpenMP with other parallel programming models, different evaluations demonstrate that OpenMP delivers tantamount performance and efficiency to that provided by highly-tunable models such as TBB [28], CUDA [29] and OpenCL [30]. Moreover, OpenMP has different advantages over low-level libraries such as Pthreads [31]: on the one hand, it offers robustness without sacrificing performance [32] and, on the other hand, OpenMP does not lock the software to a specific number of threads. Another important benefit is that the code can be compiled as a single-threaded application just disabling support for OpenMP, thus easing debugging, and so programmability.

Overall, the use of OpenMP presents three main advantages. First, an expert community has been constantly reviewing and augmenting the language for the past 20 years. Second, OpenMP is widely implemented by several chip and compiler vendors from both the high-performance and the embedded computing domains (e.g., GNU, Intel®, ARM, Texas Instruments, IBM, Gaisler, NVIDIA), increasing portability among multiple platforms. Third, OpenMP provides great expressiveness due to years of experience in its development; the language offers several directives for parallelization and fine-grain synchronization, along with a large number of clauses that allow it to contextualize concurrency and heterogeneity, providing fine control of the parallelism.



(a) A parallel real-time task, multi-core platform.

(b) Set of parallel real-time tasks, multi-core platform.

(c) A parallel real-time task, heterogeneous platform.

Figure 1.2: Scheduling problem of a real-time system, parallel real-time tasks.

Interestingly, the structure and syntax of the OpenMP tasking model have certain similarities with the sporadic DAG scheduling model [26]. Moreover, a recent real-time scheduling technique, the limited preemptive scheduling [33], resembles the OpenMP execution model approach. However, the sporadic DAG model under the limited preemptive scheduling approach has not been addressed yet. This thesis advances the current state of the art regarding the response time analysis of the parallel DAG model, also in combination with the limited preemptive scheduling technique. Figure 1.2 illustrates the scheduling problem of parallel real-time tasks, modeled as DAGs. Concretely, this thesis analyzes the timing behavior of: (1) a single parallel real-time task scheduled in a multi-core platform (Figure 1.2a); (2) a set of parallel real-time tasks, to implement a complete real-time system, scheduled in a multi-core platform (Figure 1.2b); and (3) a single real-time task partially executed in an accelerator device, i.e., targeting heterogeneous architectures (Figure 1.2c).

Considering OpenMP in real-time systems does not only imply the study of new scheduling techniques. Given its non "real-time nature", the OpenMP specification includes some particular features and characteristics whose impact on the timing constraints of real-time systems needs to be investigated. Moreover, there are a number of design and integration implications that must be taken into account. Therefore, this thesis also studies the concrete implications, in both the OpenMP specification

and its runtime implementations, of using OpenMP to parallelize and implement real-time systems.

1.3 Thesis Contributions

This thesis advances the current state of the art towards the safe adoption of OpenMP in critical real-time systems. From a timing analysis perspective, based on the similarities between the DAG model and the OpenMP tasking model, this thesis builds the response time analysis upon which the timing requirements of parallel DAG-based real-time systems are guaranteed. The proposed timing analysis techniques are not exclusive to OpenMP, but they can be also applied to any other task-based parallel programming model, provided that the DAG model is used to represent the parallel work.

The main contributions of this thesis and the corresponding published articles in which the contributions were presented (see Section 1.5 for the complete list), are summarized as follows:

1. Timing characterization of the OpenMP specification to implement and parallelize real-time systems.

    1.1. Analyze the timing and scheduling features of the OpenMP specification, identifying the similarities with current real-time scheduling practices; propose new features to be incorporated in the OpenMP specification; and provide a set of guidelines to implement real-time systems with OpenMP (publication number 5).

    1.2. Analyze the features and constraints of the OpenMP tasking model that must be taken into account for the timing analysis of OpenMP applications (publication number 1).

    1.3. Extend the DAG model to support heterogeneous computing, being compatible with the OpenMP accelerator model (publication number 4).

2. Development of response time analysis to provide evidence on the satisfaction of the timing constraints of DAG-based real-time systems.

    2.1. Develop a response time analysis for a single DAG-based real-time task, compatible with the OpenMP tasking model (publication number 1).

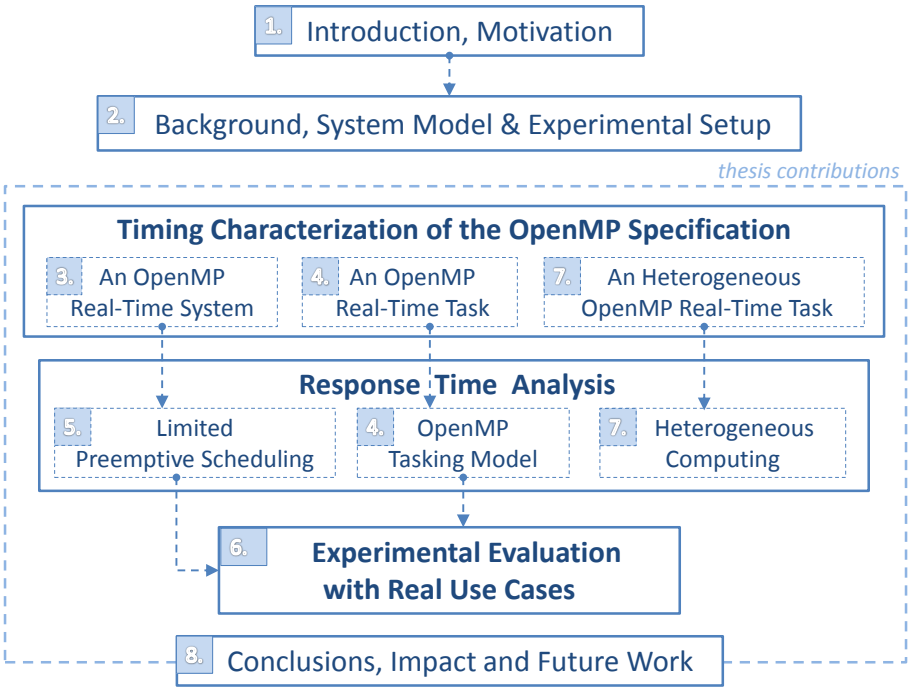


Figure 1.3: Thesis contributions and organization.

2.2. Develop a response time analysis for a real-time system composed of DAG-based parallel real-time tasks, under limited preemptive scheduling (publications number 2 and 3).

2.3. Develop a response time analysis for a DAG-based real-time task supporting heterogeneous computing, and compatible with the OpenMP accelerator model (publication number 4).

3. Study the timing behavior of real uses cases and demonstrate the validity of the proposed timing analysis techniques, considering two task-based programming models: a real-time system implemented and parallelized with OpenMP, and an AUTOSAR automotive application.

1.4 Thesis Organization

Figure 1.3 shows an overview of the contributions of the thesis, and how the document is organized. The number included in the blue squares of the Figure corresponds to the chapter number associated to each block.

The rest of this document is organized as follows: Chapter 2 presents the background, including the foundations of real-time systems, scheduling techniques and

the OpenMP API for parallel programming, the system model, and the experimental setup considered in this thesis. Chapter 3 presents the guidelines to implement real-time systems with OpenMP, and the analysis of the timing features of the OpenMP specification (contribution 1.1). Chapter 4 shows the study regarding the timing characterization and the response time analysis of the OpenMP tasking model (contributions 1.2 and 2.1). Chapter 5 introduces the response time analysis of DAG-based real-time systems under the limited preemptive scheduling (contribution 2.2). Chapter 6 presents the evaluation with real use cases (contribution 3). Chapter 7 introduces the timing characterization and the response time analysis of a DAG task supporting heterogeneous computing (contributions 1.3 and 2.3). Finally, Chapter 8 presents the conclusions and impact of this thesis, as well as the future work.

1.5 List of Publications

The list of publications that the research of this thesis has produced is presented below.

1. **Maria A. Serrano**, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna and Eduardo Quiñones. *Timing Characterization of OpenMP4 Tasking Model.* In proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES). Amsterdam, The Netherlands. October, 2015. © 2015 IEEE.
2. **Maria A. Serrano**, Alessandra Melani, Marko Bertogna and Eduardo Quiñones. *Response-Time Analysis of DAG Tasks under Fixed Priority Scheduling with Limited Preemptions.* In proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE). Dresden, Germany. March, 2016. © 2016 ACM.
3. **Maria A. Serrano**, Alessandra Melani, Sebastian Kehr, Marko Bertogna and Eduardo Quiñones. *An Analysis of Lazy and Eager Limited Preemption Approaches under DAG-based Global Fixed Priority Scheduling.* In proceedings of the 20th International Symposium on Real-time Distributed Computing (ISORC). Toronto, ON, Canada. May, 2017. © 2017 IEEE.
4. **Maria A. Serrano** and Eduardo Quiñones. *Response-Time Analysis of DAG Tasks Supporting Heterogeneous Computing.* In proceedings of the 55th Design

Automation Conference (DAC). San Francisco, CA, USA. June, 2018. © 2018 ACM/IEEE.

5. **Maria A. Serrano**, Sara Royuela and Eduardo Quiñones. *Towards an OpenMP Specification for Critical Real-time Systems.* In proceedings of the 14th International Workshop on OpenMP (IWOMP). Barcelona, Spain. September, 2018. © 2018 Springer.

Some of the contributions of this thesis have been also published in two chapters of the book *High-Performance and Time-Predictable Embedded Computing*, Luis M. Pinho, Eduardo Quiñones, Marko Bertogna, Andrea Marongiu, Vincent Nelis, Paolo Gai, Juan Sancho (Editors), River Publishers, 2018. Concretely, in:

- **Maria A. Serrano**, Sara Royuela, Andrea Marongiu and Eduardo Quiñones. *Predictable Parallel Programming with OpenMP.* Chapter 3 (pp. 33-62).
- Paolo Burgio, Marko Bertogna, Alessandra Melani, Eduardo Quiõnes and **Maria A. Serrano**. *Mapping, Scheduling, and Schedulability Analysis.* Chapter 4 (pp. 63-112).

Finally, the list presented below contains other publications that, although do not constitute a contribution of this thesis, are related to it.

- Roberto E. Vargas, Sara Royuela, **Maria A. Serrano**, Xavier Martorell and Eduardo Quiñones *A Lightweight OpenMP4 Run-time for Embedded Systems.* In proceedings of the 21st Asia and South Pacific Design Automation Conference (ASP-DAC). Macau (China), January, 2016.
- Alessandra Melani, **Maria A. Serrano**, Marko Bertogna, Isabella Cerutti, Eduardo Quiñones and Giorgio Buttazzo. *A Static Scheduling Approach to Enable Safety-Critical OpenMP Applications.* In proceedings of the 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). Chiba, Japan. January, 2017.
- Sara Royuela, Alejandro Duran, **Maria A. Serrano**, Eduardo Quiñones and Xavier Martorell. *A Functional Safety OpenMP for Critical Real-Time Embedded Systems.* In proceedings of the 13th International Workshop on OpenMP (IWOMP). New York, NY, USA. September, 2017.

Chapter 2

Background, System Model and Experimental Setup

"*The greatest enemy of knowledge is not ignorance,*
it is illusion of knowledge."

— Stephen Hawking

This chapter presents the terminology and background that constitute the basis upon which we have developed the work presented in this thesis. In particular, this chapter introduces basic concepts about real-time scheduling, including the way real-time workload is modeled and the type of tests and analysis to provide evidence on the timing behavior of a system. The sporadic DAG tasks model is deeply described as it is used in this thesis to represent the parallelism exposed by real-time tasks. Also, this chapter presents the key aspects of OpenMP used in this thesis, as well as the previous works that motivate the use of OpenMP in real-time systems. Finally, this chapter also introduces the experimental setup used along this thesis. Concretely, it describes the algorithms and tools to generate the DAG tasks used to evaluate the techniques proposed in this thesis.

With the purpose of being more concrete, the work related to this thesis is presented in a dedicated section within the chapters that describe the contributions of this thesis.

2.1 Real-Time Scheduling

This thesis investigates the use of parallel programming models in general, and OpenMP in particular, in CRTES. The timing behavior of these systems should be analyzable. This implies the necessity of verifying the timing behavior of the system before execution time. Real-time scheduling theory targets this requirement, providing (1) the algorithms to manage the available shared resources in a predictable manner, and (2) the analytical methods to verify the timing constraints imposed by the system. The former, known as *scheduling algorithms*, take into account the properties of the system, for instance, urgency of tasks, and given the available processors (or any other resource), organize the execution of the tasks. The latter, known as *schedulability analysis* or tests, take the set of tasks and a given scheduling algorithm, and verify prior to system run time that all deadlines will be met.

In real-time scheduling, it is also fundamental to represent the system under analysis. A *system model* or *real-time tasks model* represents and describes the properties of the real-time tasks.

2.1.1 Real-time tasks modeling

Any real-time scheduling framework must refer to specific assumptions and a way of representing the tasks of a system. Typically, CRTES are represented as a set of recurrent (sporadic or periodic) and independent [34] real-time tasks $\mathcal{T} = \{\tau_1, \tau_2, ...\tau_n\}$. Each real-time task is said to be *recurrent* because its execution is repeatedly triggered either by an internal clock or by the occurrence of an external event, e.g., the arrival of new data from a sensor. Each execution of a real-time task is known as *job*; a task generates a potentially infinite sequence of jobs. The time at which a job is triggered is known as *release time*. Moreover, a recurrent task may be *periodic*, if there is an exact inter-arrival time between two consecutive jobs, or *sporadic*, if there is a minimum, but not a maximum, inter-arrival time between jobs. Since the periodic behavior is one of the possible behaviors of a sporadic system, real-time scheduling theory usually considers sporadic real-time systems. Tasks are *independent* in the sense that the runtime behavior of a task should not depend upon the behavior of other tasks.

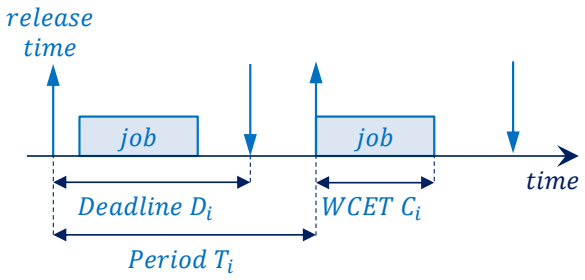


Figure 2.1: Representation of a sequential real-time task $\tau_i = \langle C_i, T_i, D_i \rangle$, as considered by the three-parameter sporadic tasks model.

2.1.1.1 Sequential real-time tasks model

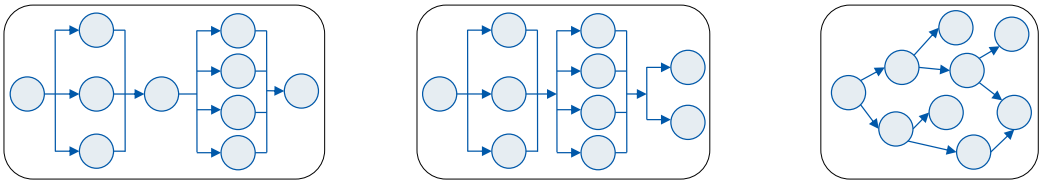
Traditionally, the *three-parameter sporadic tasks model* [35] is used to characterize a real-time system composed of sequential real-time tasks. Tasks are *sequential* in the sense that each job is assumed to represent a single thread of computation, which may execute upon at most one processor at any time instant. Each real-time task τ_i is represented as the tuple $\langle C_i, T_i, D_i \rangle$, where:

- C_i is the *Worst-Case Execution Time (WCET)* of the task, i.e., an estimation of the longest possible execution time of τ_i.
- T_i is the *period*, or the minimum inter-arrival time between two consecutive jobs of τ_i.
- D_i is the *relative deadline*, which defines the time at which τ_i must finish after its release time.

Based on the relation between D_i and T_i, a task-set $\mathcal{T}$ can be classified as follows: (1) in a *implicit deadline* task system, the relative deadlines are equal to the periods, i.e., $D_i = T_i$, for all the tasks $\tau_i \in \mathcal{T}$; (2) in a *constrained deadline* task system, the relative deadline of each task is not larger than the task's period, i.e., $D_i \leq T_i$, for all $\tau_i \in \mathcal{T}$; and (3) in an *arbitrary deadline* task system, there is no specific relation between the relative deadline and the period of each task.

Figure 2.1 shows a graphical representation of a real-time task, as considered by the three-parameter sporadic tasks model. Concretely, it shows two jobs of a real-time task τ_i, within a constrained deadline system.

The three-parameter sporadic tasks model enables to represent and exploit the inherent *concurrency* of a multi-core or many-core processor platform. Sequential real-time task-sets exploit *parallelism at system level*, because several sequential real-time tasks can execute at the same time, in different cores.



(a) Fork-join model.

(b) Parallel synchronous task model

(c) Sporadic DAG task model

Figure 2.2: Representation of parallel real-time task models.

2.1.1.2 Parallel real-time tasks model

In recent years, the complexity of real-time tasks has significantly increased to incorporate advanced functionalities. With the increasing performance demand and the newest highly-parallel embedded architectures used in critical real-time systems, the number and variety of available cores have increased significantly. This is the case for instance, of the Kalray MPPA platform, featuring a fabric of 256 cores [25]. Therefore, it is reasonable and necessary to exploit fine-grain parallelism within each real-time task.

The first parallel real-time tasks model proposed was the *fork-join* model [27], where each real-time task is represented as an alternating sequence of parallel (fork) and sequential (join) segments (see Figure 2.2a). Later, this model was enhanced by the *parallel synchronous task* model [36] [37], which allows consecutive parallel segments with an arbitrary degree of parallelism (see Figure 2.2b). Still, synchronization is enforced at every segment's boundary. The *sporadic Directed Acyclic Graph (DAG) tasks model* [13] generalizes the two previous models (see Figure 2.2c). This model represents each real-time task as a directed acyclic graph, which allows to represent both structured and unstructured parallelism.

Overall, parallel real-time tasks models allow to exploit coarse-grain and fine-grain parallelism at both *system* and *task levels*. This thesis focuses on the sporadic DAG tasks model, which is described in detail in the next section.

2.1.2 System model: the sporadic DAG tasks model

The sporadic DAG scheduling model has been introduced to characterize the parallel execution of real-time tasks. A real-time system is composed of n DAG tasks $\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$. Each real-time task $\tau_k \in \mathcal{T}$ is represented as a DAG $G_k = (V_k, E_k)$. $V_k = \{v_{k,1}, \ldots, v_{k,n_k}\}$ is the set of nodes, being $n_k = |V_k|$ the total number of nodes.

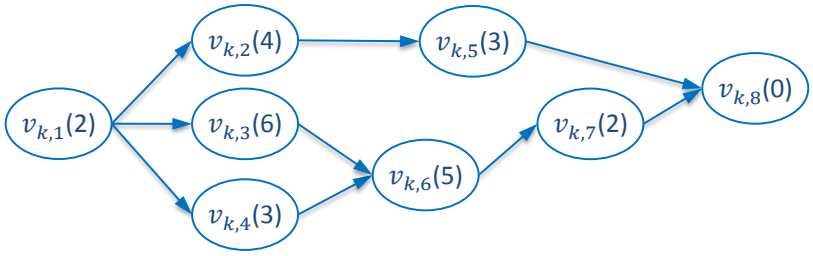


Figure 2.3: Real-time DAG task example. Nodes are labeled with WCET in parenthesis.

Each node $v_{k,i} \in V_k$ represents a sequential operation or *sub-task*, and is characterized by its *worst-case execution time* (WCET), denoted by $C_{k,i}$. $E_k \subseteq V_k \times V_k$ is the set of edges representing precedence constraints or dependencies between nodes in V_k. If $(v_{k,1}, v_{k,2}) \in E_k$, then node $v_{k,1}$ must complete before node $v_{k,2}$ can begin its execution.

We use the following nomenclature to describe particular nodes in the DAG:

- *Source node.* A node with no incoming edges.
- *Sink node.* A node with no outgoing edges.
- *Direct predecessor node.* If $(v_i, v_j) \in E$, then v_i is a direct predecessor node of v_j.
- *Direct successor node.* If $(v_i, v_j) \in E$, then v_j is a direct successor node of v_i.
- *Sibling node.* If $(v_i, v_j) \in E$ and $(v_i, v_k) \in E$, then v_j is a sibling node of v_k and vice versa.
- *Predecessor node.* v_i is a predecessor of another node v_j if there exists a path in the DAG where v_i appears before v_j.
- *Successor node.* v_i is a successor of another node v_j if there exists a path in the DAG where v_j appears before v_i. It is said that v_i is reachable from v_j.

Without loss of generality, each DAG is assumed to have exactly one source node, denoted by v_k^{source}, and one sink node, denoted by v_k^{sink}. If this is not the case, a dummy source/sink node with zero WCET can be added to the DAG, with edges to/from all the original source/sink nodes. Figure 2.3 depicts an example of a parallel real-time DAG task τ_k, composed of eight nodes $V_k = \{v_{k,1}, \ldots, v_{k,8}\}$ (labeled with their corresponding WCET in parenthesis), and nine edges representing precedence constraints. $v_{k,1} \equiv v_k^{source}$ is the source node, and $v_{k,8} \equiv v_k^{sink}$ is the (dummy) sink node (with WCET equal to 0).

Similarly to the three-parameter sporadic tasks model, each task τ_k releases an infinite sequence of *jobs* with a minimum inter-arrival time (period) of T_k time-units. When a task τ_k is released at time t, all sub-tasks in V_k are ready to execute whenever precedence constraints are fulfilled. All sub-tasks are expected to finish before time $t + D_k$, being D_k the relative deadline of τ_k.

The sporadic DAG tasks model defines a *chain* or *path* as a sequence of nodes $\lambda_k = (v_{k,i}, v_{k,j}, \ldots, v_{k,l})$ such that each pair of consecutive nodes in λ_k, $(v_{k,i}, v_{k,j})$, is an edge in E_k. The *length* of this chain, denoted by $len(\lambda_k)$, is the sum of the WCETs of all its nodes.

Definition 1. *The* critical path *of a DAG task τ_k, denoted by λ_k^*, is the chain in the DAG with the largest length.*

Definition 2. *The* length *of a DAG task τ_k, denoted by $len(G_k)$ or $len(\lambda_k^*)$, is the length of the critical path of τ_k.*

Notice that $len(G_k)$ corresponds to the minimum amount of time needed to safely execute the task τ_k on a sufficiently large number of processors. $len(G_k)$ can be computed in linear time in the number of nodes and the number of edges in G_k by first obtaining a topological sorting[1] of the nodes of the graph and then running a straightforward loop over all the nodes in topological sorting.

Definition 3. *The* volume *of a DAG task τ_k, denoted by $vol(G_k)$, is the sum of all WCETs of its nodes, i.e.,*

$$vol(G_k) \stackrel{def}{=} \sum_{v_{k,i} \in V_k} C_{k,i}$$

This value corresponds to the worst-case execution time needed to execute the DAG task sequentially on a dedicated single-core platform.

Definition 4. *The* utilization *of a task τ_k, denoted by U_k, is the ratio of its volume to its period; the* utilization *of the task-set $\mathcal{T}$, denoted by $U_\mathcal{T}$, is the sum of the utilization of all tasks, i.e.,*

$$U_k \stackrel{def}{=} \frac{vol(G_k)}{T_k}; \quad U_\mathcal{T} \stackrel{def}{=} \sum_{k=1}^{n} U_k$$

In the example of Figure 2.3, the length of the task τ_k is $len(G_k) = 15$, given by the path $\lambda_k^* = (v_{k,1}, v_{k,3}, v_{k,6}, v_{k,7}, v_{k,8})$, and its volume is $vol(G_k) = 21$.

[1]The *topological sorting* (or *topological order*) [38] is such that if there is an edge from node u to node v in the DAG, then u appears before v.

$\mathcal{T}$	Set of DAG tasks	m	Number of cores
$U_{\mathcal{T}}$	Utilization of $\mathcal{T}$	n	Number of DAG tasks in $\mathcal{T}$
τ_k	k-th DAG task in $\mathcal{T}$	G_k	DAG representation of τ_k
V_k	Set of nodes in G_k	E_k	Set of edges in G_k
$v_{k,i}$	i-th node (sub-task) of V_k	$(v_{k,i}, v_{k,j})$	Edge between nodes i and j
v_k^{source}	Source node of V_k	v_k^{sink}	Sink node of V_k
$n_k = \|V_k\|$	Number of nodes in V_k	$C_{k,i}$	WCET of the i-th node
λ_k	Any chain/path of τ_k	λ_k^*	Critical path of τ_k
$len(\lambda_k)$	Length of λ_k	$len(G_k)$	Length of τ_k (or length of λ_k^*)
$vol(G_k)$	Volume of τ_k	U_k	Utilization of τ_k
T_k	Period of τ_k	D_k	Relative deadline of τ_k

Table 2.1: System model notation.

Computing platform model. This thesis considers parallel architectures, and the number of processors in the platform, the type of these processors and if they have different computing capabilities must be completely specified.

For the most part of this thesis, real-time DAG tasks are executed upon a multiprocessor platform composed of m identical cores. Thus, each processor in the platform has the same computing capabilities as every other processor. This platform model is used interchangeably to refer to a multi-core, a many-core or a multiprocessor. Moreover, the terms *core*, *processor* and *thread* interchangeably refer to a single hardware computing unit. The last chapter of this thesis considers an heterogeneous architecture composed of a host m-core processor and an accelerator device.

Table 2.1 summarizes the notation described in this section. Notice that the subscript k in the parameters associated to a task τ_k can be omitted whenever the reference to the task is clear in the discussion.

2.1.3 Schedulability problem: the response-time analysis

Given a real-time system, we are interested in finding a schedule that allows to meet the timing constraints of the system, given by the deadline of all the jobs of all the real-time tasks. If it is the case, the system is said to be *feasible*.

Definition 5. ***Feasibility.*** *A real-time system is said to be feasible upon a spec-*

ified platform if there exists a schedule that meets all timing constraints for all the collections of jobs that could legally be generated by the task system.

As an example, since $len(G_k)$ represents the minimum amount of time needed to execute a DAG task τ_k, a necessary condition for the feasibility of such task τ_k is $len(G_k) \leq D_k$. Considering a set of real-time tasks, and given the platform and system model considered in the previous section, a simple necessary condition for the feasibility of the real-time system is $U_{\mathcal{T}} \leq m$.

However, feasibility is a very general property as it merely requires that a correct schedule *exists*, but it may not always be possible to construct such a schedule. Therefore, it is not sufficient to know if a real-time system is feasible; in addition, it is fundamental to know, prior to run time, if the timing constraints will be met. This is known as the schedulability problem.

Definition 6. ***Schedulability problem.*** *Given a set of sporadic real-time tasks and a scheduling algorithm upon a specified platform, the schedulability problem finds if all the potentially infinite jobs generated by the system meet their deadlines using the given scheduling algorithm.*

The idea behind the schedulability problem (also known as schedulability test) is to determine, given a scheduling algorithm and the task-set representation, the worst case behavior of the system, and check if the timing constraints previously defined are met. However, predicting the behavior of a multiprocessor system is not trivial and requires a significant computing effort. To simplify the analysis, it is often necessary to consider pessimistic assumptions.

In order to determine, at each point in time, which task should be executed, real-time tasks typically have a *priority* assigned. Therefore, a higher priority task should have the preference to execute. Moreover, for the most part of this thesis, we focus on *preemptive scheduling*, in which a task can be preempted while it is executing (in favor of a higher priority task, for instance), being later resumed.

A well-known schedulability test is the *Response Time Analysis (RTA)* [12]. It is based on the computation of the worst-case response time of each task in the system.

Definition 7. ***Worst-Case Response Time.*** *Given a real-time task τ_k, its worst-case response time, denoted by R_k^{ub}, is the longest interval between the release time and the completion of all its jobs.*

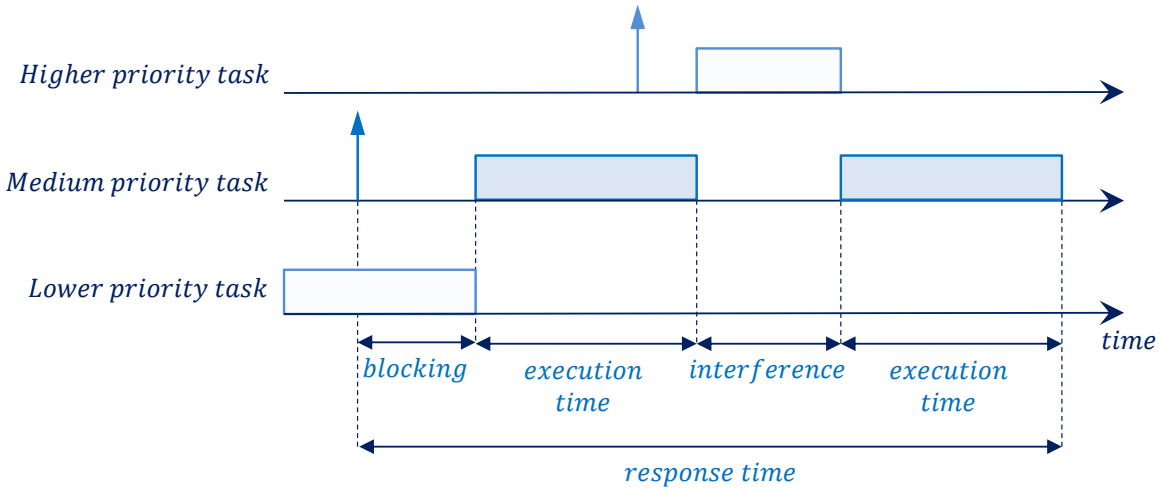


Figure 2.4: Overview of the response time analysis of a medium priority task.

The worst-case response time of a task τ_k can be longer than the actual worst-case execution time of the task due to *interference* and *blocking times*. The interference is the time spent executing higher priority tasks while τ_k is ready and waiting to execute. The blocking time is the time spent executing lower priority tasks, while τ_k is ready and waiting to execute. Also preemptions, i.e., context switches, cause overheads that may delay the execution of a task.

Figure 2.4 shows an example of the response time of a task. It considers that the task of interest has medium priority and that there exist two other tasks with higher and lower priority. The three tasks run in a single-core processor for simplicity. The task is first blocked by the lower priority task. Then, the high priority task is released and the medium priority task is preempted, suffering interference. Overall, the response time of the task considers not only its execution time but also the interference due to the higher priority task and the blocking time due to the lower priority task.

Since the exact interference and blocking times suffered by each task is difficult to compute when considering multiprocessor systems, the response time analysis applied to such systems computes an upper bound of the interference and blocking times.

The timing constraints are met whenever the worst-case response time of each real-time task in the system is less than or equal to the task's deadline. Therefore, a set of real-time tasks is said to be *schedulable* under a given scheduling algorithm if the following condition holds for all tasks $\tau_k \in \mathcal{T}$: $R_k^{ub} \leq D_k$.

2.2 The OpenMP Parallel Programming Model

OpenMP (Open Multi Processing) is an Application Programming Interface (API) for expressing parallelism in C/C++ and Fortran programs for shared-memory processor architectures. OpenMP provides a very convenient abstraction layer by means of a set of constructs and directives, described in the OpenMP specification, to define parallel regions and synchronization operations. The constructs and directives are processed by the compiler and executed by the runtime, which implements the parallel OpenMP functionalities.

Initial versions of OpenMP, up to version 2.5 [39], implemented a *thread-centric model* with a shared-memory space. It was limited to a standard fork-join type of parallelism to exploit massively data-parallel and loop-intensive applications, enforcing a rather structured parallelism. In this model, *OpenMP threads* work as an intermediary for physical processors, hence the specification somehow exposes the underlying resources.

From version 3.0 [40], OpenMP has evolved to a *task-centric model* that enables very sophisticated types of fine-grain, both structured and unstructured, parallelism. The OpenMP tasking model allows the programmer to define explicit tasks[2] and the data dependencies existing among them. An OpenMP task is a unit of work, specified by an instance of executable code and its data environment. At run-time, tasks are executed by OpenMP threads, being the programmer oblivious of the physical resources. This allows to effectively exploit the performance capabilities of parallel architectures while hiding their complexity to the programmer.

Versions 4.0 [41] and 4.5 [20] of OpenMP include support for heterogeneous architectures through a *host-centric accelerator model*. This model considers a parallel heterogeneous architecture composed of a host processor and one or more accelerator devices (e.g., a FPGA, GPU or DSP fabric). The host device is the one in charge of offloading code and data to the accelerator device and collecting the results. The OpenMP specification incorporates easy-to-use device constructs to define the offloaded code, and data clauses to express data directionality when moving data to/from the device memories.

[2]Notice the difference between real-time tasks and OpenMP tasks. We define their relationship in Chapter 3.

2.2.1 The OpenMP tasking model

An OpenMP program starts with an *implicit task*[3] surrounding the whole program. This implicit task is executed by a single thread, called the *initial* OpenMP thread, which runs sequentially.

When the thread encounters a `parallel` construct, it creates a new *team of threads*, composed of itself, as the *master* thread, and $M - 1$ additional threads (M can be specified with the `num_threads` clause). Each individual OpenMP thread executes the region inside the `parallel` construct, by means of an implicit task. When a thread encounters a `master` construct, it creates an implicit task that will be executed by the *master* thread of the team. Similarly, the `single` construct defines a block that will be executed by one thread of the team (not necessarily the master thread). The other threads in the team, which do not execute the single block, wait at an implicit barrier at the end of the single construct unless a `nowait` clause is specified.

When a thread encounters a `task` construct, a new *explicit task* is created, consisting of all code within the *task region* (C/C++ Fortran code block). The OpenMP specification defines the following tasks:

- *Child task.* A task is a *child task* of its generating task region.
- *Sibling tasks.* Tasks that are child tasks of the same task region.
- *Descendant task.* A task that is the child task of a task region or of one of its descendant task regions.

Additionally, we define:

- *Parent task.* The task region encountering a task construct.
- *Predecessor task.* A task that is the parent task of a task region or of one of its predecessor task regions.

When an explicit task is created, it can be assigned to one of the threads in the current team for immediate or deferred execution, based on additional clauses: `depend`, `if`, `final` and `untied`.

- The `depend` clause forces sibling tasks to be executed in a given order based on dependencies defined among data items. A task that cannot be executed until its task dependencies are fulfilled is a *dependent task.*

[3] An implicit task is not created by the programmer but by the runtime; tasks created by the programmer using the `task` construct are commonly referred to as *explicit tasks.*

- The `if` clause makes the new task to be *undeferred*, meaning that it must be executed by a thread of the team, suspending the current task region until the new task completes.
- Similarly, the `final` clause makes all descendants of the new task to be *included*, meaning that they must execute immediately by the encountering thread.
- The `untied` clause makes the new generated task not being tied to any thread and so, in case it is suspended, it can later be resumed by any thread in the team. By default, OpenMP tasks are *tied* to the thread that first starts their execution. Hence, if such tasks are suspended, they can later only be resumed by the same thread.

Moreover, OpenMP defines some clauses that allow to specify the *data-sharing attributes* of the variables in the `task` construct:

- `private`: specifies that the variables are private to the task.
- `firstprivate`: specifies that the variables are private to the task, and initializes each of them with the value that the corresponding original variable has when the `task` construct is encountered.
- `shared`: specifies that the variables are shared among tasks.

The completion of a subset or all explicit tasks bound to a given parallel region may be specified through the use of synchronization constructs, e.g., the `taskwait` and the `taskgroup` constructs. The `taskwait` construct specifies a wait on completion of child tasks of the current task. The `taskgroup` specifies a wait on completion of child tasks of the current task and their descendant tasks. All tasks are guaranteed to have completed at the implicit barrier at the end of the parallel region, as well as at any other explicit `barrier` construct. The `barrier` construct (from the thread-centric model) specifies an explicit barrier where all threads of the team must complete execution before any of them is allowed to continue execution beyond the barrier.

Listing 2.1 shows an OpenMP program example. The code enclosed in the `parallel` construct at line 1 defines a team of M threads. The `single` construct at line 3 is used to specify that only one of the threads in the team has to execute the block between brackets in lines 4 and 19, denoted as T_0. When the thread executing T_0 encounters the `task` constructs at lines 6, 14 and 17, new tasks T_1, T_3 and T_4 are generated. Any thread of the team can execute these tasks as soon as the data dependencies are resolved. For instance, T_4 will not start its execution until T_1 finishes

```
1  #pragma omp parallel num_threads(M)
2  {
3    #pragma omp single nowait            // T0
4    {
5      part00
6      #pragma omp task depend(out:x)     // T1
7      {
8        part10
9        #pragma omp task                 // T2
10       { part20 }
11       part11
12     }
13     part01
14     #pragma omp task                   // T3
15     { part30 }
16     part02
17     #pragma omp task depend(in:x)      // T4
18     { part40 }
19   }
20 }
```

Listing 2.1: Example of an OpenMP program (tasking model).

because there exists a data dependency (T_1 produces variable x and T_4 consumes it). Moreover, the thread executing task T_1 creates task T_2. All tasks are guaranteed to have completed at the *implicit barrier* at the end of the parallel region at line 20. As an example of predecessor and descendant tasks, predecessor tasks of T_2 are T_0 and T_1, and the descendant tasks of T_0 are T_1, T_2, T_3 and T_4.

2.2.2 The OpenMP accelerator model

The OpenMP accelerator model incorporates several *Device Constructs* to create *target regions* and execute them in an accelerator device. The execution model is host-centric meaning that the host device (on which the OpenMP program begins execution) offloads target regions to target devices. When a `target` construct is encountered, a new *target task* is generated. An initial thread on the device starts the execution of the target task. If the accelerator device does not exist or the implementation does not support it, all the target regions associated with that device are executed on the host. Parallelism can be exploited within the target device through the `parallel` construct.

Some of the clauses that can be added to a `target` construct are the following:

- `if`: along with an expression which, if evaluates to false, specifies that the target task is executed in the host.

```
1 int i;
2 float p[D], v1[D], v2[D];
3 ...
4 #pragma omp target map(to: v1, v2) map(from: p)
5 {
6   for (i=0; i<D; i++)
7     p[i] = v1[i]*v2[i];
8 }
9 ...
```

Listing 2.2: Example of an OpenMP program (accelerator model).

- `device`: specifies the accelerator device identifier.
- `map`: specifies how variables are mapped between the data environments in the host and the device.
- `nowait`: allows the region in the host to continue its execution, not being necessary to wait for the device to finish its execution.
- `depend`: similarly to the `task` construct, allows to define data dependencies between target tasks and tasks executed in the host.
- `firstprivate` and `private`: similarly to the `task` construct, allows to define data-sharing attributes.

Listing 2.2 shows an example [42] of an OpenMP program using the accelerator model. It implements a simple loop over an array (lines 6-7) that may be executed in an accelerator device. The `map` clause controls the data movement of the variables $v1$, $v2$ and p to/from the memory device.

2.2.3 OpenMP tasks scheduling

The OpenMP API defines *task scheduling points* (*TSP*) as points in the program where the encountering OpenMP task can be preempted, and the hosting thread can be rescheduled to a different task. As a result, *TSPs* divide task regions into *task parts* (or simply *parts*) executed uninterrupted from start to end. The example shown in Listing 2.1 identifies the parts in which each task region is divided. For instance, T_0 is composed of $part_{00}$, $part_{01}$ and $part_{02}$.

As defined in the OpenMP specification, *TSPs* occur upon (1) explicit tasks creation; (2) implicit and explicit tasks completion; (3) explicit synchronization points such as `taskwait` directives and `taskgroup` directives; (3) implicit barriers; (4) the

`taskyield` directive, in which the current task can be suspended in favor of the execution of a different task; and (5) target regions creation and different points during the data movement between host and accelerator.

2.3 Considering OpenMP in Real-time Systems

The use of parallel programming models like OpenMP in real-time systems involves many challenges to assure that software satisfies both functional and non-functional requirements. OpenMP has been already considered as a convenient interface to describe real-time applications and deal with two features that are mandatory in such restricted systems: timing analysis and functional safety.

2.3.1 Timing analysis: the OpenMP-DAG

From a timing perspective, there is a significant amount of work considering the time predictability properties of OpenMP. Both, the fork-join and the parallel synchronous task models, were firstly considered to characterize and analyze the timing guarantees of the OpenMP thread-centric model [27]. However, the OpenMP tasking model seems to be more suitable to define fine-grain, both structured and unstructured parallelism. Vargas et al. [26] presented a first attempt to link the sporadic DAG tasks model and the OpenMP tasking model. They studied how to construct an OpenMP task graph that contains enough information to allow the use of real-time DAG scheduling models, from which timing guarantees can be derived.

Despite the current OpenMP specification lacks any notion of real-time semantics, the structure, syntax and execution of an OpenMP program, based on the tasking model, have certain similarities with the DAG tasks model. Vargas et al. presented the *OpenMP-DAG* as the DAG task representation of an OpenMP program. Moreover, the compilation techniques for automatically derive the OpenMP-DAG from an OpenMP application have been also presented [43] [44].

Thus, the execution of a *task part* in the OpenMP program resembles the execution of a sub-task (node) in a DAG task, for which WCET estimation can be derived. Edges in the DAG model can be used to represent OpenMP semantics: (1) synchronizations through the `depend` clause, which forces tasks not to be scheduled until all its predecessors have finished; (2) implicit and explicit synchronizations, for instance, through the `taskwait` construct; (3) the `if` and `final` clauses, which make the gen-

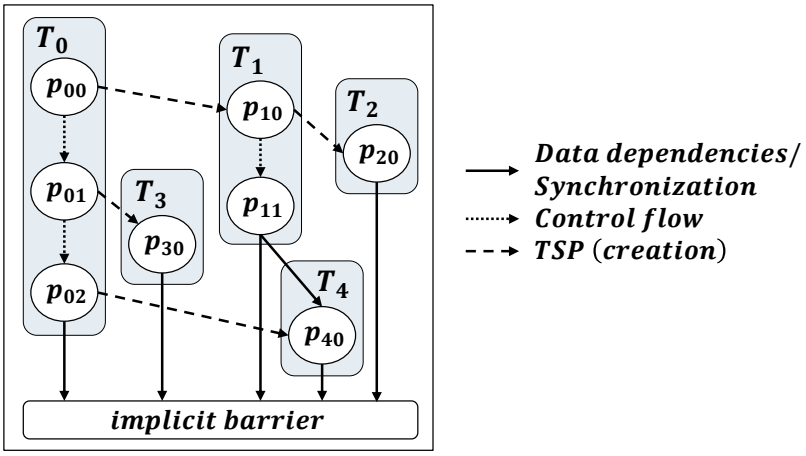


Figure 2.5: OpenMP-DAG corresponding to the OpenMP program in Listing 2.1.

erating task to be suspended until the new generated task completes execution; (4) TSPs; and (5) control flow constraints (defined by the sequential execution of task parts from the same OpenMP task). All edges express precedence constraints.

As an example, Figure 2.5 illustrates the OpenMP-DAG corresponding to the OpenMP program presented in Listing 2.1. Tasks parts, abbreviated to p_{ij}, define the nodes. For instance, the task region executed within the `single` construct, T_0, is composed of three parts or nodes p_{00}, p_{01} and p_{02}, which are sequentially executed. The creation of the OpenMP tasks T_1, T_2, T_3 and T_4 defines the edges (p_{00}, p_{10}), (p_{10}, p_{20}), (p_{01}, p_{30}) and (p_{02}, p_{40}). Moreover, the data dependency between tasks T_1 and T_4 define the edge (p_{11}, p_{40}). Finally, the implicit barrier at the end of the `parallel` construct defines the edges at the end of each OpenMP task T_0 to T_4.

2.3.2 Functional safety

This thesis focuses on the timing and scheduling requirements of OpenMP to be considered in real-time systems. However, from a functional safety perspective, OpenMP has been already considered as a convenient candidate to implement real-time systems.

Recent works study the functional verification of OpenMP programs, demonstrating the benefits of using OpenMP in real-time embedded systems, even though some features and restrictions must be addressed [45]. Based on the potential of existent correctness techniques for OpenMP, both at compile time [46, 47, 44] and run time [48, 49] levels, it could be introduced in safe languages such as Ada [50, 51, 52], widely used to implement safety-critical systems. The Ada Rapporteur Group is cur-

rently considering the introduction of OpenMP into Ada [53] to exploit fine-grain parallelism. The functional correctness of OpenMP is out of the scope of this thesis.

2.4 Experimental Setup

In order to evaluate the effectiveness of the techniques proposed in this dissertation, we face the problem of generating a large number of DAG-based task-sets with different characteristics. To do so, a mechanism to randomly generate synthetic DAG tasks (section 2.4.1) and task-sets with different parameters (section 2.4.2) has been developed and implemented in MATLAB®. Section 2.4.3 demonstrates how our algorithm can be used to obtain the DAG representation of a real OpenMP application.

2.4.1 Synthetic DAGs generation

An algorithm to randomly generate synthetic DAG tasks has been developed, based on the simulation environment presented in [54].

A DAG task is recursively created by expanding it in each iteration, either to a single node, or to a new parallel sub-graph. Sub-graphs consist of a source node, a sink node, and a random number of parallel branches, further expanded in the successive iterations. The recursive procedure finishes when one of these situations occurs:

1. all branches of the parallel sub-graph are expanded to single nodes;
2. a maximum number of nodes, given by max_{nodes}, is reached; or
3. a maximum recursion depth, given by max_{depth}, is reached.

The probabilities that control if a branch is expanded to a single node or to a parallel sub-graph are p_{term} and p_{par}, respectively (subject to the relation $p_{term} + p_{par} = 1$). The maximum number of branches of parallel sub-graphs is max_{par}. Based on this value, whenever a given branch is expanded to a new parallel sub-graph, the new number of branches is uniformly selected in $[0, max_{par}]$. An additional parameter $p_{dep} \in [0, 1]$ is used to add edges between non-connected nodes[4] so that unstructured parallelism is represented. Notice that if $p_{dep} > 0$, transitive edges[5] may be created in

[4] A pair of nodes u, v are *non-connected* if there is no path from u to v, and no path from v to u.

[5] An edge (u, v) is transitive if v can be reached from u following an alternative path, for instance (u, w) and (w, v).

the DAG. Finally, the WCET of each node is uniformly selected as a positive integer in the interval $[C^{Min}, C^{Max}]$.

Algorithm 1 presents the pseudo-code implementation of the synthetic DAG task generation tool. In order to facilitate the explanation of the algorithm, Figure 2.6 presents a graphical representation of the recursive iterations needed to randomly generate a DAG task, assuming the following input global parameters: $p_{term} = 0.4$ (thus, $p_{par} = 0.6$), $p_{dep} = 0.1$, $max_{par} = 4$, $max_{depth} = 5$ and $max_{nodes} = 30$.

The algorithm starts with the function RANDOM_DAG that

1. initializes the set of nodes V with the source v_1 and sink v_2 nodes (line 2);
2. initializes the set of edges E to empty (line 3);
3. randomly selects the number of parallel branches par in the interval $[0, max_{par}]$; as max_{nodes} could be reached, the minimum between max_{par} and the current number of nodes allowed to be created, $max_{nodes} - |V|$, is selected (line 4);
4. initializes the variable nds that stores the number of nodes that, at least, will be created, i.e., the current number of nodes, $|V|$, in addition to the number of branches that will be created, par, with at least one node per branch (line 5);
5. continues the DAG task generation by calling the recursive function EXPAND_TASK (line 6); and
6. randomly adds extra edges to the set E by calling the function EXTRA_EDGES (line 7).

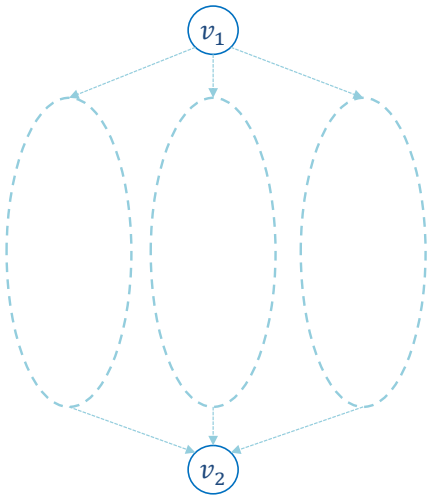
This algorithm considers two variables to count the number of nodes: nds and $|V|$. The reason is that $|V|$ considers the current number of nodes already created and nds considers the nodes that, at least, will be created (given the number of branches, i.e., the par variable) in the successive iterations of the loop in line 13. The final number of nodes created will depend on the value of the random numbers within each iteration. Figure 2.6a shows the results of the first part of the function RANDOM_DAG, before calling to EXPAND_TASK. It assumes that par is set to 3, and so 3 parallel sub-graphs, represented as dashed ovals, will be further expanded in the successive iterations of the algorithm. Consequently, nds is equal to 5.

The function EXPAND_TASK recursively expands each of the par branches. It proceeds as follows: if there are no branches to create ($par = 0$), an edge between current source and sink nodes is created (line 11), otherwise it iterates over each branch (line 13) and

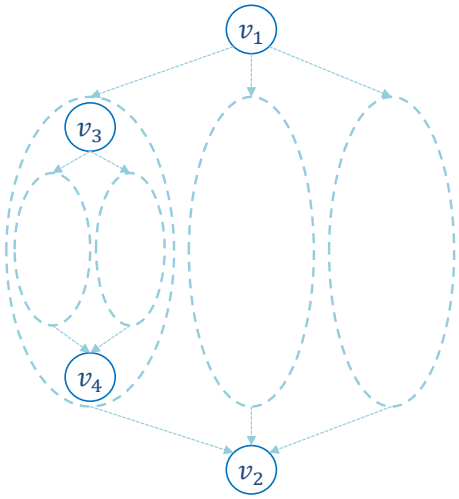
Algorithm 1 Generate random DAG $G = (V, E)$

Global: p_{term}: Probability of a branch to be a terminal node
p_{dep}: Probability of adding an edge between non-connected nodes
max_{par}: Maximum number of branches of each parallel sub-graph
max_{depth}: Maximum recursion depth of the DAG task
max_{nodes}: Maximum number of nodes of the DAG task

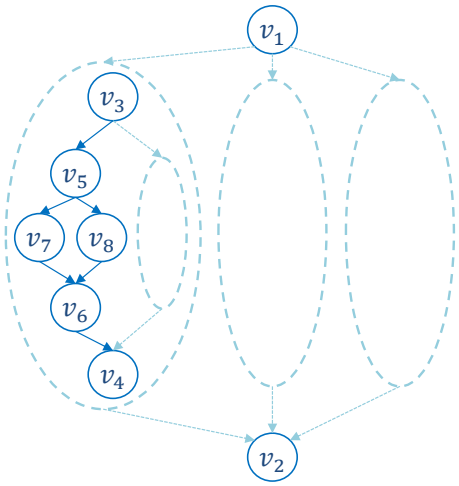
```
1  function RANDOM_DAG return (V, E)
2      V ← {v_1, v_2}
3      E ← ∅
4      par ← RANDOM([0, min(max_nodes − |V|, max_par)])
5      nds ← |V| + par
6      (V, E) ← EXPAND_TASK(V, E, v_1, v_2, max_depth − 1, par, nds)
7      E ← EXTRA_EDGES(V, E)
8  end function
9  function EXPAND_TASK(V, E, v^source, v^sink, depth, par, nds) return (V, E, nds)
10     if par == 0 then
11         E ← E ∪ (v^source, v^sink)
12     else
13         for each i ∈ [1, par] do
14             j ← |V| + 1
15             p ← RANDOM([0, 1])
16             if (p ≤ p_term) ‖ (nds == max_nodes) ‖ (depth == 0) then
17                 V ← V ∪ {v_j}
18                 E ← E ∪ (v^source, v_j) ∪ (v_j, v^sink)
19             else
20                 V ← V ∪ {v_j, v_{j+1}}
21                 E ← E ∪ (v^source, v_j) ∪ (v_{j+1}, v^sink)
22                 parS ← RANDOM([0, min(max_nodes − (nds + 1), max_par)])
23                 nds ← nds + 1 + parS
24                 (V, E, nds) ← EXPAND_TASK(V, E, v_j, v_{j+1}, depth − 1, parS, nds)
25             end if
26         end for
27     end if
28 end function
29 function EXTRA_EDGES(V, E) return E
30     for each v_i ∈ |V| do
31         for each v_j ∈ |V| do
32             if NON_CONNECTED(v_i, v_j) && (RANDOM([0, 1]) ≤ p_dep) then
33                 E ← E ∪ (v_i, v_j)
34             end if
35         end for
36     end for
37 end function
```



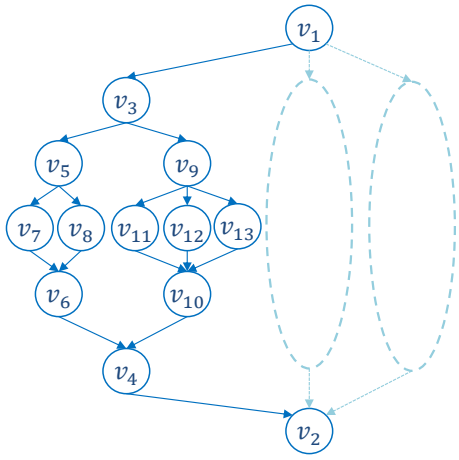
(a) RANDOM_DAG ($par = 3$, $nds = 5$).



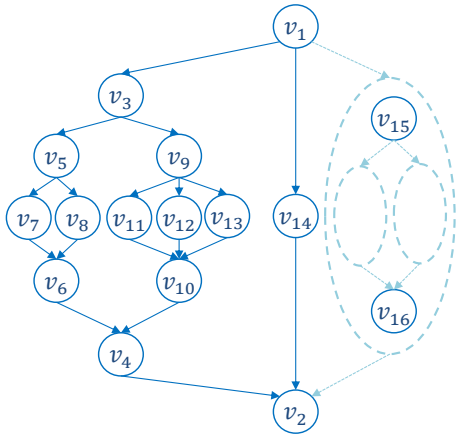
(b) EXPAND_TASK: recursive iter. 1 ($parS = 2$, $nds = 8$).



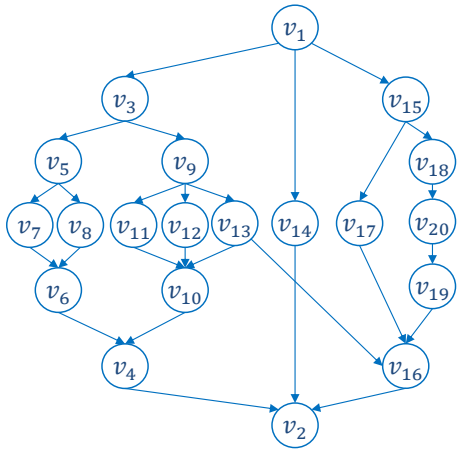
(c) EXPAND_TASK: recursive iters. 2 ($parS = 2$, $nds = 11$), 3 & 4.



(d) EXPAND_TASK: recursive iters. 2 ($parS = 3$, $nds = 15$), 5, 6 & 7.



(e) EXPAND_TASK: recursive iter. 1 ($parS = 2$, $nds = 18$).



(f) EXPAND_TASK: recursive iters. 8, 9 ($parS = 1$, $nds = 20$) & 10. EXTRA_EDGES.

Figure 2.6: Recursive random DAG generation.

1. enumerates the new node, by giving the j value (line 14); (2) randomly computes $p \in [0, 1]$ which, compared to p_{term}, determines if the branch is a single node ($p \leq p_{term}$) or a parallel sub-graph ($p > p_{term}$) (line 15);
2. if $p \leq p_{term}$, or max_{nodes} or max_{depth} have been reached, then a new single node v_j is created (line 17) and also its edges from current source and sink nodes (line 18);
3. otherwise, a new parallel sub-graph is created. For this new sub-graph:
   3.1. v_j and v_{j+1} will be the new source and sink nodes (line 20);
   3.2. edges to the current source and sink nodes are created (line 21);
   3.3. the new number of parallel branches $parS$ is randomly selected in the interval $[0, max_{par}]$, again unless max_{nodes} can be reached (line 22);
   3.4. variable nds is updated to the nodes already considered plus the extra node v_{j+1} of the current parallel subgraph and the parallel branches of the recursive parallel sub-graph (line 23);
   3.5. the parallel sub-graph is further expanded (line 24).

Figures 2.6b to 2.6f show an example of the recursive iterations of the function EXPAND_TASK.

- Figure 2.6b shows the first recursive iteration:
  - The first branch is further expanded to a new parallel sub-graph, delimited by nodes v_3 and v_4, with two new branches ($parS = 2$ and $nds = 8$).
- Figure 2.6c shows:
  - The recursive iteration 2: a branch is further expanded to a new parallel sub-graph, delimited by nodes v_5 and v_6, with two new branches ($parS = 2$ and $nds = 11$).
  - Since the maximum recursion depth 3 has been reached, the recursive iterations 3 and 4, expand these two branches to single nodes v_7 and v_8, respectively.
- Similarly, Figure 2.6d shows:

  - Back to recursive iteration 2, the second branch is further expanded to a new parallel sub-graph, delimited by nodes v_9 and v_{10}, with two new branches ($parS = 3$ and $nds = 15$).
  - Since $max_{depth} = 3$, recursive iterations 5, 6 and 7, expand these three branches to single nodes v_{11}, v_{12} and v_{13}, respectively.

- Figure 2.6e shows, back to recursive iteration 1:
  - The second branch is expanded to a single node v_{14} because $p \leq p_{term}$.
  - The third branch is further expanded to a new parallel sub-graph $p > p_{term}$, delimited by nodes v_{15} and v_{16}, with two new branches ($parS = 2$ and $nds = 18$).

- Figure 2.6f shows:
  - The recursive iteration 8: the first branch is expanded to a single node v_{17}.
  - The recursive iteration 9: the second branch is further expanded to a new parallel sub-graph, delimited by nodes v_{18} and v_{19}, with only one new branch ($parS = 1$ and $nds = 20$). Since the maximum recursion depth 3 has been reached, the recursive iteration 10 expands this branch to a single node v_{20}.

Finally, the function EXTRA_EDGES includes additional edges into the set E. It is a simple procedure that iterates over all pair of nodes, $v_i \in V$ (line 30) and $v_j \in V$ (line 31). If v_i and v_j are non-connected, and based on the probability p_{dep} (line 32), a new edge (v_i, v_j) is created (line 33). Figure 2.6f shows an example of the result of this function. An additional edge between nodes v_{13} and v_{16} is created.

Although not included in Algorithm 1, the MATLAB implementation of the DAG task generation tool also assigns a WCET to each node in V, as a random integer in the set $[C^{Min}, C^{Max}]$.

2.4.2 Parametrized task-sets generation

Based on the synthetic DAG task generation tool presented in the previous section, a task-set composed of n DAG tasks is built. Given a number of cores m, this allows to evaluate the scheduling policies at system level. We consider two methods to

randomly generate task-sets: with a target utilization $U_{\mathcal{T}}$, or with a target number of tasks n.

Task-sets with a target utilization. Given a fixed number of cores m, task-sets with a target utilization $U_{\mathcal{T}}$ is generated. The total number of DAG tasks n is randomly selected in the interval $[n_{min}, n_{max}]$. To do so, an iterative procedure is implemented based on the following steps:

1. Generate a random DAG task $G_k = (V_k, E_k)$.
2. Compute $len(G_k)$ and $vol(G_k)$.
3. Randomly select the period T_k as an integer in the interval

$$\left[\frac{vol(G_k)}{U_{\mathcal{T}}/n_{min}}, \frac{vol(G_k)}{U_{\mathcal{T}}/n_{max}}\right].$$

   As a result, the utilization U_k of each DAG task is selected in the interval $\left[\frac{U_{\mathcal{T}}}{n_{max}}, \frac{U_{\mathcal{T}}}{n_{min}}\right]$.
4. Set the deadline D_k considering the implicit deadline case, i.e., $D_k = T_k$.

The system utilization $U_{\mathcal{T}}$ is accumulatively computed at the end of each iteration. The iterative procedure finishes whenever the desired utilization is exceeded. Then the period of the last task is increased so that the exact system utilization is reached.

Task-sets with a target utilization and number of tasks. Similarly, given a fixed number of cores m and a fixed system utilization $U_{\mathcal{T}}$, task-sets with a target number of tasks n are generated. In this case, the procedure is identical to the one described before, except that $n_{min} = n_{max} = n$. As a result, all the DAG tasks have the same utilization U_k.

Table 2.2 summarizes the notation described in this section.

2.4.3 Synthetic DAG tasks and OpenMP applications

It may seem that the synthetic generation tool, presented in Section 2.4.1, creates DAG tasks with a structured fork-join parallelism. However, the use of the p_{dep} parameter, as well as the transitive edges, allow also to represent and characterize the unstructured parallelism that can be expressed with the OpenMP tasking model.

p_{term}	Probability of a branch to be expanded to a node
p_{par}	Probability of a brach to be expanded to a parallel sub-graph
p_{dep}	Probability of adding an edge between non-connected nodes
max_{par}	Maximum number of branches of a parallel sub-graph
max_{nodes}	Maximum number of nodes of any random DAG task
max_{depth}	Maximum recursion depth when creating a random DAG task ($max_{depth} \times 2 + 1$ represents the maximum number of nodes in λ_k^*)
C^{Min}	Minimum WCET $C_{k,i}$ of any node in a DAG task
C^{Max}	Maximum WCET $C_{k,i}$ of any node in a DAG task
n_{min}	Minimum number of DAG tasks n of any task-set
n_{max}	Maximum number of DAG tasks n of any task-set

Table 2.2: Experimental setup notation.

As an example, we show how it is possible to obtain the OpenMP-DAG of a real OpenMP application with the methodology presented in Algorithm 1. The Cholesky factorization [55] is a useful application for efficient linear equation solvers and Monte Carlo simulations. Moreover, Cholesky can also be used to accelerate *Kalman filter*, implemented in autonomous vehicle navigation systems to detect pedestrians and bicycle positions [56]. The Cholesky factorization is the classical application that exploits unstructured parallelism and so perfectly fits the OpenMP tasking model. The most representative function of this application can be found in Appendix B.3.

Figure 2.7 shows the Task Dependency Graph (TDG) of the Cholesky application, as obtained from the source code by the compiler technique presented by Royuela in her PhD dissertation [57], when setting the variable $NB = 4$. The TDG is the graphical representation, similar to a DAG, of the OpenMP tasks and the synchronizations among them. Each node is labeled with a number that unambiguously identifies it. Each color represents a different OpenMP task construct, hence red, light green, purple and blue nodes correspond to the task constructs in lines 9, 12-13, 18-20 and 23-24, of the code in Appendix B.3, respectively.

Figure 2.8 shows the graphical representation of Algorithm 1 operations which provide the Cholesky OpenMP-DAG. Certainly, the input parameters are not relevant for this purpose, but we can establish a lower bound: $p_{term} > 0$, $p_{dep} > 0$, $max_{par} \geq 3$, $max_{depth} \geq 3$ and $max_{nodes} \geq 20$.

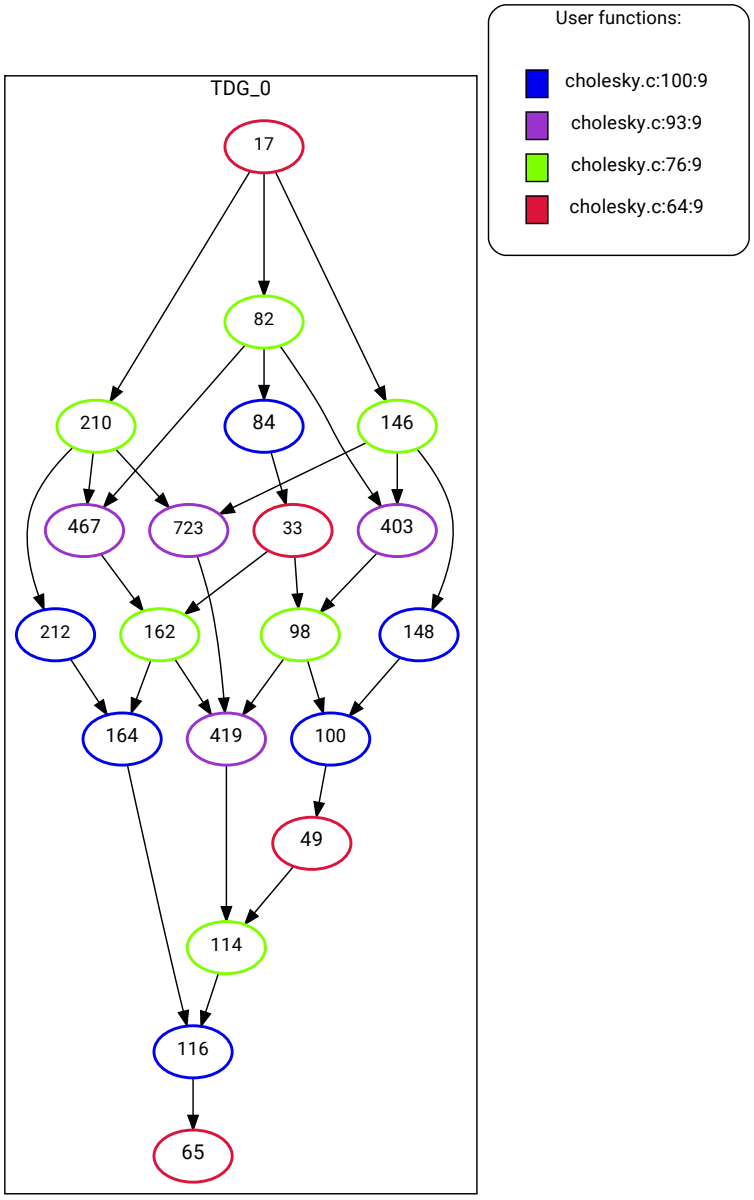
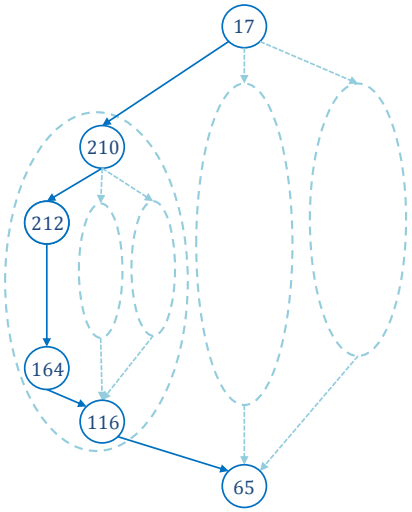
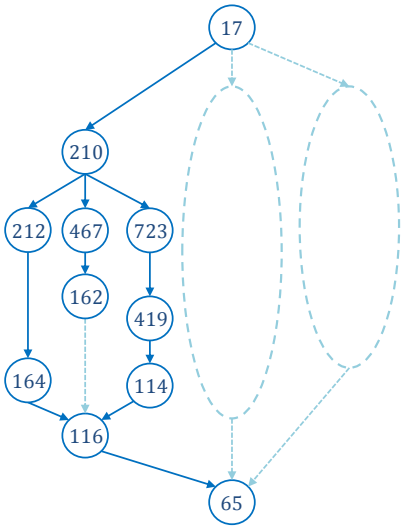


Figure 2.7: Cholesky TDG.

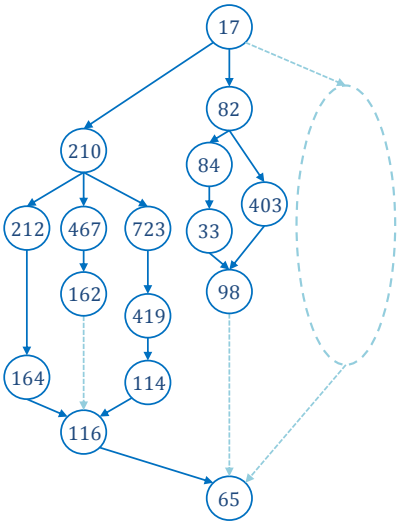
- Figure 2.8a shows:
  - The RANDOM_DAG function, which creates the source and sink nodes, 17 and 65, respectively, and starts the recursive procedure ($par = 3$ and $nds = 5$).
  - The EXPAND_TASK function, recursive iteration 1, where the first branch is further expanded to a new parallel sub-graph, delimited by nodes 210 and 116, with three new branches ($parS = 3$ and $nds = 9$).
  - The EXPAND_TASK function, recursive iteration 2, where the first branch



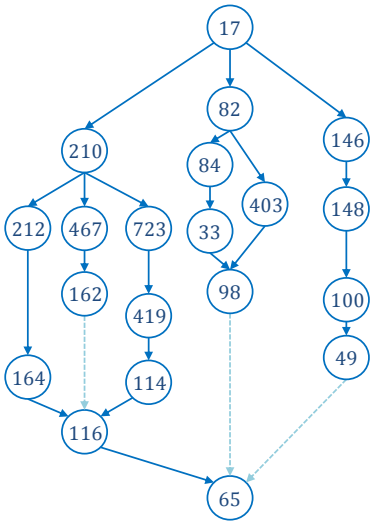
(a) Initialization. Recursive iters. 1, 2 and 3.



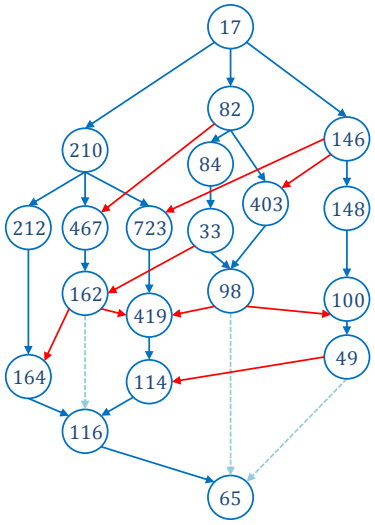
(b) Recursive iters. 2, 4 and 5.



(c) Recursive iters. 1, 6 and 7.



(d) Recursive iters. 1, 8 and 9.



(e) Additional edges created.

Figure 2.8: Cholesky OpenMP-DAG generation.

is further expanded to a new parallel sub-graph, delimited by nodes 212 and 164, without new branches ($parS = 0$ and $nds = 10$).

- The EXPAND_TASK function, recursive iteration 3, where only the edge $(212, 164)$ is created, because par equals to 0.

- Figure 2.8b shows the EXPAND_TASK function:
  - Back to recursive iteration 2, where the second branch is further expanded to a new parallel sub-graph, delimited by nodes 467 and 162, without new branches ($parS = 0$ and $nds = 11$).
  - Recursive iteration 4, where only the edge $(467, 162)$ is created.
  - Back to recursive iteration 2, where the third branch is further expanded to a new parallel sub-graph, delimited by nodes 723 and 114, with a new branch ($parS = 1$ and $nds = 13$).
  - Recursive iteration 5, where the branch is expanded to a terminal node 419.

- Similarly, Figure 2.8c shows the EXPAND_TASK function:
  - Back to recursive iteration 1, the second branch is further expanded to a new parallel sub-graph, delimited by nodes 82 and 98, with two new branches ($parS = 2$ and $nds = 16$).
  - Recursive iteration 6, expands the first branch to a new parallel sub-graph, delimited by nodes 84 and 33, without new branches ($parS = 0$ and $nds = 17$).
  - Recursive iteration 7, where only the edge $(84, 33)$ is created.
  - Back to recursive iteration 6, the second branch is expanded to a terminal node 403.

- Figure 2.8d shows the EXPAND_TASK function:
  - Back to recursive iteration 1, the third branch is expanded to a new parallel sub-graph, delimited by nodes 146 and 49, with only a new branch ($parS = 1$ and $nds = 19$).
  - Recursive iterations 8 expands the branch to a new parallel sub-graph, delimited by nodes 148 and 100, without new branches ($parS = 0$ and $nds = 20$).

  - Recursive iteration 9, where only the edge $(148, 100)$ is created.

- Figure 2.8e shows in red the additional edges created by the EXTRA_EDGES function.

As a result, if the dashed light-blue edges $(162, 116)$, $(98, 65)$ and $(49, 65)$ are ignored, we obtain the TDG shown in Figure 2.7. These edges can be ignored because they are transitive edges, and the execution order imposed by them is honored by alternative edges. As an example, the execution order imposed by the edge $(98, 65)$ is honored by the edges $(98, 100)$, $(100, 49)$ and $(49, 65)$. Nevertheless, the response time analysis provides the same results for a DAG with and without transitive edges.

Chapter 3

Developing Critical Real-Time Embedded Systems with OpenMP

"*A ship in port is safe; but that is not what ships are built for.*"

— Grace Murray Hopper

The similarities between the DAG tasks model and the OpenMP tasking model allow, as shown in previous Section, to parallelize real-time tasks based on the OpenMP tasking model, and to represent their execution as a DAG. This chapter analyzes the use of OpenMP to implement critical real-time embedded systems. We also focus on the design implications and the scheduling decisions to efficiently exploit fine-grain parallelism within real-time tasks and concurrency among them. The goal is twofold: (1) to use OpenMP to represent the recurrence of real-time tasks, and to exploit parallelism at system and tasks levels, and (2) to extend the OpenMP specification to incorporate the missed properties that are common in real-time systems. And all this while guaranteeing the timing behavior of the system, according to current real-time practices. We also evaluate three available OpenMP runtime implementations to show their strengths and limitations when targeting real-time systems.

3.1 The OpenMP Tasking Model to Implement Critical Real-Time Embedded Systems

This section analyzes the use of the OpenMP tasking model to implement critical real-time embedded systems, from two different perspectives: (1) how to efficiently

```
1 // τ1
2 void RT_task_1()
3 {
4   for (...) {
5     #pragma omp task
6     ...
7   }
8 }
```

```
1 // τ2
2 void RT_task_2()
3 {
4   #pragma omp task
5   ...
6   #pragma omp task
7   ...
8 }
```

```
1  // τn
2  void RT_task_n()
3  {
4    #pragma omp task depend(out:x)
5    ...
6    #pragma omp task
7    ...
8    #pragma omp task depend(in:x)
9    ...
10 }
```

Listing 3.1: Example of real-time tasks parallelized with OpenMP.

exploit parallelism within real-time tasks and among them, and (2) how to express the recurrence of real-time tasks.

3.1.1 Parallelizing real-time systems

In critical real-time systems, the scheduler plays a key role as it must be guaranteed that all real-time tasks execute before their deadline. To do so, real-time schedulers implement the following features (presented in Section 3.2): (1) *tasks priorities*, which determine the urgency of each real-time task to execute (e.g., the smaller priority value, the more urgent task); (2) *preemption strategies*, which determine when a real-time task can be temporarily interrupted if, for instance, a more urgent task is ready to execute; and (3) *allocation strategies*, which determine the computing resources (cores) in which tasks can execute.

The first approach that one might consider to develop a real-time system with OpenMP is to implement each real-time task of the system as an independent OpenMP application, i.e., each using its own instance of the OpenMP runtime, as considered in previous works [26, 43]. In order to illustrate this approach, first consider the example of a set of real-time tasks $\{\tau_1, \tau_2, \ldots \tau_n\}$, as shown in Listing 3.1. Then, Listing 3.2 shows the parallelization strategy of a real-time system in which each real-time task

```
1 // τ1 : OpenMP-DAG1
2 void main()
3 {
4   #pragma omp parallel
5   #pragma omp single nowait
6   {
7      RT_task_1();
8   }
9 }
```

```
1 // τ2 : OpenMP-DAG2
2 void main()
3 {
4   #pragma omp parallel
5   #pragma omp single nowait
6   {
7      RT_task_2();
8   }
9 }
```

```
...
1 // τn : OpenMP-DAGn
2 void main()
3 {
4   #pragma omp parallel
5   #pragma omp single nowait
6   {
7      RT_task_n();
8   }
9 }
```

Listing 3.2: Example of real-time system implemented as independent OpenMP applications.

$\tau_i \in \mathcal{T}$ is independently encapsulated within an OpenMP application. However, this approach presents a fundamental problem: the OpenMP parallel environment becomes a black box for a common scheduler, which can not control how the resources are used internally by each real-time task. For instance, TSP are not exposed to the common scheduler. Therefore, a different approach must be considered.

In order for the scheduler to have full control over the execution of the real-time tasks (and their parallel execution), the complete task-set must be included within a single parallel environment, i.e., a single OpenMP application. To do so, one option is to exploit nested parallel regions, i.e., to enclose the real-time tasks, each defining its own parallel region, within an outer parallel region. In this case, the OpenMP framework manages two scheduling levels: one in charge of scheduling the real-time tasks (outer parallel region), and another one in charge of scheduling the parallel execution within each real-time task (inner parallel regions). Listing 3.3 shows an example of a real-time system where each real-time task $\tau_i \in \mathcal{T}$ (as shown in Listing 3.1) is encapsulated within an OpenMP parallel region. However, this solution is, again, not valid as the first-level scheduler cannot control the parallel execution of each real-time task. In this case, the team of threads of each real-time task is a black box for the first-level scheduler.

```
1  #pragma omp parallel
2  #pragma omp single nowait
3  {
4     #pragma omp parallel                    // τ1 : OpenMP-DAG1
5     #pragma omp single nowait
6     {
7       RT_task_1();
8     }
9     #pragma omp parallel                    // τ2 : OpenMP-DAG2
10    #pragma omp single nowait
11    {
12      RT_task_2();
13    }
14    ...
15    #pragma omp parallel                    // τn : OpenMP-DAGn
16    #pragma omp single nowait
17    {
18      RT_task_n();
19    }
20 }
```

Listing 3.3: Example of real-time system implemented as a single OpenMP application with nested parallel regions.

The control of the OpenMP threads executing each of the real-time tasks is key to implement the real-time scheduling mechanisms over the whole parallel execution. To do so, we propose to define *a common team of OpenMP threads to execute all the real-time tasks*. In this approach, a single real-time scheduler will be in charge of scheduling both, the OpenMP tasks implementing the real-time tasks (with an associated priority given by the `priority` clause), and the nested OpenMP tasks implementing the parallel execution of each real-time task. Interestingly, this approach enables the scheduler to use the `priority` clause associated to the `task` construct to determine the priority of each real-time task (see Section 3.2). Listing 3.4 shows the implementation of a real-time system in which each real-time task $\tau_i \in \mathcal{T}$ (as shown in Listing 3.1) is encapsulated within an OpenMP task, and parallelized with nested OpenMP tasks. A `taskwait` synchronization construct must be included at the end of each real-time task τ_i since the OpenMP task implementing τ_i cannot finish until the nested OpenMP tasks finish their execution.

Notice the overlapping use of the term "task". According to this approach, a real-time task (represented as a DAG task) is implemented as an OpenMP task and parallelized with nested OpenMP tasks.

```
1  #pragma omp parallel
2  #pragma omp single nowait
3  {
4     #pragma omp task priority(p_1)      // τ_1 : OpenMP-DAG_1
5     {
6       RT_task_1();
7       #pragma omp taskwait
8     }
9     #pragma omp task priority(p_2)      // τ_2 : OpenMP-DAG_2
10    {
11      RT_task_2();
12      #pragma omp taskwait
13    }
14    ...
15    #pragma omp task priority(p_n)      // τ_n : OpenMP-DAG_n
16    {
17      RT_task_n();
18      #pragma omp taskwait
19    }
20 }
```

Listing 3.4: Example of real-time system implemented as OpenMP nested tasks (with a common team of threads).

3.1.2 Implementing recurrent real-time tasks in OpenMP

Despite the suitability of the OpenMP tasking model to implement critical real-time systems based on DAG scheduling models, OpenMP lacks an important feature: *the notion of recurrence*. Real-time tasks can be either periodic or sporadic, triggered by an event, e.g., an internal clock or a sensor.

With the objective of including recurrence in the OpenMP execution model, we propose to incorporate a new clause, named `event`, associated to the `task` construct. This clause enables to define the release time of the OpenMP tasks implementing real-time tasks. The syntax of the `event` clause is as follows:

`#pragma omp task event(`*event-expression*`)`

where only if *event-expression* evaluates to true, the associated OpenMP task is created. This expression represents the exact moment in time[1] at which the real-time task release occurs, or the condition of the external event to occur and release a new job of the associated real-time task. The task is released whenever the expression is true, and the expression shall evaluate to false after the task creation. Interestingly, this new `event` clause would allow to unequivocally identify which OpenMP tasks

[1]Real-Time Operating Systems (RTOS) provide time management mechanisms and timers to determine the release time or deadline of real-time tasks.

implement real-time tasks, differentiating them from the OpenMP tasks used to parallelize each real-time task. The real-time system implemented in Listing 3.4 must therefore include the `event` clause associated to each `task` construct at lines 4, 11 and 19.

However, the `event` clause is not enough to state the synchrony between the event that triggers a real-time task and the actual execution of that task. In languages such as Ada, which are intrinsically concurrent, events are treated at the base language level, thus an Ada task triggering an event will launch an *entry* (a functionality) of a different task [58]. But OpenMP is defined on top of C, C++ and Fortran, languages intrinsically sequential[2], that do not typically provide these kind of features. Following, we analyze three different approaches to associate the occurrence of an event and the execution of a real-time task in OpenMP:

- *Managed by the base language*: a simple approach would use the base language to implement an infinite control loop containing the set of real-time tasks with their corresponding events and priorities. In Listing 3.4, this loop could wrap lines between 4 and 27. Then, the creation of the OpenMP real-time tasks could be managed by the `event` clause. This solution however renders one thread useless, executing the control loop.

- *Managed by the operating system*: based on the previous approach, the thread executing the control loop may be freed at the end of each iteration, and the operating system may return the thread to the control loop in a period of time shorter than the minimum period of a task (ensuring no job is missed).

- *Managed by the OpenMP API*: a different approach would be implementing the concept of *persistent* task [59] in the OpenMP API, pushing the responsibility for checking the occurrence of an event to the OpenMP runtime.

While this chapter focuses on the analysis of the OpenMP specification, a deeper evaluation of the most suitable solution to implement recurrence is of paramount importance to promote the use of OpenMP in critical real-time environments. This evaluation is out of the scope of this thesis and remains as a future work.

[2] C++11 introduced multi-threading support, adding features to define concurrent execution.

3.2 Real-Time Scheduling Features in the OpenMP Task Scheduler

One of the most important components of critical real-time systems is the *real-time scheduler*, in charge of assigning the execution of real-time tasks to the underlying computing resources. The real-time scheduler behavior must conform to the scheduling policy considered in the schedulability analysis, so that it can be guaranteed that all tasks execute before their deadline. In the context of real-time systems, when several tasks are considered, scheduling algorithms are normally priority driven [60], i.e., real-time tasks (or jobs) have a *priority* assigned and the preference to execute is given to the higher-priority tasks. Hence, the scheduler is allowed to interrupt (preempt) a running task if a more urgent (higher priority) task is ready to execute. The preempted task can later resume its execution. Moreover, scheduling algorithms place additional restrictions as to where tasks are allowed to executed. Therefore, real-time schedulers are commonly classified based on (1) task priorities, (2) preemption strategies and (3) migration strategies.

3.2.1 Priority-driven scheduler algorithms

There exist several priority-based scheduling algorithms, which are classified based on the restrictions on how to assign priorities to real-time tasks [34]:

- In *Fixed Task Priority (FTP) scheduling*, each real-time task has a unique fixed priority. For instance, the Rate-Monotonic (RM) scheduling algorithm establishes the priorities based on the period T_i, i.e., tasks with smaller periods have greater priority.
- In *Fixed Job Priority (FJP) scheduling*, different jobs of the same real-time task may have different priority. For instance, the Earliest Deadline First (EDF) scheduling algorithm establishes the priorities based on the deadline D_i, i.e., jobs with earlier deadlines have greater priority.
- In *Dynamic Priority (DP) scheduling*, there are no restrictions on the manner priorities are assigned, i.e., the priority of each job may change between its release time and its completion. For instance, the Least Laxity (LL) scheduling algorithm assigns the priorities based on the *laxity* of a job, which at any instant

in time, is defined as its deadline minus the sum of its remaining processing time and the current time.

In OpenMP, the `priority(`*priority-value*`)` clause associated to the `task` construct can be used to represent the priority of real-time tasks for the FTP scheduling. The *priority-value* is a non-negative numerical scalar expression. A higher numerical value indicates a higher priority. However, the OpenMP specification (version 4.5) states that *"the priority clause is a hint for the priority of the generated task [..] Among all tasks ready to be executed, higher priority tasks are recommended to execute before lower priority ones. [...] A program that relies on task execution order being determined by this priority-value may have unspecified behavior"*. As a result, the current behavior of the `priority` clause does not guarantee the correct priority-based execution order of real-time tasks. Therefore, the development of OpenMP task schedulers in which the `priority` clause truly leads the scheduling behavior is essential for real-time systems. Moreover, the *priority-expression* value defined at real-time task level must be inherited by the corresponding child tasks implementing parallelism within each real-time task. By doing so, the OpenMP task scheduler can preempt the inner OpenMP tasks exploiting parallelism of low priority real-time tasks in favor of inner OpenMP tasks exploiting parallelism of higher priority real-time tasks.

Regarding the implementation of EDF and LL schedulers, a new clause, named `deadline`, associated to the `task` construct is needed. This clause will enable to define the deadline of the real-time task upon which EDF and LL schedulers are based. We define the syntax of the `deadline` clause as follows:

`#pragma omp task deadline(`*deadline-expression*`)`

where the *deadline-expression* is the expression that determines the time instant at which the OpenMP task must finish. Similarly to the `priority` clause, the *deadline-expression* associated to an OpenMP task implementing a real-time task must be inherited by all its child tasks. This allows the scheduler to identify those OpenMP tasks with the farthest deadline, and preempt them to assign the corresponding OpenMP threads to those tasks with the closest deadline.

Listing 3.5 shows an example of an OpenMP real-time system, when the scheduler is EDF or LL, and so the `deadline` clause is required. Real-time tasks $\tau_1, \tau_2 ... \tau_n$ have a deadline and an event associated to them. All child tasks inherit the deadline of the OpenMP parent task, for instance, for the OpenMP real-time task τ_1, OpenMP

```
1  #pragma omp parallel
2  #pragma omp single nowait
3  {
4    while(1) {
5      #pragma omp task deadline(D_1) event(e_1)  // τ_1 : OpenMP-DAG_1
6      {
7        #pragma omp task depend(out:x)    // T_1
8        { ... }
9        #pragma omp task                  // T_2
10       { ... }
11       #pragma omp task depend(in:x)     // T_3
12       { ... }
13       #pragma omp taskwait
14     }
15     #pragma omp task deadline(D_2) event(e_2)  // τ_2 : OpenMP-DAG_2
16     {  ...  }
17     ...
18     #pragma omp task deadline(D_n) event(e_n)  // τ_n : OpenMP-DAG_n
19     {  ...  }
20   }
21 }
```

Listing 3.5: OpenMP real-time system desig for a deadline-based scheduler.

tasks T_1, T_2 and T_3 inherit the deadline D_1. Notice that, compared to a fixed task priority scheduler, the only difference is that the `deadline` clause would be replaced by a `priority` clause. The `deadline` clause is not compatible with the `priority` clause, if both are meant for determining the priority of a task for different scheduling algorithms. However, the deadline may be compatible with a fixed task priority scheduler, if timing correctness is addressed (as an upper bound of the response time of the task).

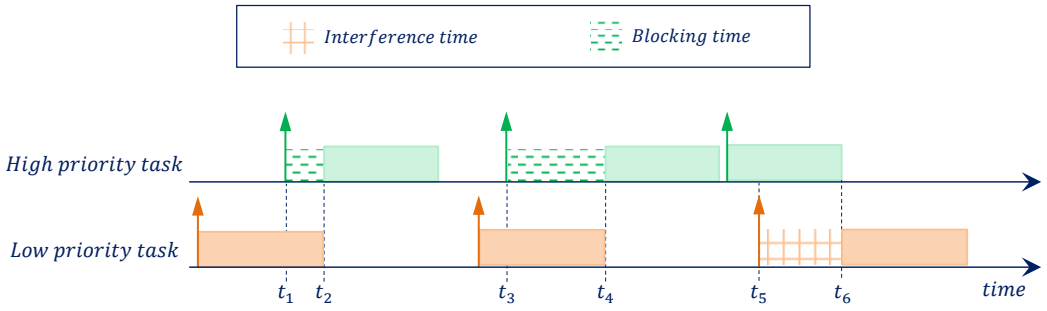
3.2.2 Preemption strategies

Preemptive scheduling permits a task executing upon any processor to be interrupted by the scheduler, and to be resumed at a later point in time. There are several preemption strategies that can be considered in real-time systems:

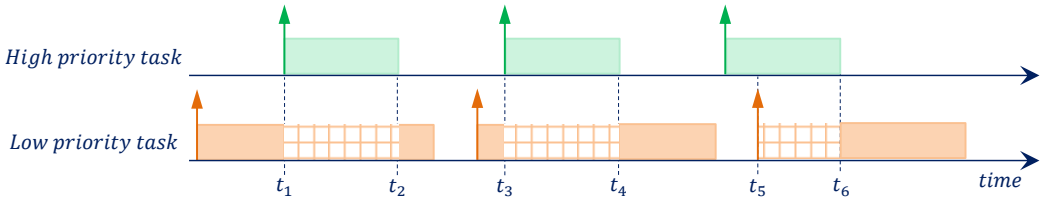
- In *non-preemptive* scheduling [61], preemption is completely forbidden, i.e., jobs are executed until completion, without interruption. This strategy achieves higher degree of predictability, at the cost of higher blocking times and blocking effects to higher-priority tasks. For instance, a high priority task τ may have access to less cores than available/needed, if there exists a lower priority task running and having an execution time longer than τ's deadline.

- By contrast, in *fully-preemptive* scheduling [62], a job can be preempted at any point of its execution, if a higher priority real-time task becomes ready. In this case, higher-priority tasks does not suffer blocking times, but lower priority tasks suffer a possibly high number of preemptions. This may lead to prohibitively high context switch overheads, cache related preemption and migration delays, and network contention costs [63], which may degrade the schedulability of the system and potentially cause deadline misses. Moreover, accurately accounting for preemption delays is very difficult (if not impossible) due to the potentially "infinite" preemption points, i.e., at any execution point of the task.
- A midway alternative is the *limited preemptive* scheduling [33] in which some restrictions are placed upon the occurrence of preemptions. The limited preemptive scheduling with *Fixed Preemption Points* approach allows preemptions only at predefined locations within the real-time task, which divides it into fixed *non-preemptive regions* (NPR). Limited preemptive scheduling has been proposed as an effective scheduling scheme that allows to reduce the number of preemptions of lower priority tasks, compared to the fully preemptive scheduling, while also reducing the blocking time to higher priority tasks, compared to the non-preemptive scheduling, thus improving schedulability. Moreover, with the limited preemptive scheduling model, a tighter analysis of the preemption-related overhead is possible and the preemption overhead may be significantly reduced by an optimized placement of preemption points [64].

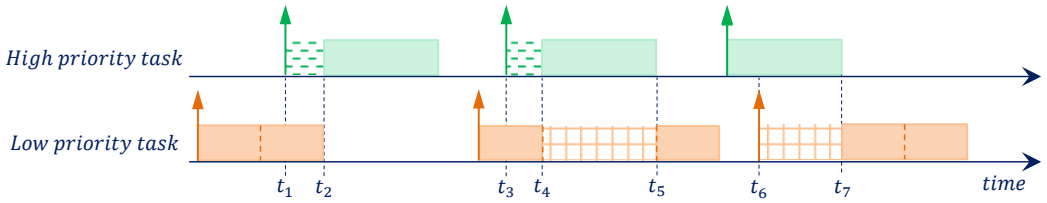
Figure 3.1 shows an example of the three scheduling strategies presented above. In order to facilitate the explanation, two sequential real-time tasks are scheduled in a single core. Moreover, the implicit deadline case is considered, so arrows represent the release time of a given job, and the deadline of the previous job. Figure 3.1a shows the non-preemptive scheduling scheme: when the high priority task is released, at time instants t_1 and t_3, the low priority task is running. Therefore, the high priority task waits until time instants t_2 and t_4, when the low priority task completes. However, at time instant t_5, the low priority task must wait for the high priority task to complete its execution at t_6, since it is released (and starts executing) before. In this case, both high and low priority tasks may suffer blocking times and interference, respectively. Figure 3.1b shows the fully-preemptive scheduling scheme: as soon as the high priority task is released, at time instants t_1 and t_3, the low priority task is preempted. It resumes as soon as the high priority task finishes, at t_2 and t_4. In the



(a) Non-preemptive scheduling.



(b) Fully-preemptive scheduling.



(c) Limited preemptive scheduling with fixed preemption points.

Figure 3.1: Preemption strategies in a single core.

last high priority task release, there is no preemption since it starts executing before, but still, the low priority task suffers interference from time instant t_5 till t_6. In this case, high priority tasks never suffer blocking times, but only low priority tasks suffer interference. Finally, Figure 3.1c shows the limited preemptive scheduling scheme: in this case, the low priority task has one fixed preemption point (dashed lines). When the high priority task is released at time instant t_1, the preemption point has already passed and so the high priority task must wait for the low priority task to complete at t_2. However, when the high priority task is released again at time instant t_3, it waits only until the preemption point of the low priority task is reached, at t_4. Then, the low priority task resumes as soon as the high priority one completes, at t_5. The same as in the non-preemptive and fully preemptive schemes, when the low priority task is released at t_6, it must wait for the high priority task to finish.

OpenMP and the limited preemptive scheduling.

Interestingly, the OpenMP tasking model implements the limited preemptive strategy. The OpenMP API specifies that OpenMP tasks can be suspended (preempted) only at task scheduling points (TSPs), dividing the task into multiple non preemptive *task part* regions. Accordingly, the OpenMP runtime can preempt OpenMP tasks at TSPs, and assign its corresponding threads to a different OpenMP task based on the priorities. Therefore, we can establish a one-to-one correspondence between NPRs in the limited preemptive scheduling and task parts in the OpenMP specification, which are represented as nodes (sub-tasks) in the OpenMP-DAG. It is worth noting that OpenMP provides the `taskyield` construct, which allows the programmer to explicitly define additional TSPs. However, regarding task scheduling points, the OpenMP API states that *"the implementation may cause it to perform a task switch"* and regarding the `taskyield` clause, *"the current task can be suspended in favor of execution of a different task"*. This means that an implementation is not forced to suspend a task in favor of another one in any case, not even if there is a higher priority task ready to execute (see evaluation in section 3.3). However, in real-time scheduling a TSP must be evaluated, meaning that if a higher priority task is ready at that point, the lower priority task must be preempted. Therefore, limited preemptive OpenMP schedulers must implement the evaluation of each TSP occurrence when targeting real-time systems.

Interestingly, this laxity in the OpenMP specification, which establishes that threads are allowed to, but not forced to suspend a task at TSPs, supports the implementation of non-preemptive scheduling. By simply disabling the suspension of tasks at those points, the OpenMP scheduler would be non-preemptive. In fact, for sequential real-time tasks, this is the default preemption strategy, since there are no implicit TSPs. In this case, it is worth noting that the `taskyield` construct also allows the implementation of the limited preemptive strategy in sequential real-time tasks.

Finally, OpenMP does not support the implementation of fully-preemptive scheduling strategies because that would require the runtime to preempt the execution of OpenMP tasks at any point of its execution, causing the implementation to be non-compliant. In any case, as we stated above, fully-preemptive scheduling can cause high preemption overheads, which degrade the predictability of the system.

3.2.3 Allocation and migration strategies

Based on the restrictions as to where real-time tasks are permitted to execute, there exist two scheduling schemes [34]:

- *Global scheduling* allows jobs from real-time tasks to execute upon any core. Jobs are *dynamically allocated* to cores, based on runtime information, such as the state of the platform (e.g., computing and communication resources available), the set of ready tasks, or the location of input data. Real-time tasks are allowed to migrate between cores, so that a preempted job can resume its execution in a core different to the one it started.
- *Partitioned scheduling* allows each job of a real-time task to execute only upon the core to which it has been mapped. Jobs are *statically allocated* to cores at design time, with the objective of increasing the predictability. Ideally, an analysis of the real-time tasks execution times and the available resources, provides a task-to-core mapping that minimizes the response time of the overall system.
- *Hybrid allocation* strategies, which allows a real-time task to be scheduled only on a subset of the available cores, have been proposed as well [65]. There exists an interesting approach, based on the hybrid allocation, called *federated scheduling* [66] in which some tasks are statically assigned to a group of cores and some others are globally scheduled.

Although the OpenMP API does not specify anything about allocation strategies, current OpenMP runtime implementations are based on dynamic allocation. However, the OpenMP *tied* tasking model (the default one) limits the implementation of global schedulers since tasks are tied to the thread that started its execution, i.e., migration of tied tasks is not allowed. This is not the case of *untied* tasks, that can be resumed by any thread in the team. A deeper analysis of the timing implications of the OpenMP tied and untied tasking models is presented in chapter 4.

OpenMP task to OpenMP thread Mapping

With the objective of increasing time predictability, most of the real-time schedulers consider a direct mapping between real-time tasks and cores. This includes two conditions: (1) threads are mapped to cores in a one-to-one manner, and (2) threads are not allowed to migrate between cores.

OpenMP threads are an abstraction of the computing resources upon which OpenMP tasks execute. In this thesis, we propose the use of a single team of threads to execute all the tasks of the system (see Section 3.1.1). This enables the real-time scheduler to have full control over the execution of OpenMP tasks over threads. However, OpenMP threads are further assigned to the operating system threads, hardware threads and cores, referred to as *places* in OpenMP. As a result, other levels of scheduling exist, out of the control of the OpenMP scheduler.

Fortunately, the OpenMP specification provides mechanisms to fulfill the two conditions stated above. On one hand, the `requires` directive, provisionally defined in the proposal of the OpenMP version 5.0 specification [67], allows to specify *"the features an implementation must provide in order for the code to compile and execute correctly"*. This may be useful to express the minimum number of cores that the target architecture must provide to guarantee a one-to-one mapping, as required by the system. On the other hand, OpenMP defines the *bind-var* internal control variable, together with the `proc_bind` clause, which allow to control the binding of OpenMP threads to cores. Both enable to define different thread-affinity policies. Finally, the *place-partition-var* internal control variable controls the list of places available.

Overall, an OpenMP framework intended to implement a critical real-time system must obey the following constraints:

1. *place-partition-var* := `cores`, so that each OpenMP place corresponds to a single core;

2. *bind-var* := `close`, so that OpenMP threads are consecutively assigned to places (forbidding threads migration between places). Once OpenMP threads are assigned to cores, this affinity must not be modified. Therefore, the `proc_bind` clause must be forbidden or ignored.

Moreover, we propose to use the `requires` directive along with the `ext_min_cores` clause and an integer value, to determine the minimum number of threads (and so, cores) necessary to correctly execute the system.

3.3 Evaluation of Current OpenMP Runtime Implementations

This thesis focuses on the analysis of the timing and scheduling behavior of OpenMP, and not on the efficient implementation of the runtime. However, to better understand the support that current OpenMP implementations have to develop CRTES, this section evaluates how priorities and preemptions are treated in three widely used OpenMP runtime implementations.

3.3.1 Experimental setup

Runtimes. We test three runtime implementations of the OpenMP version 4.5, provided by GCC 8.1 [68], Intel C++ 18.0.3 [69] and Nanos++ [70].

Performance monitoring tools. We use two instrumentation libraries to obtain the traces of the OpenMP executions: (1) Extrae [71], that captures the information of the performance of parallel applications, and generates traces in files, and (2) Paraver [72], a performance visualization and analysis tool that uses Extrae traces.

Application. Listing 3.6 shows the synthetic application implemented to properly exercise the features we want to test. Three simple real-time tasks, τ_1, τ_2 and τ_3, are created with low, medium and high priority, respectively. τ_1 includes an explicit TSP by means of the `taskyield` construct. Therefore, according to the limited preemptive scheduling strategy, τ_1 is divided into two non-preemptive task parts. Sequential real-time tasks and two threads have been considered for simplicity. Current OpenMP implementations only support dynamic allocation and global scheduling.

3.3.2 OpenMP execution traces: limited preemptive scheduling and the `priority` clause.

Critical real-time systems must honor the priority of each task because it determines preeminence of some tasks over others. Moreover, in the limited preemptive strategy, tasks must be preempted at preemption points in favor of ready tasks of higher priority. Therefore, knowing the execution time of each task part, the expected behavior during three iterations of the OpenMP real-time system presented in Listing

```
1  #pragma omp parallel
2  #pragma omp single nowait
3  {
4    while (1) {
5      #pragma omp task untied priority(1)  // τ1
6      {
7        part11();
8        #pragma omp taskyield
9        part12();
10     }
11     part01();
12     #pragma omp task untied priority(2)  // τ2
13     { part21(); }
14     #pragma omp task untied priority(3)  // τ3
15     { part31(); }
16     part02();
17   }
18 }
```

Control loop
$part_{01}()$
$part_{02}()$

τ_1 Low priority task
Task part $part_{11}()$
Task part $part_{12}()$

τ_2 Medium priority task
Task part $part_{21}()$

τ_3 High priority task
Task part $part_{31}()$

Listing 3.6: OpenMP real-time system example for the evaluation of current runtimes.

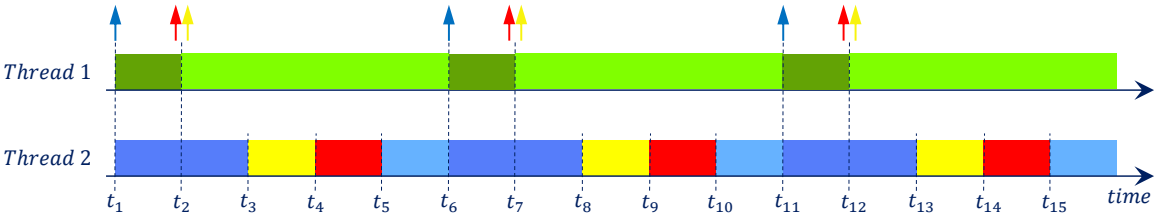


Figure 3.2: Expected behavior of the OpenMP real-time system in Listing 3.6.

3.6 is shown in Figure 3.2. Green blocks represent the execution of the code within the `single` construct ($part_{01}$ and $part_{02}$) in thread 1, blue blocks represent the execution of τ_1 (task parts $part_{11}$ and $part_{12}$), red blocks represent the execution of τ_2 ($part_{21}$) and yellow blocks represent the execution of τ_3 ($part_{31}$). τ_1, τ_2 and τ_3 execute in thread 2. Blue, red and yellow arrows denote the time instants at which tasks τ_1, τ_2 and τ_3 are ready, respectively.

τ_1 gets first the idle thread 2, at time instants t_1, t_6 and t_{11}, because it is created before the higher priority tasks τ_3 and τ_2. At time instants t_2, t_7 and t_{12}, the highest and medium priority tasks, τ_3 and τ_2, are created. As a result, when τ_1 reaches its task scheduling point, defined by the `taskyield`, at time instants t_3, t_8 and t_{13}, it is

preempted and the highest priority task τ_3 starts its execution. When τ_3 finishes, at time instants t_4, t_9 and t_{14}, τ_2 and the second task part of τ_1 are ready to execute. Since τ_2 has higher priority, it starts its execution. Finally, when τ_2 finishes, at time instants t_5, t_{10} and t_{15}, τ_1 can resume its execution.

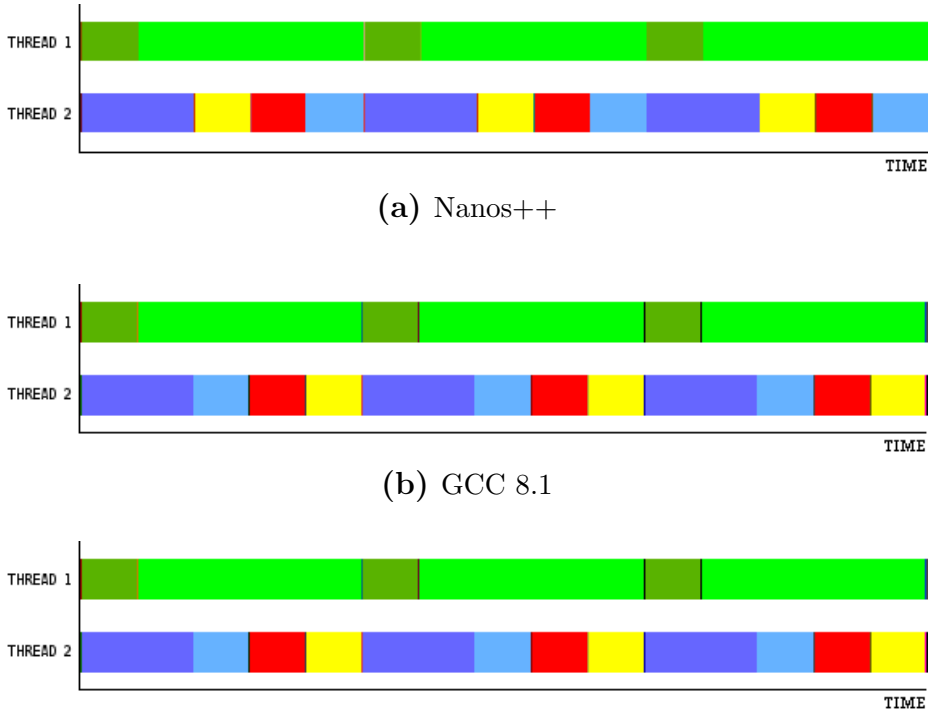
The execution traces of three iterations of the source code presented in Listing 3.6 are shown in Figure 3.3a for Nanos++, Figure 3.3b for GCC 8.1, and Figure 3.3c for Intel C++ 18.0.3. The observed behavior in Nanos++ is exactly as expected. Nevertheless, it is worth mentioning that the behavior is different if tasks are tied, i.e., if there is no `untied` clause associated to the `#pragma omp task`. In this case, τ_2 does not execute immediately after τ_3 finishes. Instead, τ_1 resumes its execution. The reason is that, at this point, τ_1 is tied to a thread and this prevails over the priority.

In GCC and Intel, the behavior is exactly the same, and contrary to what is expected based on the priorities of the tasks. Neither the preemption point of τ_1 nor the priorities of τ_3 and τ_2 are honored. Instead, τ_1 is executed uninterruptedly from beginning to end, i.e., both task parts are executed consecutively. Also, τ_2 is executed before τ_3, even though τ_3 has higher priority than τ_2. This execution order is established by the task creation order in a FIFO manner.

Overall, the runtime behavior in the three cases is correct with respect to the OpenMP v4.5 specification. The reasons are: (1) the `taskyield` construct is defined such that the executing task *"can"*, but it is not forced to, be suspended in favor of any other task; and (2) the `priority` directive is only a *"hint"* for the priority of tasks. Although current OpenMP runtimes are not ready to support the development and execution of critical real-time systems, Nanos++ already implements some of the fundamental features needed by critical real-time systems. This is not the case of GCC 8.1 nor Intel C++ 18.0.3.

3.4 Related Work

The performance requirements of advanced embedded critical real-time systems entails a booming trend to use multi-core, many-core and heterogeneous architectures. A recent work [73] describes the challenges of parallel real-time systems, and provides an overview of the research conducted by authors. They investigate a scheduling system for parallel applications (OpenMP-based, for instance) in real-time systems and a fault-tolerant approach which provides resilience against hardware faults on



(a) Nanos++

(b) GCC 8.1

(c) Intel C++ 18.0.3

Figure 3.3: Execution traces of the OpenMP real-time system in Listing 3.6

application level.

OpenMP has been already considered to cope with the performance needs of embedded real-time systems [74, 21]. In this context, OpenMP has been analyzed regarding two features that are mandatory in such restricted systems: timing analysis and functional safety.

From a timing perspective, there is a significant amount of work considering the time predictability properties of OpenMP. Despite the fork-join model was firstly considered [27], the tasking model seems to be more suitable given its capabilities to define fine-grain, both structured and unstructured parallelism. For this reason several works [26, 75, 76], included the next chapter of this thesis, studied the OpenMP tasking model and its similarities with the sporadic DAG scheduling model. From a functional safety perspective, as seen in Section 2.3.2, OpenMP is also considered as a convenient candidate to implement real-time systems.

Finally, as embedded systems usually have tight constraints regarding resources such as memory (e.g., the Kalray MPPA has 2MB shared memory [25]), different approaches for developing lightweight OpenMP runtime systems coexist [43, 77]. These

studies are meant to efficiently support OpenMP in such constrained environments. For instance, the memory used at runtime is reduced when the task dependency graph of the applications is statically derived.

3.5 Summary

OpenMP is a solid candidate to address the performance challenges of critical real-time embedded systems. However, OpenMP was originally intended for a different purpose than such systems, for which guaranteeing the correct output is as important as guaranteeing it before the deadline. In this chapter, we evaluate the use of the OpenMP tasking model to develop and execute critical real-time embedded systems. The OpenMP tasking model has been shown to have similarities with the sporadic DAG-based scheduling model, upon which many critical real-time systems are based on, e.g., AUTOSAR [78], used in automotive systems. We focus on the design implications and the scheduling decisions to efficiently exploit fine-grain parallelism within real-time tasks and concurrency among them, while guaranteeing the timing behavior according to current real-time practices.

Concretely, we propose the use of a single team of threads to implement and execute both concurrent real-time tasks and the parallelism within them. Two new clauses, `event` and `deadline`, are proposed to allow the implementation of recurrent real-time tasks and deadline-based schedulers, respectively. Moreover, we analyze some important features already provided in the OpenMP API: the `priority` clause and the TSPs. We conclude that the behavior of these two features, as defined in the OpenMP API, is not conforming to the expected behavior of real-time schedulers. In both cases, the clause must be a prescriptive modifier, instead of a hint (the case of the `priority` clause) or a possibility of occurrence (the case of TSPs). It must be guaranteed that, at each preemption point (i.e., at each TSP), if there is a high priority task ready, the running task is suspended in favor of the high priority task. This behavior is required to implement limited preemptive scheduling, the most suitable preemptive strategy for OpenMP real-time systems. Overall, correctly addressing all these features in the specification is of paramount importance to enable the use of OpenMP in critical real-time embedded systems.

Chapter 4

Timing Characterization of the OpenMP Tasking Model

"El ayer querella, añora y el mañana condiciona, en cambio, vivo el instante antes de después, después de antes, el eterno presente"[1]

— David Martínez Álvarez (Rayden)

Originally focused on a *thread-centric* model to exploit massively data-parallel and loop-intensive types of applications, OpenMP has evolved to a *task-centric* model which enables very sophisticated types of fine-grain and unstructured parallelism. The OpenMP tasking model, introduced in version 3.0 [40], allows the programmer to define explicit tasks as independent units of parallel work. Moreover, from version 4.0 [41], OpenMP defines an accelerator model that, coupled with the tasking model, enables to efficiently offload computation to specialized accelerator devices. These two models allow to effectively utilize parallel architectures, while hiding their complexity to the programmer.

Several practical issues have been addressed by the OpenMP language committee when designing the tasking model specification, considering simplicity of use, compatibility with the existing specification and performance, as the main metrics of interest [79]. However, the requirements for the co-existence of a legacy thread-based execution model and a new task-based execution model led to conflicting needs for choosing the default settings. Unfortunately, none of the considered design choices took time predictability into account, as this is traditionally not a relevant metric in

[1] English translation: "*Yesterday hurts, longs for and Tomorrow conditions, instead, I live the moment before the afterwards, after the previous one, the eternal present.*"

the HPC domain. The aim of this chapter is to revisit such design choices, as found in the OpenMP specification version 4.0, introducing *timing predictability* as a new key metric of interest.

4.1 The OpenMP Tasking Model

As seen in Section 2.3, the OpenMP tasking model has been already considered as a firm candidate to be adopted in critical real-time systems. On the one hand, the execution model of OpenMP tasks resembles the real-time sporadic DAG tasks model [26, 43]. On the other hand, functional safety has been also considered in OpenMP [57] as a key element in critical real-time systems. Moreover, the previous chapter analyzes the scheduling features that must be addressed in the specification of OpenMP to safely target critical real-time embedded systems.

This section analyzes the OpenMP tasking model, identifying the issues that affect the timing analysis.

4.1.1 From the thread-centric to the task-centric model

Up to specification version 2.5, OpenMP assumed a thread-centric execution model. The programmer could determine the thread in which a code segment was executing, with the OpenMP routine `omp_get_thread_num()`. Following the *single program, multiple data (SPMD)* programming paradigm, the programmer was also allowed to explicitly perform different works on different threads, based on their *id*. Moreover, the programmer could assign private storage to the thread (marking the target variables with the `threadprivate` directive) that persisted across executions of different parallel regions.

The tasking model (with the associated `task` construct) was first introduced in OpenMP specification 3.0. The OpenMP tasking model provides programmer with a very convenient abstraction of parallelism, being the runtime in charge of scheduling tasks to threads. Version 4.0 of the OpenMP specification introduced advanced features to express dependencies between tasks.

However, for backward compatibility reasons, both models need to coexist in the OpenMP specification. This leads to conflicting needs for choosing the default settings. Probably the most notable example of a "trade-off" design choice between the old (thread-centric) and the new (task-centric) specification is the distinction be-

tween *tied* and *untied* tasks. In state-of-the-art *tasking* programming models, e.g., Cilk [80], there are points in the execution of a program where a thread can suspend the execution of the current task and switch to another task. The suspended task can resume execution on a different thread, if available. This execution model implements a *work-conserving* policy, which ensures that no thread remains idle if there is work to be done. Ultimately, this behavior guarantees an efficient exploitation of a parallel architecture and facilitates the timing characterization of parallel execution (see Section 4.2 for further details). Unfortunately, the thread-centric nature of OpenMP exposes a number of issues if migration of a task from one thread to another is allowed. To give a few examples:

- work-sharing among threads based on the thread id, at the core of classic OpenMP programming practices (up to version 2.5), would break the semantics of the program;
- mutually-exclusive code regions (e.g., `critical` construct) could result in deadlock scenarios, as critical section locks are owned by threads;
- private data to a thread (e.g., `threadprivate` variables) should also migrate with the task, which is neither easy nor efficient to implement.

As a solution to the problem, the OpenMP specification states that, by default, an OpenMP task must be *tied* to the thread which started its execution. Tied tasks cannot migrate to a different thread when the task is suspended, even if there are idle threads available. Moreover, the OpenMP specification defines an extra restriction, known as the *task scheduling constraint 2* (TSC 2), that does not allow tied tasks to be scheduled in threads in which other non-descendant tied tasks are suspended. Overall, the *tied* tasking model results in a non work-conserving task scheduling approach. A knowledgeable programmer can specify a work-conserving approach by using *untied* tasks, which are allowed to resume execution on a different thread when suspended. As it always happens in OpenMP, the programmer takes responsibility for guaranteeing correct execution of the program. As a result, the use of the default *tied* tasking model may have serious implications on the predictability of the timing behavior of OpenMP applications, a fundamental property to apply OpenMP in real-time systems.

The remaining of this section analyzes the behavior of the `tied` and `untied` task models, from a scheduling and timing analyzability point of view. Next sections

describe the impact that such models have on the capability of our analysis to provide precise and tight timing guarantees.

4.1.2 OpenMP task-to-thread scheduling

In OpenMP, the execution of tasks explicitly generated by the `task` construct is assigned to one of the threads in the team, subject to the thread's availability to execute work. Thus, execution of a new task could be immediate or deferred according to *task scheduling constraints (TSCs)* and thread availability. Moreover, threads are allowed to suspend an executing task at *task scheduling points (TSPs)* in order to execute a different task.

Section 2.2.3 describes the TSPs, which divide task regions into task parts and lead the OpenMP-DAG creation. However, TSPs and TSCs have certain implications when timing analysis is considered since the run-time scheduling of tasks may be different to what the schedulability analysis considers.

4.1.2.1 Task Scheduling Constraints

When a thread encounters a *TSP*, it can begin or resume the execution of a task, provided that a set of *task scheduling constraints* (*TSC*), as defined in Section 2.11.3 of the OpenMP specification [41], are fulfilled:

TSC 1: An *included* task must be executed immediately after the task is created.

TSC 2: Scheduling of new *tied* tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a `barrier` region. If this set is empty, any new *tied* task may be scheduled. Otherwise, a new *tied* task may be scheduled only if it is a descendant task of every task in the set.

TSC 3: A *dependent* task shall not be scheduled until its task data dependencies are fulfilled.

TSC 4: When a task contains an `if` clause and its associated condition evaluates to false, the task is executed immediately if the rest of the *TSC*s are met.

Am OpenMP program relying on any other *TSC* or performing a different action when a *TSP* is encountered is non-conforming[2].

[2] An OpenMP conforming program is that which follows all rules and restrictions of the OpenMP specification.

```
1  for (i=0; i < N; i++) {
2    #pragma omp task          // T1
3    {
4      foo();
5      #pragma omp critical
6      {
7        bar();
8        #pragma omp task      // T2
9        foobar();
10     }
11   }
12 }
```

Listing 4.1: Example of an OpenMP program using synchronization constructs.

TSC 2 may considerably reduce the number of threads available to tied tasks, impacting on both performance and timing predictability. Next section explains the reason of such a design choice.

4.1.2.2 Understanding TSC 2

TSC 2 prevents tied task from being scheduled in threads in which other non-descendant tied tasks are suspended. This inhibits the runtime from incurring in a deadlock situation when the `critical` synchronization construct is used within a task [79]. The `critical` construct is a synchronization mechanism inherited from the *thread-centric* model that defines a region that can be exclusively executed by a single thread at a time. The reason of the deadlock situation is because the owner of the lock is a thread and not a task, but the critical region may be within task. Hence, if the task is re-scheduled to a different thread, it will not have access to the corresponding lock.

Listing 4.1 shows an example in which the `critical` construct is used within a task. The example creates as many T_1 and T_2 task instances as for-loop iterations. When the thread executing the first instance of T_1 enters the critical section, the thread obtains the lock so that no other thread can access it. However, the execution of this task instance of T_1 can be suspended when reaching the *TSP* at line 8 (`task` construct T_2) and so its thread could be assigned to a different task. If the same thread starts executing another instance of T_1, it would eventually reach the critical section again, but this time, it would not be able to enter it as this thread already has the lock. This leads to a deadlock situation in which the thread has the lock due to the first instance of T_1 and, at the same time, is blocked in the critical section due

to the second instance of T_1. Notice that the `critical` construct does not imply a *TSP*, so that the thread is stalled in the second T_1 task instance. In order to avoid this situation, the OpenMP specification defines the *TSC 2*, which prevents the same thread from executing any tied task that is not descendant of T_1. Note that T_2 is a descendant task of T_1 and so the thread executing T_1 is allowed to execute T_2.

When untied tasks are used, the responsibility of the utilization of critical sections or thread-specific information lies on the programmer.

4.1.2.3 Task Scheduling Algorithms

When a task encounters a *TSP*, the program execution branches into the OpenMP runtime system, where task-to-thread schedulers can: (1) begin the execution of a task region bound to the current team or (2) resume any previously suspended task region bound to the current team. The order in which these two actions are applied is not specified by the standard. An ideal task scheduler will schedule tasks for execution in a way that maximizes concurrency while accounting for load imbalance and locality to facilitate better performance. Current runtime implementations of OpenMP are based on two main task scheduling policies [81]:

BFS (Breadth-First Scheduling). When a task is created, it is placed into a pool of tasks and the encountering thread continues the execution of the parent task. Tasks placed in that pool can then be executed by any available thread from the team. Due to *TSC 2*, when a tied task is suspended in a *TSP*, it is placed into the private pool of tasks associated to its thread. Untied tasks instead are queued into a pool of tasks accessible by all threads in the team. Access to these pools can be LIFO (i.e., last queued tasks will be executed first) or FIFO (i.e., oldest queued tasks will be executed first). Threads will always try to schedule first a task from their local pool. If it is empty then they will try to get tasks from the team pool. An example of BFS is shown in [82].

WFS (Work-First Scheduling). New tasks are executed immediately after they are created by the parent's thread, suspending the execution of the parent task. When a task is suspended in a *TSP*, it is placed in a thread local pool that can be accessed in a LIFO or FIFO manner. When looking for tasks to execute, threads will look into their local pool. If it is empty, they will try to steal work from other threads. When stealing from another thread pool, to comply with OpenMP restrictions, tied tasks cannot be stolen from its associated thread.

The Cilk scheduler [80] belongs to this family. In particular, it is a WFS where access to the local pool is LIFO, tries to steal the parent task first and otherwise steals from another thread pool in a FIFO manner.

WFS tends to obtain better performance results than BFS due to two reasons [81]: (1) the WFS strategy tries to follow the serial execution path hoping that if the sequential algorithm was well designed, it will lead to better data locality; and (2) it also has the property of minimizing space. The reason is that in a BFS strategy all tasks coexist simultaneously since all child tasks are created before executing them. On the contrary, WFS creates the same number of tasks, but fewer tasks have to exist at the same time because they are executed immediately after they are created. However, OpenMP implementations typically use BFS due to the tied tasks default restriction: if WFS is implemented, when a tied task T_i creates a child tied task T_{i+1}, this one starts its execution in T_i's thread. Then, T_i is suspended and it cannot resume its execution until T_{i+1} finishes or suspends in a *TSP* because it is tied to a thread. Therefore, WFS turns a parallel program with tied tasks into a sequential execution, as will be shown in Section 4.4.1.

Overall, *TSC 2* and the semantics of *tied* tasks prevent the implementation of work-conserving schedulers. We discuss in the next section the implications that the tied and untied models have on the timing analysis of task-based OpenMP applications.

4.2 The Schedulability Problem of an OpenMP application

Once the OpenMP-DAG of an OpenMP application is derived, as shown in Section 2.3.1, the problem of schedulability reduces to the problem of determining whether the DAG can be scheduled on the available threads to complete within a specified relative deadline D.

The OpenMP specification is agnostic to the task-to-thread scheduling implemented by the runtime. It is therefore the responsibility of the runtime developer to implement the most suitable scheduler for the OpenMP system, guaranteeing that the *TSCs* are fulfilled.

In high-performance systems, the main goal of task-to-thread schedulers is to maximize the occupancy of threads. In real-time systems, the main goal is not only

maximizing the use of resources but also to provide timing guarantees. Considering global scheduling, the use of *work-conserving* schedulers facilitates the timing characterization of parallel execution.

Definition 8. ***Work-conserving scheduling.*** *A scheduling algorithm is work-conserving if the following situation does never occur in the system: (1) there exists a ready task awaiting execution and (2) there exists at least one idle thread.*

For work-conserving schedulers, the problem of determining the schedulability of an OpenMP-DAG has a strong correspondence with the makespan[3] minimization problem of a set of precedence constrained nodes (or OpenMP task *parts*) on identical processors (or OpenMP threads in a team), which is known to be strongly NP-hard by a result of Lenstra and Rinnooy Kan [83]. However, the Graham's *List Scheduling* algorithm [84], which can be implemented in polynomial time complexity, provides an approximation of $2 - \frac{1}{m}$ for this problem, being m the total number of threads in a team. This means that this algorithm is able to produce for any input task graph a value of the makespan that is at most $2 - \frac{1}{m}$ times the optimal one. The *List Scheduling* algorithm simply maps tasks to available threads in a team without introducing idle times if not needed, i.e., it implements a *work-conserving* scheduling algorithm.

For real-time systems, the use of work-conserving schedulers in the the OpenMP runtime implementations seems to be the best option. Current OpenMP runtime implementations already incorporate work-conserving schedulers, i.e., BFS and WFS.

Unfortunately, the *TSC 2* and the execution semantics of tied tasks force these schedulers not to be work-conserving. On the one hand, *TSC 2* forbids a new `tied` task to be scheduled to a thread where it is not a descendant of all the other suspended *tied* tasks already assigned to this thread. This may potentially reduce the number of threads in the team that can be assigned to new tied tasks. On the other hand, *tied* task parts cannot migrate when the task is resumed and its corresponding thread is being used by another descendant *tied* task or an *untied* task. These constraints impose extra conditions on the schedulability analysis of OpenMP programs.

This is not the case of the execution semantics of untied tasks, which are not subject to *TSC 2*, and so *parts* of the same task are allowed to execute on different threads i.e., when a task is suspended, the next *part* to be executed can be resumed

[3]The *makespan* of a set of precedence constrained nodes is defined as the total length of the schedule (i.e., response-time) of the collection of nodes.

on a different thread. Hence, the execution model of untied tasks allows BFS and WFS to be work-conserving.

Overall, the additional requirements imposed by the use of tied tasks suggest devising distinct timing characterizations for the two types of OpenMP tasks, i.e., tied and untied. Hence, in the rest of this chapter we analyze both types of tasks to characterize their timing behavior, outlining the major challenges posed by the use of tied tasks in a real-time domain.

4.3 Schedulability Analysis of Untied Tasks

The `untied` clause allows a task to be executed in any thread and, in case it is suspended, to be resumed by any thread in the team. In other words, the task can freely migrate across threads during its execution. This flexibility in the task allocation is exploited at the analytical level in order to derive a direct solution to the schedulability problem.

Given an OpenMP-DAG, as derived in Section 2.3.1, we build upon the result in [84] to derive a response-time upper bound of an OpenMP-DAG composed of `untied` tasks, by considering that each task *part* represents a sequence of operations that can be executed in one of the available threads as soon as all its three types of dependencies have been fulfilled (control flow, *TSP* creation/resume and synchronizations). Whenever more parts than available threads are ready to be executed, any allocation order is possible, provided that the scheduling strategy remains work-conserving. This is the case of BFS and WFS strategies.

We derive an upper-bound on the response-time, denoted by R^{ub}, of an OpenMP program composed of `untied` tasks and represented as an OpenMP-DAG G. Such a bound can be computed starting from the proof of the $2-\frac{1}{m}$ approximation bound in [84], in conjunction with some additional considerations. Here, we first establish two lower-bounds on the minimum makespan R^{opt} of an OpenMP program, which will be useful to derive an upper-bound on its response-time.

Proposition 1.

$$R^{opt} \geq \frac{1}{m} \sum_{v_i \in V} C_i = \frac{1}{m} vol(G). \tag{4.1}$$

Proposition 2.

$$R^{opt} \geq \max_{\lambda \in G} \sum_{v_i \in \lambda} C_i = len(G). \tag{4.2}$$

Equation (4.1) trivially follows from the fact that the total amount of work should be executed on m threads, while Equation (4.2) is obtained by noticing that parts belonging to a chain must be executed sequentially. This is true for any chain of the OpenMP-DAG, and in particular for its longest one, i.e., its critical path.

We now review the proof in [84] to derive the approximation bound of *List Scheduling* on the minimum makespan of a generic set of precedence-constrained nodes (task parts), which applies to OpenMP-DAGs with `untied` tasks as well.

Theorem 1. *Graham's List Scheduling algorithm gives a* $2-\frac{1}{m}$ *approximation for the makespan minimization problem of a set of precedence-constrained nodes expressed by means of a task graph* G*, scheduled on* m *identical processors (or threads).*

Proof. Let v_z be the node in G that completes last, and t_z its starting time. Let v_{z-1} be the predecessor of v_z that completes last. By the precedence relation between the two nodes, we have that $t_z \geq t_{z-1}+C_{z-1}$. Proceeding in this way until a node without predecessors is reached, we construct a particular chain of nodes $\lambda = (v_1, \ldots, v_z)$. The fundamental observation that must be made is that, between the completion time $t_i + C_i$ of each node of λ^*, and the starting time of the next node, all threads must be busy, otherwise node v_{i+1} would have started earlier. The same applies to the time interval between 0 and t_1. Note also that some node belonging to λ is executing at every time instant when not all the threads are busy.

The response-time of the OpenMP-DAG, denoted by R, is given by the sum of the time instants when some of the threads are idle and the time instants when all the threads are busy. The former contribution cannot exceed $len(\lambda)$, while the latter cannot exceed $\frac{1}{m}(vol(G) - len(\lambda))$, since the total amount of workload executed in such time slots is no more than $vol(G) - len(\lambda)$. Hence,

$$R \leq len(\lambda) + \frac{1}{m}\left(vol(G) - len(\lambda)\right). \tag{4.3}$$

Now, by combining Equations (4.1), (4.2) and (4.3) and reordering the terms, we obtain:

$$\begin{aligned} R &\leq len(\lambda) + \frac{1}{m}\left(vol(G) - len(\lambda)\right) = len(\lambda) + \frac{1}{m}vol(G) - \frac{1}{m}len(\lambda) \leq \\ &\leq R^{opt} + R^{opt} - \frac{1}{m}R^{opt} = \left(1 - \frac{1}{m} + 1\right) R^{opt} = \left(2 - \frac{1}{m}\right) R^{opt} \end{aligned}$$

□

Equation (4.3) cannot be directly used as an upper-bound to the response-time of the OpenMP-DAG, because the chain λ is not known a priori. However, a simple upper-bound can be found for Equation (4.3) by upper-bounding the length of the chain λ with the length of the critical path λ^*, as it is longer than any possible chain in the OpenMP-DAG. The following lemma formalizes this result[4].

Lemma 1. *An upper-bound on the response-time of an OpenMP-DAG composed of* `untied` *tasks is given by* R^{ub}*:*

$$R^{ub} = len(G) + \frac{1}{m}\left(vol(G) - len(G)\right) \tag{4.4}$$

Proof. The upper-bound R^{ub} simply follows from Equation (4.3) by definition of critical path and by considering that $1 \geq \frac{1}{m}$. More explicitly:

$$\begin{aligned} R &\leq len(\lambda) + \frac{1}{m}\left(vol(G) - len(\lambda)\right) = \left(1 - \frac{1}{m}\right) len(\lambda) + \frac{1}{m} vol(G) \leq \\ &\leq len(\lambda^*) + \frac{1}{m}\left(vol(G) - len(\lambda^*)\right) = len(G) + \frac{1}{m}\left(vol(G) - len(G)\right) \end{aligned}$$

□

Factor $\frac{1}{m}\left(vol(G) - len(G)\right)$ is known as the *self-interference* (or intra-task interference) i.e., the interference contribution of the task itself to the critical path. The result of Lemma 1 suggests that, whenever an OpenMP program is composed of `untied` tasks, a timing analysis can be easily performed by checking Equation (4.4) against the relative deadline D of the OpenMP-DAG.

4.4 Impact of Tied Tasks on Scheduling

The execution semantics of the tied task model, presents some conceptual difficulties that significantly affect the complexity of the schedulability problem.

Tied tasks are constrained by *TSC 2*, which reduces the number of available threads for the execution of new tied tasks. Also due to the fact that tied tasks must always resume on the same thread where they started executing. Overall, these two constraints impact both performance and timing analyzability.

[4]The response time upper bound has been proposed in collaboration with Alessandra Melani.

4.4.1 Reduction of available threads

This section analyzes the implications of using tied tasks from a schedulability point of view. In particular, we compute the number of threads available to a new task due to *TSC 2*, and the number of tasks that can prevent another task from resuming its execution in its thread. In this way, we demonstrate that the *tied* task execution model results in a non-conserving policy, and explain why analyzing tied tasks under current scheduling algorithms without introducing unacceptable pessimism is prohibitive, or at least conceptually very difficult to achieve.

The rest of this section analyzes these two scenarios assuming a generic scheduling approach, denoted by GenS, and the breadth-first and work-first schedulers. GenS represents that a concrete scheduling policy has not been specified. For the BFS and WFS strategies, a FIFO policy (see Section 4.1.2.3) has been considered. Notice that the possible scheduling solutions derived by BFS and WFS strategies are included in GenS. The OpenMP program in Listing 4.2 (and its corresponding OpenMP-DAG in Figure 4.1) is used to illustrate the explanation.

4.4.1.1 Number of threads available to a new OpenMP *tied* task

The number of available threads to a new tied task may be reduced because other *tied* tasks suspended in a *TSP* prevent the new tied task from being scheduled in the same thread. According to *TSC 2*, the new tied task can be scheduled to a thread in which other tied tasks are suspended only if it is a descendant of all the tasks tied to this thread. In the extreme case, a new tied task could even not start its execution despite existing available threads in the team. Hereunder, we consider basic notions of set theory to derive the number of tasks affecting the effective number of threads available to new tied tasks, for each considered scheduling solution.

GenS. Given an OpenMP task T_i and a generic scheduling strategy GenS, we define $BlockCT_i(GenS)$ as the set of potential tasks that may prevent T_i from executing on the same threads in which they are suspended:

$$BlockCT_i(GenS) = (T \setminus \{T_i\} \setminus DesT_i \setminus PreT_i \setminus DDepT_i) \cap TSPT_{gens}, \qquad (4.5)$$

where T is the set of all OpenMP tasks, $DesT_i$ is the set of descendant tasks of T_i, $PreT_i$ is the set of predecessor tasks of T_i, $DDepT_i$ is the set of tasks that depends

```
1 #pragma omp parallel
2 #pragma omp single nowait { // T0
3   part00
4   #pragma omp task // T1
5                depend(out:x)
6   { part10
7     #pragma omp task // T2
8     { part20 }
9     part11
10    #pragma omp taskwait
11    part12
12  }
13  part01
14  #pragma omp task // T3
15               depend(in:x)
16  { part30
17    #pragma omp task // T4
18    { part40 }
19    #pragma omp taskwait
20    part31
21  }
22  part02
23  #pragma omp task // T5
24  { part50
25    #pragma omp task { // T6
26      part60
27      #pragma omp task  // T7
28      { part70
29        #pragma omp task // T8
30        { part80 }
31        #pragma omp taskwait
32        part71
33      }
34      part61
35      #pragma omp task // T9
36      { part90 }
37      part62
38    }
39    part51
40    #pragma omp taskwait
41    part52
42  }
43  #pragma omp taskwait
44  part03
45 }
```

Listing 4.2: Example of an OpenMP program using tasking model.

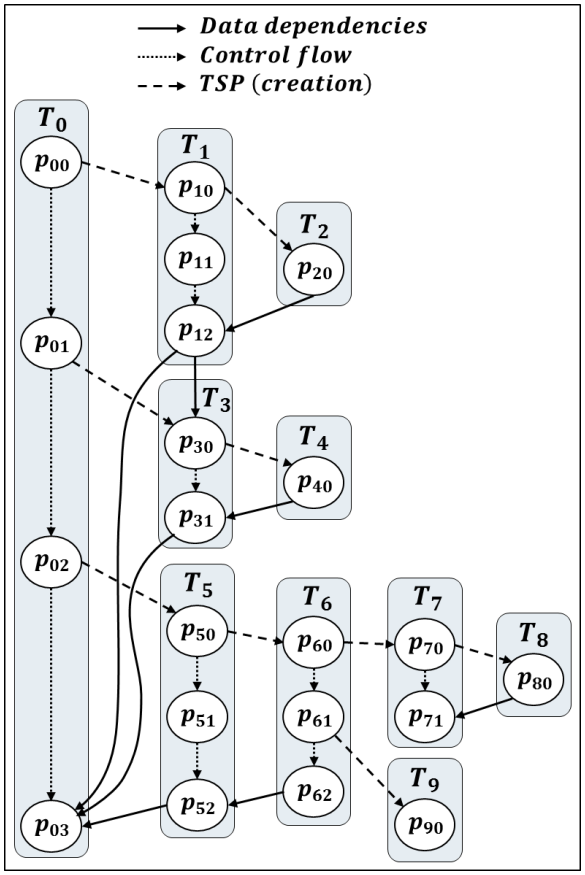


Figure 4.1: OpenMP-DAG corresponding to the OpenMP program in Listing 4.2.

on T_i and $TSPT_{gens}$ is the set of tasks with at least one *TSP* (e.g., contain a `task` or a `taskwait` construct) and so, can suspend their execution. $DDepT_i$ considers (1) the set of tasks with data dependencies, through `depend` clauses, to or form T_i; and (2) the descendant tasks of this set, if a synchronization dependency exists (e.g., a

T	$TSPT_{gens}$
$\{T_0, T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9\}$	$\{T_0, T_1, T_3, T_5, T_6, T_7\}$

T_i	$DesT_i$	$PreT_i$	$DDepT_i$	$BlockCT_i(GenS)$
T_0	$\{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9\}$	$\emptyset$	$\emptyset$	$\emptyset$
T_1	$\{T_2\}$	$\{T_0\}$	$\{T_3, T_4\}$	$\{T_5, T_6, T_7\}$
T_2	$\emptyset$	$\{T_0, T_1\}$	$\{T_3, T_4\}$	$\{T_5, T_6, T_7\}$
T_3	$\{T_4\}$	$\{T_0\}$	$\{T_1, T_2\}$	$\{T_5, T_6, T_7\}$
T_4	$\emptyset$	$\{T_0, T_3\}$	$\{T_1, T_2\}$	$\{T_5, T_6, T_7\}$
T_5	$\{T_6, T_7, T_8, T_9\}$	$\{T_0\}$	$\emptyset$	$\{T_1\}$ *or* $\{T_3\}$
T_6	$\{T_7, T_8, T_9\}$	$\{T_0, T_5\}$	$\emptyset$	$\{T_1\}$ *or* $\{T_3\}$
T_7	$\{T_8\}$	$\{T_0, T_5, T_6\}$	$\emptyset$	$\{T_1\}$ *or* $\{T_3\}$
T_8	$\emptyset$	$\{T_0, T_5, T_6, T_7\}$	$\emptyset$	$\{T_1\}$ *or* $\{T_3\}$
T_9	$\emptyset$	$\{T_0, T_5, T_6\}$	$\emptyset$	$\{T_1, T_7\}$ *or* $\{T_3, T_7\}$

Table 4.1: Sets to compute the number of threads available to new tied tasks (generic scheduling approach), for the example in Figure 4.1.

`taskwait`).

Only parallel tasks can simultaneously be suspended in different threads. Therefore, only parallel tasks can simultaneously prevent a new tied task T_i from executing on the threads in which they are suspended. This means that, if $BlockCT_i = \{T_j, T_k\}$, but T_j cannot execute in parallel with T_k, then $BlockCT_i = \{T_j\}$ or $BlockCT_i = \{T_k\}$.

In other words, $BlockCT_i$ contains the set of tasks that are not descendant or predecessor of T_i, that do not depend on T_i, and that can be suspended in a *TSP*. The descendant tasks of T_i have not been created yet at the point T_i is created, hence we can neglect them. Similarly, the dependent tasks of T_i and their descendant tasks are not considered because they have to wait until T_i completes to start executing. Also, the predecessor tasks of T_i can be neglected because, according to *TSC 2*, T_i can be scheduled in the threads of all its predecessor tasks.

Table 4.1 shows, for the example in Figure 4.1, the sets T and $TSPT_{gens}$, the sets $DesT_i$, $PreT_i$ and $DDepT_i$, for each task T_i, and the computed values of $BlockCT_i(GenS)$. Consider task T_1 as an example: $DesT_1$ is equal to $\{T_2\}$ because T_1 creates T_2; $PreT_1$ equals to $\{T_0\}$ because T_0 creates T_1; and $DDepT_1$ is equal to $\{T_3, T_4\}$ because T_3 and its descendant task T_4 have a data dependency relationship with T_1: T_3 due to the `depend` clause, and T_4 due to the `taskwait` in T_3. As a result, $BlockCT_1(GenS)$

is equal to $\{T_5, T_6, T_7\}$, and so these tasks can suspend their execution and block a thread that T_1 could not use. In the case of task T_5, $BlockCT_5$ equals to $\{T_1\}$ or $\{T_3\}$ because there is a data dependency between T_1 and T_3 and then, both tasks are executed sequentially, T_3 after T_1. Therefore, only T_1 or T_3 may be simultaneously suspended in a thread, which will not be available for executing T_5.

BFS. When considering the BFS strategy (and a FIFO policy), $BlockCT_i(BFS)$ is defined as follows:

$$\begin{aligned} BlockCT_i(BFS) &= BlockCT_i(GenS) \setminus SAftT_i = \\ &= ((T \setminus \{T_i\} \setminus DesT_i \setminus PreT_i \setminus DDepT_i) \cap TSPT_{bfs}) \setminus SAftT_i, \end{aligned} \tag{4.6}$$

where $BlockCT_i(GenS)$ is the set defined in Equation (4.5) and $SAftT_i$ is the set of sibling (and their descendant) tasks starting their execution after T_i, according to the FIFO policy. In other words, in order to compute $BlockCT_i(BFS)$, it is necessary to remove the tasks that start executing after T_i from $BlockCT_i(GenS)$. $SAftT_i$ includes only the tasks for which the execution order is known prior to run-time. $TSPT_{bfs}$ is different from $TSPT_{gens}$ because the *TSPs* defined at task creations are not considered in BFS. The reason is that the parent task is not suspended when it creates a child task, but rather it continues its execution in the same thread. The same consideration regarding the parallel tasks applies for $BlockCT_i(BFS)$.

Table 4.2 shows, for the example in Figure 4.1, the new set $TSPT_{bfs}$ and for each task T_i, the extra sets $SAftT_i$ needed for computing the values of $BlockCT_i(BFS)$. As an example, it is guaranteed that T_5 starts executing after T_1 ($T_5 \in SAftT_1$), since T_1 is created before and hence, it enters first the FIFO queue. However, whether T_2, a descendant task of T_1, starts executing before or after T_5 is unknown ($T_2 \notin SAftT_5$ and $T_5 \notin SAftT_2$). Moreover, although T_3 is created before T_5, it is not possible either to know if T_3 executes before T_5 ($T_3 \notin SAftT_5$ and $T_5 \notin SAftT_3$), because there exists a data dependency between T_3 and T_1 that affects the order in which T_3 can be executed. T_3 becomes ready only when T_1 completes, and weather T_5 starts executing before or after T_1 finishes, is unknown. For the task T_1, the computed $BlockCT_1(BFS)$ equals to $\emptyset$ because, according to the BFS policy, T_5, T_6 and T_7 are created after T_1 ($\{T_5, T_6, T_7\} \in SAftCT_i$) and T_6 never suspends its execution ($T_6 \notin TSPT_{bfs}$).

$TSPT_{bfs}$
$\{T_0, T_1, T_3, T_5, T_7\}$

T_i	$SAftCT_i$	$BlockCT_i(BFS)$
T_0	$\emptyset$	$\emptyset$
T_1	$\{T_3, T_4, T_5, T_6, T_7, T_8, T_9\}$	$\emptyset$
T_2	$\emptyset$	$\{T_5, T_7\}$
T_3	$\emptyset$	$\{T_5, T_7\}$
T_4	$\emptyset$	$\{T_5, T_7\}$
T_5	$\emptyset$	$\{T_1\}$ *or* $\{T_3\}$
T_6	$\emptyset$	$\{T_1\}$ *or* $\{T_3\}$
T_7	$\{T_9\}$	$\{T_1\}$ *or* $\{T_3\}$
T_8	$\emptyset$	$\{T_1\}$ *or* $\{T_3\}$
T_9	$\emptyset$	$\{T_1, T_7\}$ *or* $\{T_3, T_7\}$

Table 4.2: Sets to compute the number of threads available to new tied tasks (BFS approach), for the example in Figure 4.1.

WFS. Finally, in case of the WFS strategy, the set $BlockCT_i(WFS)$ is empty, because all tasks T_i start executing immediately after their creation in the parent task thread:

$$BlockCT_i(WFS) = \emptyset. \tag{4.7}$$

Overall, the cardinality[5] of the sets $BlockCT_i(GenS)$ and $BlockCT_i(BFS)$ determines the maximum number of idle threads which will not be available to execute a new tied task T_i when it is created. In the worst case, all these threads will be blocked by suspended tasks due to the *TSC 2*. In case of the WFS strategy, even if tied task can also block threads, when a new task T_i is created, it executes in the parent task thread. Hence, $|BlockCT_i(WFS)| = 0$. In any case, it is also necessary to consider the constraints a tied task has to resume its execution. Next section analyzes this situation.

4.4.1.2 At resumption time

When a suspended *tied* task T_i is ready to resume, it may not restart its execution, even though there may be idle available threads. The reason is that the thread to which T_i is tied to could be executing another task. This situation occurs only when T_i has been suspended in a *TSP* and, at resumption time, another predecessor, descendant or *untied* task is executing in the thread. There may be other idle threads

[5]The *cardinality* of a set A, expressed as $|A|$, is a measure of the number of elements of the set.

T_i	$BlockRT_i(GenS)$ and $BlockRT_i(WFS)$	$BlockRT_i(BFS)$
T_0	$\{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9\}$	$\{T_7, T_8, T_9\}$
T_1	$\{T_0, T_2\}$	$\emptyset$
T_3	$\{T_0, T_4\}$	$\emptyset$
T_5	$\{T_0, T_6, T_7, T_8, T_9\}$	$\{T_7, T_8, T_9\}$
T_6	$\{T_0, T_5, T_7, T_8, T_9\}$	—
T_7	$\{T_0, T_5, T_6, T_8\}$	$\{T_0, T_5\}$

Table 4.3: Tasks that may block threads at resumption time for each task $T_i \in TSPT_{gens}$ in Figure 4.1.

but the task cannot resume its execution because it is tied to its thread.

GenS. Given a GenS strategy and an OpenMP task $T_i \in TSPT_{gens}$, we define $BlockRT_i(GenS)$ as the set of potential tasks that may prevent (block) T_i from resuming its execution in the thread to which T_i is tied to:

$$BlockRT_i(GenS) = DesT_i \cup PreT_i \cup uT, \tag{4.8}$$

where $DesT_i$ is the set of descendant tasks of T_i, $PreT_i$ is the set of predecessor tasks of T_i and uT is the set of untied tasks. $BlockRT_i(GenS)$ contains only predecessor and descendant tasks of T_i and all the untied tasks because, due to *TSC 2*, while T_i is suspended, only these tasks can be scheduled to the same thread that executes T_i. Therefore only these tasks can prevent T_i to resume its execution.

Second column of table 4.3 shows, for the example in Figure 4.1, the computed values for the sets $BlockRT_i(GenS)$, for all tasks $T_i \in TSPT_{gens}$. The sets $DesT_i$ and $PreT_i$ shown in Table 4.1, are needed to compute $BlockRT_i$, as well as the set uT, which in this example equals to the empty set ($uT = \emptyset$). Hence, given task T_0, $BlockRT_0(GenS)$ contains all its descendant tasks because all of them can be scheduled in the same thread and prevent T_0 from resuming its execution. For the rest of tasks, descendant and predecessor tasks are considered to compute $BlockRT_i(GenS)$.

BFS. When considering the BFS strategy, to compute $BlockRT_i(BFS)$ it is necessary to discard from $BlockRT_i(GenS)$ the tasks that cannot block T_i's thread when it resumes execution, considering that:

1. TSPs upon task creation do not suspend the execution of T_i, by definition of the

BFS strategy, i.e., after the creation of a child task, T_i continues the execution. That is, as considered in the previous section, $TSPT_{bfs} \neq TSPT_{gens}$.

2. When T_i may suspend in a TSPs upon a task synchronization construct (a `taskwait`, `taskgroup`, or a `barrier` construct), only the descendant tasks not affected by the synchronization can block T_i's thread. If any other descendant task is executing on T_i's thread, then T_i is not ready to resume its execution but waiting for this task to complete.
3. The thread executing T_i may be blocked by T_i's predecessors only if they have a `taskyield` construct, or a task synchronization construct that does not affect T_i.

Third column of table 4.3 shows, for the example in Figure 4.1, the computed values for the sets $BlockRT_i(BFS)$ for all tasks $T_i \in TSPT_{bfs}$ (T_6 is not considered). AS an example, given task T_0, we analyze independently each TSP in which T_0 can be suspended, this is the `taskwait` at line 43 in Listing 4.2. Then, we remove all child tasks (and recursively, the descendant tasks) that are involved in this `taskwait`, i.e., task T_1 (and recursively its child task T_2 due to another synchronization construct at line 10), task T_3 (and similarly T_4) and task T_5 (and similarly T_6 but not its descendants T_7, T_8 and T_9 because there is no synchronization dependency with T_6). As a result, only tasks T_7, T_8 and T_9 compose the set $BlockRT_0(BFS)$. Given task T_1, T_0 is not considered in $BlockRT_1(BFS)$ since the TSP of T_0 is a `taskwait` in which T_1 is involved. T_2 is not considered either because it is involved in the synchronization construct (`taskwait`) of T_1. Similar situations occur when computing $BlockRT_3(BFS)$, $BlockRT_5(BFS)$ and $BlockRT_7(BFS)$.

WFS. In the case of WFS strategy, given an OpenMP task $T_i \in TSPT_{gens}$, the set $BlockRT_i(WFS)$ is equal to $BlockRT_i(GenS)$ since the scheduling strategy does not impose any extra condition that reduces the number of tasks that may block a thread when T_i resumes its execution.

WFS is particularly affected when *tied* tasks are implemented, because the parallel execution turns into a sequential execution. When any task T_i is created, it starts its execution in the thread that was executing the parent task. Therefore, the parent task is suspended and it cannot resume its execution in another thread because it is tied to its thread. On the contrary, if T_i is suspended in a TSP (not a task creation) and T_i's parent task resumes its execution, then T_i is blocked because of its parent

task.

Overall, the analysis done in this section demonstrates that, given a task T_i ready to resume its execution, the tasks in the sets $BlockRT_i$ may be blocking the thread to which T_i is tied to. There may be idle threads but the tied model prevents T_i to execute on these threads.

4.4.2 Issues on the timing characterization of tied tasks

The reasoning about the computation of $BlockCT_i$ and $BlockRT_i$ suggests that deriving schedulability results when tied tasks are involved is extremely challenging, unless very pessimistic assumptions are made. More specifically, in Section 4.3 we have leveraged the work-conserving policy implied by the use of *untied* tasks to derive a timing analysis simply based on three quantities: (1) the critical path length of the OpenMP-DAG; (2) the volume of the OpenMP-DAG; and (3) the available number of threads m. The response time analysis considers the critical path plus the interference of the rest of the OpenMP-DAG evenly distributed among the available threads.

However, when considering the non-work-conserving scenario induced by tied tasks, deriving such an accurate analysis is not as easy as for untied tasks, due to multiple reasons:

1. It is not correct to compute the critical path of the task graph as a whole, but rather a critical path reaching the end of each task in the OpenMP-DAG, since it is important to compute the different time offsets after which each task can start executing. In fact, since each task has its own descendant and precedence relationships, the corresponding $BlockCT_i$ and $BlockRT_i$ sets will be different, suggesting to carry out a *per-task* timing analysis.
2. The interference contribution for a tied task cannot be considered as evenly distributed. Specifically, it is necessary to differentiate the interference contribution before the task starts, which can be accounted for as evenly distributed on the threads being blocked due to $BlockCT_i$, and the interference suffered by the task at each of its *TSPs*, which includes the full contribution of the set of tasks $BlockRT_i$.
3. The critical path reaching the end of a task T_i may include parts of other tasks that can have different descendant relationships with respect to T_i. This makes

really hard to identify which tasks may actually interfere with T_i without introducing unacceptable pessimism in the analysis. In order to have an intuitive feeling of the problem, consider again the example given in Figure 4.1, where all task parts have unitary WCETs. Here, task T_3 has a data dependency with T_1, hence it cannot start executing until T_1 has finished. When computing the critical path reaching the end of T_3, we immediately observe that it is not simply composed of tasks that are predecessors of T_3, but also by parts of T_1 and T_2 p_{10}, p_{11} and p_{20} (that are not predecessors of T_3). Hence, the interference imposed on critical task parts of T_3 cannot simply be estimated based on the descendant relationships of T_3 (i.e., by the knowledge of $BlockRT_3$), but should take into account those of all the tasks involved, which hugely complicates the analysis.

4. From the analytical point of view, computing an upper-bound on the response-time of a tied task T_i would require to assume the worst-case scenario in which all the tasks that can be suspended simultaneously at the creation point of T_i are indeed suspended, inhibiting T_i to execute on the corresponding threads tied to these tasks. Therefore, besides knowing the maximum number of tasks that could be suspended at the time of T_i's creation due to *TSC 2* (i.e., $BlockCT_i$), we should provide an upper-bound on the maximum time the suspended tasks would take before being resumed.

Overall, the above considerations confirm that a timing analysis for tied tasks under the considered scheduling algorithms, besides being conceptually very difficult to achieve, would require to address sources of inherent complexity that would lead to unacceptably pessimistic response-time bounds. As a result, the makespan of the task graph may undergo large variations depending on the allocation of newly generated tasks, leading in few cases to resource under-utilization and undesirable idleness of some threads as shown in the next section.

Interestingly, based on the work presented in this chapter, Sun et al. [75] propose a new algorithm to schedule OpenMP task systems composed of tied tasks. Based on this new scheduling algorithm, which avoids tying too much workload to the same thread, they provide a response time analysis for tied tasks.

```
1  #pragma omp parallel
2              num_threads(4) {
3  #pragma omp single { // T0
4    part00
5    #pragma omp task { // T1
6      part10
7      #pragma omp task {  // T2
8        part20
9        #pragma omp task {  // T3
10         part30
11         #pragma omp task  // T4
12         { part40 }
13         #pragma omp taskwait
14         part31
15       }
16       #pragma omp taskwait
17       part21
18     }
19     #pragma omp taskwait
20     part11
21   }
22   part01
23   #pragma omp task { part50 } // T5
24   part02
25   #pragma omp task { part60 } // T6
26   part03
27   #pragma omp task { part70 } // T7
28   part04
29 }}
```

Listing 4.3: Example of an OpenMP program leading to a pessimistic scheduling of tied tasks.

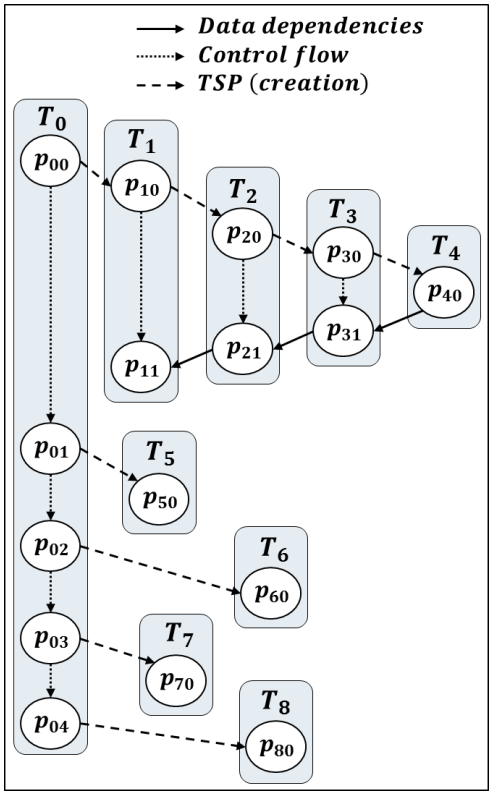


Figure 4.2: OpenMP-DAG corresponding to the OpenMP program in Listing 4.3.

4.4.3 Platform under-utilization

As previously observed, the use of tied tasks encompasses their suspension and resumption only by the same thread that first started their execution. This may lead to platform under-utilization problems, reducing the number of threads working, even if there are tasks ready to execute. We refer as m_i^* to the minimum number of threads available to task T_i at the time of its creation. Since not all threads may be available to a task when it is created, it follows that the interference suffered from other tasks cannot be considered to be evenly distributed across all threads, but only on $m_i^* \leq m$ threads.

Theorem 2. *The minimum value of m_i^* is 2, for any task graph comprising tied tasks.*

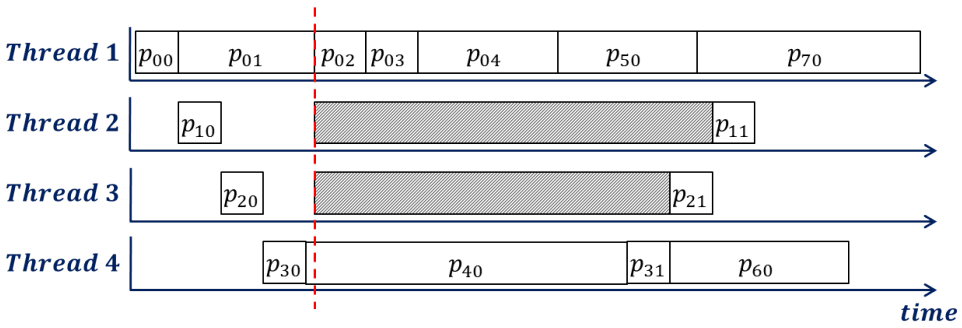
Proof. The statement can be demonstrated by the two following points: (i) providing

a configuration where $m_i^* = 2$, and (ii) showing that no configuration can be produced with $0 \leq m_i^* < 2$.

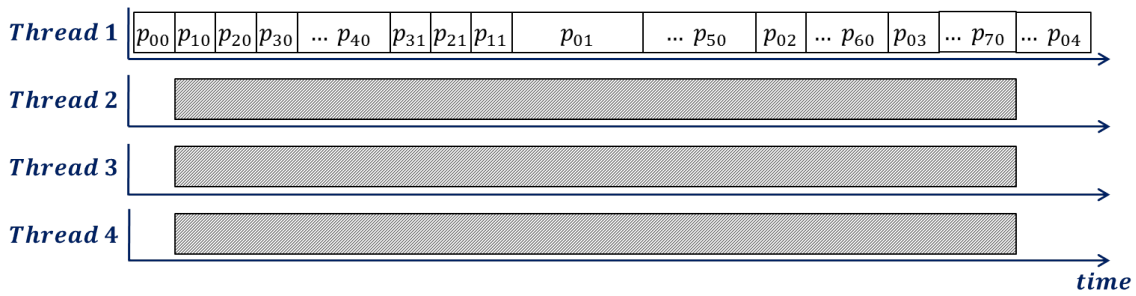
(i) There exists a scenario where $m_i^* = 2$. Consider the OpenMP program illustrated in Listing 4.3 (and its corresponding OpenMP-DAG in Figure 4.2). Suppose that the program must be executed on $m = 4$ threads and that the allocation on the available threads is as shown in Figure 4.3a. Tasks T_1, T_2 and T_3 must wait for their first-level descendants to finish, due to the `taskwait` directives. Then, if task parts p_{04} and p_{40} have a very long execution time, there is a long time interval where T_5, T_6 and T_7 cannot execute on threads 2 and 3, although they are idle, due to *TSC 2*. T_5, T_6 and T_7 can only be scheduled in threads 1 and 4 which are used by tasks T_4 and T_0, respectively. Therefore, T_5, T_6 and T_7 cannot start their execution until they finish and such a time interval can be arbitrarily long depending on the WCET of task parts p_{04} and p_{40}.

(ii) There is no configuration such that $m_i^* = 0$ or 1. It cannot be $m_i^* = 0$ because this would mean that all m threads contain tasks simultaneously suspended in a *TSP*, but then none of them would make progress (i.e., a deadlock occurs). In this case, no new task can be created, hence the blocking due to *TSC 2* cannot be experienced.

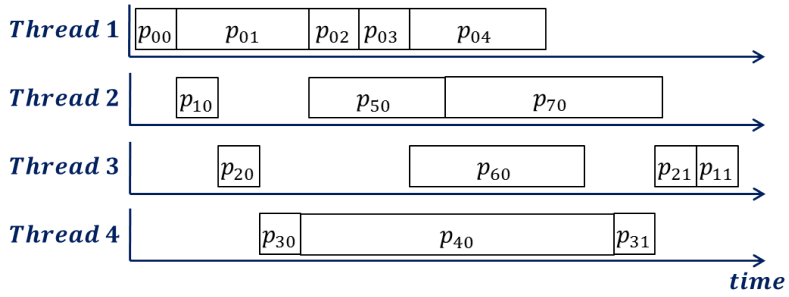
Analogously, it cannot be $m_i^* = 1$. By contradiction, assume $m_i^* = 1$. This means that when task T_i is released, $m - 1$ threads are not available to it due to *TSC 2*, i.e., $m - 1$ threads are blocked by tasks that are not predecessors of T_i. Such $m - 1$ tasks must be suspended in a *TSP*, and cannot continue executing because some of their synchronization constraints are not fulfilled. This can only happen when some task must wait for its first-level descendants, due to a `taskwait` or an `if` clause that evaluates to false. The semantics of the synchronization constraints implies that there cannot be any synchronization arrow that traverses multiple levels: synchronization arrows can either connect siblings in the case of data-dependencies, or first-level descendants (childs) to their parent task, in the case of `taskwait` or `if` clause. As a result, it follows that the $m - 1$ tasks must belong to $m - 1$ contiguous descendant levels $[l_x, l_{x+m-2}]$. Therefore, the task that generates T_i must belong to l_i, being either $i \leq x - 1$ or $i \geq x + m - 1$. In the case $i \leq x - 1$, a contradiction is reached, because each of the m threads executes at least one task, but the task belonging to $x + m - 2$ has no descendant, hence there is no reason why it should be suspended in a *TSP*. If instead $i \geq x + m - 1$, then the task that generates T_i is descendant of all the other $m - 1$ tasks, and the same holds for T_i. This facts also



(a) Pessimistic BFS example with tied tasks



(b) WFS (LIFO) with tied tasks



(c) BFS example with untied tasks

Figure 4.3: Scheduling alternatives of the program in Listing 4.3.

imply a contradiction because *TSC 2* comes into play only when the generated task is not descendant of the other ones. In conclusion, there is no situation such that $m_i^* = 1$, proving the theorem. □

Therefore, we define m_i^* as:

$$m_i^* = max(2, m - |BlockCT_i|), \tag{4.9}$$

where $|BlockCT_i|$ is the maximum number of tasks that may block threads which T_i could not use at its creation time due to *TSC 2*. As we consider all potential cases,

this number of tasks can be greater than the total number of threads, m. Therefore, $m - |BlockCT_i|$ may be negative, but it is proven by Theorem 2 that the minimum value of m_i^* is 2. Hence, in this case, an accurate timing analysis should identify which tasks compose this subset in the worst-case, since only a subset of the tasks composing $BlockCT_i$ will subtract threads to the considered task. However, when tied tasks are involved, it is absolutely non-trivial to identify the scenario that maximizes the interference imposed on T_i. This is another subtle reason (in addition to those listed in Section 4.4.2) that explains why devising a timing analysis for tied tasks is a computationally-intensive and overly pessimistic process.

Figure 4.3 illustrates possible scheduling of the OpenMP program in Listing 4.3 In particular, Figure 4.3a shows a case of resource under-utilization implied by the use of tied tasks considering BFS: if all the nested tasks are scheduled in different threads before T_5, T_6 and T_7, and being $part_{04}$ and $part_{40}$ very time-consuming, then the execution of tasks T_5, T_6 and T_7 is postponed even if threads 2 and 3 are idle (striped areas) but tied to tasks T_1 and T_2. Figure 4.3b shows the scheduling considering WFS (LIFO): as already noted, WFS turns into a sequential execution when implementing tied tasks. Notice that in this figure task parts $p_{40}, p_{50}, p_{60}, p_{70}$ and p_{04} are less time-consuming only for the sake of space-saving. If the clause `untied` is added to all the tasks in the program in Listing 4.3, we observe that the breadth-first scheduling of these untied tasks, illustrated in Figure 4.3c, determines no platform under-utilization beyond program limitations. WFS will result in a similar scheduling for untied tasks.

4.5 Related Work

The OpenMP language committee presented a comparison between the thread-centric and the task-centric models, exposing the design choices done in the tasking model due to conflicts with the thread-centric model [79]. These decisions include the definition of *tied* and *untied* tasks, data-sharing clauses and scheduling constraints. Duran et al. [81] performed an evaluation of different scheduling policies using the Nanos++ runtime system [70], and analyzed the differences existing between *tied* and *untied* tasks for an average performance point of view. However, none of these works take time predictability into account.

OpenMP has been already considered as a convenient interface to describe real-time applications to deal with parallel task models in multiprocessor systems. The

earliest parallel task models proposed to represent OpenMP parallel applications are the *fork-join* model [27] and the *parallel synchronous* model [85] [37]. These works consider the OpenMP thread-centric model. Vargas et al. firstly considered the *DAG task model* to represent OpenMP applications parallelized with the tasking model [26]. The authors studied how to construct an OpenMP task graph which contains enough information to allow the application of real-time DAG scheduling models, from which timing guarantees can be derived.

Unfortunately, besides the increasing expressiveness provided by existing real-time parallel task models, all of them neglect the real semantics of the OpenMP execution model and bypass the functionality of the runtime system. Instead, the purpose of the work presented in this chapter is to demonstrate that the OpenMP tasking model can be applied to real-time systems if work-conserving schedulers, such as BFS and WFS, are used. We present the first scheduling analysis of an OpenMP-DAG composed of untied tasks. Based on this work, Sun et al. [75] presented an interesting work that addresses the scheduling analysis of an OpenMP-DAG composed of `tied` tasks. The authors provide a new scheduling algorithm and develop two response time bounds for the new algorithm, whose difference is a trade-off between simplicity and analysis precision.

4.6 Summary

OpenMP is a firm candidate to address the performance challenges of critical real-time systems. However, OpenMP was originally intended for a different purpose than critical real-time systems, for which guaranteeing the correct output is as important as guaranteeing it within a predefined time window. In this chapter, we evaluate the use of OpenMP in critical real-time systems. We take into account the timing constraints of such systems by considering the use of the sporadic DAG tasks model, used in real-time systems for providing timing guarantees of parallel applications.

Concretely, we analyze from a timing perspective the two tasking execution models existing in OpenMP, *tied* and *untied*. The existence of these two models results from the coexistence of the thread-centric and task-centric models, for backward compatibility reasons. The considerations drawn in this chapter suggest that using *tied* tasks inside time-critical applications is not recommendable. The reason is the inherent pessimism that underlies the timing analysis of such tasks and the conceptual

difficulties behind the construction of an accurate schedulability test. On the other hand, we have shown that a simple schedulability analysis of OpenMP programs is possible whenever *untied* tasks are involved. This definitely suggests that the use of *untied* tasks is preferable for parallel applications in the real-time context, since it allows to exploit a parallel execution model in a predictable way.

Chapter 5

Response Time Analysis under the Limited Preemptive Scheduling

"*Nunca el tiempo es perdido.*"[1]

— Manolo Garcia

Previous chapters demonstrate the convenience of using OpenMP to parallelize computationally intensive functionalities of critical real-time systems. One of the main reasons is given by the similarities between the OpenMP tasking model and some features widely used in the real-time community: the sporadic DAG tasks model and the limited preemptive scheduling strategy. Despite there is plenty of literature about the DAG scheduling model on the one hand, and the limited preemptive scheduling on the other hand, the current state of the art does not consider the sporadic DAG scheduling model under the limited preemptive scheduling strategy.

Chapter 3 (Section 3.2.2) provides a first analysis about the similarities of the execution model of OpenMP tasks and the limited preemptive scheduling. This chapter presents a novel response time analysis for DAG-based real-time systems under the limited preemptive scheduling strategy. The set of real-time tasks is scheduled under global fixed priority scheduling, i.e., each real-time task has a unique given priority, and is allowed to execute on any of the available cores.

This new analysis is key to predict the timing behavior of critical real-time systems implemented with OpenMP. Besides its applicability to OpenMP, this analysis can also be applied to other parallel computing models, as long as the real-time tasks are represented as DAG tasks.

[1]English translation: "*Never the time is lost.*"

5.1 Limited Preemptive Scheduling

The limited preemptive scheduling approach has been proposed as an effective scheduling scheme that reduces preemption-related overheads of the fully preemptive approach while constraining the amount of blocking of the non-preemptive approach [33] (see Section 3.2.2 in Chapter 3 for further details). In the limited preemptive scheduling with fixed preemption points, preemptions can only take place at certain points during the execution of a task, dividing its execution into non-preemptive regions (NPR).

5.1.1 Extending the system model

The system model presented in Section 2.1.2 must be extended to incorporate the particularities of the limited preemptive scheduling and the priorities of the tasks. According to this model, each task τ_k of the system is represented as a DAG $G_k = (V_k, E_k)$. V_k is the set of nodes, that now represent non-preemptive regions (NPRs) as defined by the limited preemptive scheduling. Therefore a DAG task can only be preempted at node's boundaries, being $q_k = |V_k| - 1$ the number of potential preemption points of τ_k. E_k is the set of edges.

Under a fixed task priority scheduling algorithm, real-time tasks are ordered according to their decreasing unique priority, i.e., τ_i has a higher priority than τ_j if $i < j$. For each real-time task $\tau_k \in \mathcal{T}$, the sets $hp(k)$ and $lp(k)$ are identified as the sets of real-time tasks with higher and lower priorities than τ_k, respectively. In this chapter, we consider a sporadic task system with constrained relative deadline $D_k \leq T_k$. Tasks $\tau_k \in \mathcal{T}$ are globally scheduled on a platform composed of m identical cores.

5.1.2 Similarities with the OpenMP tasking model

As seen in Chapter 2, the OpenMP API defines the task scheduling points (TSPs) as points in the program where the encountering OpenMP task can be suspended (preempted), being the executing thread rescheduled to a different task. As a result, OpenMP tasks are divided into task parts, that are represented as nodes in the OpenMP-DAG.

If we apply a direct mapping between OpenMP threads and cores, as considered

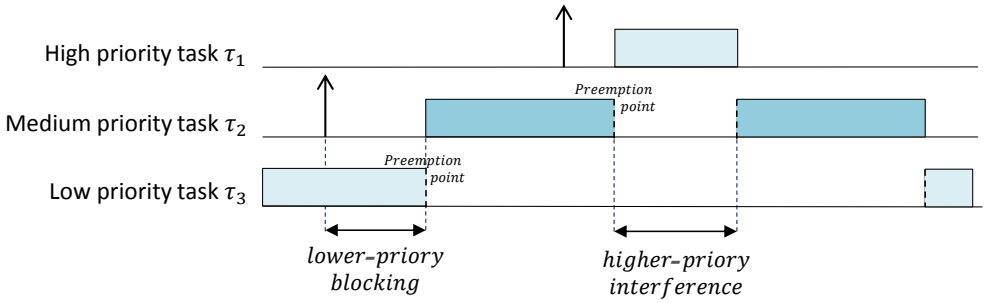


Figure 5.1: Example of limited preemptive scheduling for three sequential tasks.

in Chapter 3 (see Section 3.2.3), the execution model for the OpenMP tasking model resembles the limited preemptive scheduling. There is a direct correspondence between nodes in the OpenMP-DAG (task parts) and NPRs in the limited preemptive scheduling. That is, given an OpenMP-DAG, preemptions are only allowed at task parts' boundaries.

5.2 Response Time Analysis

When considering a set of tasks globally scheduled under the limited preemptive strategy, they can suffer (1) *higher-priority interference* and (2) *lower-priority blocking* times (also called *lower-priority interference*). The reasons are: (1) a ready task waits for the higher priority tasks to finish their execution and (2) the execution of a task cannot be suspended until a preemption point is reached, so a ready high priority task may have to wait for one, or more, running lower priority tasks to finish their execution. As we will see in Section 5.2.2, the number of running tasks of lower priority that a given task must wait depends on the limited preemptive scheduling strategy.

Figure 5.1 shows a simple example of limited preemptive scheduling for three sequential tasks, τ_1, τ_2 and τ_3 (in decreasing priority order), running on an single core. The task of interest is τ_2, which has a higher priority task τ_1 and a lower priority task τ_3. As a result, τ_2 suffers lower-priority blocking time from its release time until τ_3 reaches a preemption point. Moreover, since τ_1 is released while τ_2 is running, τ_2 must suspend its execution at its preemption point, until τ_1 finishes its execution, suffering higher-priority interference.

Therefore, the response time upper bound presented in Chapter 4 (Section 4.3) for a single DAG task (with only intra-task interference), must be extended to consider

a DAG-based task-set under limited preemptive scheduling (with both intra-task and inter-task interference). The response time analysis must incorporate the *inter-task interference*, given by both the higher-priority and the lower-priority interference. As a result, the response time upper bound of each task τ_k of a DAG-based task-set, under limited preemptive global fixed priority scheduling is computed as:

$$R_k^{ub} \leftarrow len(G_k) + \frac{1}{m}\big(vol(G_k) - len(G_k)\big) + \frac{1}{m}\big(I_k^{hp} + I_k^{lp}\big) \tag{5.1}$$

where $len(G_k)$ is the length of the critical path, $vol(G_k)$ is the volume of the task, I_k^{hp} denotes the higher-priority interference factor and I_k^{lp} denotes the lower-priority interference factor. In the next sections, we describe how to compute a valid upper bound for these two factors. The schedulability of the system is checked by comparing $R_k^{ub} \leq D_k$ for all the real-time tasks $\tau_k \in \mathcal{T}$.

5.2.1 Higher-priority interference

Melani et al. [54] considered a fully-preemptive scheduler in which DAG tasks only suffer interference from higher priority tasks. The reason is that a low priority task is preempted as soon as a higher priority task is ready to execute. They compute the higher-priority interference I_k^{hp} as the amount of time, in the worst case, during which each higher priority task executes, hence, being task τ_k pending and not executing.

The computation of I_k^{hp} and therefore, the response time analysis, is based on the notion of *problem window*, in which I_k^{hp} is computed considering a time interval named *window of interest*. Then higher-priority interference contribution of each interfering task τ_i in the problem window is divided between *carry-in job*, *body jobs*, and *carry-out job*, where:

- The carry-in job is the first instance of τ_i that is part of the problem window. It is released before the window of interest and has the deadline within it.
- The carry-out job is the last instance of τ_i executing in the problem window. It is released within the window of interest and has the deadline after it.
- All other instances of τ_i are named body jobs.

The following Lemma, rephrased from [54], provides a valid upper bound for I_k^{hp} by taking, for each task in $hp(k)$, the densest possible packing of parallel nodes i.e., the volume, given a problem window of length R_k^{ub}.

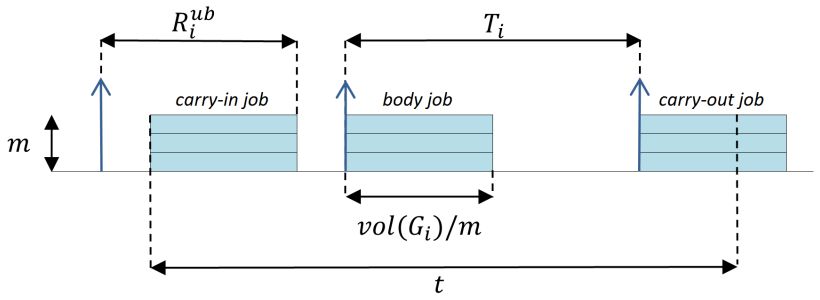


Figure 5.2: Worst-case workload of a task τ_i in a window on length t.

Lemma 2 (From [54]). *An upper-bound on the higher-priority interference of a task τ_k in a window of length R_k^{ub} is given by*

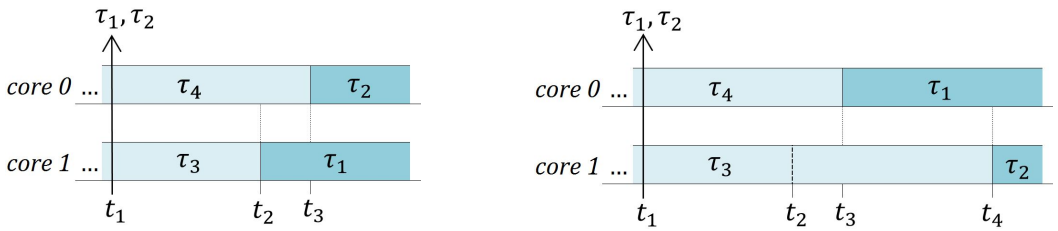
$$I_k^{hp} \leq \sum_{i \in hp(k)} \mathcal{W}_i(R_k^{ub}) \tag{5.2}$$

where

$$\mathcal{W}_i(t) = \left\lceil \frac{t + R_i^{ub} - vol(G_i)/m}{T_i} \right\rceil vol(G_i) \tag{5.3}$$

$\mathcal{W}_i(t)$, represented in Figure 5.2, is the maximum workload of an interfering task τ_i in a window of length t. It happens when (i) the volume of the higher priority task τ_i is evenly divided among all m cores; (ii) the carry-in job executes as late as possible, i.e., close to its worst-case response time which is upper-bounded by R_i^{ub}; and (iii) later instances execute as soon as possible, i.e., when they are released with the minimum inter-arrival time. By considering a full contribution of both carry-in and carry-out instances, the lemma follows. The corresponding Lemma V.1 in [54] does not consider a full carry-out contribution, but only the share that fits the considered problem window. However, we found that such a tighter estimation does not improve the analysis, since the response-time iteration will always continue until a full carry-out instance is considered. This observation allowed us to simplify the formula without introducing pessimism.

Since R_k^{ub} is needed to compute R_k^{ub}, the response time analysis is an iterative procedure: the initial window of interest of a task τ_k is $R_k^{ub} = len(G_k) + \frac{1}{m}\big(vol(G_k) - len(G_k)\big)$, for which I_k^{hp} (equation 5.2) and I_k^{lp} (equation in the next section) are computed. Then, a new window of interest R_k^{ub} is iteratively computed by equation 5.1 until a fixed point is reached. Moreover, the response time upper bound is computed for all the real-time tasks in decreasing priority order, i.e., starting from the highest priority task τ_1, for which $I_1^{hp} = 0$. The reason is that, for computing I_k^{hp} for the



(a) Eager approach **(b)** Lazy approach

Figure 5.3: Limited preemptive scheduling of sequential tasks τ_1, τ_2, τ_3 and τ_4 on a 2-core processor.

rest of tasks $\tau_k, k > 1$, the response time upper bound of the higher priority tasks $R_i^{ub}, i < k$ must be previously computed (see equation 5.3).

5.2.2 Lower-priority interference

Under the limited preemptive scheduling, we consider two approaches that lead to different values of the lower-priority interference.

- The *eager* approach, where a high priority task preempts the first lower priority executing task that encounters a preemption point.
- The *lazy* approach, where a high priority task waits until the lowest priority executing task reaches a preemption point.

Figure 5.3 shows an example of the eager and lazy approaches for a system composed of four sequential tasks executing on two cores. The priority order of these tasks, from higher to lower, is: $\tau_1, \tau_2, \tau_3, \tau_4$. Assume that tasks τ_3 and τ_4 are already executing when τ_1 and τ_2 are released at time instant t_1. Under the eager preemption approach (Figure 5.3a), τ_1 starts executing as soon as the lower priority task τ_3 reaches a preemption point, which occurs at time instant t_2. Similarly, task τ_2 starts the execution at time instant t_3 when the next lower priority task τ_4 reaches a preemption point. Under the lazy approach instead (Figure 5.3b), τ_1 waits until τ_4, the lowest priority running task, reaches a preemption point at time t_3. Notice that, another low priority task, τ_3, reached a preemption point before, at time instant t_2, but it has not been preempted because it was not the lowest priority executing task. Hence, task τ_2 is further blocked by τ_3 until τ_3 reaches its next preemption point at time instant t_4.

Given these two approaches, to compute an upper bound on the lower-priority interference, it is necessary to identify:

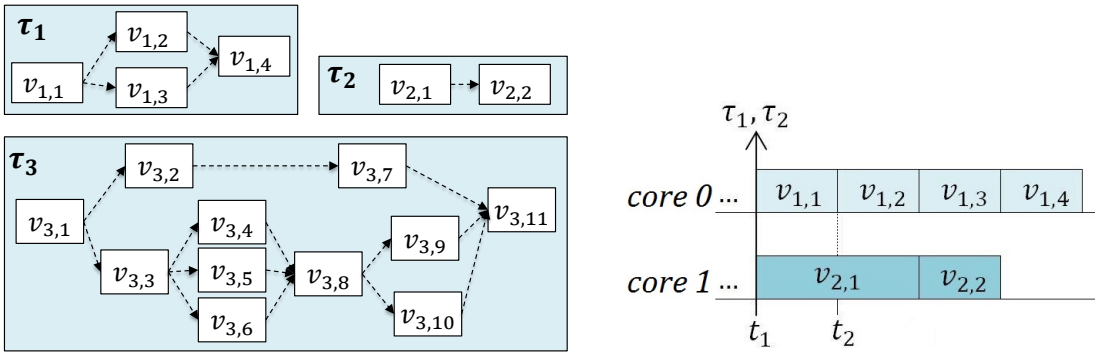
1. The situations in which a *priority inversion* may occur, i.e., when a task is blocked while other lower priority tasks are executing. As noted in previous works addressing global limited preemptive schedulability analysis [86, 87], a task may be blocked before the beginning of its execution by lower priority tasks that already started executing, and it may also suffer additional priority inversions due to later lower priority instances.
2. The amount of *blocking time* for each of the priority inversions, i.e., how much time a task is blocked by lower priority instances.

To understand why a task may be blocked by lower priority tasks after it started executing, consider a task-set composed of sequential tasks, with eager preemptions. In this setting, a task τ_k may start executing along with one or more lower priority instances $\tau_{i>k}$. If a higher priority task is released, τ_k may be preempted if it is the first task reaching a preemption point, even if it is not the lowest priority executing task. This causes τ_k to be preempted, while lower priority tasks $\tau_{i>k}$ continue executing, leading to a priority inversion. This situation lasts until one of the running tasks $\tau_{i>k}$ reaches a preemption point and so τ_k can resume its execution. In Figure 5.3a, τ_3 suffers lower-priority interference after it started executing between time instants t_2 and t_3.

When considering DAG-based task-sets, this scenario becomes more complex. This is due to the parallelism exposed by DAG tasks, which may dynamically vary depending on which portion of the DAG is being executed. Hence, a DAG task τ_k may experience lower-priority blocking, even without being preempted by higher priority tasks. This occurs when τ_k requires additional cores to fork two or more parallel nodes. If cores are busy executing lower priority instances, τ_k experiences blocking time on the forked nodes until lower priority instances reach a preemption point.

Figure 5.4 shows an example of this scenario. We consider a system composed of two DAGs, only τ_1 and τ_2 from Figure 5.4a (τ_3 is not considered here), executing on $m = 2$ cores. τ_1 has higher priority than τ_2. All nodes of τ_1 and τ_2 have unitary WCET, except node $v_{2,1}$ with $C_{2,1} = 2$. Despite task τ_1 is the highest priority task, it may be blocked by τ_2 after τ_1 starts its execution. This scenario is shown in Figure 5.4b. The two tasks start executing at the same time instant t_1. When $v_{1,1}$ finishes, only one of the forked nodes, $v_{1,2}$, is scheduled at time instant t_2, while $v_{1,3}$ is blocked by the lower priority task τ_2 executing on the other core.

Overall, the worst-case scenario for lower-priority interference occurs (1) when a



(a) Task-set composed of three DAG tasks. **(b)** Scheduling of tasks τ_1 and τ_2.

Figure 5.4: Example of DAG task-set under limited preemptive scheduling.

task τ_k is released, if all the m cores are executing lower priority tasks, and (2) after τ_k starts executing, if it is blocked by lower priority tasks each time a priority inversion occurs. In this case, at most $m-1$ lower priority tasks may be executing (since τ_k is running in the other core [87]). Therefore, the lower-priority interference I_k^{lp} is computed as:

$$I_k^{lp} = \Delta_{k,m} + p_k(R_k^{ub}) \times \Delta_{k,m-1} \tag{5.4}$$

where $p_k(R_k^{ub})$ is an upper-bound on the number of *additional priority inversions* suffered by τ_k during its response time after it starts executing, and $\Delta_{k,m}$ and $\Delta_{k,m-1}$ are upper-bounds on the lower-priority blocking time, when τ_k is released (on the source node, or first NPR) and after it starts executing (on the rest of nodes or NPRs), respectively. The values of these three factors depend on the selected limited preemptive scheduling approach. Sections 5.3 and 5.4 present how these factors must be computed for the eager and lazy approaches. Before that, the next section presents how to compute the number of additional cores requested by a DAG task after it starts executing. This factor is needed to compute the number of additional priority inversions.

5.2.2.1 Additional core requests

With the objective of identifying the number of additional priority inversions that a DAG task τ_k may suffer after starting its execution, we introduce the new parameter sw_k.

Definition 9. *sw_k is the maximum number of core requests that a DAG task τ_k additionally needs after starting its execution.*

sw_k identifies all the points at which a priority inversion may occur after τ_k starts executing. To better understand the reasoning behind this definition, consider the example in Figure 5.4a, where a DAG-based task-set composed of three tasks is represented. For the first task τ_1, $sw_1 = 1$ because one additional core is required when node $v_{1,1}$ forks nodes $v_{1,2}$ and $v_{1,3}$. For task τ_2, $sw_2 = 0$ because it does not require additional cores after it starts executing. Finally, $sw_3 = 4$ because τ_3 requires: (i) 1 additional core when node $v_{3,1}$ forks nodes $v_{3,2}$ and $v_{3,3}$ ($sw_3 = 1$); (ii) 2 additional cores when node $v_{3,3}$ forks nodes $v_{3,4}$, $v_{3,5}$ and $v_{3,6}$ ($sw_3 = 1+2$); and (iii) 1 additional core when node $v_{3,8}$ forks nodes $v_{3,9}$ and $v_{3,10}$ ($sw_3 = 1+2+1 = 4$). Notice that, even though τ_3 may occupy three cores when nodes $v_{3,4}$, $v_{3,5}$ and $v_{3,6}$ are ideally executed in parallel, two out of these three cores are freed before executing $v_{3,8}$. Therefore, we need to account again for an additional core when $v_{3,8}$ forks nodes $v_{3,9}$ and $v_{3,10}$.

It is important to distinguish between the *additional* core requests needed by τ_k, and accounted by sw_k, and the number of forks, the total forked nodes, or the maximum degree of parallelism. Notice that for task τ_3, the number of forks is 3 (after nodes $v_{3,1}$, $v_{3,3}$ and $v_{3,8}$), the total forked nodes are 7 (nodes $v_{3,2}$, $v_{3,3}$, $v_{3,4}$, $v_{3,5}$, $v_{3,6}$, $v_{3,9}$ and $v_{3,10}$), but $sw_3 = 4$. In this case, by coincidence, sw_3 equals the maximum degree of parallelism of τ_3 (nodes $v_{3,4}$, $v_{3,5}$, $v_{3,6}$ and $v_{3,2}$ or $v_{3,7}$ may run in parallel). However, for task τ_1, $sw_1 = 1$ but the maximum degree of parallelism is 2.

5.2.2.2 Computing the number of additional core requests

To compute sw_k it is required not only to account, for each node, the number of direct successors minus one, but also to consider the edges that prevent a group of direct successors to execute in parallel. For instance, if an extra edge would exist between nodes $v_{3,4}$ and $v_{3,5}$ of τ_3 (Figure 5.4a), only one additional core, instead of two, is required at this point.

Algorithm 2 computes the exact value of sw_k[2]. It takes as input the DAG $G = (V, E)$ and, for each node $v_i \in V$ the sets $DSucc(v_i)$ and $DPred(v_i)$, which are the sets of direct successors and direct predecessors of v_i, respectively. The algorithm iterates over all the nodes in V. At each iteration the number of additional cores required after the execution of v_i is initialized to its maximum possible value: the number of direct successors of v_i minus 1 (line 5). In the second loop, the algorithm iterates over all v_i's direct successors $v_j \in DSucc(v_i)$, to check if a core has already

[2]Notice that subscript k is omitted in the algorithm for simplicity.

Algorithm 2 Additional core requests caused by a DAG task.

Input: $G = (V, E)$: DAG task
$DSucc(v_i)\ \forall\ v_i \in V$: Direct successors of v_i
$DPred(v_i)\ \forall\ v_i \in V$: Direct predecessors of v_i
Output: sw

```
1  function ADDITIONAL_CORES
2      sw ← 0
3      N ← {}
4      for each v_i ∈ V do
5          cores ← |DSucc(v_i)| − 1
6          for each v_j ∈ DSucc(v_i) do
7              if v_j ∈ N then
8                  cores ← cores − 1
9              else
10                 if DSucc(v_i) ∩ DPred(v_j) ≠ φ then
11                     cores ← cores − 1
12                 end if
13                 N ← N ∪ {v_j}
14             end if
15         end for
16         sw ← sw + max(0, cores)
17     end for
18 end function
```

been accounted for the execution of v_j (line 7), or if there is an edge between v_j and any of its sibling nodes (line 10). In both cases the number of additional cores required decreases by one. Then, the v_j is added to N (line 13) to keep track of the nodes that have been already considered for accounting additional cores. Finally, sw is updated with the additional cores required after the execution of v_i (line 16). This algorithm has quadratic complexity in the number of nodes.

sw_k is required to compute the number of additional priority inversions p_k that, along with the blocking time factors $\Delta_{k,m}$ and $\Delta_{k,m-1}$, lead to the computation of an upper bound on the lower-priority interference. In the following sections we provide the equations and algorithms to compute the lower-priority interference to any DAG task scheduled under the eager or lazy limited preemptive approaches.

5.3 Eager Preemption Analysis

This section presents how to compute the number of priority inversions p_k, and the lower-priority blocking times $\Delta_{k,m}$ and $\Delta_{k,m-1}$, suffered by a task τ_k under the eager

limited preemptive scheduling.

5.3.1 Number of priority inversions

Under the eager approach, the first lower priority task to reach a preemption point is preempted, even if it is not the lowest priority running task. That is, when a highest priority task is ready to execute (or requests additional cores), the first lower-priority task τ_k running that reaches a preemption point, is preempted. As a result, τ_k can suffer lower-priority interference, i.e., a priority inversion may occur, not only before starting its execution, but also at later preemption points. Moreover, with a DAG task model, τ_k may also suffer lower-priority interference when it requires one or more additional cores for executing its forked nodes.

The next lemma provides an upper bound on the number of higher priority instances that may be released within the scheduling window of a job of a task τ_k.

Lemma 3. *In any time interval of length t, a job of task τ_k may be preempted by higher priority tasks at most h_k times:*

$$h_k(t) = \sum_{\tau_i \in hp(k)} \left\lceil \frac{t + R_i^{ub}}{T_i} \right\rceil (1 + sw_i) \tag{5.5}$$

where $hp(k)$ is the set of tasks with higher priority than τ_k, R_i^{ub} is the response time upper-bound of task τ_i, T_i is the period of τ_i and sw_i is the number of additional cores requested by τ_i after it starts executing.

Proof. Assume the job of τ_k is released at time $t_0 = 0$. During a time interval of length t, each higher priority task τ_i can be released at most $\left\lceil \frac{t+R_i^{ub}}{T_i} \right\rceil$ times. Following the definition of sw_i in the previous section, it descends that the number of cores requests (and so potential preemptions of τ_k) by a single instance of τ_i is at most $1 + sw_i$: one when τ_i is released plus sw_i after it starts executing. If we consider all the higher priority tasks in $hp(k)$, the lemma simply follows. □

To determine the number of additional priority inversions experienced by τ_k under the eager approach, after it starts executing, the following lemma identifies the conditions under which this situation occurs.

Lemma 4. *Under the limited preemptive eager approach, a DAG task τ_k that already started executing, may experience additional priority inversions (and then lower-priority blocking times) only if all the following conditions are simultaneously satisfied:*

1. *τ_k encounters a preemption point.*
2. *A higher priority task is released or τ_k requires additional cores to fork nodes.*
3. *There are lower priority tasks being executed.*

Proof. Condition (1) guarantees that, following the limited preemptive scheduling model, a task cannot be preempted within the execution of a node (NPR). Condition (2) follows from the observation that a task cannot be preempted by a lower priority instance; therefore, in order for τ_k to experience blocking from other lower priority running instances, it must either be preempted by a higher priority task or require additional cores to fork new nodes. Condition (3) is trivially derived by noticing that no lower-priority blocking may be experienced without lower priority instances being executed. □

Lemma 4 allows to upper-bound the number of additional priority inversions for the eager approach as follows.

Lemma 5. *Under the limited preemptive eager approach, in any time interval of length t, an upper bound on the number of priority inversions that a DAG task τ_k may additionally experience after starting its execution is*

$$p_k^{eager}(t) = min\Big(q_k, sw_k + h_k(t), \sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t + R_i^{ub}}{T_i} \right\rceil \times |V_i|\Big) \tag{5.6}$$

Proof. Condition (1) in Lemma 4 ensures that the number of additional priority inversions cannot exceed the number of potential preemption points of the task τ_k, i.e., q_k. Condition (2) provides an upper bound given by the number of preemption requests from higher priority instances during a time interval of length t i.e., $h_k(t)$ (see Equation 5.5) plus the number of additional core requests by τ_k in fork operations sw_k. Finally, condition (3) allows deriving one last upper bound given by the number of nodes of the lower priority tasks that may be released within the considered scheduling window of length t $\left(\sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t+R_i^{ub}}{T_i} \right\rceil \times |V_i|\right)$. Given that the three conditions must be satisfied, a minimum operation between the three bounds provides the final upper bound. □

5.3.2 Blocking time

Under the eager approach, the worst case scenario that must be considered to compute the lower-priority blocking time factors is:

- $\Delta^{eager}_{k,m}$, when τ_k is released, all the m cores are running lower priority tasks.
- $\Delta^{eager}_{k,m-1}$, after τ_k starts executing, when a priority inversion occurs, $m-1$ cores are running lower priority tasks (since τ_k is running in the other core).

In both cases, the first lower priority task $\tau_i \in lp(k)$ reaching a preemption point will be preempted. Therefore, the lower-priority blocking time factors $\Delta^{eager}_{k,m}$ and $\Delta^{eager}_{k,m-1}$ must consider the m and $m-1$ longest (with higher WCET) nodes, respectively.

Thekkilakattil et al. [87] compute these factors for task-sets composed of sequential tasks, considering first the set of the longest nodes of each lower priority task and then, the sum of the m or $m-1$ longest nodes of this set:

$$\begin{aligned}\Delta^{eager}_{k,m} &= \sum \max_{\tau_i \in lp(k)}^{m} \left(\max_{1 \le j \le |V_i|} C_{i,j} \right) \\ \Delta^{eager}_{k,m-1} &= \sum \max_{\tau_i \in lp(k)}^{m-1} \left(\max_{1 \le j \le |V_i|} C_{i,j} \right)\end{aligned} \tag{5.7}$$

where $\max_{1 \le j \le |V_i|} C_{i,j}$ denotes the longest node of task τ_i and $\sum \max^{m}_{\tau_i \in lp(k)}$ and $\sum \max^{m-1}_{\tau_i \in lp(k)}$ denote the sum of the m and $m-1$ longest values (of $\max_{1 \le j \le |V_i|} C_{i,j}$) among all tasks $\tau_i \in lp(k)$, respectively.

However, this analysis does not hold for task-sets composed of DAG tasks, because multiple nodes from the same task can execute in parallel. Therefore, there may be two nodes of the same task with longer WCET than two nodes from different tasks. Next subsections present two methods to compute the lower-priority blocking time in DAG-based task-sets.

5.3.2.1 Blocking time impact of the longest nodes

The easiest way of deriving the lower-priority blocking time factors is to account for the m and $m-1$ longest nodes among all the lower priority tasks:

$$\begin{aligned}\Delta^{eager}_{k,m} &= \sum \max_{\tau_i \in lp(k)}^{m} \left(\max_{1 \le j \le |V_i|}^{m} C_{i,j} \right) \\ \Delta^{eager}_{k,m-1} &= \sum \max_{\tau_i \in lp(k)}^{m-1} \left(\max_{1 \le j \le |V_i|}^{m-1} C_{i,j} \right)\end{aligned} \tag{5.8}$$

where $\max^{m}_{1 \le j \le |V_i|} C_{i,j}$ and $\max^{m-1}_{1 \le j \le |V_i|} C_{i,j}$ denote the m and $m-1$ longest nodes of task τ_i, respectively, and $\sum \max^{m}_{\tau_i \in lp(k)}$ and $\sum \max^{m-1}_{\tau_i \in lp(k)}$ denote the sum of the m and $m-1$ longest nodes among all tasks $\tau_i \in lp(k)$, respectively

Despite its simplicity, this strategy is pessimistic because it considers that the

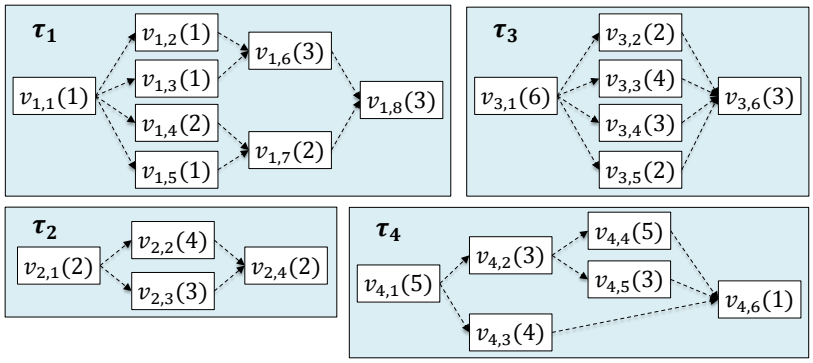


Figure 5.5: Example of a set of lower priority DAG tasks, $lp(k)$. Each node is labeled with its WCET $C_{i,j}$ in parenthesis.

longest m and $m-1$ nodes among all the lower priority tasks can execute in parallel, regardless of the precedence constraints defined in the DAG.

5.3.2.2 Blocking time impact of the longest parallel nodes

The edges (precedence constraints) in the DAG determine the maximum level of parallelism a task may exploit on m cores, which in turn determines the amount of blocking time impacting on higher priority tasks. This information must therefore be incorporated in the analysis to better upper-bound the lower-priority interference. To do so, we propose a new analysis method that incorporates the precedence constraints among nodes, as defined by the edges in the DAG, to compute the blocking time.

Given a task τ_k, our analysis derives the blocking time of $lp(k)$ over τ_k by computing new $\Delta^{eager}_{k,m}$ and $\Delta^{eager}_{k,m-1}$ factors in a three-step process: (1) identify the *worst-case workload* of each task in $lp(k)$ when executing on 1 to m cores; (2) compute the *overall worst-case workload* of $lp(k)$ for all possible *execution scenarios*; and (3) select the scenario that maximizes the blocking time.

In order to facilitate the explanation of the three steps, consider the Figure 5.5 which shows an example of a set $lp(k)$, composed of four DAG tasks $\{\tau_1, \tau_2, \tau_3, \tau_4\}$, executed on a $m = 4$ core platform. The nodes (NPRs) of $\tau_i \in lp(k)$ are labeled as $v_{i,j}$, with their WCET ($C_{i,j}$) in parenthesis.

Step 1. Identify the *worst-case workload* for all the tasks in $lp(k)$ when executing on 1 to m cores.

Definition 10. ***Worst-case workload.*** *The worst-case workload $\mu_i[c]$ of a task τ_i executing on c cores is the sum of the WCET of the c longest nodes that can execute*

c	$\boldsymbol{\mu_1}$[c]	$\boldsymbol{\mu_2}$[c]	$\boldsymbol{\mu_3}$[c]	$\boldsymbol{\mu_4}$[c]
1	$C_{1,6}$ or $C_{1,8} = 3$	$C_{2,2} = 4$	$C_{3,1} = 6$	$C_{4,1}$ or $C_{4,4} = 5$
2	$C_{1,6} + C_{1,7} = 5$	$C_{2,2} + C_{2,3} = 7$	$C_{3,3} + C_{3,4} = 7$	$C_{4,4} + C_{4,3} = 9$
3	$C_{1,6} + C_{1,4} +$ $+C_{1,5} = 6$	0	$C_{3,3} + C_{3,4} + C_{3,2} = 9$ or $C_{3,3} + C_{3,4} + C_{3,5} = 9$	$C_{4,4} + C_{4,3} +$ $+C_{4,5} = 12$
4	$C_{1,2} + C_{1,3} +$ $+C_{1,4} + C_{1,5} = 5$	0	$C_{3,2} + C_{3,3} + C_{3,4} +$ $+C_{3,5} = 11$	0

Table 5.1: Worst-case workload $\mu_i[c]$ of each task $\tau_i, i = \{1 \dots 4\}$ shown in Figure 5.5, when executing on $c = \{1, \dots, m\}$ cores.

in parallel.

This step computes an array $\mu_i[c]$, $c = \{1, \dots, m\}$, for all the tasks $\tau_i \in lp(k)$. Each element $\mu_i[c]$ is computed as follows:

$$\mu_i[c] = \sum \max_{c}^{parallel} \{C_{i,j}\} \tag{5.9}$$

where $\sum \max_c^{parallel}$ is the sum of the c longest nodes of τ_i that can execute in parallel, maximizing the interference when using c cores. To this aim, the sum must consider the edges of τ_i's DAG to determine which nodes can actually execute in parallel.

Table 5.1 shows the array $\mu_i[c]$ for each of the tasks τ_i shown in Figure 5.5 for $c = \{1, \dots, m\}$. For example, the worst-case workload occurs when nodes $v_{4,3}$ and $v_{4,4}$ execute in parallel, with an overall impact of $\mu_4[2] = 9$ time units. τ_2 has a maximum parallelism of 2, so $\mu_2[3] = 0$ and $\mu_2[4] = 0$.

Appendix A.1 presents an algorithm and the Integer Linear Programming (ILP) formulation to derive $\mu_i[c]$. The algorithm computes, for each node of τ_i, the set of nodes from the same task that can potentially execute in parallel with it (quadratic complexity). The ILP formulation takes this information as input and computes $\mu_i[c]$, i.e., the combination of c parallel nodes of τ_i that provides the worst-case sum of their WCET.

Step 2. Compute the *overall worst-case workload* of $lp(k)$ for all possible *execution scenarios.*

Definition 11. ***Execution scenarios.*** *The set of different execution scenarios* $e^m = \{s_1, s_2, ...s_{p(m)}\}$ *for a given number of cores* m *is given by the integer partitions*

$\mathbf{s_p \in e^4}$	$\mathbf{\lvert s_p \rvert}$	**Execution scenario description**
$s_1 = \{1, 1, 1, 1\}$	4	4 tasks, each running on 1 core
$s_2 = \{2, 2\}$	2	2 tasks, each running on 2 cores
$s_3 = \{2, 1, 1\}$	3	1 task running on 2 cores, and 2 tasks on 1 core each
$s_4 = \{3, 1\}$	2	1 task running on 3 cores, and 1 task on 1 core
$s_5 = \{4\}$	1	1 task running on 4 cores

Table 5.2: Set of execution scenarios $e^4 = \{s_1, s_2, s_3, s_4, s_5\}$.

of m, being $p(m) = |e^m|$ the total number of integer partitions[3]*. A given execution scenario $s_l \in e^m, l = 1 \dots p(m)$ represents the number of tasks $|s_l|$ running on the m cores and how the cores are occupied by these tasks.*

Given that m represents the number of cores, it corresponds to a relatively small integer. Therefore, despite its complexity, we can find efficient ways to compute both e^m and $p(m)$ in the literature [88]. Table 5.2 shows the set of execution scenarios assuming $m = 4$ cores, i.e., e^4. The total number of execution scenarios is $p(4) = 5$.

Definition 12. ***Overall worst-case workload.*** *The overall worst-case workload $\rho_k[s_l]$ of a set of tasks $lp(k)$ executing on m cores, is the maximum time used for executing this set according to a given execution scenario $s_l \in e^m$.*

This step computes all the execution scenarios $s_l \in e^m$ and then, the overall worst-case workload $\rho_k[s_l]$ for each execution scenario. Each element $\rho_k[s_l]$ is computed as follows:

$$\rho_k[s_l] = \sum \max_{|s_l|}^{s_l} \{\mu_i\} \tag{5.10}$$

where $\sum \max_{|s_l|}^{s_l}$ is the sum of the $|s_l|$ values of μ_i that fits in the scenario s_l and maximizes $\rho_k[s_l]$.

Table 5.3 shows the array $\rho_k[s_l]$ for each execution scenario $s_l \in e^m, l = 1 \dots p(m)$ presented in Table 5.2. It considers the array $\mu_i[c]$ shown in Table 5.1. For instance, the overall worst-case workload of s_3, $\rho_k[s_3] = 19$ is given when τ_4 executes on 2 cores ($\mu_4[2] = 9$), and τ_2 and τ_3 execute on 1 core each ($\mu_2[1] = 4$ and $\mu_3[1] = 6$).

[3] In number theory and combinatorics, the integer partition of a positive integer m is the way of writing m as a sum of positive integers. Two sums that differ only in the order of their summands are considered the same partition. The total number of integer partitions $p(m)$ can be computed with the pentagonal number theorem from the Euler's formulation: $p(m) = \sum_q (-1)^{q-1} p(m - q(3q-1)/2)$, where the sum is over all nonzero integers q (positive and negative)[88].

$\mathbf{s_l}$	$\boldsymbol{\rho_k[s_l]}$
$s_1 = \{1, 1, 1, 1\}$	$\mu_1[1] + \mu_2[1] + \mu_3[1] + \mu_4[1] = 18$
$s_2 = \{2, 2\}$	$\mu_2[2]$ *or* $\mu_3[2] + \mu_4[2] = 16$
$s_3 = \{2, 1, 1\}$	$\mu_4[2] + \mu_2[1] + \mu_3[1] = 19$
$s_4 = \{3, 1\}$	$\mu_4[3] + \mu_3[1] = 18$
$s_5 = \{4\}$	$\mu_3[4] = 11$

Table 5.3: Overall worst-case workload $\rho_k[s_l]$ of tasks within the set $lp(k)$ for each of the scenarios $s_l \in e^4$.

Appendix A.2 presents an ILP formulation to compute $\rho_k[s_l]$ as given by Equation 5.10. It takes as input the worst-case workload $\mu_i[c]$, $c = \{1, \ldots, m\}$, for all the tasks $\tau_i \in lp(k)$, as computed in the previous section.

Step 3. Select the scenario that maximizes the lower-priority blocking time.

Finally, given the overall worst-case workload $\rho_k[s_l]$ for each scenario $s_l \in e^m$, the lower-priority blocking factors for a given task τ_k can be computed as the maximum overall worst-case workload among all scenarios:

$$\begin{aligned} \Delta_{k,m}^{eager} &= \max_{s_l \in e^m} \rho_k[s_l] \\ \Delta_{k,m-1}^{eager} &= \max_{s_l \in e^{m-1}} \rho_k[s_l] \end{aligned} \tag{5.11}$$

where $\max_{s_l \in e^m}$ and $\max_{s_l \in e^{m-1}}$ provide the maximum overall worst-case workload among all the executing scenarios in e^m and e^{m-1}, respectively.

5.3.2.3 Comparing the eager blocking time factors

Given the set $lp(k)$ shown in Figure 5.5, Table 5.4 presents the lower-priority blocking time factors $\Delta_{k,m}^{eager}$ and $\Delta_{k,m-1}^{eager}$, using the pessimistic but easy-to-compute approach presented in Section 5.3.2.1 (named LP-eager-max), and the optimal but computationally intensive approach presented in Section 5.3.2.2 (named LP-eager-ilp).

On one side, $\Delta_{k,m}^{eager}$ for LP-eager-max is given by the sum of the m longest nodes among all lower priority tasks, i.e., $\Delta_{k,4}^{eager} = C_{3,1} + C_{4,1} + C_{4,4} + C_{2,2} = 20$. By contrast, $\Delta_{k,m}^{eager}$ for LP-eager-ilp is given by the maximum $\rho_k[s_l]$ from Table 5.3, i.e., $\Delta_{k,4}^{eager} = \max_{s_l \in e^4} \rho_k[s_l] = max(\rho_k[s_1], ...\rho_k[s_5]) = 19$. The pessimism added by the LP-eager-max eager approach comes from the fact that nodes $v_{4,1}$ and $v_{4,4}$ cannot

	Equation 5.8 (LP-eager-max)	Equation 5.11 (LP-eager-ilp)
$\Delta_{k,4}^{eager}$	20	19
$\Delta_{k,3}^{eager}$	16	15

Table 5.4: Lower-priority blocking factors for a given task τ_k.

be executed in parallel. Similarly, $\Delta_{k,3}^{eager} = 16$ according to LP-eager-max, while $\Delta_{k,3}^{eager} = 15$ according to LP-eager-ilp.

Overall, for this small task-set, the deeper analysis provided by the LP-eager-ilp approach computes a tighter estimation of the lower-priority blocking factors. This comparison is further analyzed in the evaluation Section 5.5.

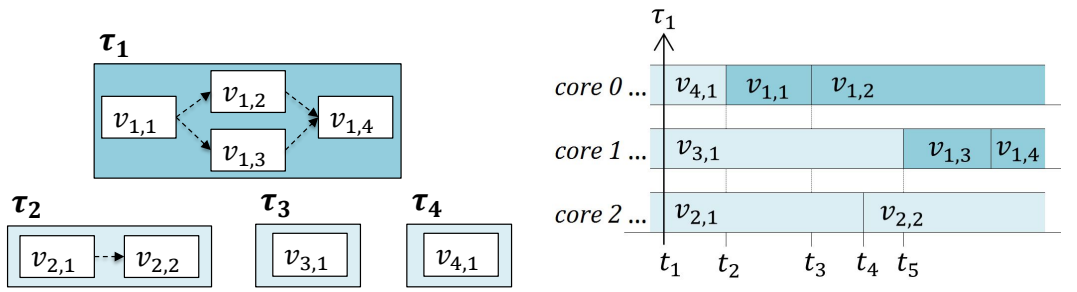
5.4 Lazy Preemption Analysis

This section presents how to compute the number of priority inversions p_k, and the lower-priority blocking times $\Delta_{k,m}$ and $\Delta_{k,m-1}$, suffered by a task τ_k under the lazy limited preemptive scheduling.

5.4.1 Number of priority inversions

Under the lazy approach, preemption is delayed until the lowest priority running task reaches a preemption point. That is, when a highest priority task is ready to execute (or requests additional cores) if there are several lower priority tasks running, the lowest priority task τ_k is preempted when it reaches a preemption point. As a result, τ_k can suffer lower-priority interference, i.e., a priority inversion may occur, before starting its execution. At later preemption points τ_k does not suffer additional priority inversions (because in case of being preempted, τ_k is the lowest priority task as there is no lower priority running tasks to interfere) unless additional cores are required to fork nodes.

Figure 5.6 shows an example of the lazy approach when considering a DAG-based task-set composed of four tasks τ_1, τ_2, τ_3 and τ_4, in decreasing priority order (Figure 5.6(a)), scheduled on $m = 3$ cores (Figure 5.6(b)). We assume that tasks τ_2, τ_3 and τ_4 are executing its first node when the highest priority task τ_1 is released at time t_1. Under the lazy approach, τ_1 starts executing the first node $v_{1,1}$ when the lowest priority running task τ_4 reaches a preemption point at time t_2. At time t_3, nodes $v_{1,2}$



(a) Task-set composed of four DAG tasks. **(b)** Scheduling on a 3-core processor.

Figure 5.6: Scheduling of a DAG-based task-set under the lazy approach.

and $v_{1,3}$ are ready to start executing, but only the core in which $v_{1,1}$ is being executed is available to start executing $v_{1,2}$. As a result, the lower priority tasks τ_2 and τ_3 block the execution of the node $v_{1,3}$. The reason is that, at time instant t_3, τ_1 forks two parallel nodes, requesting one additional core. At time instant t_4, τ_2 reaches a preemption point, but since it is not the lowest priority task running, it continues the execution. Meanwhile, node $v_{1,3}$ of τ_1 is still blocked, until τ_3 reaches a preemption point at time instant t_5. Overall, when considering DAG-based task-sets, a task τ_k may suffer priority inversion not only on the first node before it starts executing, but also when requesting additional cores to execute the rest of nodes.

To determine the number of additional priority inversions experienced by τ_k under the lazy approach, after it starts executing, the following lemma identifies the conditions under which this situation occurs.

Lemma 6. *Under the limited preemptive lazy approach, a DAG task τ_k, that already started executing, may experience additional priority inversions (and then lower-priority blocking times) only if all the following conditions are simultaneously satisfied:*

1. *τ_k encounters a preemption point.*
2. *τ_k requires additional cores to fork nodes.*
3. *There are lower priority tasks being executed.*

Proof. Similarly to Lemma 4, condition (1) guarantees that, following the limited preemptive scheduling model, a task cannot be preempted within the execution of a node. Condition (2) follows from the observation that τ_k can only be preempted by a higher priority task, but if it is the case, τ_k is the lowest priority running task, and then there are not other lower priority instances running and blocking τ_k. However, if τ_k requires additional cores to fork new nodes, all cores may be occupied by lower

priority tasks blocking τ_k. Condition (3) is trivially derived by noticing that no lower-priority blocking may be experienced without lower priority instances being executed. □

Lemma 6 allows to upper-bound the number of additional priority inversions for the lazy approach as follows:

Lemma 7. *Under the limited preemptive lazy approach, in any time interval of length t, an upper bound on the number of priority inversions that a DAG task τ_k may additionally experience after starting its execution is*

$$p_k^{lazy}(t) = min\Big(sw_k, \sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t + R_i^{ub}}{T_i} \right\rceil \times |V_i|\Big) \tag{5.12}$$

Proof. Condition (2) in Lemma 6 provides an upper bound on the number of additional priority inversions given by the number of additional cores requests sw_k (see Definition 9). Condition (3) provides an upper bound given by the number of nodes of the lower priority tasks that may be released within the considered scheduling window of length t $\left(\sum_{\forall \tau_i \in lp(k)} \left\lceil \frac{t+R_i^{ub}}{T_i} \right\rceil \times |V_i|\right)$. Condition (1) trivially follows since the number of potential preemption points is greater than the number of additional core requests, i.e., $q_k > sw_k$. To demonstrate it, consider a DAG task with a node that forks m different nodes. In this case, $q_k = m$ while $sw_k = m - 1$ as the core executing the first node can execute one of the forked nodes. Given that the three conditions must be satisfied, a minimum operation between the two bounds provides the final upper bound. □

5.4.2 Blocking time

Marinho et al. [89] estimated the worst-case blocking time due to lower priority tasks when considering a system composed of sequential tasks, under the lazy preemption strategy.

Figure 5.7 illustrates an example of the worst-case blocking time scenario generated by lower priority tasks. Concretely, it considers a task-set composed of eight sequential tasks $\tau_1, ..., \tau_8$ (in decreasing priority order) running on $m = 4$ cores. Assume that lower priority tasks τ_5, τ_6, τ_7 and τ_8 are already executing on the processor, when the higher priority tasks τ_1, τ_2, τ_3 and τ_4 are simultaneously released at time instant t_1. The first task to be preempted is τ_8 (the lowest priority task) at time

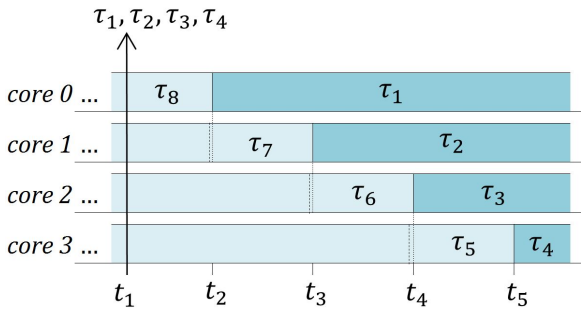


Figure 5.7: Worst-case lower-priority blocking suffered by τ_4 under the lazy approach (sequential tasks).

instant t_2, when a preemption point is reached, and so the highest priority ready task τ_1 can start its execution. In the worst case scenario, task τ_7 reaches a preemption point at time instant $t_2 - \varepsilon$ in which the lowest priority task τ_8 is still executing. Therefore, τ_7 continues executing (an so blocking τ_2) until its next preemption point is reached at time instant t_3, when τ_2 can start executing. Subsequently, in the worst-case situation, τ_6 reaches a preemption point, just before the preemption point of τ_7 is reached at time instant t_3, blocking τ_3 until time instant t_4. Finally, τ_4 is able to start its execution at time t_5. Overall, the worst-case blocking time that task τ_4 can suffer is equal to $(t_2 - t_1) \times 4 + (t_3 - t_2) \times 3 + (t_4 - t_3) \times 2 + (t_5 - t_4) \times 1$. In general, an upper bound of the maximum blocking time is computed by adding the longest node of the set $lp(k)$ multiplied by m, the second longest node from the $lp(k)$ multiplied by $m-1$, the third longest node from the $lp(k)$ multiplied by $m-2$, and so on, until the m-th longest node from $lp(k)$ is considered.

This scenario can be directly applied to this work. Under the lazy approach, the worst case scenario that must be considered to compute the lower-priority blocking time factors is:

- $\Delta^{lazy}_{k,m}$, when τ_k is released, all the m cores are running lower priority tasks and $m-1$ higher priority tasks are ready but not running. The worst case scenario explained in Figure 5.7 applies to this situation.
- $\Delta^{lazy}_{k,m-1}$, after τ_k starts executing, when a priority inversion occurs, $m-1$ cores are running lower priority tasks (since τ_k is running in the other core) and $m-2$ higher priority tasks are ready but not running. The worst-case scenario explained in Figure 5.7 applies to this situation, but considering $m-1$ instead of m cores.

Therefore, the lower-priority blocking time factors can be computed as Marinho

et al. did (ADS blocking estimation 2 [89]):

$$\Delta_{k,m}^{lazy} = \sum_{l=1}^{m} Q_k^l \times (m - l + 1)$$
$$\Delta_{k,m-1}^{lazy} = \sum_{l=1}^{m-1} Q_k^l \times (m - l) \tag{5.13}$$

where Q_k^l denotes the l^{th} *longest node* of all the nodes of all the tasks in the set $lp(k)$. $\Delta_{k,m}^{lazy}$ estimates the lower-priority blocking time by considering that the worst-case interference is obtained when the longest node, of all the nodes, of all the tasks in the set $lp(k)$ is accounted for m times (assuming the lowest priority for this task), the second longest node is added up $m-1$ times (assuming the second lowest priority for this task), etc., until the m^{th} longest node is reached, which is only considered once. $\Delta_{k,m-1}^{lazy}$ is similarly computed until the $(m-1)^{th}$ longest node is reached.

5.5 Experimental Results

This section evaluates the response time analysis of the limited preemptive scheduling approach presented in this chapter, for both strategies, eager and lazy. The evaluation considers the following metrics:

1. Schedulability ratio, i.e., a percentage of schedulable task-sets, when varying the overall system utilization and number of tasks.
2. Number of priority inversions considered by the response time analysis, as computed in Equations 5.6 and 5.12.
3. Number of preemptions that actually occur when deploying the system using a scheduling simulator.
4. Impact of the interference and blocking times from higher priority and lower priority tasks, respectively, over the response time.

Concretely, we evaluate the response time analysis of the lazy strategy, labeled as *LP-lazy*, and the eager strategy, for which two methods have been presented to compute the blocking time, labeled as *LP-eager-max* and *LP-eager-ilp*. Moreover, the limited preemptive scheduling strategies are compared against an ideal fully-preemptive scheduling (labeled as *FP-ideal*). In fully-preemptive scheduling, the impact of lower

priority tasks is null ($I_k^{lp} = 0$). Therefore, the fully-preemptive response time analysis always performs better than the limited preemptive scheduling. However, it is important to remark that the performance of a real fully-preemptive approach in which the preemption overheads is included in the analysis may significantly decrease compared to limited preemptive scheduling. Accurately accounting for preemption overheads in fully-preemptive is very difficult (if not impossible) since the execution of each task can be preempted at any time instant. Preemption overheads in the case of limited preemptive scheduling have neither been considered. Nevertheless, a safe upper bound could be easily computed by multiplying the maximum number of preemptions a task may suffer q_k by the maximum time required for a context switch.

As explained in Section 5.2.1, the response time upper bound of a task-set is computed starting from the highest priority τ_1 to the lowest priority task τ_n. Therefore, given a task τ_k, the response time upper bound of its lower priority tasks $R_i^{ub}, \forall \tau_i \in lp(k)$, is not computed yet when needed for the computation of p_k^{eager} or p_k^{lazy} (see equations 5.6 and 5.12). As a result, we consider D_i as a safe upper bound of R_i^{ub} (if D_i is greater than R_i^{ub} then the task-set is not schedulable) when computing p_k^{eager} or p_k^{lazy}.

5.5.1 Experimental setup

All the experiments presented in this section consider the algorithms to randomly generate task-sets composed of DAG task, presented in 2.4. The concrete values used for the DAG tasks generation are:

- Probabilities of a branch to be expanded to a single node or to a parallel sub-graph, $p_{term} = 0.4$ and $p_{par} = 0.6$, respectively.
- Probabilities of adding extra edges, $p_{dep} = 0.1$.
- Maximum number of branches of a parallel sub-graph, $max_{par} = 6$.
- Maximum recursion depth, $max_{depth} = 3$.
- The WCET of each node varies in the interval $[C^{min}, C^{max}] = [1, 100]$.

Given the complexity of the technique presented in Section 5.3.2.2, we consider two different experiment sizes, in terms of number of DAG tasks within each task-set (n) and number of nodes of each DAG task (max_{nodes}):

- *Small* DAG task-sets: $max_{nodes} = 30$ and $n < 10$.

- *Large* DAG task-sets: $max_{nodes} = 50$ and $n < 50$.

The evaluation is carried out for different numbers of cores, concretely, for $m = 2$, 4, 8 and 16. Finally, for each experiment, we generate 500 DAG task-sets and consider the implicit deadline case ($D_k = T_k$).

The schedulability analysis for all the scheduling approaches, and the algorithm 2 presented in Section 5.2.2.2 have been implemented in MATLAB®. The ILP formulations presented in Appendix A have been coded and solved with the IBM ILOG CPLEX Optimization Studio [90].

The evaluation of the response time analysis introduced in this chapter, considering real use-cases, is presented in Chapter 6.

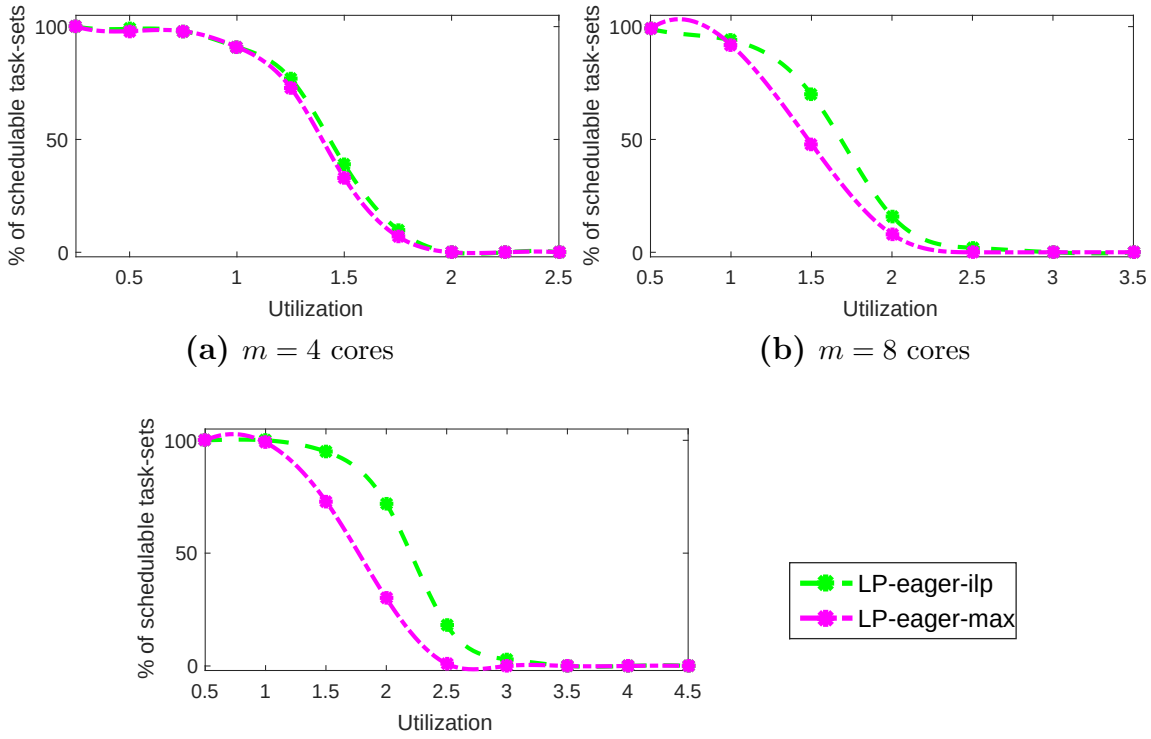
5.5.2 Schedulability analysis

This section evaluates the schedulability ratio given by the response time analysis presented in Section 5.2, and used for the fully-preemptive and the limited preemptive scheduling. In case of the fully-preemptive scheduling, the lower-priority interference is null, $I_k^{lp} = 0$. In case of the limited preemptive scheduling, the lower-priority interference I_k^{lp} is computed using Equation 5.4, for which the blocking time factors and the number of priority inversions are computed following Sections 5.3 and 5.4, for the eager and lazy approaches, respectively.

5.5.2.1 Evaluation of the two methods to compute the eager blocking time factors

This section evaluates the two methods proposed to compute the impact of lower priority tasks when considering the eager strategy. These two methods for computing the blocking time are LP-eager-max and LP-eager-ilp, presented in Sections 5.3.2.1 and 5.3.2.2, respectively.

Figure 5.8 shows the percentage of schedulable task-sets, composed of small DAG tasks, i.e., *Small* DAG task-sets, when varying the total system utilization $U_{\mathcal{T}}$, for $m = 4$, $m = 8$ and $m = 16$ cores. In all cases, LP-eager-ilp and LP-eager-max perform very similar, decreasing the schedulability ratio as the utilization of the system increases. The difference between both techniques is that LP-eager-max considers as lower-priority interference the nodes with maximum WCET, but that may not execute in parallel. LP-eager-ilp instead, selects only the nodes that can actually execute



(a) $m = 4$ cores

(b) $m = 8$ cores

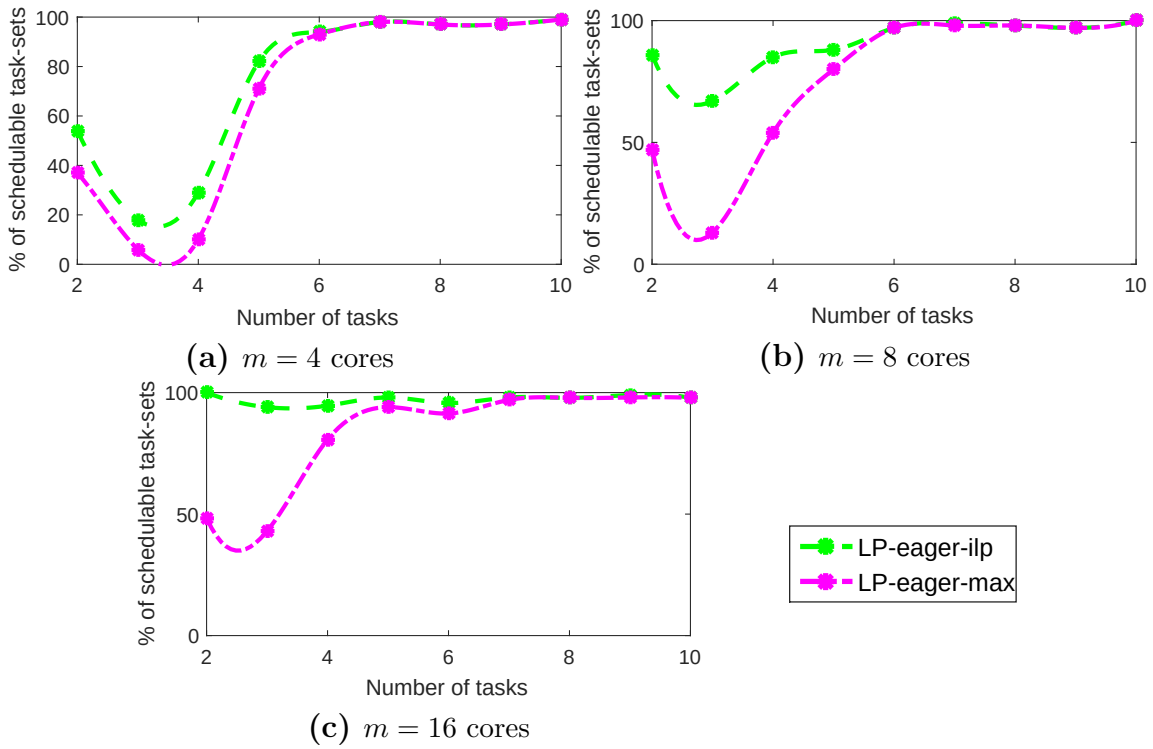
(c) $m = 16$ cores

Figure 5.8: Percentage of schedulable *Small* DAG task-sets as a function of $U_{\mathcal{T}}$.

in parallel.

Figure 5.8a shows the case in which $m = 4$ cores are considered, ranging the utilization $U_{\mathcal{T}}$ from 0.25 to 2.5. Both approaches are able to schedule nearly all the task-sets until the utilization reaches 1. From this point on, the performance of LP-eager-ilp and LP-eager-max drop, e.g., when $U_{\mathcal{T}} = 1.5$, the schedulability ratio is 39% and 33% for LP-eager-ilp and LP-eager-max, respectively. The schedulability ratio is 0% when the utilization is equal to 2. Figure 5.8b shows the schedulability ratio when $m = 8$ cores, ranging $U_{\mathcal{T}}$ from 0.5 to 3.5. Assuming $U_{\mathcal{T}} = 1.5$, the schedulability ratio is 70% and 48% for LP-eager-ilp and LP-eager-max, respectively. Finally, Figure 5.8c shows the schedulability ratio when $m = 16$ cores, ranging $U_{\mathcal{T}}$ from 0.5 to 4.5. In this case, the trend is maintained; when $U_{\mathcal{T}} = 2$, the schedulability ratio is 72% and 30% for LP-eager-ilp and LP-eager-max, respectively. As the number of cores increases, the difference between LP-eager-ilp and LP-eager-max is higher because a higher number of nodes must be selected to compute the interference, being LP-eager-ilp more accurate in the selection.

In order to have a better understanding of the response time analysis performance,



(a) $m = 4$ cores **(b)** $m = 8$ cores **(c)** $m = 16$ cores

Figure 5.9: Percentage of schedulable *Small* DAG task-sets as a function of the number of tasks $n \in [2, 10]$.

given a fixed value of the system utilization, $U_{\mathcal{T}} = 1.5$, Figure 5.9 presents the percentage of schedulable *Small* DAG task-sets when varying the total number of tasks n from 2 to 10, for $m = 4$ (Figure 5.9a), $m = 8$ (Figure 5.9b) and $m = 16$ (Figure 5.9c) cores. As shown before, for a given number of tasks, the schedulability rate increases as the number of cores increases. However, what is remarkable in this figures is that the performance of the response time analysis increases as the number of tasks increases, conforming to the intuition that scheduling a large number of light tasks (with low individual utilization) is easier than scheduling fewer heavy tasks (with high individual utilization). This is the case of both schedulability tests, LP-eager-max and LP-eager-ilp, that achieve the same rate, around 100% when $n \geq 6$.

ILP Complexity. Regarding the ILP complexity, we measure the execution time of the response time analysis of the LP-eager-ilp approach on an Intel(R) Xeon(TM) CPU 5148 at 2.33GHz. Table 5.5 shows the average execution time (in seconds) of the schedulability test of a task-set. We consider different task-sets and system configurations, as shown in Figure 5.9: varying the number of tasks $n \in [2, 10]$; varying the number of cores $m = 4$, 8 or 16; and assuming a system utilization $U_{\mathcal{T}} = 1.5$.

	n				
m	**2**	**4**	**6**	**8**	**10**
4	1.2528	5.6357	8.0497	13.535	18.971
8	13.348	45.477	56.504	76.652	95.452
16	23.703	124.55	166.16	249.42	453.00

Table 5.5: Average execution time (seconds) of the LP-eager-ilp schedulability test of a *Small* DAG task-set.

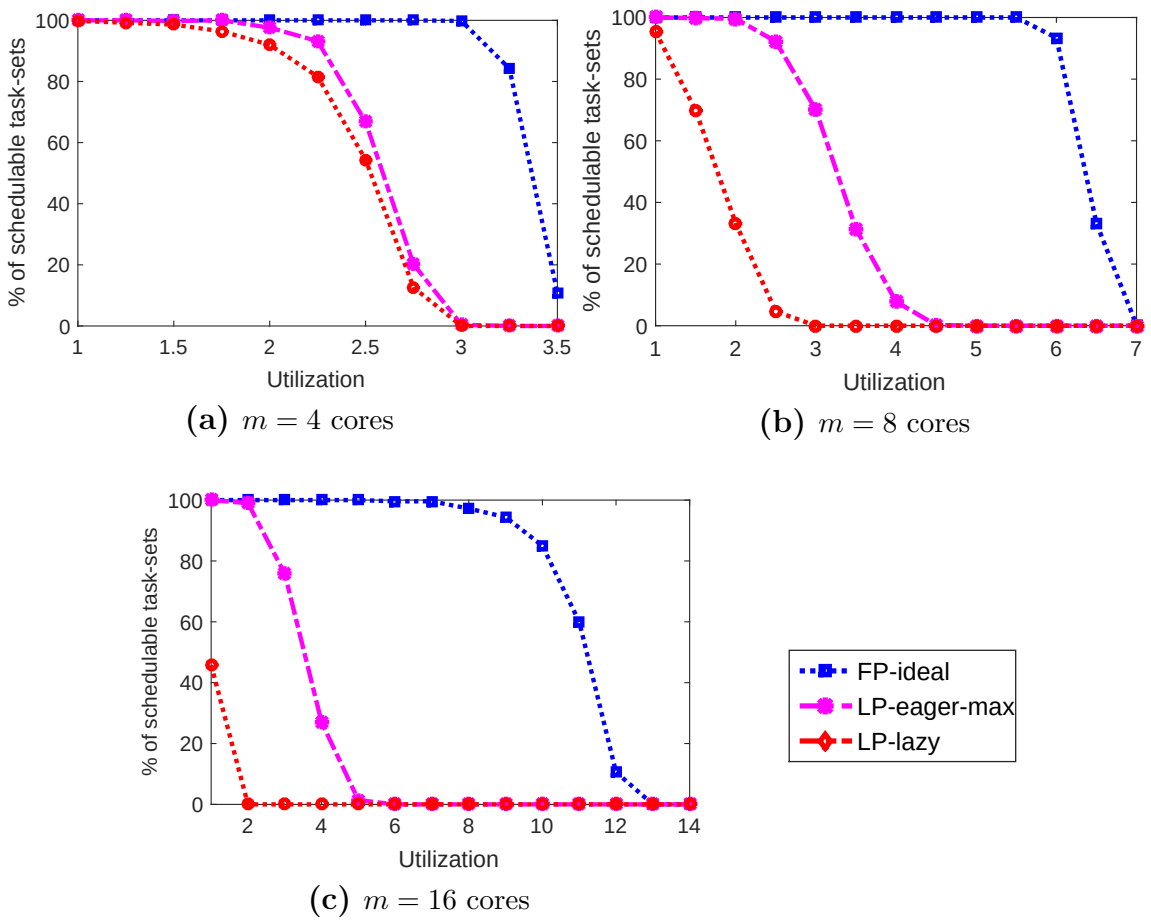
The ILP complexity makes the schedulability test of a task-set to take, in average, up to 18 seconds, 95 seconds and 7.5 minutes, when considering $m = 4$, $m = 8$ and $m = 16$ cores, respectively.

5.5.2.2 Comparing limited preemptive and fully-preemptive scheduling

This section evaluates and compares the two proposed limited preemptive approaches LP-eager-max and LP-lazy, and a fully-preemptive approach FP-ideal when considering *Large* DAG task-sets. Given the complexity of the ILP solver to compute the LP-eager-ilp solution, and the similar performance of LP-eager-max and LP-eager-ilp when the number of tasks increases, we only consider the LP-eager-max approach in the rest of this evaluation section.

Figure 5.10 shows the percentage of schedulable *Large* DAG task-sets, $n \in [30, 50]$, when varying the task-set utilization $U_\mathcal{T}$. Figure 5.10a shows the results for $m = 4$, ranging $U_\mathcal{T}$ from 1 to 3.5. When $U_\mathcal{T} = 2.25$, the percentage of schedulable task-sets is 100%, 93% and 81% for FP-ideal, LP-eager-max and LP-lazy, respectively. Figure 5.10b shows the results for $m = 8$, ranging $U_\mathcal{T}$ from 1 to 7. When $U_\mathcal{T} = 2$, the percentage of schedulable task-sets is 100%, 99% and 33% for FP-ideal, LP-eager-max and LP-lazy, respectively. Finally, Figure 5.10c shows the results for $m = 16$, ranging $U_\mathcal{T}$ from 1 to 14. When $U_\mathcal{T} = 2$, the percentage of schedulable task-sets is 100%, 99% and 0% for FP-ideal, LP-eager-max and LP-lazy, respectively.

As expected, the FP-ideal dominates the limited preemptive approaches since only the interference caused by higher priority tasks is considered in FP-ideal. Regarding the limited preemptive approaches, LP-eager-max dominates the LP-lazy approach. Interestingly, and contrary to the intuition, for a given $U_\mathcal{T}$, the LP-lazy approach achieves less schedulability rate as the number of cores increases. The reason is that the blocking factor for the lazy approach dominates the response time analysis adding



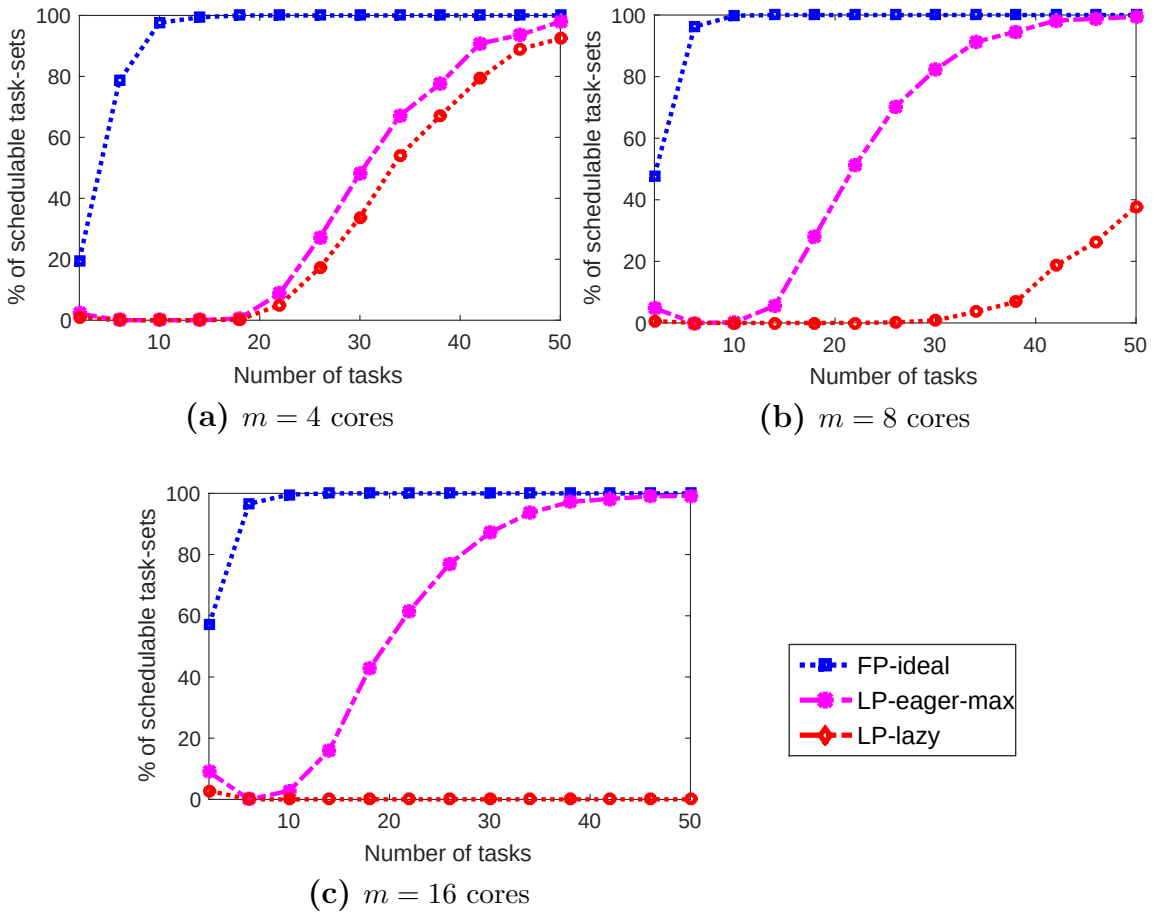
(a) $m = 4$ cores **(b)** $m = 8$ cores **(c)** $m = 16$ cores

Figure 5.10: Percentage of schedulable *Large* DAG task-sets as a function of $U_{\mathcal{T}}$.

huge pessimism, which increases with the number of cores. Next section describes this phenomenon in detail, by analyzing each of the factors needed to compute the lower-priority interference.

Given a fixed value for the system utilization $U_{\mathcal{T}} = 2.5$, Figure 5.11 shows the percentage of schedulable *Large* DAG task-sets when varying the number of tasks n from 2 to 50 (in steps of 4), and considering $m = 4$, $m = 8$ and $m = 16$ cores (Figures 5.11a, 5.11b and 5.11c, respectively).

Intuitively, the schedulability rate should increase as the number of tasks increases because, as shown in the previous section, scheduling a large number of light tasks is simpler than scheduling fewer heavy tasks. This trend is observed for the FP-ideal and LP-eager-max strategies, for which the schedulability ratio is around 100% for task-sets composed of 50 DAG tasks. However, for the LP-lazy strategy, the lower-priority interference, I_k^{lp}, hugely increases as the number of cores increases, resulting in a very pessimistic response-time analysis, in which no task-set can be scheduled,



(a) $m = 4$ cores **(b)** $m = 8$ cores **(c)** $m = 16$ cores

Figure 5.11: Percentage of schedulable *Large* DAG task-sets as a function of the number of tasks $n \in [2, 50]$.

even with an utilization of 2.5 in a 16-core processor. As an example, when the number of tasks is 30, the LP-lazy strategy is able to schedule 33%, 0.8% and 0% of task-sets, for $m = 4$, 8 and 16 cores, respectively. By contrast, LP-eager-max is able to schedule 48%, 82% and 87% of task-sets, for $m = 4$, 8 and 16 cores, respectively. This phenomenon is also detailed in the next section.

5.5.3 Impact of priority inversions and preemptions

This section evaluates the number of priority inversions p_k^{eager} and p_k^{lazy} considered by our response time analysis, and the actual number of preemptions occurring at system deployment. Moreover, the interference generated by higher priority and lower priority tasks, I_k^{hp} and I_k^{lp}, respectively, is also analyzed. All experiments consider *Large* DAG task-sets, with an overall task-set utilization of 2.5.

Figure 5.12 compares the number of additional priority inversions considered by

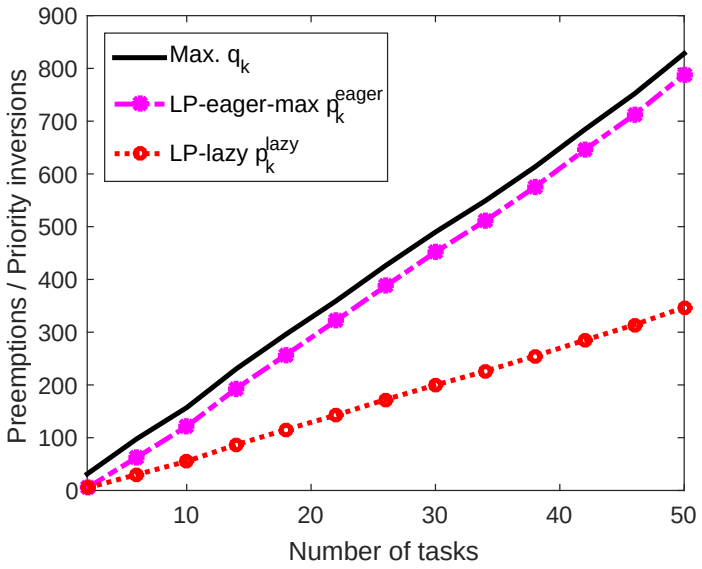
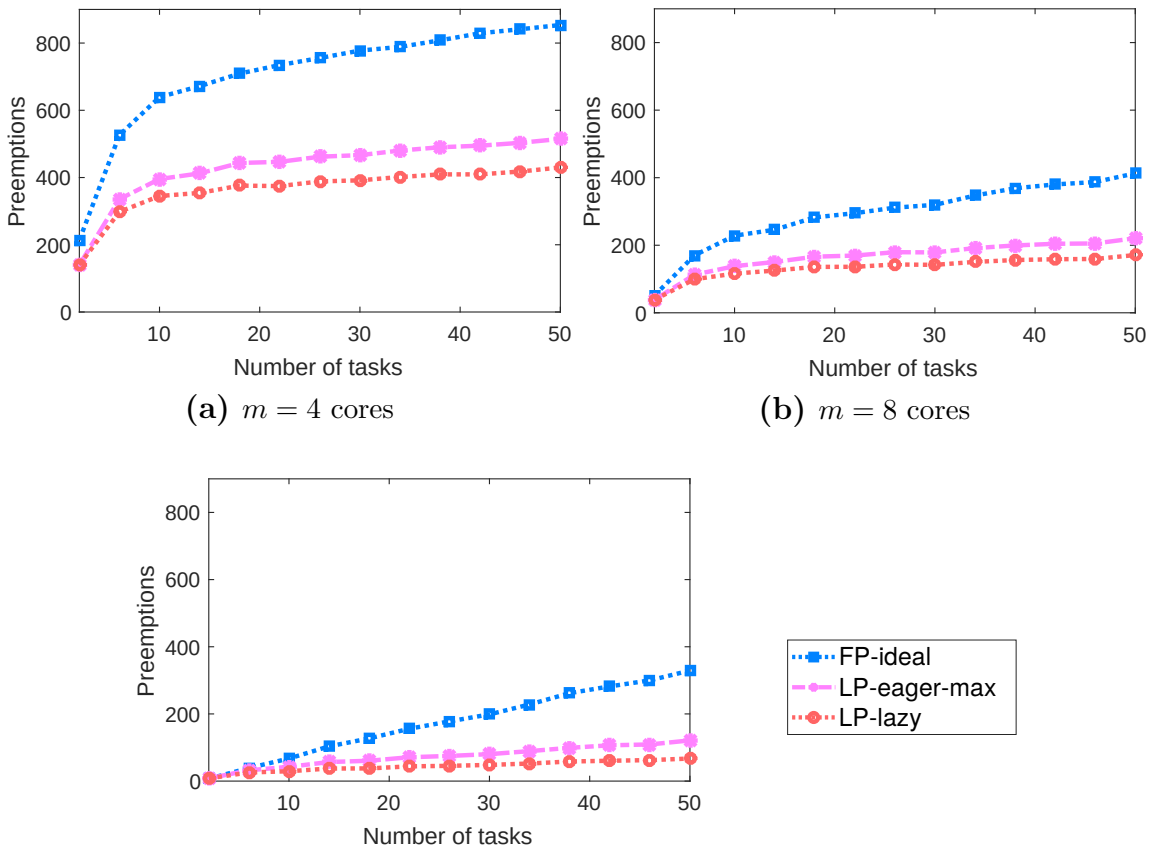


Figure 5.12: Number of additional priority inversions and maximum number of preemption points, as a function of the number of tasks $n \in [2, 50]$ (*Large* DAG task-sets).

the response time analysis of the LP-eager-max and LP-lazy strategies i.e., p_k^{eager} and p_k^{lazy}, computed by Equations 5.6 and 5.12, and the maximum number of preemptions a task may suffer, i.e., q_k, when varying the number of tasks from 2 to 50 (in steps of 4). Notice that q_k also represents the maximum number of additional priority inversions a task τ_k may suffer after it starts its execution.

As expected, Figure 5.12 confirms that the response time analysis of the eager approach considers a higher number of additional priority inversions than the lazy approach, being very close to the maximum number of preemptions. The reason is that in the eager approach, lower-priority blocking can come from (1) higher priority tasks preemptions at the end of each node while there are lower priority tasks running, and (2) the request of additional cores to fork new parallel nodes (see Lemma 4). Under the lazy approach instead, only *fork* operations can generate blocking from lower priority tasks (see Lemma 6). In fact, in most cases, p_k^{eager} is given by q_k except for (1) the highest priority task, for which $p_k^{eager} = sw_k$ because $h_k = 0$ and $sw_k < q_k$, and (2) the lowest priority task, for which $p_k^{eager} = 0$ since there are no lower priority tasks causing blocking. The impact of these two tasks is shown in Figure 5.12, in the small difference between q_k and p_k^{eager}. In most cases, p_k^{lazy} is given by sw_k, except for the lowest priority task, for which $p_k^{lazy} = 0$ for the same reason than in LP-eager-max.

Such a trend is also observed when simulating the execution of the *Large* DAG task-sets. Figure 5.13 shows the observed preemptions when executing the task-sets



(a) $m = 4$ cores **(b)** $m = 8$ cores

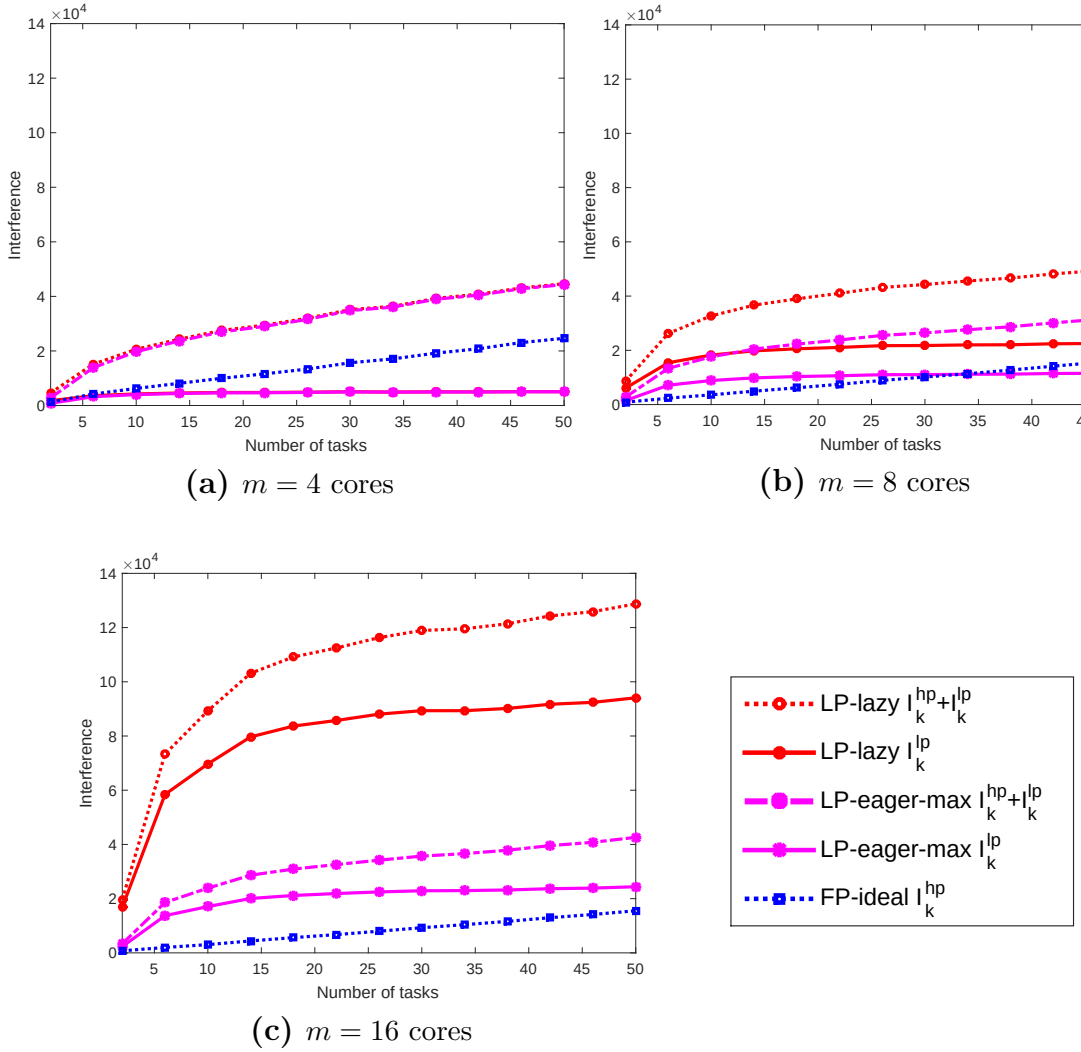
(c) $m = 16$ cores

Figure 5.13: Observed preemptions as a function of the number of tasks $n \in [2, 50]$ (*Large* DAG task-sets).

in a scheduling simulation running for 10^5 time units (in which task are released multiple times), varying the number of DAG tasks from 2 to 50 (in steps of 4), and considering $m = 4$, $m = 8$ and $m = 16$ cores (Figures 5.13a, 5.13b and 5.13c, respectively). In this case, a fully-preemptive scheduling strategy has been considered as well, for comparison purposes.

As expected, the limited preemptive eager approach generates more preemptions than the lazy approach. Clearly, the number of preemptions in both cases decreases as more cores are available for the same number of tasks. In case of the fully-preemptive scheduling strategy, the number of preemptions is much higher than the limited preemptive, since more scheduling opportunities exist (resulting in a higher schedulability rate, as shown in Section 5.5.2.2). However, this would (seriously) complicate the response time analysis if preemption overheads would be included, as the points at which tasks are preempted are unknown.

Despite LP-eager-max enforces a higher number of priority inversions compared



(a) $m = 4$ cores **(b)** $m = 8$ cores

(c) $m = 16$ cores

Figure 5.14: Average higher-priority and lower-priority interference as a function of the number of tasks $n \in [2, 50]$ (*Large* DAG task-sets).

to LP-lazy, as shown in Figures 5.12 and 5.13, LP-eager-max results in a better schedulability ratio in the response time analysis, as shown in Figure 5.10. The reason for this is explained in Figure 5.14, which shows the interference due to higher priority and lower priority tasks, when varying the number of tasks $n \in [2, 50]$, and considering $m = 4$, $m = 8$ and $m = 16$ cores (Figures 5.14a, 5.14b and 5.14c, respectively). Concretely, the figure shows the absolute value (in time units) of the contribution of I_k^{lp} and the sum of $I_k^{hp} + I_k^{lp}$ to the response time analysis of both LP-eager-max and LP-lazy, and the contribution of I_k^{hp} to the response time analysis of FP-ideal.

The interference factor due to lower priority tasks I_k^{lp} for the LP-lazy and the LP-

eager-max approaches is almost equivalent when $m = 4$ cores. However, the difference between the factor I_k^{lp} of both approaches drastically increases as the number of cores increases. In case of $m = 16$ cores, I_k^{lp} for the LP-lazy approach becomes the dominant factor in the response time. Regarding the higher-priority interference I_k^{hp}, it is alike computed for FP-ideal, LP-eager-max and LP-lazy (see Equation 5.2). However, the factor I_k^{hp} for the LP-lazy approach is always worse than for LP-eager-max, which in turn, is worse than for FP-ideal. The reason is that I_k^{hp} is computed considering the *window of interest* in which higher priority tasks can interfere, i.e., the response time upper bound, which is iteratively computed by Equation 5.1. As I_k^{lp} for LP-lazy or LP-eager-max increases, the window of interest increases as well, impacting on I_k^{hp}. Overall, factors $\Delta_{k,m}^{lazy}$ and $\Delta_{k,m-1}^{lazy}$, used for the computation of I_k^{lp} for the LP-lazy approach, add huge pessimism to the response time upper bound, increasing the window of interest and negatively impacting on the system schedulability of LP-lazy, as shown in the previous section.

5.6 Related Work

The DAG model allows to represent each parallel real-time task as a directed acyclic graph. Baruah et al. [13] first introduced the scheduling problem for a unique DAG task, and showed that the problem *"is computationally intractable, but amenable to efficient approximate solution"*. They provide schedulability tests for determining whether a given DAG task can be scheduled by earliest deadline first (EDF) to always meet the deadlines for all jobs on a specified number of processors. The global EDF scheduling problem of multiple DAG tasks was studied by Bonifaci et al. [91]. Other works that proposed different global EDF schedulability tests are [92][93][94][92]. The partitioned [95] or the federated [66] scheduling approaches for DAG tasks have also been studied. Moreover, conditional DAG tasks [96][54][97][98] have been considered to enrich the parallel task model with control-flow information. Finally, Fonseca et al. [99] provide a response time analysis for DAG tasks scheduled by partitioned fixed priority scheduling.

Despite the significant amount of work on parallel task models, none of the existing works investigates the potential of combining the limited preemptive framework with the current schedulability analysis for DAG task-systems. The potential of limited preemptive scheduling schemes has been mostly investigated in the case of task-sets

composed of sequential tasks. We refer to [33] for an exhaustive survey on the limited preemptive scheduling framework in a single-core scenario. We analyze the limited preemption with *fixed preemption points*, but other techniques have been proposed: (1) *preemption thresholds scheduling* [100], where the preemption of a task is allowed only when the priority of the arriving task is higher than a priority level (threshold), that is assigned to each task; and (2) *deferred preemptions scheduling* [101], where each task specifies the longest interval that can be executed without being preempted. Optimized preemption point placement techniques [64] [102] have also been proposed to reduce the cost of preemption related overheads incurred by a task. These works propose methods for placing suitable preemption points in each task in order to maximize the chances of finding a schedulable solution. Interestingly, similar methods could be exploited for task-sets composed of DAG tasks, in particular, the OpenMP `taskyield` construct would allow to place such optimal preemption points. This is not considered in this thesis but remains as a future work.

In a multi-core system, schedulability analysis have been developed under both the lazy and eager approaches. In the former strategy, an analysis based on link-based scheduling has been proposed by Block et al. [103]. Under link-based scheduling, any newly-released higher priority task is linked to the processor where the lowest priority running task is executing, and can preempt it only when the lowest priority task encounters a preemption point. A response time analysis targeting global fixed priority scheduling with eager preemptions has been proposed by Davis et al. [86], under the assumption that each sequential task has a single final non-preemptive region (the rest of the task is fully-preemptive). This work also showed that an appropriate choice of the length of this region can improve schedulability. Moreover, the authors showed that the limited preemptive approach under global fixed priority scheduling with eager preemptions is incomparable to that with lazy preemptions. The reason is that, for sequential tasks, there is a trade-off between the blocking time of lazy preemptions and the number of preemptions of the eager approach. In this chapter, we demonstrate that such trade-off does not exist for task-sets composed of DAG tasks, for which the blocking time factors of the lazy strategy is very pessimistic. Marinho et al. [89] encompasses tasks with multiple non-preemptive regions and a lazy approach but it mostly provides an analysis of blocking effects. A complete schedulability analysis in the case of eager preemptions and multiple NPRs for task-sets composed of sequential tasks has been proposed by Thekkilakattil et al. [87].

5.7 Summary

This chapter provides a response time analysis for DAG-based task-sets under global fixed priority limited preemptive scheduling. Given the limited preemptive scheduling strategy, two different approaches have been analyzed: *eager* and *lazy*. In case of a higher priority task becomes ready, the former selects the first lower priority running task to reach a preemption point as the one being preempted; the latter selects the lowest priority running task to be preempted whenever it reaches a preemption point. We show the necessary conditions under which DAG tasks may experience lower-priority interference for both approaches. Concretely, we formally proved which are these conditions and compute (1) the number of priority inversions a task may suffer and (2) the lower-priority blocking time. As a result, we derive a novel response time analysis for DAG-based task-sets scheduled under the eager or lazy limited preemptive scheduling.

Finally, we evaluate and compare the response time analysis for the eager and lazy approaches with randomly generated task-sets. Our analysis demonstrates that, despite the eager approach generates a higher number of priority inversions, the blocking factor of the lazy approach dominates the response time upper bound. Therefore, contrary to what has been demonstrated when considering sequential task-sets, the limited preemptive lazy scheduling approach has been proven to be a very inefficient scheduling strategy when task-sets composed of DAG tasks are considered, and so not suitable for parallel execution.

Overall, we advance the state of the art to provide a response time analysis for DAG-based task-sets under global fixed priority limited preemptive scheduling. Given the similarities between the OpenMP tasking model and the DAG tasks model under limited preemptive scheduling, this work allows to provide timing guarantees to real-time systems parallelized with OpenMP, provided that such scheduling strategy is supported.

Chapter 6

DAG-based Parallel Real-Time Systems: Two Real Use Cases

"I tore myself away from the safe comfort of certainties through my love for truth - and truth rewarded me."

— Simone de Beauvoir

This chapter applies the response time analysis presented in previous chapters to two real use cases. We first consider a real-time system composed of several independent OpenMP applications that we integrate together. Secondly, we consider an AUTomotive Open System ARchitecture (AUTOSAR) application, from the automotive domain: a diesel engine management system (EMS).

6.1 Real-Time Tasks Parallelized with OpenMP

This section evaluates the response-time analysis of three real and independent applications parallelized with OpenMP, and integrated within a single real-time system conforming the strategy presented in Chapter 3 (Section 3.1.1). Given the reticence of the industry to provide real use cases, we experiment with three different applications that do not necessarily conform a real system but help us to test our proposed timing analysis techniques. Concretely, we consider a pre-processing sampling application for infra-red H2RG detectors, from the space domain, a pedestrian detector and a cholesky factorization, both useful in the automotive domain to support advanced vehicle functionalities.

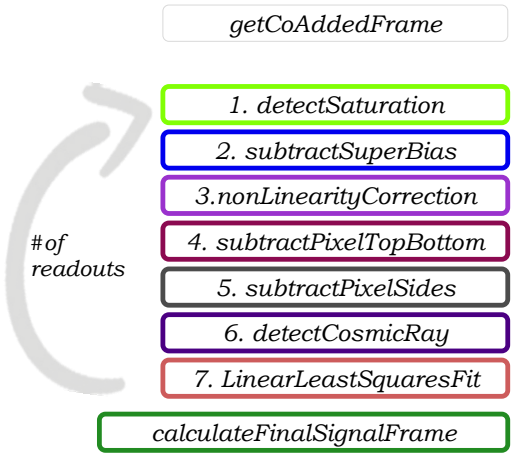


Figure 6.1: Stages of the pre-processing sampling application.

We first describe the applications and the implemented parallelization strategies. Then, we present the methodology to extract the DAG and the experimental setup. Finally, we present the timing analysis for each application individually, i.e., running in isolation, considering the response time analysis presented in Chapter 4, and for the real-time system, i.e., running concurrently, considering the response time analysis presented in Chapter 5. In both cases, we compare the results with the average and maximum observed execution time in a real platform.

6.1.1 Description of the OpenMP applications

This section presents the three C/C++ applications considered and the implemented parallelization strategy.

6.1.1.1 Pre-processing sampling for infra-red detectors

The pre-processing sampling application for the infra-red H2RG detectors (provided by Airbus Defense and Space and developed by the European Space Agency (ESA) under a GPL license), is planned to be used in the Euclid spacecraft, whose objective is to better understand the geometry of dark energy and dark matter by measuring the red-shift of galaxies at varying distances from Earth.

Figure 6.1 illustrates the description of this application. It processes frames of 2048×2048 pixels, provided by the H2RG sensor, through seven stages. It also includes an extra stage, *getCoadedFrame*, that simulates the acquisition of a given number of readouts from the H2RG sensor frame into a 2048x2048 array structure.

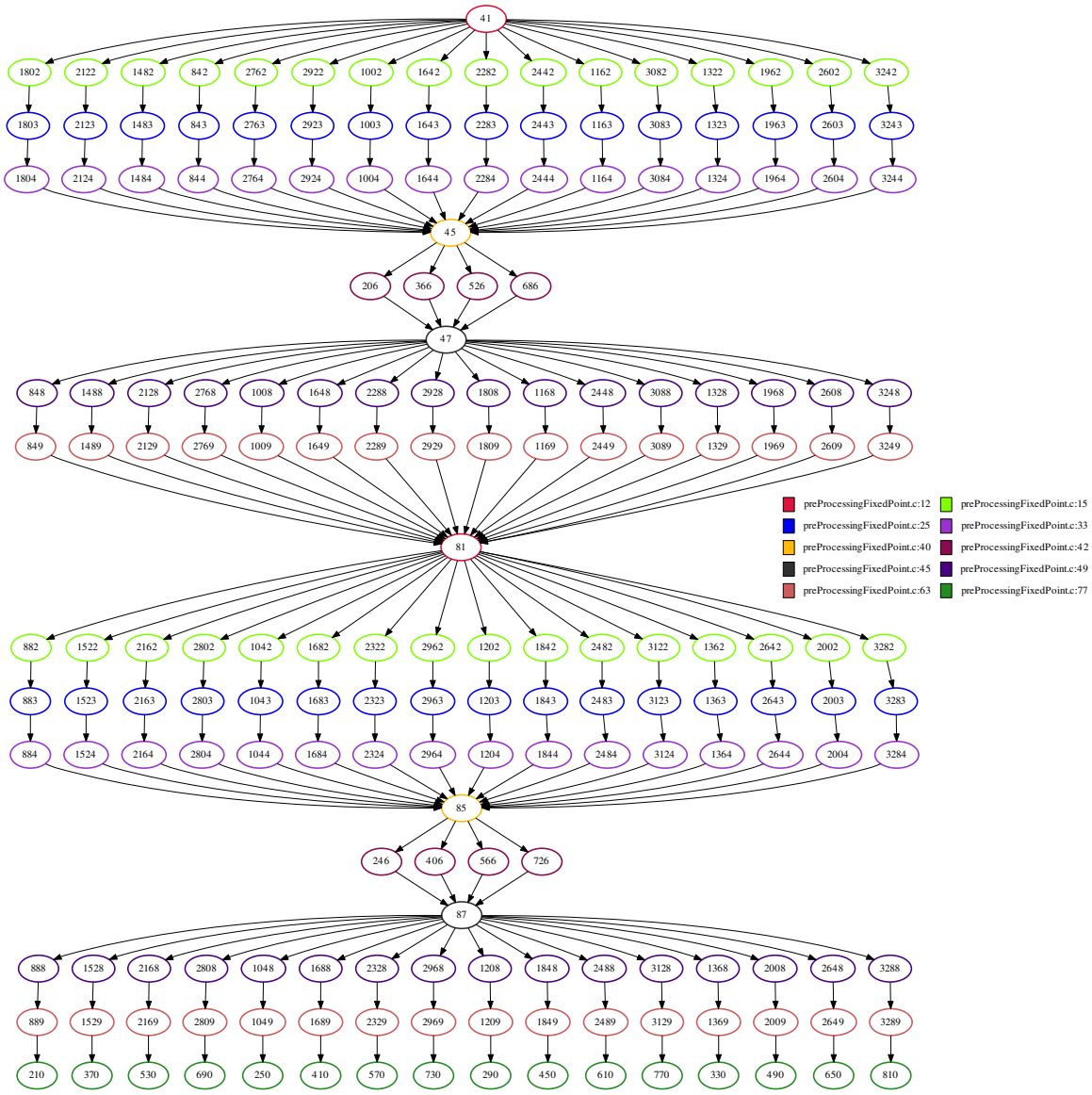


Figure 6.2: DAG of the pre-processing sampling application when $BS = 512$ (190 nodes).

In this case, we consider two readouts of the sensor. Since this stage is a simulation we do not consider it into the parallelization strategy. Moreover, we assume that the sensor data acquisition overlaps with the computation that processes a previously acquired frame. The last stage, *calculateFinalSignalFrame*, is a final stage to compute the metrics and provide the results that summarize the previously computed frames.

Parallelization strategy. We parallelized the pre-processing sampling application following a wave-front strategy, in which the frame is divided into blocks, of size

$BS \times BS$, and being potentially processed in parallel. Therefore, the computation of each frame block for each stage is potentially assigned to an OpenMP task. The data dependencies existing among the different stages are defined by the `depend` clause. The source code of the most representative function of this application, that includes the OpenMP directives, is shown in Appendix B.1.

Figure 6.2 shows the resultant DAG of the pre-processing sampling application when $BS = 512$, i.e., when the frame is divided into 16 blocks. The numbers in the legend correspond to the lines of the source code where the OpenMP `task` or `taskwait` directive start (see Appendix B.1). Also the colors shown in Figure 6.1 correspond to the color of the nodes implementing the stages. The first frame (from the first readout of the sensor) is processed by the tasks represented as nodes 41 to 81. Similarly, the nodes starting after node 81 process the second frame (from the second readout), except the last "row of green nodes" that represent the final stage. Nodes 41, 45, 81 and 85 represent the `taskwait` constructs.

6.1.1.2 Pedestrian detector

Probably, one of the most popular pedestrian detectors is the Histograms of Oriented Gradients (HOG) feature descriptor (or simply HOG descriptor), trained with a Support Vector Machine (SVM) approach [104]: A HOG feature descriptor is a data structure used to encode the digital image of an object (in our case a pedestrian) independently of modest changes in viewing conditions, e.g., changes in scale, orientation, contrast. The descriptor counts occurrences of gradient orientation in localized portions of the image, and it is based on global features (rather than a collection of local features) to describe pedestrians. A SVM is a type of machine learning algorithm used, in this case, to classify pedestrians based on HOG descriptors.

The pedestrian detector uses a *sliding detection window* (64×128 pixels) that is moved around the input image to be analyzed. For each detection window, a HOG descriptor is computed and processed by the SVM, which classifies it as "a pedestrian" or "not a pedestrian". The HOG descriptor of a given window is computed by further dividing the window into *cells* (8×8 pixels) and overlapping *blocks* (2×2 cells). Figure 6.3 shows an example of the divisions made to an input image.

Our implementation of the parallel pedestrian detector is based on the computation of the HOG descriptor included in the open-source VLFeat library [105].

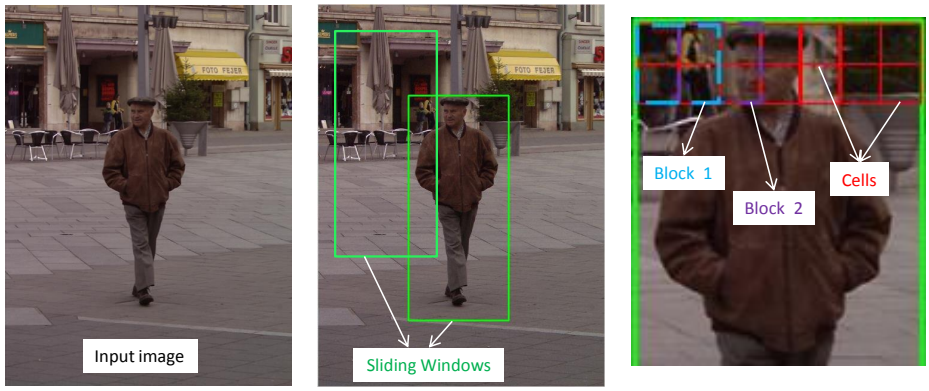


Figure 6.3: Pedestrian detector description: divisions in the input image.

Parallelization strategy. Given a Full HD input image divided into blocks, the computation of $NBLOCKS \times NBLOCKS$ blocks is assigned to an OpenMP task. Since blocks can overlap, the `depend` clause is used to define the dependencies among tasks, i.e., if the four cells of a block are processed, an overlapping block only processes the non-common cells (see Figure 6.3). As a result, the parallelization follows a wavefront strategy, meaning that the computation of block (x, y) depends on blocks $(x-1, y)$, $(x, y-1)$ and $(x-1, y-1)$. The OpenMP task computing the last block of a sliding window, also computes the final HOG descriptor and compares it with the reference SVM. The source code of the most representative function of the pedestrian detector, that includes the OpenMP directives, is presented in Appendix B.2.

Figure 6.4 shows the resultant DAG of the person detector when $NBLOCKS = 20$, i.e., when each OpenMP task processes 400 blocks. In this case, the number of nodes in the DAG is 84. The number in the legend correspond to the lines of the source code where the OpenMP `task` directives are included (see Appendix B.2).

6.1.1.3 Cholesky factorization

The Cholesky factorization [55] is an useful function commonly used for efficient linear equation solvers and Monte Carlo simulations. Cholesky can also be used to accelerate Kalman filters, implemented in autonomous vehicle navigation systems to detect objects positions and compute trajectories. The application processes a matrix of 4096×4096 real floating-point numbers, using the Intel Math Kernel Library to compute the matrix factorization.

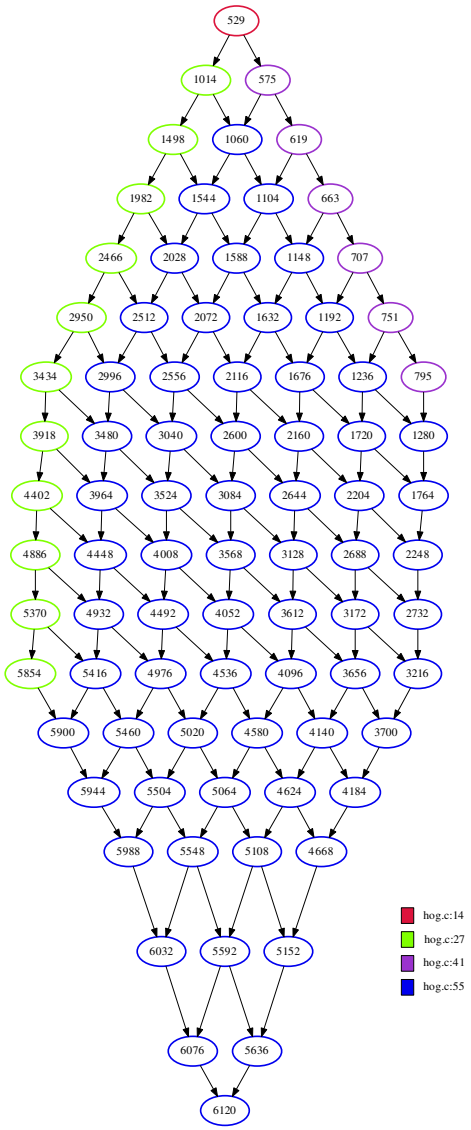


Figure 6.4: DAG of the pedestrian detector when $NBLOCKS = 20$ (84 nodes).

Parallelization strategy. The parallelization approach considered for this application, divides the matrix into $NB \times NB$ blocks. Similarly to the previous applications, the computation of each matrix block is assigned to an OpenMP task. The data dependencies between blocks are defined by the `depend` clause. As NB increases, the number of OpenMP task increases as well, since there are more blocks to process. However, the task granularity decreases as NB increases, since the size of each block becomes smaller. The source code of the most representative function of this application, that includes the OpenMP directives, is presented in Appendix B.3.

Figure 6.5 shows the DAG representation of the cholesky factorization when $NB = 8$, i.e., when the matrix is divided into 64 blocks. The number in the legend correspond

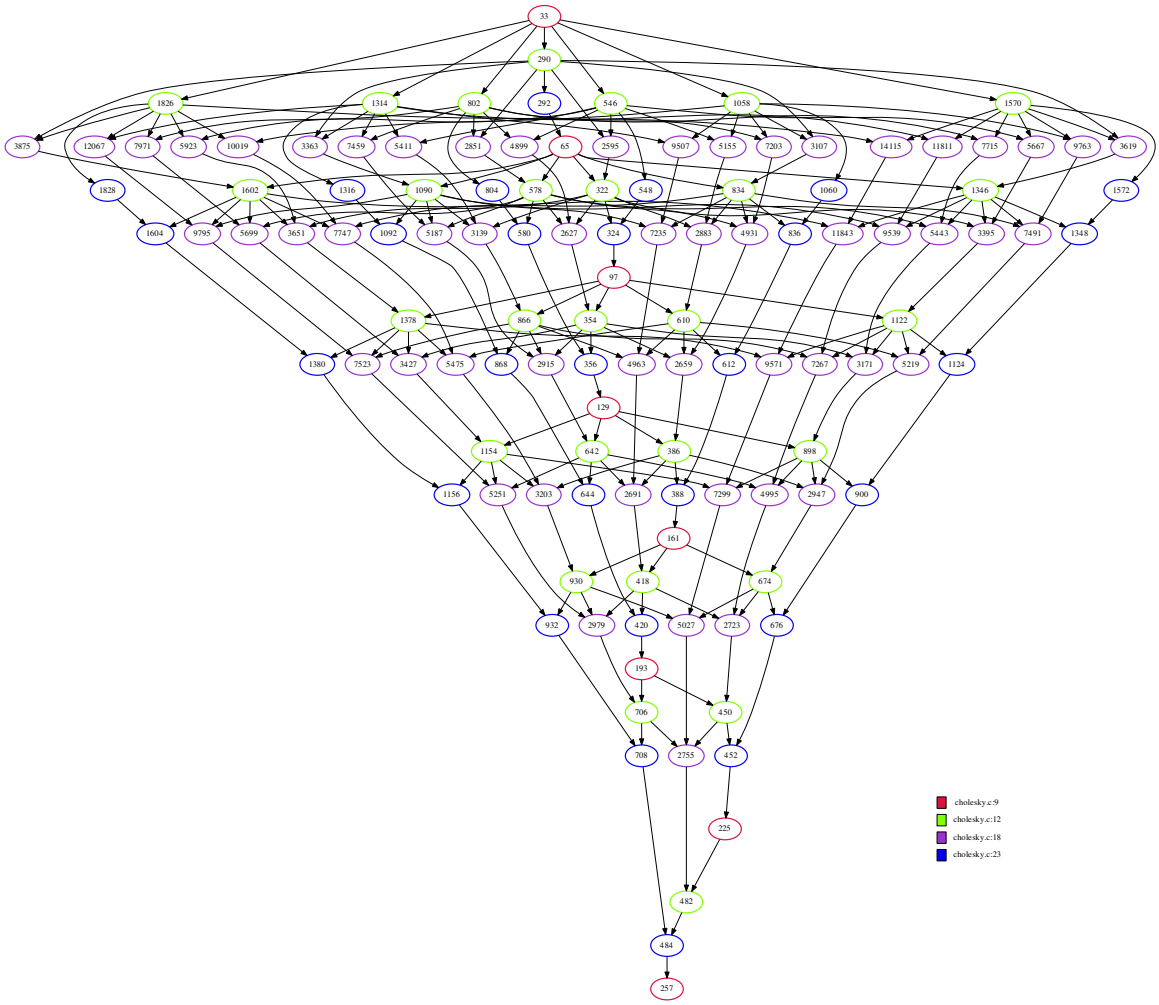


Figure 6.5: DAG of the cholesky factorization application when $NB = 8$ (120 nodes).

to the lines of the source code where the OpenMP `task` directives start (see Appendix B.3).

6.1.2 Extraction of the DAG task

The DAG representations of the three OpenMP applications (as the examples shown in Figures 6.2, 6.4 and 6.5) have been statically obtained from the source code of each application, with a compiler technique [57], implemented in Mercurium [106]. At runtime level, the DAG is also used to execute OpenMP tasks while honoring their dependencies. The scheduler follows the statically derived DAG, instead of using the `depend` clauses (already analyzed by the compiler) [43]. This feature in the runtime is implemented on top of the GNU libgomp library [68] included in GCC version 5.4.0 (labeled as *libgomp**). Mercurium and libgomp* also provide a trace utility to get

the execution time of the OpenMP tasks, task parts and the runtime overhead.

These methods are used to compute the WCET of each node by taking the highest execution time observed of the corresponding OpenMP task running in isolation. Then, a safety margin of 60% is added, which is a common industrial practice to obtain WCET values, relying on software simulation and testing, and reinforcing by the application of safety margins [107].

6.1.3 Experimental setup

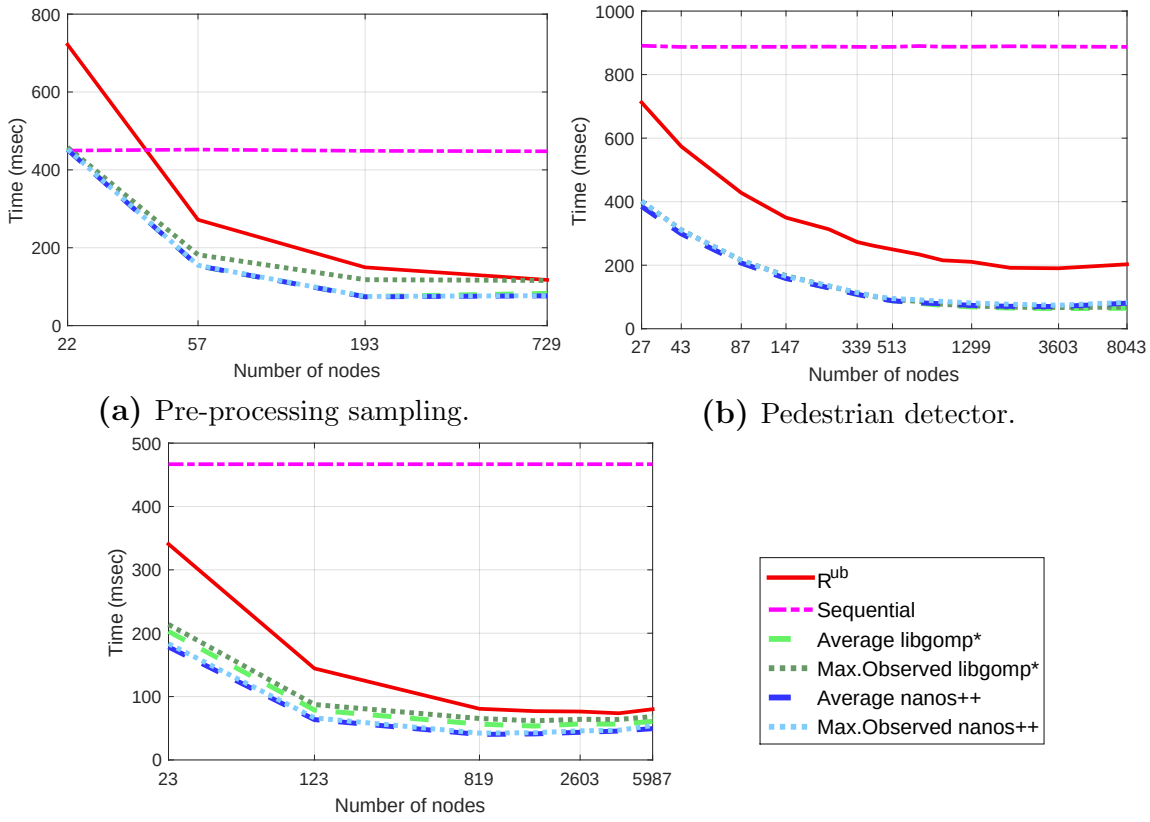
In order to evaluate the response time analysis of each OpenMP application is isolation, and being executed concurrently as a part of a real-time system, we compare the results with the average and maximum observed execution times in a real platform. To do so, we consider two different OpenMP runtime implementations: libgomp* and Nanos++ [108]

The reason of also using Nanos++ is that, as seen in Chapter 3, libgomp does not implement the priorities and the TSPs as required by the fixed priority limited preemptive scheduling approach presented in Chapter 5.

As seen in previous chapter, the response time analysis have been implemented in MATLAB®. The experiments run in an Intel Xeon Platinum 8160 CPU with 24 cores, which operates at 2.10GHz. This platform is mainly used in the HPC domain. The reason of using it rather than an embedded platform is twofold: (1) it allows us to explore up to a higher number of cores; and (2) it supports OpenMP runtime implementations, i.e., libgomp and Nanos++, with all the required features (`depend`, `priority` clause, etc.). As a drawback, the variability of the execution time of the applications is higher than in an embedded platform. It remains as a future work the evaluation of our response time analysis in a parallel embedded architecture.

6.1.4 Individual timing analysis

This section presents the timing analysis of the three applications running in isolation. We compute for each application (1) the response time upper bound R^{ub}, as provided by Equation 4.4 in Chapter 4; (2) the average execution time when running sequentially, i.e., when OpenMP directives are ignored; and (3) the average and maximum observed execution time when running in parallel. The average and maximum observed execution times have been computed for *libgomp** and *nanos++*. The



(a) Pre-processing sampling.

(b) Pedestrian detector.

(c) Cholesky factorization.

Figure 6.6: Individual timing analysis when $m = 16$.

reason is that, as explained in the previous section, the WCET of each node in the DAG has been computed considering the execution time in *libgomp**, while the final system uses *nanos++*. Therefore, we compare the execution times for *libgomp** and *nanos++* and, as shown in the next sections, the WCET estimations obtained with libgomp* remains valid when executing the applications in Nanos++.

Figure 6.6 shows the R^{ub}, the average sequential and parallel execution times, and the maximum observed parallel execution times (in ms), when varying the number of nodes of the DAG of the three applications, the pre-processing sampling, the pedestrian detector and the cholesky factorization (Figures 6.6a, 6.6b and 6.6c, respectively). The number of nodes, i.e., the number of OpenMP task instances, is determined by the block size BS in case of the pre-processing sampling, the number of blocks $NBLOCKS$ in case of the pedestrian detector, and the total number of blocks NB in case of the cholesky factorization. The figures show the results when the number of cores used is $m = 16$. Similar trends have been observed when the number of cores is $m = 4$ and 8.

First of all, the figures show the improved performance of the parallel version of these application with respect to the sequential version. Concretely, the performance increases as the number of nodes in the DAG increases, because more parallelism is exploited. In average, the best speedup factors are 6x, 13x and 8.7x, for the pre-processing sampling, the pedestrian detector and the cholesky factorization, with *libgomp** and obtained when the number of nodes is 193, 3603 and 1543, respectively. In case of *nanos++*, this values are 6x, 12.7x and 11.6x, when the number of nodes is 193, 3603 and 819, respectively.

When comparing the average and maximum observed execution times in *libgomp** and *nanos++*, both provide similar performance (the results in *libgomp** are not always visible in the figures since the curves overlap with the results in *nanos++*). Depending on the number of nodes, the difference between the average execution time in *libgomp** and *nanos++* is between the range $[-0.3\%, 7.74\%]$, $[-24\%, 0.25\%]$, and $[12\%, 28\%]$, for the pre-processing sampling, the pedestrian detector and the cholesky factorization, respectively. The positive values means that *nanos++* is better than *libgomp** and, since the WCET estimations consider the maximum observed execution time, *nanos++* is safely covered by the analysis of *libgomp**. In case of the pedestrian detector, the negative value of -24% means that *libgomp** is better than *nanos++*, but it is found for the highest value of the number of nodes 8043, when the performance in both cases starts to decrease. When the number of nodes is 3603 (and the best performance is observed in both *libgomp** and *nanos++*) the difference decreases up to -10%. This small percentage is safely covered by the safety margin.

Notice also the small difference between the average and the maximum observed execution times. The reason is that the applications execute in isolation, so the variation in the timing behavior is small. When the best performance is observed, the maximum observed time increases 57%, 5% and 15% over the average time, in *libgomp** for the pre-processing sampling, the pedestrian detector and the cholesky factorization, respectively. In *nanos++* these values are 1.4%, 5% and 5%, respectively.

Finally, the response time analysis provides a safe upper bound R^{ub} for the maximum observed execution times. When the best performance is considered, the maximum observed execution time represents 63%, 35% and 80% over the response time upper bound in *libgomp**, for the pre-processing sampling, the pedestrian detector and the cholesky factorization, respectively. In *nanos++* these values are 50%, 38% and

52%, respectively. These values indicate the actual system utilization, which provides an overview of the resources overestimation that is common in real-time systems due to the required timing guarantees.

6.1.5 Timing analysis of an OpenMP real-time system

In this section we present the timing analysis when integrating the three OpenMP applications as a unique real-time system. We evaluate the eager limited preemptive scheduling approach as implemented in *nanos++*. To the best of our knowledge, there is no implementation of a lazy limited preemptive scheduler in any OpenMP implementation, and OpenMP does not support a fully-preemptive execution model. Hence, with the objective of comparing the eager and lazy limited preemptive scheduling and the fully-preemptive scheduling, we also present the results of a simulation.

6.1.5.1 System configuration

For the system integration, it is required to consider a DAG configuration for each application, i.e., the number of nodes, that we selected based on two criteria: (1) the performance of the application and (2) the performance of the response time analysis. Thus, the DAGs with 193, 1299 and 819 nodes have been considered, for the pre-processing sampling, the pedestrian detector and the cholesky factorization, respectively. In case of the pre-processing sampling and the cholesky factorization, these values correspond to the best performance of the application in isolation for *nanos++*, as seen in the previous section. In case of the pedestrian detector, we select the DAG with 1299 nodes instead of 3603, because both configurations provide similar performance, but a higher number of nodes in the DAG imposes more lower priority interference, as the number of potential preemption points is higher. This would lead to a worse response time upper bound.

Similar criteria have been considered to assign priorities to each real-time task, also considering that, as seen in the previous chapter, the lower priority interference is the dominant factor in the response time analysis. The pre-processing sampling is the highest priority task since it has nodes in the DAG with high WCET, compare to the rest of nodes in the system. Since this nodes would be considered in the lower-priority interference if the pre-processing sampling would have lower priority than any other task, this would lead to pessimistic response time upper bounds. In case of the pedestrian detector and the cholesky factorization, the latter has a smaller volume,

1,705 ms vs 734 ms, so we assign to the cholesky factorization the lowest priority, so that it does not suffer lower priority interference. Finally, the deadline, and period, of each application is 410 ms, 780 ms and 400 ms, for the pre-processing sampling, the pedestrian detector and the cholesky factorization, respectively. Overall, the utilization of the system is computed as:

$$U_{\mathcal{T}} = \frac{vol(G_{pre})}{T_{pre}} + \frac{vol(G_{ped})}{T_{ped}} + \frac{vol(G_{cho})}{T_{cho}} = \frac{722}{410} + \frac{1,705}{780} + \frac{734}{400} = 5.78$$

The analysis is performed for the execution of the real-time system during 18 s. During this time, the pre-processing sampling, the pedestrian detector and the cholesky factorization, are released 44, 24 and 46 times, respectively.

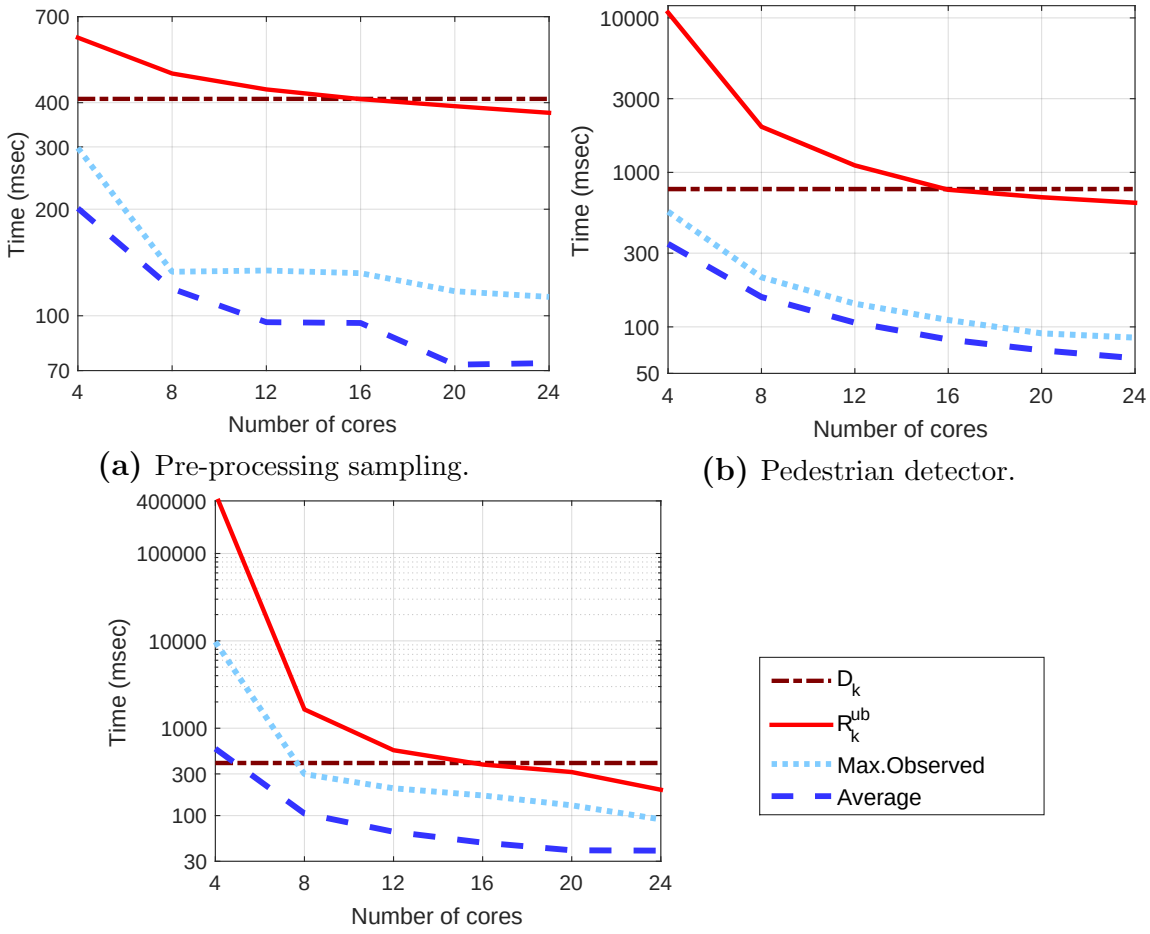
6.1.5.2 Eager limited preemptive scheduling

Figure 6.7 shows the response time upper bound R_k^{ub}, the deadline D_k, and the average and maximum observed execution times (in ms), when varying the number of cores available for executing the system, for the pre-processing sampling, the pedestrian detector and the cholesky factorization (Figures 6.7a, 6.7b and 6.7c, respectively).

The real-time system is not schedulable, i.e., $R_k^{ub} > D_k$ when the number of cores is low. As expected, the response time upper bound (and the average and maximum observed execution time) decreases, in all the cases, as the number of cores increases, being the system schedulable when the number of cores is $m \geq 16$.

Figures show the increasing difference between the deadline and the response time upper bound, as the priority of the applications decreases (even though the y-axis are in different logarithmic scale for each application). For instance, when $m = 4$, the difference between D_k and R_k^{ub} represents the 32%, 92% and 99.9% of R_k^{ub}, for the pre-processing sampling, the pedestrian detector and the cholesky factorization, respectively. The reason is that tasks with lower priority experience higher interference. In case of the cholesky factorization, not even the average execution time is below the deadline when $m = 4$, meaning that, in average, all the deadlines are missed.

When the number of cores is $m = 16$, the average execution time of each application increases 22%, 11% and 18% with respect to the average execution time in isolation, for the pre-processing sampling, the pedestrian detector and the cholesky factorization, respectively. In case of the response time upper bound, it increases



(a) Pre-processing sampling.

(b) Pedestrian detector.

(c) Cholesky factorization.

Figure 6.7: Timing analysis of the OpenMP applications within the real-time system, under eager limited preemptive scheduling (y-axis in logarithmic scale).

63%, 72% and 78%, due to the impact that both the higher and lower priority tasks interference have on the computation of R_k^{ub}.

When the system is schedulable, i.e., for $m \geq 16$, the maximum observed time represents up to 32%, 14% and 45%, over the response time upper bound R^{ub}, for the pre-processing sampling, the pedestrian detector and the cholesky factorization, respectively. Similarly to the individual timing analysis, these values indicates the actual system utilization of the platform.

6.1.5.3 Comparing eager, lazy limited preemptive and fully preemptive scheduling

Similarly to the execution of the system in a real platform, the simulation considers 18 ms of execution, during which the pre-processing sampling, the pedestrian detector

		Time (ms)		
		Infra-red (193 nodes)	**Pedestrian (1299 nodes)**	**Cholesky (819 nodes)**
$D_k = T_k$		410	780	400
LP-eager-max	R_k^{ub}	374.50	635.84	197.52
	Max.	114.56	129.47	142.29
	(Avg.)	(112.18)	(116.93)	(66.54)
LP-lazy	R_k^{ub}	3,419.25	1,234.02	580.58
	Max.	117.57	128.93	142.24
	(Avg.)	(112.43)	(116.80)	(66.50)
FP-ideal	R_k^{ub}	136.99	207.45	167.41
	Max.	111.52	127.04	142.52
	(Avg.)	(111.52)	(84.73)	(54.39)

Table 6.1: Response time analysis and scheduling simulation maximum observed and average time when $m = 24$.

and the cholesky factorization, are released 44, 24 and 46 times, respectively.

Table 6.1 shows the response time upper bound R_k^{ub}, and the maximum observed and average execution time of the scheduling simulation, for each application and for the three scheduling strategies, LP-eager-max, LP-lazy and FP-ideal, when $m = 24$.

Notice that the scheduling simulation considers the WCET of each node in the DAGs, instead of the actual execution time in a real platform. Moreover, the maximum observed and average execution times of the simulation are computed considering all the times the different tasks are released within the simulation interval. In case of the eager strategy, we compare the maximum observed execution time obtained in the real platform and the simulator. When the $m = 24$, the maximum observed execution time in the simulator increases 1.3%, 34% and 36%, over the real maximum observed time, for the pre-processing sampling, the pedestrian detector and the cholesky factorization, respectively.

Table 6.1 confirms the conclusion reached in Chapter 5, for the synthetic DAG task-sets (Section 5.5). The response time analysis of the LP-eager-max strategy clearly outperforms the LP-lazy approach. The system is not schedulable under LP-lazy strategy, that adds huge pessimism over the average and maximum observed simulated times. Thus, the maximum observed simulated time represents only 3%,

	Infra-red (193 nodes)	Pedestrian (1299 nodes)	Cholesky (819 nodes)
LP-eager-max	0	287	935
LP-lazy	0	245	928
FP-ideal	0	231	1,418

Table 6.2: Preemptions during a 18 ms simulation when $m = 24$.

10% and 24% of the R_k^{ub}, for the pre-processing sampling, the pedestrian detector and the cholesky factorization, respectively. In case of the LP-eager-max strategy, this percentage increases up to 30%, 20% and 72%.

As also shown in previous experiments, the FP-ideal strategy outperforms the LP-eager-max, representing the average execution time in the simulator the 81%, 40% and 32% of the R_k^{ub} for the FP-ideal strategy, for the pre-processing sampling, the pedestrian detector and the cholesky factorization, respectively. However, while the average and maximum observed execution times are similar for the three scheduling strategies, the number of preemption points (not considered in the response time upper bound) increases in the fully-preemptive with respect to the limited-preemptive scheduling. Table 6.2 shows the number of preemptions that each application experiments in the scheduler simulator of the LP-eager-max, LP-lazy and FP-ideal strategies. While the difference between the LP-lazy and LP-eager is small, the FP-ideal significantly increases the number of preemptions, being almost 500 more for the cholesky factorization being in 18 ms. Besides the difficulties of properly accounting for the preemptions in the response time analysis of a fully-preemptive scheduling strategy, this would significantly degrade the average and maximum observed execution times in a real platform.

6.2 Automotive Use Case

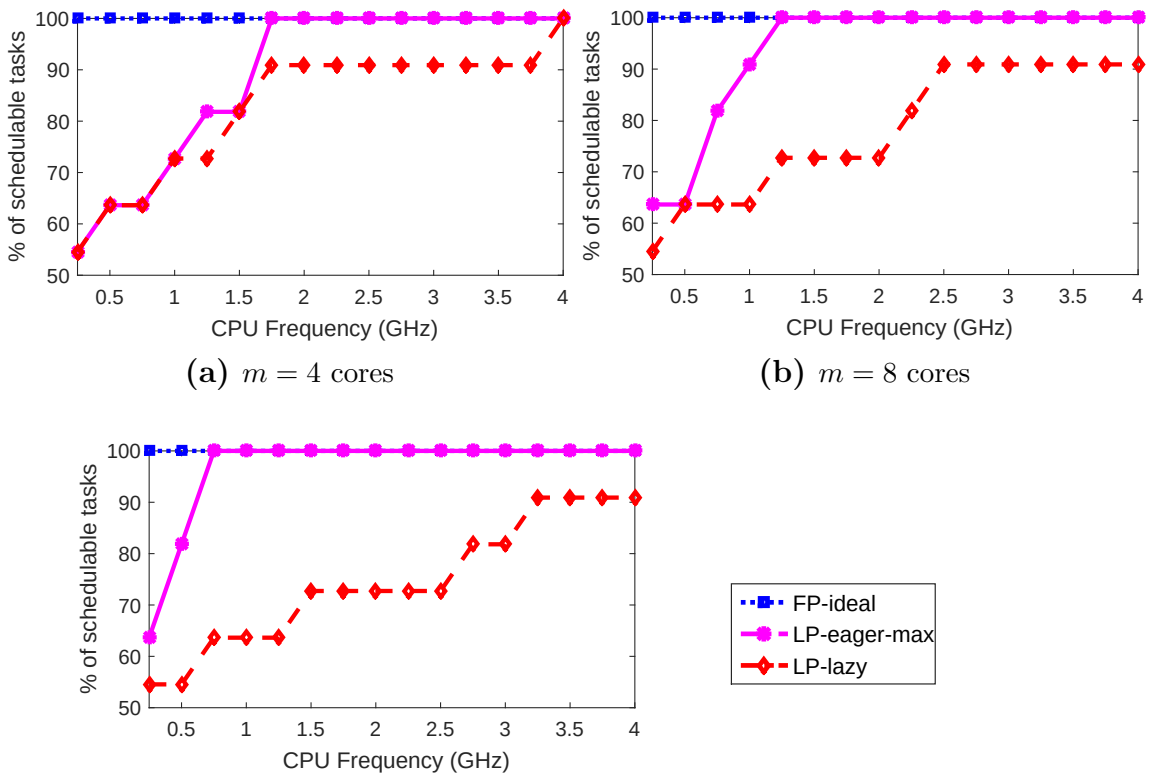
The response time analysis presented in Chapter 5 can be also applied to any other task-based parallel model than OpenMP. Therefore, this section evaluates the response time analysis of an AUTomotive Open System ARchitecture (AUTOSAR) application: a diesel engine management system (EMS). This system has been provided by DENSO Deutschland GmbH, a leading supplier of automotive technology.

6.2.1 Description of the EMS DAG-based task-set

AUTOSAR is a standardized system software architecture upon which automotive applications are built and executed [78]. An AUTOSAR application is composed of a set of functions, named *runnables*, that communicate among them through well-defined communication methods. Runnables, that can be executed periodically or triggered by an interrupt, are grouped into AUTOSAR tasks, which are the unit of scheduling (UoS) of the AUTOSAR Operating System. The nature of AUTOSAR execution model fits very well the system model considered in this thesis: an AUTOSAR task can be modeled as a DAG task, as proposed in [109, 110, 111], where nodes correspond to runnables and edges correspond to communication methods among runnables. Runnables are executed uninterruptedly, defining preemption points at runnable boundaries.

An EMS is a typical automotive application, that controls the amount of fuel and the fuel injection times, which are fundamental for the smooth revolution of the engine. The amount of fuel and when it is injected depends on the state and the rotation speed of the engine, which changes continuously during its operation. The EMS is composed of eleven AUTOSAR tasks, that are periodically time-triggered, and a crank-angle task, that is triggered based on position of the crankshaft. The periodic tasks have periods (and implicit deadlines) of 1, 4, 5, 8, 16, 20, 32, 64, 96, 128 and 1024 ms, and the crank-angle triggered task has a period that varies depending on the revolutions of the engine, being the minimum period equal to 1.25 ms, and then as considered in the analysis. Overall, the EMS is composed of twelve DAG tasks comprised of roughly 1,200 runnables (nodes).

Denso provided the DAG representations of the EMS application, instead of the source code of the system. The WCET estimates of nodes ($C_{k,i}$), given in CPU cycles, were computed with a static timing analysis tool OTAWA [112, 113], which models a generic multi-core processor architecture. Concretely, a 4-core, 8-core or 16-core processor setup with private per-core scratchpads for instructions and write-through data caches. For all processor configurations, cores are connected through a tree NoC to the on-chip RAM memory. The impact of interferences due to the access to shared processor resources is not considered in the WCET computation. The approach presented in this thesis is independent of the multi-core processor architecture and the timing analysis method, so other architectures and tools can be used to compute the WCET estimates of runnables.



(a) $m = 4$ cores

(b) $m = 8$ cores

(c) $m = 16$ cores

Figure 6.8: Percentage of schedulable tasks from the EMS AUTOSAR application as a function of the CPU frequency.

The period of each DAG task, also provided by the supplier, is in milliseconds while the WCET of the nodes is in CPU cycles. Therefore, in order to evaluate the EMS application under different utilization scenarios, we range the CPU frequency. Hence, the processor frequency is used to derive the timing value (in ms) of the WCET. Increasing the CPU frequency is equivalent to decrease the overall task-set utilization. For instance, when a processor operates at 250 MHz, the overall EMS utilization equals to 0.57; 4 GHz corresponds to a system utilization of 0.03.

6.2.2 Schedulability analysis

This section evaluates the response time analysis for the two limited preemptive scheduling strategies, eager and lazy, *LP-eager-max* and *LP-lazy*, respectively, and compared them with an ideal fully preemptive strategy *FP-ideal*.

Figure 6.8 shows the percentage of schedulable tasks when ranging the CPU frequency from 250 MHz to 4 GHz and considering a 4-core, 8-core and 16-core processor (Figures 6.8a, 6.8b and 6.8c, respectively). The trend shown in these figures is similar

to the one observed in the experimental results of previous chapter, when considering synthetic DAG tasks: LP-eager-max outperforms LP-lazy in all cases, and FP-ideal outperforms the LP-eager-max, as the blocking impact of low-priority tasks is not considered. LP-lazy cannot schedule the EMS application in any processor frequency configuration, except assuming a 4-core processor operating at 4 GHz. The pessimism added by the LP-lazy approach increases as the number of cores increases, resulting in the counter-intuitive result where the schedulability decreases as the number of cores increases, as also shown with the randomly generated DAG task-sets. Instead, under the LP-eager-max approach, the EMS application is schedulable when the CPU frequency is equal or higher than 1.75 GHz, 1.25 GHz and 750 MHz for a 4, 8 and 16-core configuration, respectively (with an overall utilization of $0.08, 0.11$ and 0.2). As expected, the schedulability increases as the number of cores increases.

Overall, from this experiment, we demonstrate that the conclusion reached in previous chapter for synthetic DAG task-sets is also valid for this automotive case study. The response time analysis provided by the eager limited preemptive scheduling strategy (LP-eager-max) clearly outperforms the lazy approach.

6.3 Summary

This chapter aims to check the viability of the response time analysis presented in this thesis, with real parallel applications. We evaluate (1) a real-time system implemented with OpenMP, where tasks can only be preempted at task scheduling points, and (2) an AUTOSAR application, where tasks can only be preempted at runnable boundaries. Interestingly, these two execution models are compatible with the limited preemptive scheduling strategy widely used in real-time systems.

Our analysis confirms the huge pessimism of the lazy limited preemptive scheduling, which provides an intractable response time upper bound. However, the eager limited preemptive scheduling provides a more efficient response time analysis, with an acceptable system utilization, even considering the over estimation of the resources, an ordinary practice in critical real-time systems.

The evaluation presented in this chapter allows us to comprehend that more reliable techniques and more realistic systems must be considered. From a timing analysis perspective, analysis techniques are needed to consider parallel execution, specially to compute the WCET of parallel applications. This is not an easy task,

which still relies on safety margins even when considering single core architectures. However, adapted methods could provide tighter estimations. From an implementation perspective, specific OpenMP runtimes targeting real-time systems must be implemented to also evaluate embedded architectures.

We also conclude that the choice of the best system configuration in terms of parallelism granularity, real-time tasks' priorities, number of cores, system utilization, etc., becomes a trade off between performance of each real-time task and the interference imposed and suffered by other tasks. Moreover, this trade-off affects, not only to the response time analysis, but also to the deployment of the system in a real platform. As a future work remains the consideration of other factors, like the preemptions overhead, and the analysis of a real industrial system, not always easy to obtain.

Overall, our response time analysis can be directly applied to a real-time system implemented and parallelized with OpenMP. Moreover, our analysis also applies to other tasking models, such as AUTOSAR, widely used to design automotive applications.

Chapter 7

Response Time Analysis Supporting Heterogeneous Computing

"*Ideas do not last long. We must do something with them.*"

— Santiago Ramón y Cajal

Parallel and heterogeneous hardware architectures are becoming mainstream in the embedded domain to cope the increasing performance requirements. These architectures integrate low power general-purpose multi-cores (known as *host*) with dedicated accelerator devices like many-cores, DSP fabrics, GPUs or FPGAs. Some examples are the NVIDIA Tegra X1 [114], the Texas Instruments Keystone II [22], the Kalray MPPA [25] or the Xilinx UltraScale [115]. The use of parallel programming models is fundamental to effectively exploit the huge performance capabilities of these architectures.

OpenMP incorporates a *host-centric accelerator model*, coupled with the tasking model, used to efficiently offload code and data to accelerator devices. This is a very common design implemented by many processor vendors in which a low power general-purpose multi-core host processor is coupled with a dedicated accelerator device such as Application-Specific Integrated Circuits (ASICs), Graphics Processing Units (GPUs) or Digital Signal Processing (DSP) fabrics. Interestingly, OpenMP is being adopted in some heterogeneous architectures targeting embedded systems.

As an example, the TI keystone II and the Kalray MPPA support OpenMP in its software development kit [23][25].

This chapter extends the response time analysis introduced in Chapter 4 to support heterogeneous computing. Under this scenario, the workload offloaded into the accelerator device does not cause any interference on the parallel workload executed in the host, and vice versa. Our analysis takes this into account and identifies the portion of the DAG that can potentially execute in parallel with the offloaded workload. Then, DAG transformation techniques are used to guarantee the overlap between the computation on the host and the device. As a result, an interference reduction can be safely incorporated into the response time analysis.

7.1 Heterogeneous System Model

7.1.1 Extending the system model

In order to incorporate heterogeneous computing in our response time analysis, we need first to extend the system model presented in Chapter 2 (Section 2.1.2). We consider a *host-centric accelerator model* in which the host (m cores) is responsible for offloading code and data to a single accelerator device, and collecting results.

The system model considered in this Chapter is composed of a real-time task τ, represented as a DAG $G = (V, E)$. Nodes in V represent sub-tasks, and edges in E represent precedence constraints. The DAG task model is extended to include a special node, representing the workload executed in the accelerator device. This node is named *offloaded node*, denoted by v_{Off}, and characterized by its WCET C_{Off}, which corresponds to the worst-possible execution time of the offloaded workload into the accelerator device. We consider that the overhead due to code and data transfers between host and device, is included in the host and offloaded nodes.

In the new system model supporting heterogeneous computing, $vol(G)$ represents the WCET of the DAG task when executing on a single core in the host and the accelerator, assuming that host and accelerator cannot execute in parallel. $len(G_k)$ corresponds to the minimum amount of time needed to execute the task assuming a sufficiently large number of cores in the host.

```
1  #pragma omp parallel
2  #pragma omp single nowait   // T1
3  {
4    v11
5    #pragma omp task          // T2
6                 depend(out:A)
7    {   v21   }
8    v12
9    #pragma omp target nowait
10                depend(in:A,out:B)
11                map(to:A, from:B)
12   {
13     vOff // Device Workload
14   }
15   v13
16   #pragma omp task          // T3
17   {   v31   }
18   #pragma omp task          // T4
19                depend (in:B)
20   {   v41   }
21   v14
22 }}
```

Listing 7.1: Example of an OpenMP program using task and target constructs.

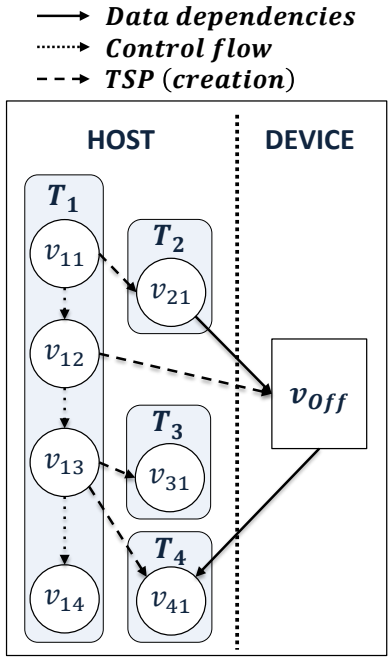


Figure 7.1: Heterogeneous OpenMP-DAG corresponding to the program in Listing 7.1.

7.1.2 Including the accelerator model in the OpenMP-DAG

OpenMP incorporates an advanced host-centric accelerator model, coupled with the tasking model, in which the host is responsible for orchestrating the execution on the host, and also on the device accelerator. This model is supported by the `#pragma omp target` directive, that defines the code to be offloaded, and the data-mapping clauses to express directionality when moving data to/from the device memories (see Section 2.2.2).

Listing 7.1 shows an example of an OpenMP program using the tasking and accelerator models. The execution of the program starts on the host, where the `parallel` construct creates the team of OpenMP threads (line 1) and the `single` construct specifies that only one thread executes the associated block of code (line 2). When this thread encounters the `task` constructs at lines 5, 16 and 18, it creates the associated OpenMP tasks that can be executed by any thread in the team. Similarly to the `task` construct, when the thread executing T_1 encounters the `target` construct (line 9), the code included within the target v_{Off} is offloaded[1] to the accelerator device,

[1]In OpenMP, the target task can also be executed in the host if all the devices are busy (or do

transferring the data specified in the `data(to:)` clause to the memory of the accelerator. The clause `data(from:)` specifies the data to be transferred from the memory of the accelerator to the host memory, once the execution completes. Therefore A must be copied to the accelerator memory and B must be copied-back from the accelerator memory. The clause `nowait` specifies that the execution in the host can continue once the code and data have been offloaded (asynchronous model[2]). Finally, the `depend` clause can also be used to define the data dependencies existing between tasks and targets. Hence, T_2 generates the variable A that is consumed by v_{Off}. Therefore, the target defers its execution until T_2 finishes. Similarly, v_{Off} generates the variable B that is consumed by T_4, making T_4 wait until v_{Off} finishes.

Incorporating these features of the OpenMP accelerator model into the OpenMP-DAG is straightforward, if we differentiate the nodes that execute in the host from those that execute in the device. We define the *Heterogeneous OpenMP-DAG* as the DAG representation of an OpenMP program using the tasking and accelerator models. Figure 7.1 shows the corresponding heterogeneous OpenMP-DAG of the program shown in Listing 7.1. It shows the task or target creation precedence constraints between nodes of T_1 and nodes v_{21}, v_{31}, v_{41} and v_{Off}; the control-flow dependencies between nodes of T_1; and the data dependencies existing between v_{21}, v_{Off} and v_{41}. We distinguish between host and offloaded nodes by showing different shapes for these nodes: circle for the host nodes and square for the offloaded node. The *nowait* clause prevents the existence of an edge between nodes v_{Off} and v_{13}.

7.2 Impact of Heterogeneous Computing on the Response-Time Analysis

This section analyzes the impact that heterogeneous computing has on the response time analysis. To do so, we first describe the homogeneous response time analysis, and then, evaluate the implications of offloading a node to the accelerator device. Finally, we present an algorithm to transform the DAG representation of a real-time task, which allows to properly compute its response time upper bound.

not exist). However, we assume that v_{Off} is executed in the device since only one target task and an available device are considered in our model.

[2]By default, OpenMP implements a synchronous model in which the execution of the task encountering the target is blocked till the execution in the device finishes.

7.2.1 Starting point: *homogeneous* computing

We denoted as R^{hom} the response time upper bound of a DAG task τ running on m homogeneous cores, which can be computed as (from Equation 4.4 in Chapter 4):

$$R^{hom}(\tau) = len(G) + \frac{1}{m}\big(vol(G) - len(G)\big) \tag{7.1}$$

where $len(G)$ is the length of the DAG task, and $vol(G)$ its volume. The factor $\frac{1}{m}\big(vol(G) - len(G)\big)$ upper-bounds the *self-interference* i.e, the interference contribution from the task itself to its critical path. In order to verify the schedulability of task τ, the result provided by Equation 7.1 must be compared with τ's relative deadline D, i.e., $R^{hom}(\tau) \leq D$.

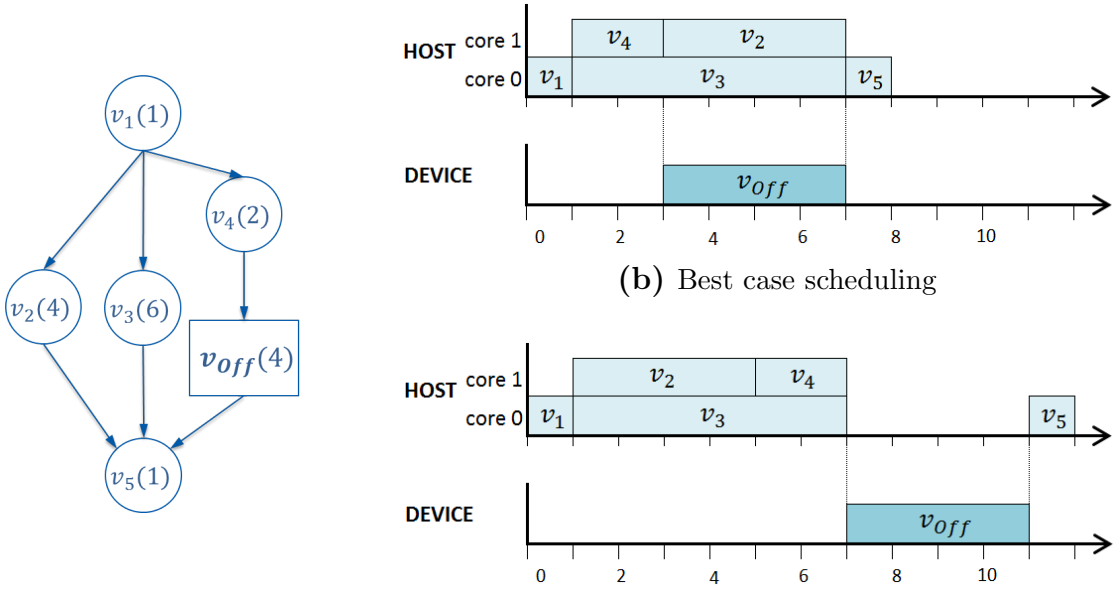
7.2.2 Towards heterogeneous computing

Clearly, *heterogeneous computing reduces the actual interference compared to homogeneous computing*, as the offloaded node does not occupy resources in the host. However, this interference reduction in the host may not imply a reduction of the response time upper bound, as the precedence constraints defined in E may defeat heterogeneous benefits.

In order to illustrate this phenomenon, consider the DAG task τ shown in Figure 7.2a composed of six nodes $v_1, \ldots v_5$, v_{Off} (with WCET shown in parenthesis). The critical path is $\{v_1, v_3, v_5\}$ (or $\{v_1, v_4, v_{Off}, v_5\}$), being $len(G) = 8$. Assuming $m = 2$, the self-interference factor is $\frac{1}{2}\big(18 - 8\big) = 5$. As a result, the response time upper bound is $R^{hom}(\tau) = 13$. Since v_{Off} does not execute in the host (see Figure 7.2b), one might subtract its contribution to the self-interference factor, being $R^{hom}(\tau) = 11$.

However, the subtraction of C_{Off} from the self-interference factor does not guarantee a trustworthy response time upper bound, because v_{Off} may not necessarily execute in parallel with the nodes running in the host. Figure 7.2c shows an alternative (and valid) scheduling in which all cores in the host remain idle while v_{Off} is running. In this case, the actual response time is 12, which is higher than the reduced response time upper bound computed above, $R^{hom}(\tau) = 11$.

Overall, the DAG portion that *potentially executes* in parallel with the offloaded node (and so reducing the interference) is *not guaranteed to actually execute* in parallel with it. Next section analyzes how to guarantee the parallel execution of the workload in the host and the offloaded node, so that the self-interference factor can be safely



(a) Heterogeneous DAG task

(b) Best case scheduling

(c) Worst case scheduling

Figure 7.2: Scheduling example of an heterogeneous DAG task. © 2018 IEEE.

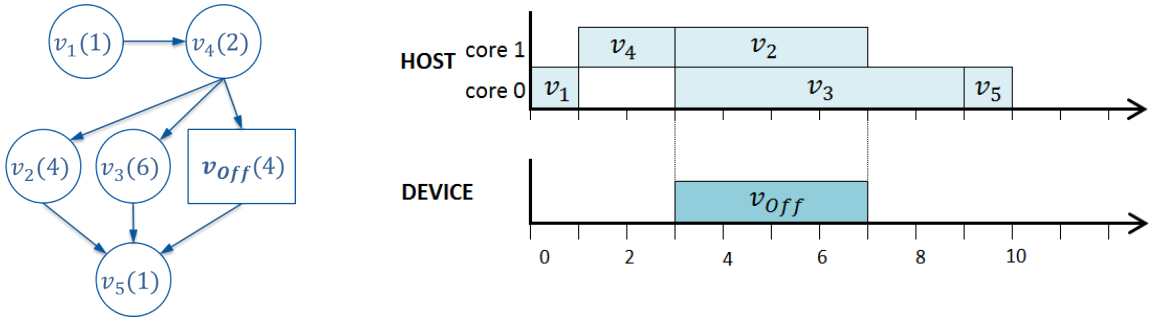
reduced.

7.2.3 Safe self-interference reduction.

In order to safely reduce the self-interference factor, it is first necessary to guarantee that there is enough workload to be executed in the host in parallel with v_{Off}. To do so, we propose an algorithm that: (1) identifies the portion of the DAG that may potentially execute in parallel with v_{Off}, named $G^{Par} = (V^{Par}, E^{Par})$, and (2) adds a synchronization point to guarantee that G^{Par} and v_{Off} actually execute in parallel.

Figure 7.3a shows the proposed transformation applied to the DAG presented in Figure 7.2a. An extra synchronization point between nodes v_4 and v_2, v_3, guarantees that v_{Off} and $\{v_2, v_3\}$ execute in parallel. Figure 7.3b shows the scheduling of the transformed DAG. Synchronization forces v_1 and v_4 to be scheduled first, avoiding the scheduling scenario shown in Figure 7.2c.

Clearly, this strategy may impact on the average performance of the tasks because: (1) the critical path can potentially enlarge (e.g., the length of the transformed DAG in Figure 7.3a is 10 instead of 8 in the original DAG) and (2) the potential parallelism is reduced due to the synchronization point (e.g., in Figure 7.3a, v_4 can not longer be executed in parallel with v_2 and v_3). Interestingly, our experiments with randomly generated DAG tasks demonstrate the opposite effect when the offloaded workload is



(a) Transformed DAG

(b) Scheduling of the transformed DAG

Figure 7.3: Transformation of the heterogeneous DAG task in Figure 7.2a. © 2018 IEEE.

large enough (see Section 7.4.2). The reason is that, ensuring the parallel execution of G^{Par} and v_{Off} avoids scheduling scenarios in which the offloaded node is running while the host processor remains idle, as shown in Figure 7.2c.

Next section introduces the algorithm to transform the DAG representation of the task, upon which a trustworthy response-time analysis supporting heterogeneous computing can be derived.

7.2.4 DAG transformation algorithm

The algorithm presented in this section considers a new restriction included in the DAG model. The transitive edges do not exist in the DAG, i.e., if $(v_1, v_2) \in E$ and $(v_2, v_3) \in E$, then $(v_1, v_3) \notin E$. This restriction does not limit the representativeness or the parallelism of real-time tasks, as it is just a property of the DAG. An easy algorithm can be used to remove transitive edges if the DAG representation of a real-time task includes them.

Given a DAG task $G = (V, E)$ with an offloaded node $v_{Off} \in V$, Algorithm 3:

1. identifies the sub-DAG $G^{Par} = (V^{Par}, E^{Par})$ that includes all the nodes executing in the host, which can potentially execute in parallel with v_{Off};
2. generates a transformed DAG $G' = (V', E')$, equivalent to G, that includes a new *synchronization node*, denoted by v_{sync}, with WCET $C_{sync} = 0$. v_{sync} is introduced just before v_{Off} and G^{Par}, so that v_{sync} guarantees that v_{Off} and G^{Par} execute in parallel.

In order to facilitate the explanation of the algorithm, consider the example shown in Figure 7.4 . Figure 7.4a shows the original DAG G, in which the synchronization

Algorithm 3 Transform Heterogeneous DAG $\tau \Rightarrow \tau'$

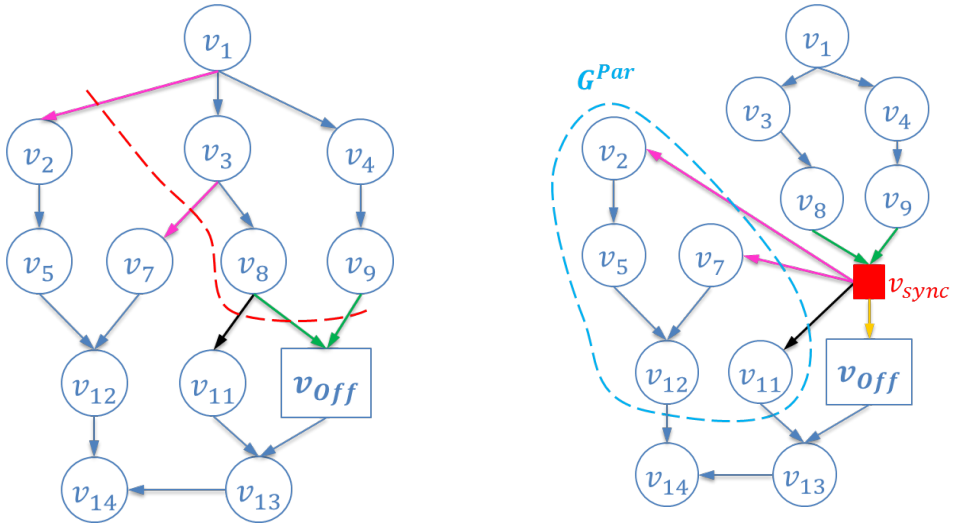
Input: $G = (V, E)$: Original heterogeneous DAG
Output: $G' = (V', E')$: Transformed heterogeneous DAG
$G^{Par} = (V^{Par}, E^{Par})$: sub-DAG with all the nodes parallel to v_{Off}

```
1  function TRANSFORM_DAG
2      Pred(v_Off) ← COMPUTE_PRED(v_Off)
3      Succ(v_Off) ← COMPUTE_SUCC(v_Off)
4      V' ← V ∪ {v_sync}
5      E' ← E
6      directPred ← ∅
7      for each (v_i, v_Off) ∈ E' do
8          directPred ← directPred ∪ {v_i}
9          E' ← E' ∪ {(v_i, v_sync)} \ {(v_i, v_Off)}
10         for each (v_i, v_j) ∈ E' do
11             if v_j ≠ v_sync then
12                 E' ← E' ∪ {(v_sync, v_j)} \ {(v_i, v_j)}
13             end if
14         end for
15     end for
16     E' ← E' ∪ {(v_sync, v_Off)}
17     for each v_i ∈ {Pred(v_Off) \ direcPred} do
18         for each (v_i, v_j) ∈ E' do
19             if v_j ∉ Pred(v_Off) then
20                 E' ← E' ∪ {(v_sync, v_j)} \ {(v_i, v_j)}
21             end if
22         end for
23     end for
24     V^Par ← V \ Pred(v_Off) \ Succ(v_Off)
25     for each (v_i, v_j) ∈ E do
26         if v_i ∈ V^Par and v_j ∈ V^Par then
27             E^Par ← E^Par ∪ {(v_i, v_j)}
28         end if
29     end for
30 end function
```

point to be included is represented with a dashed red line. Figure 7.4b shows the resultant DAG G', including the new synchronization node v_{sync} (represented as a red square node), and G^{Par} (surrounded by a blue dashed line).

Next, we describe the different phases of the algorithm:

Initialization. The algorithm first computes $Pred(v_{Off})$ (line 2), that is the set of predecessor nodes of v_{Off} (nodes from which v_{Off} can be reached), and $Succ(v_{Off})$



(a) Original DAG $G = (V, E)$ **(b)** Transformed DAG $G' = (V', E')$

Figure 7.4: Heterogeneous DAG task transformation $\tau \Rightarrow \tau'$. © 2018 IEEE.

(line 3), that is the set of successor nodes of v_{Off} (nodes reachable from v_{Off}). Then, the algorithm initializes V', which includes all the original nodes in V plus the synchronization node v_{sync}, and E', which includes all the edges in E (lines 4 and 5). A local variable *directPred* is used to store v_{Off}'s direct predecessors[3] (line 6).

Loop over v_{Off}'s direct predecessors. The first part of the algorithm (line 7) iterates over v_{Off}'s direct predecessors, denoted by v_i. At each iteration, the algorithm (1) adds v_i to *directPred*, (2) adds an edge from v_i to the extra synchronization node v_{sync} and (3) removes (v_i, v_{Off}) edge. In Figures 7.4a and 7.4b this loop operates over nodes v_8 and v_9 to remove their edge with v_{Off} and to add new edges with the new node v_{sync} (green edges). The nested loop in line 10 updates the edges between v_i and v_i's successors (parallel nodes to v_{Off}) since they are now v_{sync}'s successors. In Figures 7.4a and 7.4b this loop removes (v_8, v_{11}) and adds (v_{sync}, v_{11}), see black edges. In line 16 a new edge between the extra synchronization node v_{sync} and the offloaded node v_{Off} is added. This corresponds to the yellow edge (v_{sync}, v_{Off}) in Figure 7.4b.

Loop over other v_{Off}'s predecessors. The second part of the algorithm (line 17) iterates over all the nodes v_i from which v_{Off} can be reached, except its direct predecessors. Then, a nested loop is used to check if v_i's successors, denoted by v_j, are parallel to v_{Off} in line 19. If this is the case, then v_j is now a v_{sync}'s successor

[3] If $(v_i, v_j) \in E$ then v_i is a direct predecessor of v_j.

instead of a v_i's successor (line 20). Notice that, since transitive edges do not exist, it is not required to check if v_j is in $Succ(v_{Off})$ to determine if v_j is parallel to v_{Off}. In Figures 7.4a and 7.4b, these nested loops are used to remove edges (v_1, v_2) and (v_3, v_7) and to add (v_{sync}, v_2) and (v_{sync}, v_7), see pink edges.

Creating G^{Par}. Finally, the parallel sub-DAG G^{Par} is created. It contains all the parallel nodes to v_{Off} (line 24) and the corresponding edges involving these nodes (line 27). In Figure 7.4b, G^{Par} is surrounded by a dashed blue line.

7.3 Response-Time Analysis of Heterogeneous DAG Tasks

In this section we extend the response time analysis presented in Equation 7.1 to support heterogeneous computation. Our analysis is based on the transformed DAG task τ' in which G^{Par} and v_{Off} are guaranteed to execute in parallel. This allows to safely reduce the self-interference factor, being the new response time upper bound more accurate than R^{hom}.

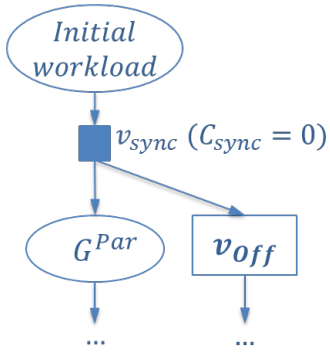
Figure 7.5a shows the generic structure of a transformed DAG task τ' and Figures 7.5b and 7.5c present the two only scheduling possibilities. That is, since there is a synchronization node v_{sync} before the execution of G^{Par} and v_{Off}, there are two possibilities:

1. the response time upper bound of G^{Par}, denoted as $R^{hom}(G^{Par})$ [4], is bigger or equal than the offloaded workload C_{Off} (see Figure 7.5b); or
2. C_{Off} is bigger than $R^{hom}(G^{Par})$ (see Figure 7.5c).

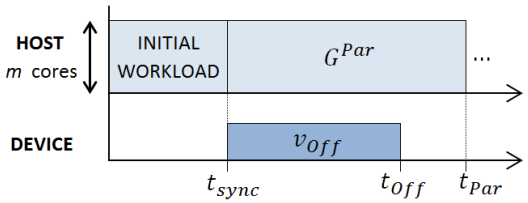
From this execution conditions, the following theorem considers three possible execution scenarios, in order to derive a new response time analysis supporting heterogeneous computing:

Theorem 3. *Consider an heterogeneous DAG task τ', with the following restrictions in its DAG representation $G' = (V', E')$: $v_{Off} \in V'$; there if an identified sub-DAG G^{Par}, containing the nodes parallel to v_{Off}; $v_{sync} \in V'$ ($C_{sync} = 0$); there exists an edge in E' between v_{sync} and all the source nodes of G^{Par}; and $(v_{sync}, v_{Off}) \in E'$. The*

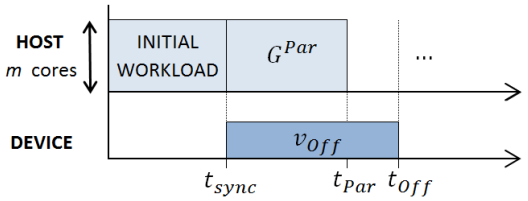
[4] $R^{hom}(G^{Par})$ is computed with Equation 7.1. Notice that, for simplicity, the input is a DAG structure G^{Par} instead of a task τ.



(a) Generic heterogeneous DAG.



(b) Scheduling scenarios 1 and 2.2.



(c) Scheduling scenario 2.1.

Figure 7.5: Scheduling possibilities of a generic heterogeneous DAG task. © 2018 IEEE.

following three execution scenarios must be considered to compute the response time upper bound of τ'*:*

- *Scenario 1.* v_{Off} *does not belong to the critical path.*

$$R^{het}(\tau') = len(G') + \frac{1}{m}\big(vol(G') - len(G') - C_{Off}\big) \tag{7.2}$$

- *Scenario 2.1.* v_{Off} *belongs to the critical path and* $C_{Off} \geq R^{hom}(G^{Par})$.

$$R^{het}(\tau') = len(G') + \frac{1}{m}\big(vol(G') - len(G') - vol(G^{Par})\big) \tag{7.3}$$

- *Scenario 2.2.* v_{Off} *belongs to the critical path and* $C_{Off} \leq R^{hom}(G^{Par})$.

$$R^{het}(\tau') = len(G') - C_{Off} + len(G^{Par}) + \frac{1}{m}\big(vol(G') - len(G') - len(G^{Par})\big) \tag{7.4}$$

Proof. The generic structure of a DAG task τ' is shown in Figure 7.5a. The synchronization node v_{sync} ($C_{sync} = 0$) guarantees that G^{Par} and v_{Off} start their execution at the same time (t_{sync}, as shown in Figures 7.5b and 7.5c).

In case of *Scenario 1*, represented in Figure 7.5b, since v_{Off} does not belong to the critical path, there exists at least one path in G^{Par} whose length is greater than C_{Off}, i.e., $len(G^{Par}) > C_{Off}$. Therefore, $R^{hom}(G^{Par}) = len(G^{Par}) + \frac{1}{m}\big(vol(G^{Par}) - len(G^{Par})\big)$ must be greater than C_{Off}, and so $t_{Par} > t_{Off}$ (see Figure 7.5b) is always

true. As a consequence, C_{Off} does not generate interference that may increase the response time of τ' and it can be safely subtracted from the self-interference factor, as done in Equation 7.2.

In case of *Scenarios 2.1 and 2.2*, since v_{Off} belongs to the critical path, none of the nodes in G^{Par} belong to it and so they contribute to the self-interference factor. In the former scenario, represented in Figure 7.5c, C_{Off} is greater (or equal) than $R^{hom}(G^{Par})$, then $t_{Par} \leq t_{Off}$ and so G^{Par} cannot generate interference that may increase the response time of τ'. Hence, its complete workload $vol(G^{Par})$ can be safely subtracted from the self-interference factor, as done in Equation 7.3. In the latter scenario, represented in Figure 7.5b, C_{Off} is smaller (or equal) than $R^{hom}(G^{Par})$, and so $t_{Off} \leq t_{Par}$. Therefore, even though v_{Off} belongs to the critical path, it does not dominate the response time of τ', but G^{Par} does instead. In this case, we can safely replace C_{Off} by $R^{hom}(G^{Par})$ in the critical path. Since the contribution of G^{Par} is also considered in the self-interference factor, $vol(G^{Par})$ can be subtracted from it, in order not to count twice for it. By replacing the mentioned terms and subtracting $vol(G^{Par})$ we obtain:

$$\begin{aligned} R^{het}(\tau) &= len(G') - C_{Off} + R^{G}(G^{Par}) + \frac{1}{m}\big(vol(G') - len(G') - vol(G^{Par})\big) \\ &= len(G') - C_{Off} + len(G^{Par}) + \frac{1}{m}\big(vol(G^{Par}) - len(G^{Par})\big) \\ &\quad + \frac{1}{m}\big(vol(G') - len(G') - vol(G^{Par})\big) \end{aligned}$$

By simplifying the terms, Equation 7.4 follows. □

It is important to remark that scenarios 2.1 and 2.2 are equivalent when $C_{Off} = R^{hom}(G^{Par})$. Hence, if starting from Equation 7.4 we replace C_{Off} by $R^{hom}(G^{Par}) = len(G^{Par}) + \frac{1}{m}\big(vol(G^{Par}) - len(G^{Par})\big)$, we rapidly reach Equation 7.3.

Theorem 3 allows to provide a response-time upper bound to DAG-based real-time tasks supporting heterogeneous computing.

7.4 Experimental Results

This section evaluates the proposed response time analysis supporting heterogeneous computing (see Theorem 3) and compares it with respect to the baseline homogeneous response time analysis (see Equation 7.1). Moreover, in order to evaluate the accuracy

of the proposed response time analysis, we compare it with respect to the minimum makespan provided by the ILP formulation presented in Appendix C.

7.4.1 Experimental setup

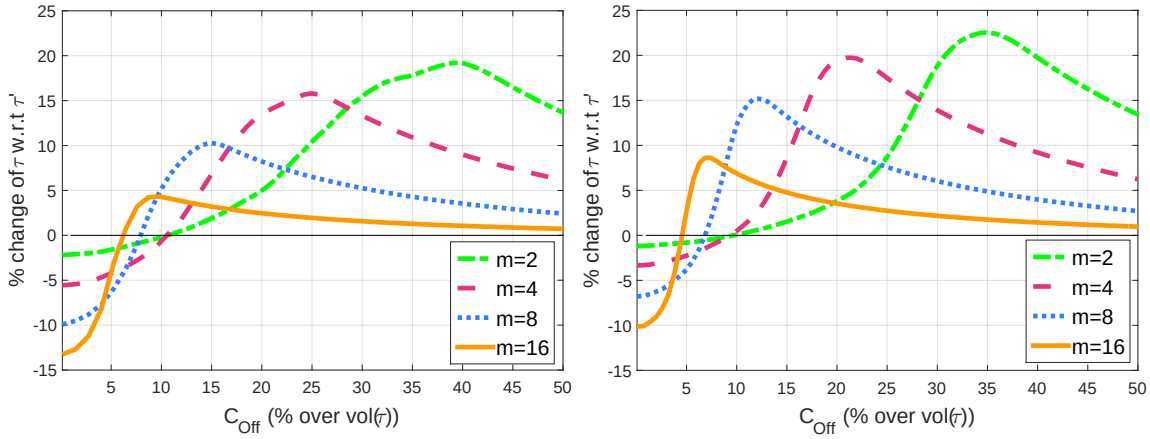
All the experiments presented in this section consider randomly generated DAG tasks, as presented in the experimental setup in Chapter 2 (Section 2.4.1). In order to include a v_{Off} node in the DAG task, the Algorithm 1, presented in Section 2.4.1, has been modified as follows: (1) *par* is randomly selected in the interval $[1, min(max_{nodes} - |V|, max_{par})]$ (line 4), i.e., the interval starts in 1 instead of 0 because at least, one extra node must be created, the offloaded node v_{Off}; (2) once the DAG task is created (line 6), and additional edges are included in the DAG (line 7), the v_{Off} node is randomly selected among all nodes in the DAG (except the source and sink nodes). As a consequence, the maximum number of nodes must be greater than, or equal to 3, $max_{nodes} \geq 3$ (source and sink nodes, and v_{Off}).

The concrete values used for the DAG tasks generation are:

- Probabilities of a branch to be expanded to a single node or to a parallel subgraph, $p_{term} = 0.5$ and $p_{par} = 0.5$, respectively.
- Probabilities of adding extra edges, $p_{dep} = 0$. Otherwise, transitive edges could be included in the DAG.
- The WCET of each node varies in the interval $[C^{min}, C^{max}] = [1, 100]$.
- C_{Off} varies in the interval $[1, C^{MAX}_{Off}]$, being C^{MAX}_{Off} a percentage (up to 60%) of DAG's volume.
- Moreover, we consider two types of heterogeneous DAG tasks:

  1. *Small* DAG tasks, with $max_{nodes} = 100$, $max_{par} = 6$ and $max_{depth} = 3$, used for the ILP solution not capable of dealing with larger tasks.
  2. *Large* DAG tasks, with $max_{nodes} = 400$, $max_{par} = 8$ and $max_{depth} = 5$.

The evaluation is carried out for different numbers of cores in the host, i.e., $m = 2$, 4, 8 or 16. For each experiment, we generate 100 heterogeneous DAG tasks for each target value of C_{Off}.



(a) $|V| \in [100, 250]$ **(b)** $|V| \in [250, 400]$

Figure 7.6: Percentage change of the average execution time of τ w.r.t. τ'.

7.4.2 Impact of the DAG transformation

This section evaluates the impact that the extra synchronization point v_{sync} has on the average performance of the transformed DAG task τ', with respect to the original DAG task τ. To do so, we simulate the execution of the original and transformed DAG tasks, assuming the *breadth-first scheduler* implemented in GOMP, the OpenMP implementation in the GNU GCC Compiler [68].

Figure 7.6 shows the percentage change[5] of the average execution time of τ with respect to τ', when varying the offloaded workload C_{Off} with respect to τ's volume, from 0.1% to 50%. This experiment considers $m = 2$, 4, 8 and 16 cores and a number of nodes $|V| \in [100, 250]$ (Figure 7.6a) and $|V| \in [250, 400]$ (Figure 7.6b). Since both figures show similar results, we focus our explanation in Figure 7.6a.

As expected, adding a synchronization node v_{sync} has a negative impact on the average performance of τ', compared to τ, when C_{Off} represents a small portion of DAG's volume (less than 11%, 10.5%, 8% and 6.5% for $m = 2$, 4, 8 and 16, respectively). The reason is that an extra synchronization point limits the parallelism. This negative impact increases as the number of cores increases, since the DAG task cannot exploit the increasing resources available to exploit parallelism. When C_{Off} represents 1% of the DAG's volume, τ is 2.2% faster than τ' for $m = 2$, and 13% faster for $m = 16$.

Surprisingly, when C_{Off} increases the trend is inverted; τ results 19.2% slower than

[5]The *percentage change* computes the relative change of two values from the same variable; in our case the average execution time.

	$\|V\| \in [100, 250]$		$\|V\| \in [250, 400]$	
	τ' over τ	$\%C_{Off}$	τ' over τ	$\%C_{Off}$
$m = 2$	19.2%	39.6	22.5%	34.7
$m = 4$	15.8%	25	19.7%	21.2
$m = 8$	10.3%	14.9	15.1%	12.2
$m = 16$	4.3%	9.7	8.62%	7.3

Table 7.1: Maximum percentage change of the average execution time of τ w.r.t. τ'.

τ' for $m = 2$ when C_{Off} represents the 39.6% of DAG's volume, and 4.3% slower for $m = 16$ when C_{Off} represents the 9.7%. The reason is that v_{sync} guarantees that the host processor is not idle while executing v_{Off} (see Figure 7.2c). Table 7.1 shows the maximum average performance benefit of τ' over τ, and the value of C_{Off} for which it was observed, for each value of m and for both intervals of $|V|$. The performance benefit of v_{sync} decreases as m increases because the self-interference factor has less impact as the number of cores increases (see Theorem 3).

The exact same trend is observed in Figure 7.6b, that shows the results for a different interval for the number of nodes, $|V|$. Comparing Figures 7.6a and 7.6b demonstrate that, as the number of nodes increases, the peak benefit of the DAG transformation is higher, and this peak is reached when the percentage of offloaded workload is smaller. Also, the degradation for small values of C_{Off} is smaller as the number of nodes increases.

Finally, it is worth noting that for higher values of C_{Off}, the difference between τ and τ' performance seems to decrease. However, the absolute difference remains constant. As C_{Off} increases it becomes the dominant factor in τ and τ' execution times and so both equally increase as well. The trend of the percentage of an absolute difference with respect to an increasing time is to decrease.

7.4.3 Accuracy of the response time analysis

This section analyzes the accuracy of R^{het} (Equations 7.2, 7.3, 7.4) and R^{hom} (Equation 7.1) with respect to the minimum makespan of a heterogeneous DAG task. To do so, we have developed an ILP model, presented in Appendix C, that computes such a minimum makespan, i.e., the minimum time interval needed to execute a given heterogeneous DAG task on m cores and a device. The ILP formulation has been coded and solved with the IBM ILOG CPLEX Optimization Studio [90].

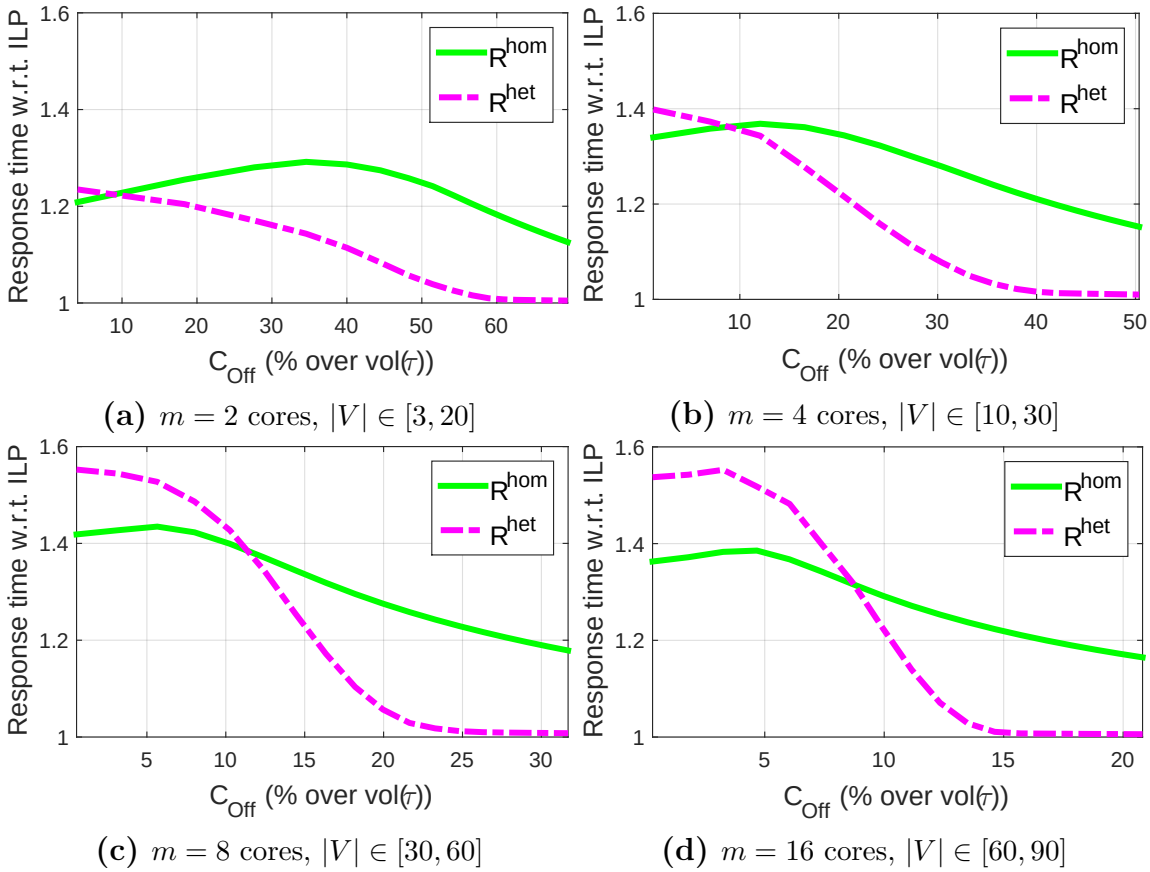


Figure 7.7: Increment of $R^{hom}(\tau)$ and $R^{het}(\tau')$ w.r.t. the minimum makespan of τ. Notice that x-axes are different.

Given the ILP complexity, we only consider different subsets of *Small* DAG tasks for which the ILP solver is able to provide an optimal solution in less than 12 hours.

Figure 7.7 shows the increment of the response time upper bound provided by $R^{hom}(\tau)$ and $R^{het}(\tau')$ with respect to the minimum makespan of τ computed by the ILP solver, when varying C_{Off} with respect to τ's volume. We evaluated 2, 4, 8 and 16 cores (Figures 7.7a, 7.7b, 7.7c and 7.7d, respectively).

When C_{Off} represents less than 5% of $vol(\tau)$, $R^{het}(\tau')$ is around 23%, 40%, 54% and 57% higher than the minimum makespan for $m = 2$, 4, 8 and 16, respectively. This pessimism however decreases as C_{Off} increases, being around 1% when C_{Off} represents more than 56%, 40%, 23% and 15% of $vol(\tau)$, for $m = 2, 4, 8$ and 16, respectively. The reason is that C_{Off} becomes the dominant factor of $R^{het}(\tau')$ and so G^{Par} is not relevant any more (see Figure 7.5c).

$R^{hom}(\tau)$ provides more accurate results than $R^{het}(\tau')$ when C_{Off} represents less than 9%, 9%, 11.4% and 8.8% of $vol(\tau)$ for $m = 2, 4, 8$ and 16, respectively. The

reason of this trend, as also shown in Section 7.4.2, is that v_{sync} impacts negatively on both, average and upper bound response time. This trend however is inverted when C_{Off} increases, and so $R^{het}(\tau')$ provides more accurate results than $R^{hom}(\tau)$. For instance, when $R^{het}(\tau')$ provides a response time only 1% higher or less than the minimum makespan, $R^{hom}(\tau)$ is around 20% higher, for all the cores configuration.

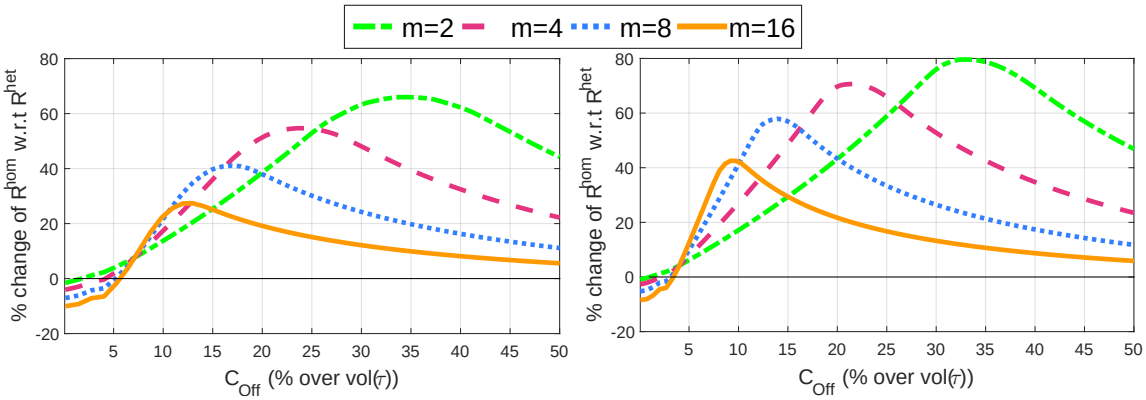
7.4.4 Homogeneous vs. Heterogeneous

This section further compares the accuracy of our response time analysis $R^{het}(\tau')$, with respect to $R^{hom}(\tau)$, considering *Large* DAG tasks, with up 400 nodes.

Figure 7.8 shows the percentage change of $R^{hom}(\tau)$ with respect to $R^{het}(\tau')$, when varying C_{Off} with respect to $vol(\tau)$ from 0.1% to 50%. This experiment considers a host processor featuring $m = 2, 4, 8$ and 16 cores, and a number of nodes $|V| \in [100, 250]$ (Figure 7.8a) and $|V| \in [250, 400]$ (Figure 7.8b). Following the same trend observed in the previous section, our response time analysis $R^{het}(\tau')$ improves over $R^{hom}(\tau)$, when considering *Large* DAG tasks. This improvement increases as C_{Off} increases due to self-interference factor reduction. R^{hom} only outperforms R^{het} for small values of C_{Off} due to the negative impact of the synchronization point. Concretely, for $|V| \in [100, 250]$ this occurs when C_{Off} represents less than 1.6%, 4.2%, 5.2% and 5.6% over $vol(\tau)$ for $m = 2$, 4, 8 and 16, respectively. For $|V| \in [250, 400]$ this occurs when C_{Off} represents less than 1%, 2.1%, 3.1% and 3.4% over $vol(\tau)$ for $m = 2$, 4, 8 and 16, respectively. Notice that, as m increases the benefit of $R^{het}(\tau')$ is smaller, because the self-interference factor is divided by m (see Equations 7.2 to 7.4).

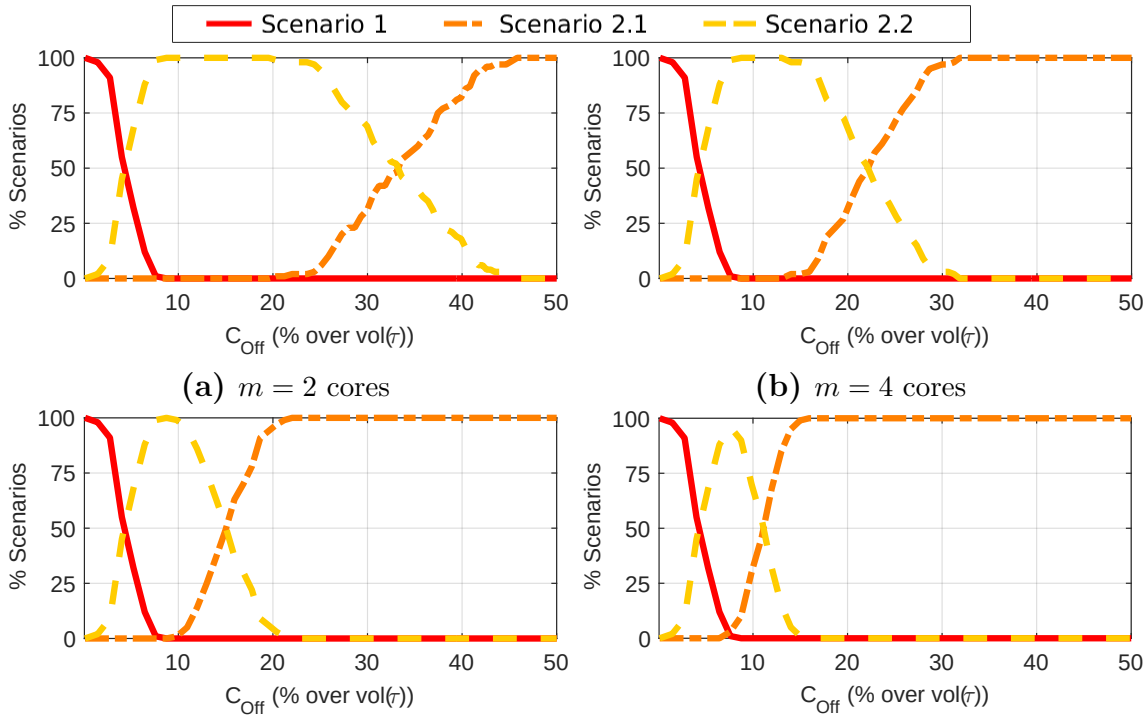
The maximum benefit observed is the following: when $|V| \in [100, 250]$, $R^{hom}(\tau)$ is $66\%, 54.7\%, 41\%$ and 27.4% higher than $R^{het}(\tau')$ for $m = 2, 4, 8$ and 16, respectively; when $|V| \in [250, 400]$, $R^{hom}(\tau)$ is 79.6%, 70.6%, 58% and 42.5% higher than $R^{het}(\tau')$ for $m = 2, 4, 8$ and 16, respectively. Results presented in Figure 7.8 correspond to an average response time upper bound over all generated DAG tasks. However, the maximum observed difference between $R^{hom}(\tau)$ and $R^{het}(\tau')$, when $|V| \in [100, 250]$, is 92.6%, 82.4%, 67.6% and 50.4% for $m = 2$, 4, 8 and 16, respectively. When $|V| \in [250, 400]$, these values are 95%, 88.2%, 76.1% and 60%.

In order to better understand the benefits brought by $R^{het}(\tau')$, it is important to understand the execution scenarios presented in Theorem 3. Figure 7.9 shows the occurrence percentage of the execution scenarios, when varying the percentage of



(a) $|V| \in [100, 250]$ **(b)** $|V| \in [250, 400]$

Figure 7.8: Percentage change of $R^{hom}(\tau)$ w.r.t. $R^{het}(\tau')$.



(a) $m = 2$ cores **(b)** $m = 4$ cores

(c) $m = 8$ cores **(d)** $m = 16$ cores

Figure 7.9: Percentage of scenarios occurrence, $|V| \in [100, 250]$.

C_{Off} over $vol(\tau)$ from 0.1% to 50%. The number of nodes $|V|$ is randomly selected in $[100, 250]$ (similar trends are observed when $|V| \in [250, 400]$). This experiment also considers a host processor featuring $m = 2, 4, 8$ and 16 cores (Figures 7.9a, 7.9b, 7.9c and 7.9d, respectively).

Scenario 1 is the dominant one when the percentage of C_{Off} over $vol(\tau)$ is less than 5%. This scenario corresponds to the case in which v_{Off} does not belong to the critical

path and therefore, is independent of m. From that point on, scenario 2.2 becomes more relevant as v_{Off} belongs to the critical path, but C_{Off} is still smaller than the response time of G^{Par}. When C_{Off} becomes higher that $R^{hom}(G^{Par})$, occurrences of scenario 2.1 increase. As m increases, occurrences of scenario 2.1 start to increase earlier because higher parallelism can be exploited in the host, and so $R^{hom}(G^{Par})$ becomes smaller.

Interestingly, the intersection of scenarios 2.1 and 2.2, i.e., when $C_{Off} = R^{hom}(G^{Par})$ (and so Equations 7.3 and 7.4 are equivalent), results in the maximum benefit of R^{het} with respect to R^{hom} (shown in Figure 7.8a). This occurs when C_{Off} is 33%, 22%, 15% and 11% over $vol(\tau)$ for $m = 2, 4, 8$ and 16, respectively. The reason is that, in this particular case, utilization of both host and device is maximized, i.e., there are less idle times.

7.5 Related Work

When considering heterogeneous architectures, real-time tasks have been traditionally modeled as self-suspending tasks, i.e., tasks that contains a region which is executed in an external device. The execution of the task is suspended until the completion of the external operations in the device.

Most of the published work considers that self-suspended tasks are scheduled on a uniprocessor platform and utilizes a device to accelerate part of the execution. Unfortunately, it has been shown that many previous works concerning the analysis of self-suspending tasks are flawed. Chen et al. [116] presented a complete review of self-suspending tasks theory and an explanation of the existing misconceptions.

When considering multiprocessor architectures, Liu and Anderson [117] analyzed self-suspending task systems and proposed a schedulability test for global earliest-deadline-first (EDF) scheduling. The schedulability test by Liu et al. [118] considers partitioned scheduling for harmonic tasks with suspensions, which have periodic job arrivals. Chen, Huang and Liu [119] studied global rate-monotonic scheduling of dynamic self-suspending tasks, and proposed a utilization-based schedulability analysis.

Finally, Biondi et al. [120] designed a framework to support real-time systems on FPGAs and provide a response time analysis to verify the schedulability of a set of tasks with software parts and hardware accelerated functions.

7.6 Summary

This chapter presents a novel response time analysis supporting heterogeneous computing. It allows to verify the schedulability of a DAG task that offloads part of its computation to an accelerator device. To do so, we first identify the portion of the DAG running in the host (named G^{Par}) that can potentially execute in parallel with the workload offloaded to the device (named v_{Off}). Secondly, we propose a DAG transformation to guarantee the parallel execution of G^{Par} and v_{Off}. Our response time analysis is built upon this transformation.

Interestingly, besides the timing guarantees provided, this DAG transformation also results in higher average performance when the offloaded workload represents more than 10% of the DAG's volume. The reason is that the probability of a scheduling scenario in which the host processor is idle waiting for the device to finish, is reduced.

Our results reveal that the proposed heterogeneous response time analysis significantly outperforms the homogeneous presented in Section 4.3 (up to 80% in average and 95% observed) when C_{Off} is large enough (more than 5% of the task volume). Moreover, for small DAG tasks (up to 100 nodes), we demonstrate that our response time upper bound is comparable to the minimum makespan derived with an ILP solution.

Overall, the benefit of using a specific response time analysis for heterogeneous architectures has been demonstrated. This benefit is higher for (a) a small number of cores in the host processor and (b) larger DAG tasks. Moreover, the heterogeneous response time analysis is favorable when the portion of workload executed in the device is sufficiently large, which complies with the heterogeneous computing philosophy.

The work presented in this chapter facilitates the use of the OpenMP accelerator model into real-time systems, to support heterogeneous architectures. However, it remains as future work the support for multiples regions within the tasks that can be offloaded to a device (or multiple devices), and the analysis when several heterogeneous DAG tasks are considered within a real-time system.

Chapter 8

Discussion

"I never am really satisfied that I understand anything; because, understand it well as I may, my comprehension can only be an infinitesimal fraction of all I want to understand."

— Ada Lovelace

8.1 Conclusions

Critical real-time embedded systems are increasingly implementing advanced functionalities that require more powerful computing platforms to provide higher performance, while guaranteeing the predictability requirements of the system. Parallel computing is fundamental to achieve the required level of performance, and parallel programming models are of paramount importance to exploit the huge computational capabilities of current and future parallel embedded architectures targeting real-time systems. This is a challenging task that requires a real convergence of high-performance and embedded domains, impacting at all levels of the design flow and execution stack.

This thesis tackles the use of the OpenMP task-based parallel programming model to develop future critical real-time embedded systems. Concretely, this thesis analyzes the time predictability properties of the OpenMP tasking and accelerator models. OpenMP was created for a very different purpose than implementing real-time applications. However, its syntax and execution model have certain similarities with real-time formalisms, such as the DAG scheduling model, that could make it a good candidate to fill the existing gap between: a convenient programming model for paral-

lel embedded architectures and the scheduling analysis techniques required to provide timing guarantees.

We first present an analysis of the OpenMP specification that shows the benefits and implications of developing and parallelizing real-time systems with OpenMP. We propose a set of implementation guidelines and OpenMP extensions to support the timing and scheduling requirements of real-time tasks, like the notion of recurrence or the need for priorities. We conclude that OpenMP is an excellent candidate to develop critical real-time systems, although some modifications on the OpenMP specification must be specifically addressed to guarantee the timing behavior of the system.

Then, we provide a deeper analysis of concrete features of the OpenMP tasking model, the *tied* and *untied* tasks and their scheduling constraints, that directly impact on the timing predictability of the real-time tasks. We provide a schedulability analysis for the untied tasking model, and show the difficulties of deriving timing guarantees for the tied model. We conclude that the use of *untied* tasks is preferable for parallel applications in the real-time context.

Moreover, we show the similarities between the OpenMP execution model and the limited preemptive scheduling strategy. Therefore, we extend the current state of the art on the schedulability analysis for the sporadic DAG tasks model under the limited preemptive scheduling. We develop a novel response time analysis considering two well-known variants of this scheduling strategy, *eager* and *lazy*. We evaluate the proposed analysis with synthetic workloads and conclude that the eager approach provides better schedulability results, being the *lazy* approach very inefficient. This also demonstrates that specific methodologies must be developed when considering parallel execution in real-time systems, since a totally different conclusion was reached when considering task-sets composed of sequential tasks, for which eager and lazy are incomparable strategies. The reason is that there are task-sets composed of sequential tasks for which eager fits better than lazy, and vice versa.

We also demonstrate the usability of our response time analysis with real use cases, an AUTOSAR application and a real-time system implemented and parallelized with OpenMP. We evaluate the systems and reinforce the conclusions obtained with synthetic workloads. Moreover, we further demonstrate the effectiveness of using OpenMP in real-time systems, as timing guarantees can be provided.

Finally, we provide a response time analysis for a restricted model when heterogeneous architectures are considered. This analysis targets the OpenMP accelerator

model, used to efficiently offload the most computationally intensive functionalities to accelerator devices. Our analysis considers an OpenMP application with a single offload operation. Our evaluation with synthetic DAG tasks let us conclude that the proposed analysis clearly outperforms the response time analysis for homogeneous architectures. Moreover, a DAG transformation, required for the analysis, leverages an average execution time improvement, since idle times in the host are avoided.

Overall, we envision a promising future in the adoption of OpenMP in critical real-time systems. As a proof of concept, the next section presents the impact of this thesis, which demonstrates the interest of both the academia and the industry, on this topic.

8.2 Impact

The work done in this thesis is having an impact, not only within the BSC, but also in the international community.

This thesis contributed to the European FP7 project *Parallel Software Framework for Time-Critical Many-core Systems (P-SOCRATES)* (http://p-socrates.github.io/) [21], which developed methodologies and tools for implementing time-predictable high-performance applications. The characterization of the OpenMP tasking model, and the response time analysis was key to analyze the timing predictability properties of the use cases considered in the project.

Moreover, the contributions and results of this thesis opened new research lines in the distributed computing domain. This is the case of the European Horizon 2020 projects, *Edge and Cloud Computation: A Highly Distributed Software for Big Data Analytics (CLASS)* (2018-2020) [121] and *A Software Architecture for Extreme-Scale Big-Data Analytics in Fog Computing Ecosystems (ELASTIC)* (2018-2021). Both aim to develop novel software architectures to efficiently distribute data and process mining along the compute continuum (from edge to cloud resources), while providing sound real-time guarantees imposed by automotive (CLASS) and railway (ELASTIC) systems. The response time analysis techniques developed in this thesis will be used for two purposes: (1) provide timing guarantees to the parallel computation in the edge nodes, and (2) characterize the timing behavior of the distributed computation, which will be implemented with a task-based framework, COMPSs [122].

The work conducted in the scope of this thesis had also an impact on three

industrial projects. The first project, *Increasing the Guaranteed Performance in Many-core Heterogeneous Architectures*, (2016-20017), with the participation of BSC and DENSO AUTOMOTIVE Deutschland GmbH (Germany), investigated parallel programming models, scheduling and timing estimation techniques to obtain high-performance and tight response-time bounds of parallel computation for automotive Advanced Driver Assistance Systems (ADAS). The second project, *Parallel Programming Models for Space Systems* (2015-2016), was a contract with the European Space Agency (ESA), BSC and Evidence Srl (Italy). This project aimed to study the benefits of using the OpenMP tasking model in space systems in order to improve performance speed-up and increase programmability, while still providing timing analyzability. The third industrial project, *High Performance Parallel Payload Processing for Space* (*HP4S*) (2018-2019), with the participation of BSC, Airbus Defense and Space (France, UK), and the ESA, aims to further evaluate the use of OpenMP in architectures that will be qualified to be used in the space domain in a short term. In all these projects, the timing analysis techniques targeting OpenMP and developed in this thesis, were or are fundamental to motivate the use of OpenMP in these domains.

The research conducted in this thesis towards the adoption of OpenMP in real-time systems also promoted a Master and a PhD student within the BSC, and enrolled in the Universitat Politècnica de Catalunya, to continue this research. They will investigate further timing-related issues, and runtime and compiler implementation requirements for adopting OpenMP in real-time systems.

Finally, this research line is having a significant impact in the OpenMP language committee. This work has been presented to the OpenMP architecture review board (ARB), motivating the creation of a discussion group to tackle real-time aspects in the OpenMP specification. We are currently defining the new features to be included in the OpenMP specification to allow the development of critical real-time systems.

8.3 Future Work

The research work conducted in this thesis poses the basis for further research on timing analysis techniques focused on OpenMP. Below, we describe the main future research lines that emanate from this thesis.

The most immediate action, which already started, is the discussion within the OpenMP language committee to address real-time features in the OpenMP specifi-

cation. Usability and expressiveness factors, or implementation issues must be considered for this research line. Interestingly, the `event` clause, and an event-driven model, are being considered as a useful feature, not only for the real-time community, but also for the HPC community.

An important and challenging research line is to consider and evaluate more realistic systems. This includes platforms, runtime implementations, compilers, analysis tools, use cases, etc. Fortunately, the industry is more and more interested in this topic, as seen in the previous sections, and this facilitates the access to real software and hardware setups.

When considering heterogeneous architectures, the work presented in this thesis must be extended to consider a more flexible model. The final goal is to study the interaction of a set of real-time tasks when accessing concurrently to one or more accelerator devices.

In our view, this thesis offers an excellent source for future works to explore the opportunities that parallel programming models offer to real-time systems, and how current parallel models can be extended to fulfill the needs of real-time systems.

Appendix A

Eager Limited Preemptive Blocking Time Factors

Section 5.3.2.2 presents a technique to compute the blocking time of lower priority tasks under the eager limited preemptive scheduling strategy. This appendix presents the algorithm and ILP formulations to compute the needed factors: (1) the worst-case workload generated by each lower-priority task τ_i (i.e., μ_i, see Equation (5.9)), and (2) the overall worst-case workload of lower-priority tasks for each execution scenario $s_l \in e^m$ (i.e., $\rho_k[s_l]$, see Equation (5.10)). The former can be computed at compile-time for each task, and is independent from the task-set; the latter requires the complete task-set knowledge, and is computed at system integration time. The last section of this appendix describes the complexity of these algorithms, and the total complexity to compute the lower priority blocking time for this technique.

A.1 Worst-case workload of τ_i executing in c cores

$\mu_i[c]$ represents the worst-case workload of a task τ_i executing in c cores, and is determined by the sum of the WCET of the c longest nodes of τ_i that can execute in parallel (see Definition 10 in Chapter 5). This is computed in two steps: (1) we identify for each node, the set of nodes that can execute in parallel with it; and (2) we compute the worst case blocking time when different number of cores are considered.

Algorithm 4 Parallel nodes of τ

Input: $G = (V, E)$: DAG task
$TopolOrder$: Topological order of G
$Sibling(v_j),\ \forall v_j \in V$: Set of sibling nodes of v_j
$Succ(v_j),\ \forall v_j \in V$: Set of successor nodes of v_j
$Pred(v_j),\ \forall v_j \in V$: Set of predecessor nodes of v_j
Output: $Par(v_j), \forall v_j \in V$: Set of nodes parallel to v_j

```
1  function PARALLEL_NPR
2      for each v_j ∈ V do
3          Par(v_j) ← ∅
4          for each v_l ∈ Sibling(v_j) do
5              if (v_j, v_l) ∉ E and (v_l, v_j) ∉ E then
6                  s ← Succ(v_l) \ Succ(v_j)
7                  Par(v_j) ← Par(v_j) ∩ {{v_l} ∪ s}
8              end if
9          end for
10     end for
11     for each v_j ∈ TopolOrder do
12         for each v_l ∈ Pred(v_j) do
13             p ← Par(v_l) \ Pred(v_j)
14             Par(v_j) ← Par(v_j) ∪ p
15         end for
16     end for
17 end function
```

A.1.1 Computing the set of parallel nodes

Given the DAG $G_i = (V_i, E_i)$, Algorithm 4 computes, for each node $v_{i,j} \in V_i$, the set of nodes in V_i that can execute in parallel with $v_{i,j}$. Notice that the i subscript has been omitted.

The algorithm takes as input the DAG representation of the task τ, i.e., $G = (V, E)$, the topological order of G, and for each node v_j, the sets: (1) $Sibling(v_j)$, the nodes that have a common predecessor with v_j, (2) $Succ(v_j)$, the nodes reachable from v_j and (3) $Pred(v_j)$, the nodes from which v_j can be reached. The algorithm computes for each $v_j \in V$, the set $Par(v_j)$, that contains the nodes that can execute in parallel with v_j.

The algorithm iterates two times over all the nodes in V. The first loop (lines 2-10) adds to $Par(v_j)$ the set of sibling nodes v_l that are not connected to v_j by an edge, and the nodes reachable from v_l, i.e., $Succ(v_l)$, discarding those connected to

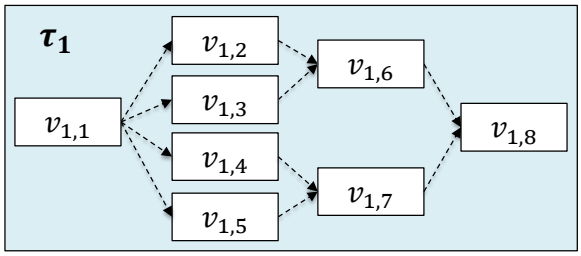


Figure A.1: Example of DAG task.

v_j by an edge. The second loop (lines 11-16), which traverses V in topological order, adds to $Par(v_j)$ the set of nodes $Par(v_l)$, previously computed, being v_l a predecessor node of v_j. From $Par(v_l)$, we discard the predecessor nodes of v_j.

As an example, consider the node $v_{1,3}$ of the DAG task τ_1 shown in Figure A.1. The first loop iterates over the sibling nodes $v_{1,2}$, $v_{1,4}$ and $v_{1,5}$. None of them is connected to $v_{1,3}$ by an edge, so they are included in $Par(v_{1,3})$. The first loop also considers the sets $Succ(v_{1,2}) = \{v_{1,6}, v_{1,8}\}$, $Succ(v_{1,4}) = \{v_{1,7}, v_{1,8}\}$ and $Succ(v_{1,5}) = \{v_{1,7}, v_{1,8}\}$. The algorithm discards from $Succ(v_{1,2})$ the nodes $\{v_{1,6}, v_{1,8}\}$, since they are successors of $v_{1,3}$ and can not be executed in parallel. This is not the case of node $v_{1,7} \in Succ(v_{1,4})$, which is included in $Par(v_{1,3})$. Hence, we obtain $Par(v_{1,3}) = \{v_{1,2}, v_{1,4}, v_{1,5}, v_{1,7}\}$. The second loop does not add new nodes to $Par(v_{1,3})$ because the unique predecessor node of $v_{1,3}$ is $v_{1,1}$, and $Par(v_{1,1}) = \emptyset$. However, when the second loop iterates over node $v_{1,7}$, the two sets $Par(v_{1,4})$ and $Par(v_{1,5})$ are considered, since $v_{1,4}, v_{1,5} \in Pred(v_{1,7})$. Then, nodes $v_{1,2}$, $v_{1,3}$ and $v_{1,6}$ are included in $Par(v_{1,7})$, since none of them belongs to $Pred(v_{1,7})$.

A.1.2 Worst-case workload of parallel nodes in c cores

This section presents an ILP formulation to compute $\mu_i[c]$, i.e., for any task τ_i, the sum of the c longest nodes in V_i that, when executed in parallel in c cores, generate the worst-case workload.

Input parameters

1. c, the number of cores used by τ_i.
2. $v_{i,j} \in V_i$, the nodes of τ_i.
3. $C_{i,j}$, the WCET of each node.

4. $IsPar_{i,j,k} \in (0,1)$, a binary variable that takes the value 1 if $v_{i,j}$ and $v_{i,k}$ can execute in parallel, 0 otherwise.

Problem variables

1. $b_j \in (0,1)$, a binary variable that takes the value 1 if $v_{i,j}$ is one of the selected parallel node, 0 otherwise.
2. $b_{j,k} = b_j \wedge b_k, b_{j,k} \in (0,1), j \neq k$, an auxiliary binary variable.

Constraints

1. Only c nodes can be selected:

$$\sum_{j=1}^{|V_i|} b_j = c$$

2. The selected nodes can execute in parallel:

$$\sum_{j=1}^{|V_i|} \sum_{k=j+1}^{|V_i|} b_{j,k} IsPar_{i,j,k} = c$$

3. This is an auxiliary constraints used to model the logical *and*:

$$b_{j,k} \geq b_j + b_k - 1; b_{j,k} \leq b_j; b_{j,k} \leq b_k$$

Objective function. The objective function aims to maximize the WCET of the c nodes that can execute in parallel, i.e.,

$$max \sum_{j=1}^{|V_i|} C_{i,j} b_j$$

A.2 Overall worst-case workload of $lp(k)$ under the execution scenario s_l

$\rho_k[s_l]$ represents the overall worst-case workload generated by the set of lower priority task $lp(k)$ under the execution scenario $s_l \in e^m$ (see Definition 12 in Chapter 5). This section presents an ILP formulation to compute $\rho_k[s_l]$.

Parameters

1. $lp(k)$, the set of lower priority tasks.
2. m, the number of available cores.
3. $s_l \in e^m$, the execution scenario.
4. $\mu_i[c], \forall \tau_i \in lp(k), \forall c = 1 \dots m$, the worst-case workload of parallel nodes of τ_i executing in c cores.

Problem variable

1. w_i^c, a binary variable that takes the value 1, when $\mu_i[c]$ contributes to the overall worst-case workload, 0 otherwise.

Constraints

1. The number of tasks contributing to the overall worst-case workload must be equal to the size of the execution scenario:

$$\sum_{c=1}^{m} \sum_{\forall \tau_i \in lp(k)} w_i^c = |s_l|$$

2. A task can be considered at most in one execution scenario:

$$\forall \tau_i \in lp(k), \sum_{c=1}^{m} w_i^c \leq 1$$

3. For each number of cores considered in s_l, there exist at least one $\mu_i[c]$ that is selected:

$$\sum_{\forall \tau_i \in lp(k)} w_i^c \geq 1, c \in s_l$$

4. The sum of number of cores considered is m:

$$\sum_{c=1}^{m} \sum_{\forall \tau_i \in lp(k)} w_i^c \cdot c = m$$

Objective function. The objective function aims to maximize worst-case workload contribution of the tasks $\tau_i \in lp(k)$ in the execution scenario, i.e.,

$$max \sum_{c=1}^{m} \sum_{\forall \tau_i \in lp(k)} w_i^c \mu_i^c$$

A.3 Complexity

Algorithm 4 requires to specify for each node in V_i the sets $Sibling$, $Succ$ and $Pred$, that can be computed in quadratic time on the number of nodes. Similarly, the complexity of Algorithm 4 is quadratic on the size of the DAG task, i.e., $O(|V_k|^2)$. The ILP formulation to compute $\mu_i[c]$ is performed for each task (except for the highest-priority one), and the number of cores ranges from 2 to m (when $c = 1, \mu_i[1] = max_{1 \leq j \leq q_{i+1}} C_{i,j}$), hence the complexity cost is $O(nm) \cdot O(ilp_A)$. It is important to remark that Algorithm 4 and the ILP that computes $\mu_i[c]$ are executed for each task, and are independent of the task-set and the system where they execute.

$\rho_k[s_l]$ is computed for the execution scenarios e^m and e^{m-1}, and for each task τ_k (except for the lowest-priority task τ_n), hence the complexity cost is: $O(n \cdot p(m)) \cdot O(ilp_B) + O(n \cdot p(m-1)) \cdot O(ilp_B)$. The cost of solving both ILP formulations is pseudo-polynomial, if the number of constraints is fixed [123]. Our ILP formulations have fixed constraints, with a function cost of $O(ilp_A)$ and $O(ilp_B)$ depending on $|V_k|$ and $(m \cdot n)$, respectively.

Therefore, the cost of computing $\rho_k[s_l]$ for e^m dominates the cost of other operations; hence, the complexity of computing the lower-priority blocking time is pseudo-polynomial in the number of tasks and execution scenarios, i.e., cores.

Appendix B

Benchmarks Source Code

This appendix contains the source code of the most representative function of the OpenMP applications used for the experimental evaluation in Section 6.1 of this thesis. Concretely, section B.1 presents a pre-processing sampling application for infra-red H2RG detectors, from the space domain. Sections B.2 and B.3 present to different application but both useful in the automotive domain to support advanced vehicle functionalities: a pedestrian detector and a cholesky factorization, respectively.

B.1 Pre-processing for infra-red detectors

```
1  void preProcessingFixedPoint()
2  {
3    initialization();
4    #pragma omp parallel
5    #pragma omp single nowait
6    {
7      INT32BIT groupNumber=1;
8      int i=0, j=0;
9      for (groupNumber=1; groupNumber<=numberOfGroupsPerExposure;
10          groupNumber++)
11     {
12       #pragma omp taskwait
13       for (i=0; i < DIM_Y; i++) {
14         for (j=0; j < DIM_X; j++) {
15           #pragma omp task firstprivate(i, j) \
16                            depend(in: currentFrame[i][j]) \
17                            depend(in: saturationLimit) \
18                            depend(inout: saturationFrame[i][j])
19           detectSaturation(currentFrame[i][j], saturationLimit,
20                            saturationFrame, i, j);
21         }
22       }
23       for (i=0; i < DIM_Y; i++) {
```

```
24          for (j=0; j < DIM_X; j++) {
25            #pragma omp task firstprivate(i, j) \
26                             depend(in: biasFrame[i][j]) \
27                             depend(inout: currentFrame[i][j])
28            subtractSuperBias(currentFrame[i][j], biasFrame, i, j);
29          }
30        }
31        for (i=0; i < DIM_Y; i++) {
32          for (j=0; j < DIM_X; j++) {
33            #pragma omp task firstprivate(i, j) \
34                             depend(in: coeffOfNonLinearityPolynomial) \
35                             depend(inout: currentFrame[i][j])
36            nonLinearityCorrectionPolynomial(currentFrame[i][j],
37                                 coeffOfNonLinearityPolynomial, i, j);
38          }
39        }
40        #pragma omp taskwait
41        for (j=0; j < DIM_X; j++) {
42          #pragma omp task firstprivate(j) depend(in: dummy)
43          subtractReferencePixelTopBottom(currentFrame[i][j], j);
44        }
45        #pragma omp task depend(inout: dummy)
46        subtractReferencePixelSides(currentFrame[i][j]);
47        for (i=0; i < DIM_Y; i++) {
48          for (j=0; j < DIM_X; j++) {
49            #pragma omp task firstprivate(i, j, groupNumber) \
50                             depend(in: dummy) \
51                             depend(in: currentFrame[i][j]) \
52                             depend(in: sumXYFrame[i][j]) \
53                             depend(in: sumYFrame[i][j]) \
54                        depend(inout: offsetCosmicFrame[i][j]) \
55                        depend(inout: numberOfFramesAfterCosmicRay[i][j])
56            detectCosmicRay(currentFrame[i][j], sumXYFrame, sumYFrame,
57                        offsetCosmicFrame, numberOfFramesAfterCosmicRay,
58                        groupNumber, i, j);
59          }
60        }
61        for (i=0; i < DIM_Y; i++) {
62          for (j=0; j < DIM_X; j++) {
63            #pragma omp task firstprivate(i, j, groupNumber) \
64                             depend(in: currentFrame[i][j]) \
65                             depend(in: offsetCosmicFrame[i][j]) \
66                             depend(in: saturationFrame[i][j]) \
67                             depend(inout: sumXYFrame[i][j]) \
68                             depend(inout: sumYFrame[i][j])
69            progressiveLinearLeastSquaresFit(currentFrame[i][j],
70                             sumXYFrame, sumYFrame, offsetCosmicFrame,
71                             saturationFrame, groupNumber, i, j);
72          }
73        }
74      }
75      for (i=0; i<DIM_Y; i++) {
76        for (j=0; j < DIM_X; j++) {
77          #pragma omp task firstprivate(i, j) \
78                           depend(in: sumXYFrame[i][j]) \
79                           depend(inout: sumYFrame[i][j])
80          calculateFinalSignalFrame(sumXYFrame, sumYFrame,
```

```
81                                      numberOfGroupsPerExposure, i, j);
82       }
83     }
84   }
85 }
```

Listing B.1: C/OpenMP implementation of the pre-processing sampling application.

B.2 Pedestrian detector

```
1  vl_float * vl_bsc_hog (vl_float const * image,
2                         vl_size width, vl_size height,
3                         Locations **winDetected)
4  {
5    int bx, by;
6    VlHog * self = vl_hog_new ();
7    #pragma omp parallel
8    #pragma omp single nowait
9    {
10     int nblocks = NBLOCKS;
11     for (by = 0 ; by < HOG_HEIGHT-1; by=by+NBLOCKS) {
12       for (bx = 0 ; bx < HOG_WIDHT-1; bx=bx+NBLOCKS) {
13         if (by == 0 && bx == 0) {
14           #pragma omp task firstprivate(by,bx) \
15                       depend(out: hog[by+nblocks-1][bx+nblocks-1])
16           {
17             int tby, tbx;
18             int ubx = bx+NBLOCKS, uby = by+NBLOCKS;
19             for (tby = by ; tby < uby; tby++) {
20               for (tbx = bx ; tbx < ubx; tbx++) {
21                 if ((tby < HOG_HEIGHT-1) && (tbx < HOG_WIDHT-1))
22                   vl_bsc_compute_block (self, image, tby, tbx);
23               }
24             }
25           }
26         } else if (by == 0 && bx != 0) {
27           #pragma omp task firstprivate(by,bx) \
28                       depend(in: hog[by+nblocks-1][bx-1]) \
29                       depend(out: hog[by+nblocks-1][bx+nblocks-1])
30           {
31             int tby, tbx;
32             int ubx = bx+NBLOCKS, uby = by+NBLOCKS;
33             for (tby = by ; tby < uby; tby++) {
34               for (tbx = bx ; tbx < ubx; tbx++) {
35                 if ((tby < HOG_HEIGHT-1) && (tbx < HOG_WIDHT-1))
36                   vl_bsc_compute_block (self, image, tby, tbx);
37               }
38             }
39           }
40         } else if (by != 0 && bx == 0) {
41           #pragma omp task firstprivate(by,bx) \
42                       depend(in: hog[by-1][bx+nblocks-1]) \
```

```
43                          depend(out: hog[by+nblocks-1][bx+nblocks-1])
44            {
45              int tby, tbx;
46              int ubx = bx+NBLOCKS, uby = by+NBLOCKS;
47              for (tby = by ; tby < uby; tby++) {
48                for (tbx = bx ; tbx < ubx; tbx++) {
49                  if ((tby < HOG_HEIGHT-1) && (tbx < HOG_WIDHT-1))
50                    vl_bsc_compute_block (self, image, tby, tbx);
51                }
52              }
53            }
54          } else {
55            #pragma omp task firstprivate(by,bx) \
56                        depend(in: hog[by-1][bx+nblocks-1]) \
57                        depend(in: hog[by+nblocks-1][bx-1]) \
58                        depend(in: hog[by-1][bx-1]) \
59                        depend(out: hog[by+nblocks-1][bx+nblocks-1])
60            {
61              int tby, tbx;
62              int ubx = bx+NBLOCKS, uby = by+NBLOCKS;
63              for (tby = by ; tby < uby; tby++) {
64                for (tbx = bx ; tbx < ubx; tbx++) {
65                  if ((tby < HOG_HEIGHT-1) && (tbx < HOG_WIDHT-1))
66                    vl_bsc_compute_block (self, image, tby, tbx);
67                }
68              }
69            }
70          }
71        }
72      }
73    }
74    vl_hog_delete(self);
75    *winDetected = windows;
76    return &features[0][0][0][0];
77  }
```

Listing B.2: C/OpenMP implementation of the pedestrian detector application.

B.3 Cholesky factorization

```
1  void cholesky_blocked(const int ts, double* Ah[NB][NB])
2  {
3    #pragma omp parallel
4    #pragma omp single nowait
5    {
6      double (*AhDep)[NB][NB] = (double (*) [NB][NB])Ah;
7      int k, i, j, l;
8      for (k = 0; k < NB; k++) {
9        #pragma omp task depend(inout:AhDep[k][k])
10       omp_potrf (Ah[k][k], ts, ts);
11       for (i = k + 1; i < NB; i++) {
12         #pragma omp task depend(in:AhDep[k][k]) \
```

```
13                          depend(inout:AhDep[k][i])
14          omp_trsm (Ah[k][k], Ah[k][i], ts, ts);
15        }
16        for (l = k + 1; l < NB; l++) {
17          for (j = k + 1; j < l; j++) {
18            #pragma omp task depend(in:AhDep[k][l])  \
19                             depend(in:AhDep[k][j])  \
20                             depend(inout:AhDep[j][l])
21            omp_gemm (Ah[k][l], Ah[k][j], Ah[j][l], ts, ts);
22          }
23          #pragma omp task depend(in:AhDep[k][l]) \
24                          depend(inout:AhDep[l][l])
25          omp_syrk (Ah[k][l], Ah[l][l], ts, ts);
26        }
27      }
28    }
29  }
```

Listing B.3: C/OpenMP implementation of the cholesky factorization.

Appendix C

Minimum Makespan for Heterogeneous DAG Tasks

In this appendix we present an ILP formulation that computes the minimum time interval needed to execute a given heterogeneous DAG task on m cores and a device. It provides a node-to-core mapping so that the heterogeneous DAG task makespan is minimized. The purpose of this ILP is to evaluate the accuracy of the response time analysis presented in Chapter 7. The ILP has been built upon the formulation presented in [76], which provides the minimum makespan of an OpenMP-DAG taking into account some OpenMP specification features.

Input parameters

1. m: Number of cores available for execution.
2. $G = (V, E)$: DAG task.
3. Off: Index number of the node v_{Off}, that represents the workload offloaded to the device.
4. $sink$: Index number of the node v^{sink}, the sink node of the heterogeneous DAG task.
5. $source$: Index number of the node v^{source}, the source node of the heterogeneous DAG task.
6. $C_i, \forall i = 1 \dots |V|$: WCET of the nodes.
7. $succ_{i,j} \in (0, 1), \forall v_i \in V, \forall v_j \in V$: Binary variable representing precedence constraints. It equals to 1 if node v_j is a direct successor of node v_i, i.e., if $(v_i, v_j) \in E$, 0 otherwise.

Problem variables

1. $y_{i,k} \in (0,1), \forall v_i \in V, \forall k = 1 \dots m$: Binary variable that is equal to 1 if node v_i is executed on core k, 0 otherwise.
2. $t_i, \forall v_i \in V$: Integer variable representing the starting time of node v_i.
3. $a_{i,j}, \forall v_i \in V, \forall v_j \in V$: Auxiliary binary variable that is equal to 1 if node v_i executes before v_j.

Initial assumptions

1. The source node of the heterogeneous DAG task must start executing at time 0, i.e., $t_{source} = 0$.

Constraints

1. v_{Off} node is executed in the device. Therefore, none of the m available cores of the host executes it:

$$y_{Off,k} = 0, \ \forall k = 1 \dots m$$

2. Each node (except v_{Off} node) is executed only by one core of the host:

$$\sum_{k=1}^{m} y_{i,k} = 1, \ \forall v_i \in V, i \neq Off$$

3. Precedence constraints are fulfilled:

$$succ_{i,j} \cdot (t_i + C_i) <= t_j, \ \forall v_i \in V, \ \forall v_j \in V$$

4. The execution of different nodes in the same core must not overlap, i.e., if two nodes are executed by the same core then, either one finishes before the other begins, or vice versa:

$$(y_{i,k} = 1 \ \wedge \ y_{j,k} = 1) \Rightarrow (t_i + C_i \leq t_j \ \vee \ t_j + C_j \leq C_i),$$
$$\forall v_i \in V, \ \forall v_j \in V, \ \forall k = 1 \dots m$$

   This constraint can be written as:

$$t_i + C_i \leq t_j \ + MC \cdot (3 - a_{i,j} - y_{i,k} - y_{j,k})$$

$$t_j + C_j \leq t_i \ + MC \cdot (2 + a_{i,j} - y_{i,k} - y_{j,k}),$$
$$\forall v_i \in V,\ \forall v_j \in V,\ \forall k = 1 \ldots m,$$
$$i \neq j, i \neq Off$$

where MC is an arbitrarily large constraint. In order to be safe, we set $MC = |V| \times \max\{C_i, \forall v_i \in V\}$.

Objective function. The objective function aims to minimize the starting time of the sink node of the heterogeneous DAG task, i.e.,

$$min\ t_{sink}$$

$t_{sink} + C_{sink}$ represents the minimum makespan.

Bibliography

[1] Crystal Market Research. Embedded System Market by Product and Application - Global Industry Analysis and Forecast to 2022, October 2017. URL: https://www.crystalmarketresearch.com/index.php/report/embedded-systems-market. 1

[2] TTI, Inc. Whatever the Future of the Automotive Industry, Electronics is the Key, September 2014. URL: https://www.ttii.com/content/ttii/en/marketeye/articles/categories/industry/me-coulon-20140909.html. 1

[3] Fortune. Intel to Buy Israeli Self-Driving Car Tech Mobileye for $15 Billion, March 2017. URL: http://fortune.com/2017/03/13/mobileye-intel-self-driving-tesla-mbly-stock. 2

[4] Mark Dowson. The ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997. 2

[5] International Organization for Standardization. ISO/DIS 26262. Road vehicles – Functional safety, November 2011. 2

[6] RTCA and EUROCAE. DO-178C/ED-12C. Software Considerations in Airborne Systems and Equipment Certification, November 2011. 2

[7] International Electrotechnical Commission. IEC 61511. Functional safety - Safety instrumented systems for the process industry sector, January 2018. 2

[8] NVIDIA. Self-Driving Safety Report, 2018. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/self-driving-cars/safety-report/NVIDIA-Self-Driving-Safety-Report-2018.pdf. 3

[9] Andy Winstanley. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade, 2015. URL: https://www.arm.com/about/newsroom/

arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php. 3

[10] Miroslaw Staron. *Automotive Software Architectures: An Introduction*. Springer, 2017. 3

[11] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, April 2008. 3

[12] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, December 2007. 4, 20

[13] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Proceeding of the IEEE 33rd Real-Time Systems Symposium (RTSS)*, December 2012. 5, 16, 119

[14] Sabri Pllana and Fatos Xhafa. *Programming Multicore and Many-core Computing Systems*, volume 86. John Wiley & Sons, 2017. 5

[15] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. 2007. 5

[16] NVIDIA. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. 5

[17] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010. 5

[18] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997. 5

[19] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008. 5

[20] OpenMP Architecture Review Board. OpenMP Application Programming Interface, version 4.5, November 2015. URL: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. 6, 22

[21] Luis Miguel Pinho, Vincent Nélis, Patrick Meumeu Yomsi, Eduardo Quiñones, Marko Bertogna, Paolo Burgio, Andrea Marongiu, Claudio Scordino, Paolo Gai, Michele Ramponi, et al. P-SOCRATES: A parallel software framework for time-critical many-core systems. *Microprocessors and Microsystems*, 39(8):1190–1203, November 2015. 6, 58, 165

[22] Texas Instruments. *66AK2Hxx Multicore Keystone II System-on-Chip (SoC)*. November 2012. 6, 143

[23] Eric Stotzer, Ajay Jayaraj, Murtaza Ali, Arnon Friedmann, Gaurav Mitra, Alistair P Rendell, and Ian Lintault. OpenMP on the low-power TI Keystone II ARM/DSP system-on-chip. In *Proceedings of the International Workshop on OpenMP (IWOMP)*, November 2013. 6, 144

[24] Barbara Chapman, Lei Huang, Eric Biscondi, Eric Stotzer, Ashish Shrivastava, and Alan Gatherer. Implementing OpenMP on a high performance embedded multicore MPSoC. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, May 2009. 6

[25] Benoît Dupont De Dinechin, Duco Van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Proceedings of the conference on Design, Automation & Test in Europe (DATE)*, March 2014. 6, 16, 58, 143, 144

[26] Roberto Vargas, Eduardo Quiñones, and Andrea Marongiu. OpenMP and timing predictability: a possible union? In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, March 2015. 6, 7, 27, 42, 58, 62, 85

[27] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceeding of the IEEE 31st Real-Time Systems Symposium (RTSS)*, November 2010. 6, 16, 27, 58, 85

[28] Philipp Kegel, Maraike Schellmann, and Sergei Gorlatch. Using OpenMP vs. Threading Building Blocks for medical imaging on multi-cores. In *Proceedings of the European Conference on Parallel Processing (Euro-Par)*, August 2009. 6

[29] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, 44(4):101–110, 2009. 6

[30] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance gaps between OpenMP and OpenCL for multi-core CPUs. In *Proceedings of the 41st IEEE International Conference on Parallel Processing Workshops (ICPPW)*, September 2012. 6

[31] Bradford Nichols, Dick Buttlar, Jacqueline Farrell, and Jackie Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996. 6

[32] Bob Kuhn, Paul Petersen, and Eamonn O'Toole. OpenMP versus threading in C/C++. *Concurrency: Practice and Experience*, 12(12):1165–1176, 2000. 6

[33] Giorgio C Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. A survey. *IEEE Transactions on Industrial Informatics*, 9(1):3–15, 2013. 7, 50, 88, 120

[34] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems.* Springer, 2015. 14, 47, 53

[35] Aloysius Mok. Task management techniques for enforcing ED scheduling on a periodic task set. In *Proceeding of the 5th IEEE Workshop on Real-Time Software and Operating Systems*, pages 42–46, 1988. 15

[36] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013. 16

[37] Cláudio Maia, Marko Bertogna, Luís Nogueira, and Luis Miguel Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems (RTNS)*, October 2014. 16, 85

[38] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, Inc., 2006. 18

[39] OpenMP Architecture Review Board. OpenMP Application Program Interface, version 2.5, May 2005. URL: http://www.openmp.org/wp-content/uploads/spec25.pdf. 22

[40] OpenMP Architecture Review Board. OpenMP Application Program Interface, version 3.0, May 2008. URL: http://www.openmp.org/wp-content/uploads/spec30.pdf. 22, 61

[41] OpenMP Architecture Review Board. OpenMP Application Program Interface, version 4.0, July 2013. URL: http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf. 22, 61, 64

[42] OpenMP Architecture Review Board. OpenMP Application Programming Interface, version 4.5. Examples., November 2016. URL: https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf. 26

[43] Roberto Vargas, Sara Royuela, Maria A. Serrano, Xavier Martorell, and Eduardo Quiñones. A Lightweight OpenMP4 Run-time for Embedded Systems. In *Proceedings of the 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2016. 27, 42, 58, 62, 129

[44] Sara Royuela, Roger Ferrer, Diego Caballero, and Xavier Martorell. Compiler analysis for OpenMP tasks correctness. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, May 2015. 27, 28

[45] Sara Royuela, Alejandro Duran, Maria A. Serrano, Eduardo Quiñones, and Xavier Martorell. A functional safety OpenMP for critical real-time embedded systems. In *Proceedings of the International Workshop on OpenMP (IWOMP)*, September 2017. 28

[46] Yuan Lin. Static nonconcurrency analysis of OpenMP programs. In *Proceedings of the International Workshop on OpenMP (IWOMP)*. June 2008. 28

[47] Vamshi Basupalli, Tomofumi Yuki, Sanjay Rajopadhye, Antoine Morvan, Steven Derrien, Patrice Quinton, and David Wonnacott. ompVerify: polyhedral analysis for the OpenMP programmer. In *Proceedings of the International Workshop on OpenMP (IWOMP)*, June 2011. 28

[48] Jianjiang Li, Dan Hei, and Lin Yan. Correctness analysis based on testing and checking for OpenMP programs. In *Proceedings of the IEEE ChinaGrid Annual Conference*, August 2009. 28

[49] Ok-Kyoon Ha, In-Bon Kuh, Guy Martin Tchamgoue, and Yong-Kee Jun. On-the-fly detection of data races in OpenMP programs. In *Proceedings of the ACM Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, July 2012. 28

[50] Sara Royuela, Xavier Martorell, Eduardo Quiñones, and Luis Miguel Pinho. OpenMP tasking model for Ada: safety and correctness. In *Proceedings of the Ada-Europe International Conference on Reliable Software Technologies*, June 2017. 28

[51] Sara Royuela, Xavier Martorell, Eduardo Quiñones, and Luis Miguel Pinho. Safe parallelism: Compiler analysis techniques for Ada and OpenMP. In *Proceedings of the Ada-Europe International Conference on Reliable Software Technologies*, June 2018. 28

[52] Sara Royuela, Luis Miguel Pinho, and Eduardo Quinones. Converging safety and high-performance domains: Integrating OpenMP into Ada. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, March 2018. 28

[53] Luis Miguel Pinho, Eduardo Quiñones, and Sara Royuela. Combining the tasklet model with OpenMP. *ACM SIGAda Ada Letters*, 38(1):14–18, July 2018. 29

[54] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *Proceeding of the IEEE 27th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2015. 29, 90, 91, 119

[55] Nerma Bašcelija. Sequential and parallel algorithms for cholesky factorization of sparse matrices. *Mathematical Applications in Science and Mechanics*, 2013. 36, 127

[56] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, et al. Towards fully autonomous driving: Systems and algorithms. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 163–168, June 2011. 36

[57] Sara Royuela Alcázar et al. *High-level compiler analysis for OpenMP*. PhD thesis, Universitat Politècnica de Catalunya, June 2018. 36, 62, 129

[58] Tucker S Taft, Robert A Duff, Randall L Brukardt, and Erhard Ploedereder. *Consolidated Ada reference manual: language and standard libraries*, volume 2219. Springer, August 2003. 46

[59] Antoniu Pop and Albert Cohen. A stream-computing extension to openmp. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, January 2011. 46

[60] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011. 47

[61] Sanjoy K Baruah and Samarjit Chakraborty. Schedulability analysis of non-preemptive recurring real-time tasks. In *Proceeding of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006. 49

[62] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceeding of the 28th IEEE International Real-Time Systems Symposium (RTSS)*, December 2007. 50

[63] Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni, Gang Yao, Francesco Esposito, and Marco Caccamo. Preemption points placement for sporadic task sets. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS)*, July 2010. 50

[64] Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, July 2011. 50, 120

[65] 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)*, July 2007. 53

[66] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *Proceeding of the IEEE 26th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2014. 53, 119

[67] OpenMP Architecture Review Board. OpenMP Technical Report 7: Version 5.0 Public Comment Draf. (Expires November 8, 2018), July 2018. URL: https://www.openmp.org/wp-content/uploads/openmp-TR7.pdf. 54

[68] GOMP project. GNU libgomp, November 2015. URL: https://gcc.gnu.org/projects/gomp/. 55, 129, 156

[69] Intel® C++ Compiler 18.0 Developer Guide and Reference. https://software.intel.com/en-us/cpp-compiler-18.0-developer-guide-and-reference. 55

[70] Barcelona Supercomputing Center. OmpSs 1.0 Specification. https://pm.bsc.es/ompss-docs/specs/, 2016. 55, 84

[71] Barcelona Supercomputing Center. Extrae Release 3.5.2. https://tools.bsc.es/extrae. 55

[72] Barcelona Supercomputing Center. Paraver Release 4.7.2. https://tools.bsc.es/paraver. 55

[73] Lukas Osinski, Tobias Langer, Ralph Mader, and Jürgen Mottok. Challenges and opportunities with embedded multicore platforms. In *Proceedings of the 9th European Congress Embedded Real Time Software and Systems (ERTS²)*, February 2018. 57

[74] Toshihiro Hanawa, Mitsuhisa Sato, Jinpil Lee, Takayuki Imada, Hideaki Kimura, and Taisuke Boku. Evaluation of multicore processors for embedded systems by parallel benchmark program using OpenMP. In *Proceeding of the International Workshop on OpenMP (IWOMP)*, June 2009. 58

[75] Jinghao Sun, Nan Guan, Yang Wang, Qingqing He, and Wang Yi. Scheduling and analysis of real-time OpenMP task systems with tied tasks. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, December 2017. 58, 80, 85

[76] Alessandra Melani, Maria A Serrano, Marko Bertogna, Isabella Cerutti, Eduardo Quiñones, and Giorgio Buttazzo. A static scheduling approach to enable safety-critical OpenMP applications. In *Proceedings of the 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2017. 58, 181

[77] Giuseppe Tagliavini, Daniele Cesarini, and Andrea Marongiu. Unleashing fine-grained parallelism on embedded many-core accelerators with lightweight OpenMP tasking. *IEEE Transactions on Parallel and Distributed Systems*, 2018. 58

[78] AUTOSAR. AUTomotive Open System ARchitecture (AUTOSAR). URL: http://www.autosar.org. 59, 138

[79] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, March 2009. 61, 65, 84

[80] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998. 63, 67

[81] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. Evaluation of OpenMP task scheduling strategies. In *Proceedings of the 4th International Workshop on OpenMP (IWOMP)*, May 2008. 66, 67, 84

[82] Girija J Narlikar. Scheduling threads for low space requirement and good locality. *Theory of Computing Systems*, 35(2):151–187, 2002. 66

[83] Jan Karel Lenstra and AHG Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978. 68

[84] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969. 68, 69, 70

[85] Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Proceeding of the 32nd IEEE International Real-Time Systems Symposium (RTSS)*, December 2011. 85

[86] Robert I Davis, Alan Burns, Jose Marinho, Vincent Nelis, Stefan M Petters, and Marko Bertogna. Global and partitioned multiprocessor fixed priority scheduling with deferred preemption. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(3):47, 2015. 93, 120

[87] Abhilash Thekkilakattil, Robert I Davis, Radu Dobrin, Sasikumar Punnekkat, and Marko Bertogna. Multiprocessor fixed priority scheduling with limited preemptions. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS)*, November 2015. 93, 94, 99, 120

[88] Pierre Audibert. *Mathematics for informatics and computer science*. John Wiley & Sons, 2013. 102

[89] José Marinho, Vincent Nélis, Stefan M Petters, Marko Bertogna, and Robert I Davis. Limited pre-emptive global fixed task priority. In *Proceedings of the 34th Real-Time Systems Symposium (RTSS)*, December 2013. 106, 108, 120

[90] IBM ILOG. Cplex optimization studio, 2014. URL: `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer`. 110, 157

[91] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic DAG task model. In *Proceedings of the IEEE 25th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2013. 119

[92] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D Gill. Parallel real-time scheduling of DAGs. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, December 2014. 119

[93] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Analysis of global EDF for parallel tasks. In *Proceeding of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2013. 119

[94] Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *Proceedings of the IEEE 26th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2014. 119

[95] José Fonseca, Geoffrey Nelissen, Vincent Nelis, and Luís Miguel Pinho. Response time analysis of sporadic dag tasks under partitioned scheduling. In *Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016. 119

[96] José Carlos Fonseca, Vincent Nélis, Gurulingesh Raravi, and Luis Miguel Pinho. A multi-dag model for real-time parallel applications with conditional execution. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*, April 2015. 119

[97] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global EDF scheduling of systems of conditional sporadic dag tasks. In *Proceeding of the IEEE 27th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2015. 119

[98] Sanjoy Baruah. The federated scheduling of systems of conditional sporadic DAG tasks. In *Proceedings of the 12th International Conference on Embedded Software (EMSOFT)*, October 2015. 119

[99] José Fonseca, Geoffrey Nelissen, and Vincent Nélis. Improved response time analysis of sporadic DAG tasks for global FP scheduling. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS)*, October 2017. 119

[100] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceeding of the IEEE 6th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, December 1999. 120

[101] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proceeding of the IEEE 17th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2005. 120

[102] Bo Peng, Nathan Fisher, and Marko Bertogna. Explicit preemption placement for real-time conditional code. In *Proceedings of the 26th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2014. 120

[103] Aaron Block, Hennadiy Leontyev, Bjorn B Brandenburg, and James H Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2007. 120

[104] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2005. 126

[105] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. http://www.vlfeat.org/, 2008. 126

[106] Roger Ferrer, Sara Royuela, Diego Caballero, Alejandro Duran, Xavier Martorell, and Eduard Ayguadé. Mercurium: Design decisions for a s2s compiler. In *Proceedings of the Cetus Users and Compiler Infastructure Workshop in conjunction with PACT*, October 2011. 129

[107] Enrico Mezzetti and Tullio Vardanega. On the industrial fitness of WCET analysis. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2011. 130

[108] Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer, and Eduard Ayguadé. Support for OpenMP tasks in Nanos v4. In *Proceedings of the conference of the center for advanced studies on Collaborative research*, October 2007. 130

[109] Miloš Panić, Sebastian Kehr, Eduardo Quiñones, Bert Boddecker, Jaume Abella, and Francisco J Cazorla. Runpar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2014. 138

[110] Sebastian Kehr, Miloš Panić, Eduardo Quiñones, Bert Böddeker, Jorge Becerril Sandoval, Jaume Abella, Francisco J Cazorla, and Günter Schäfer. Supertask: Maximizing runnable-level parallelism in AUTOSAR applications. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, March 2016. 138

[111] Martin Lowinski, Dirk Ziegenbein, and Sabine Glesner. Splitting tasks for migrating real-time automotive applications to multi-core ECUs. In *Proceedings of the 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016. 138

[112] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In *Proceedings of the IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems (SEUS)*, October 2010. 138

[113] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic WCET analysis of real-time parallel applications. In *Proceedings of the 13th Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2013. 138

[114] NVIDIA. NVIDIA Tegra X1. NVIDIA'S New Mobile Superchip. *NVIDIA White Paper*, January 2015. 143

[115] Steve Leibson and Nick Mehta. Xilinx ultrascale: The next-generation architecture for your next-generation architecture. *Xilinx White Paper WP435*, May 2013. 143

[116] Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, et al. Many suspensions, many problems: A review of self-suspending tasks in real-time systems. *Real-Time Systems*, September 2018. 161

[117] Cong Liu and James H Anderson. Suspension-aware analysis for hard real-time multiprocessor scheduling. In *Proceedings of the IEEE 25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 271–281, July 2013. 161

[118] Cong Liu, Jian Jia Chen, Liang He, and Yu Gu. Analysis techniques for supporting harmonic real-time tasks with suspensions. In *Proceedings of the IEEE 26th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2014. 161

[119] Jian-Jia Chen, Wen-Hung Huang, and Cong Liu. k2U: A general framework from k-point effective schedulability analysis to utilization-based tests. In *Proceedings of the 35th Real-Time Systems Symposium (RTSS)*, December 2015. 161

[120] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. A framework for supporting real-time applications on dynamic reconfigurable FPGAs. In *Proceedings of the 36th Real-Time Systems Symposium (RTSS)*, December 2016. 161

[121] CLASS. Edge and Cloud Computation: A Highly Distributed Software for Big Data Analytics. https://class-project.eu/. 165

[122] Rosa M Badia, Javier Conejero, Carlos Diaz, Jorge Ejarque, Daniele Lezzi, Francesc Lordan, Cristian Ramon-Cortes, and Raul Sirvent. COMP superscalar, an interoperable programming framework. *SoftwareX*, 3:32–36, 2015. 165

[123] Christos H Papadimitriou. On the complexity of integer programming. *Journal of the ACM (JACM)*, 28(4):765–768, 1981. 174