UNIVERSITY OF MODENA AND REGGIO EMILIA

Department of Physiscs, Informatics and Mathematics
Ph. D. degree in Mathematics
in agreement with the University of Ferrara and the University of Parma
Cycle XXXIII

DOCTORAL THESIS

A comprehensive analysis of DAG tasks: solutions for modern real-time embedded systems

---

Author:
Micaela VERUCCHI

Supervisor:
Prof. Marko BERTOGNA

Ph.D. Course Coordinator: Cristian Giardinà

Accademic Year 2019/2020

Ph. D. in Mathematics - DSS INF/01

A COMPREHENSIVE ANALYSIS OF DAG TASKS: SOLUTIONS FOR MODERN REAL-TIME EMBEDDED SYSTEMS

by Micaela VERUCCHI

Reviewed by **Sanjoy BARUAH**
Geoffrey NELISSEN

Abstract

Modern cyber-physical embedded systems integrate several complex functionalities that are subject to tight timing constraints. Unfortunately, traditional sequential task models and uniprocessors solutions cannotbe applied in this context: a more expressive model becomes necessary. In this scenario, the Directed Acyclic Graph (DAG) is a suitable model to express the complexity and the parallelism of the tasks of these kinds of systems.

In recent years, several methods with different settings have been proposed to solve the schedulability problem for applications featuring DAG tasks. However, there are still many open problems left.

Besides schedulability, aspects like the freshness of data or reaction to an event are crucial for the performance of those kinds of systems. For example, a typical application in the robotics field is composed of sensing the environment, planning, and actuate consequently to the elaborated data. Predictable end-to-end latency is then decisive, and it can get very complicated in real scenarios.

This thesis represents an effort in both directions: (i) the schedulability of a DAG task on a multiprocessor, and (ii) the supervision of end-to-end latency for multi-rate task sets. For the former problem, a survey of the state-of-the-art of the Directed Acyclic Graph task model is presented, with a focus on scheduling tests that are more effective, easy to implement, and adopt. Regarding the latter, a novel method is proposed to convert a multi-rate DAG task set with timing constraints into a single-rate DAG that optimizes schedulability, age, and reaction latency. Finally, three real-world use-cases of industry 4.0, smarty city and self-driving car are detailed to demonstrate how the theoretical contribution of this thesis is intrinsically linked to modern applications.

Acknowledgements

This journey could not have come to an end without the help and support of the following people, to whom I am truly grateful.

Thanks to **Marko Bertogna**, for always believing in me and overwhelming me with his constant optimism and enthusiasm.

Thanks to **Marco Caccamo**, for welcoming and guiding me during my time at the CPS group at TUM.

Thanks to **Marco Domenico Santambriogio** for being a true educator, and making me understand the importance of failure in the process of personal growth.

Thanks to the reviewers **Sanjoy Baruah** and **Geoffrey Nelissen** for helping me improving my thesis with their insightful comments.

Thanks to all my colleagues, in Modena and Munich, who taught me and helped me day by day, in each project, presentation, demo, or deadline. A special thanks goes to **Nacho Sañudo**, **Davide Sapienza**, **Giorgia Franchini**, **Luca Miccio** and **Francesco Gatti**.

Finally, thank you **Barbara**, **Stefano** and **Beatrice** for your unconditional love.

Yours sincerely,
Micaela

Contents

List of Abbreviations

CPU	Central Processing Unit
GPU	Graphics Processing Unit
FPGA	Field Programmable Gate Array
NP	Non-Deterministic Polynomial-Time
DAG	Directed Acyclic Graph
WCET	Worst-Case Execution Time
BCET	Best-Case Execution Time
FP	Fully Preemtive
FNP	Fully Non-Preemtive
LP	Limited Preemtive
EDF	Earliest Deadline First
DM	Deadline Monotonic
RM	Rate Monotonic
FTP	Fixed Task Priority
FJP	Fixed Job Priority
EST	Earliest Starting Time
EFT	Earliest Finishing Time
LST	Latest Starting Time
LFT	Latest Finishing Time
DBF	Demand Bound Function
NFJ	Nested Fork Join
SPTree	Sequential-Parallel Tree
WCRT	Worst Case Response Time
RTA	Response Time Analysis
ILP	Integer Linear Programming
NPR	Non-Preemptive Region
LET	Logical Execution Time
OD	Object Detection
NN	Neural Network
DNN	Deep Neural Network
CNN	Convolutional Neural Network
BB	Bounding Box
GPGPU	General-Purpose computing on Graphics Processing Unit
AI	Artificial Intelligence

Chapter 1

Introduction

1.1 Keep It Up With Innovation

Is Moore's law still alive? That is a question that can easily lead to a controversial outcome: many say yes, others say no, the most say it will be dead soon. In 1964 Gordon Moore, Fairchild Semiconductor's Director of R&D, predicted that the number of transistors on a chip would double every year [84, 85]. Almost ten years later, Moore, working at the time with Intel, revised his statement by saying it was going to double every two years, rather than every year. This prophecy has been incredibly accurate for more than 50 years, becoming known as Moore's Law.

Indeed, the prediction was correct back in 1971, when the Intel 4004 processor was released, featuring 2300 transistors while Moore's law forecast 1500 for that year; and yet it was correct in 2019, when it law was expecting around 40 billion transistors and AMD released the Epyc Rome processor with 39 billion transistors.

The exponential increase in the number of transistors is made possible by the progressive reduction in the characteristic dimensions of the integrating process: back in the 70s hardware producers realized technologies in the order of 10 μm, today we are in the range of $5 - 7nm$. The main benefit of reducing dimensions is to allow a higher clock speed: more gates can fit on a chip with a reduced distance among them, therefore signals have a lower path to cover, and the transitory time for a state transition decreases. However, when reducing the distance between gates, a parasite effect of leakage current increases, being the reason why the single processor frequency has a bound of 4GHz, reached in 2002.

This limitation somehow marked the end of the single processor, but not the end of Moore's law. On the contrary, this problem was revised as an opportunity, and that opportunity was the multi-processor, which kept the law alive. In the years, many experts in the field, including NVIDIA CEO Jensen Huang, declared that Moore's Law is dead. Moore himself, in 2005 stated that his law was dead, and that in 10-20 years we will reach a fundamental limit.

Even if it may be true that this law won't be valid in 2025, the point is that Moore's Law is sustained by innovations: we surely have reached the end of the road for some approaches, but not all of them.

Today, at the end of 2020, multi-cores have been surpassed by heterogeneous many-cores, which are now the cutting-edge platforms. This is where the research is going, where modern applications will run onto, and where also Real-Time community should investigate.

1.1.1 Heterogeneous Embdedded Platforms

In 2017 Intel released the i9-7900X processor family, among which also the Intel i9-7980XE, featuring 18 cores. The following year, AMD released the Ryzen Threadripper series of processors, among which Ryzen Threadripper 2990WX with 36 cores. In March 2020, Ampere Computing announced Ampere Altra, with the Q80-33 processor featuring 80 64-bit Arm cores. In September 2020, NVIDIA launched the GeForce RTX 3000 series of Graphic Processing Units (GPUs), including the RTX 3090, with 10496 cores. Besides these examples, there exist other solutions that are currently going in the direction of many-core, rather than multi-core, e.g. Field Programmable Gate Array (FPGAs) or Google Tensor Processing Units (TPUs). Even though most of the cited architectures are server solutions, this trend can be found also in the embedded world. Indeed, in the latest years, several new embedded boards have been produced by affirmed manufactured, as Xavier or Pegasus by NVIDIA, Ultrascale+ or Versal by Xilinx, and Snapdragon by Qualcomm, but also from new ones, as Kirin from Hilisicon (owned by Huawei).

The specifics of the new proposed boards are various and different, however a bottom main idea can be easily found. All of these new-generation embedded boards features a multi-core Central Processing Unit (CPU), one or more GPUs and other additional engines, such as FPGAs or Artificial Intelligence (AI) specific accelerators. Heterogeneity has become a new required feature of embedded systems, along with the well-established goals of delivering high performance while controlling power consumption.

1.1.2 Modern Real Applications Are Complex

The reason behind this new computing paradigm can be found in the intrinsic bond between hardware and applications. At the beginning of the XXI century, the advent of GPUs made the training and the adoption of neural networks possible. From that point in time, a terrific investment has been made in the AI field, both from the research and industry worlds. It became common practice to develop applications that rely on one or more neural networks. These kinds of applications have a considerable computational demand, generally have a high degree of parallelism, and work on a massively amount of data, obtained from one or more sensors. Clearly, there was a need for appropriate new-generation embedded platforms.

However, the decisive push that convinced manufacturers to go towards this direction came from a specific sector: automotive. Since 2005, when Stanley, the autonomous car developed by the Stanford team lead by Sebastian Thrun, won the DARPA Grand Challenge [115] and David Hall proposed the first prototype of 3D Velodyne LiDAR [57], researchers and companies have been focusing on self-driving cars. Nowadays hundreds of companies are involved in this self-driving revolution, among which not only all the major car manufacturer (e.g. General Motors, Ford, Mercedes Benz, Volkswagen, Audi, Nissan, Toyota, BMW, etc.), but also all major big tech corporates (e.g. Google, Apple, Huawei, Intel, NVIDIA, Xilinx, etc.).

1.1.3 Real-Time Community Commitment

Self-driving cars are also the perfect example to understand why the Real-Time community is called to contribute in this direction. Indeed, it is one real relevant application, among others, with a terrific complexity, that runs on heterogeneous embedded platforms and has intrinsic real-time requirements.

Satisfying those real-time requirements to guarantee predictability for those applications means finding the solution to two distinct, though synergistic, problems, one related to software and the other to hardware.

The first is to derive suitable models to fully represent the complexity of real applications. Liu and Layland task model [73] cannotbe adopted anymore: its expressiveness is limited and its adoption would lead to simplistic representations of the applications, too far from reality. More articulated task models are needed, as well as algorithms to check their feasibility, to compute their end to end latencies, to estimate their Worst-Case Execution Time (WCET) on this new-generation of embedded boards.

The second is to make these high performance, low power, heterogeneous boards also predictable. And this is also challenging because they were not built with that purpose. Multi-core systems are inherently unpredictable, and these new embedded platforms are even worse. Resources are not shared only among different homogeneous cores, which is still true, but they are also shared among different heterogeneous engines. The biggest issue can be found in the bandwidth and, even more exacerbated, in the several layers of memory. Sharing those resources without any precautions leads for sure to unstable performance, caused by interference overheads. Mechanisms to partition the memory and allow isolation, such as cache coloring, are then required, as well as bandwidth reservation systems.

1.2 Contribution and Organization

This thesis makes an effort trying to solve the software-related problem, by focusing on the Directed Acyclic Graph (DAG) task model, both from a theoretical and practical point of view.

Three main contributions can be found in this work:

i A detailed survey of the literature of the DAG task model, spacing from the original proposed by Baruah et. al in 2012 [11], to the Heterogeneous Parallel Conditional DAG task proposed by Zahaf et al. in 2019 [132, 133]. Several policies have been analyzed, with a focus on the schedulability test of DAG tasks onto multi-core or heterogeneous-boards. This survey aims to (i) have a clear overview of the existing literature, (ii) compare global and partitioned scheduling solutions, and (iii) understand which methods are more suitable from an industrial adoption point of view, considering both the goodness of the methods in terms of schedulability and their execution time. To accomplish that, an implementation effort has been carried on, in order to compare the best existing methods, and it has been made publicly available at `https://github.com/mive93/DAG-scheduling`. This is the content of Chapter 3.

ii A new method to convert a real-world application, modeled with a multi-rate taskest, into a DAG, that was firstly presented in *"Latency-Aware Generation of Single-Rate DAGs from Multi-Rate Task Sets"* [122] (RTAS 2020), reported in Chapter 4. The conversion procedure, detailed in the chapter, accepts in input constraints related not only to schedulabilty, but also to end-to-end latency, in order to generate a DAG with bounded data age and reaction time.

iii Three different real-world applications with real-time requirements, that involve heterogeneous embedded platforms, neural networks and several sensors, modeled as DAGs, that are described in Chapter 5. The first is a defect detection

application, object of investigation in a join collaboration with Tetra Pak, better detailed in *"A Systematic Assessment of Embedded Neural Networks for Object Detection"* [120] (ETFA 2020), and *"Embedded Neural Networks for Object Detection: A Systematic Assessment"* [121][1]. The second is a smart city application, developed in the context of the European project CLASS, to detect and track objects from pole-mounted cameras. The third one is a new method to perform sensor fusion of cameras and LiDARs on embedded boards for self-driving cars, originally proposed in *"Real-Time clustering and LiDAR-camera fusion on embedded platforms for self-driving cars"* [123] (IRC 2020).

Chapter 2 offers some background to better understand the matter of this thesis, while conclusions and open problems are discussed in Chapter 6.

[1] Journal version, still under review.

Chapter 2

Background

2.1 Real-Time Scheduling

In the real-time scheduling theory, the workload is generated by a finite collection of recurrent tasks or processes. Each such recurrent task may model a piece of code that is embedded within an infinite loop, or that is triggered by the occurrence of some external event. Each execution of the piece of code is referred to as a job; a task, therefore, generates a potentially unbounded sequence of jobs.

A recurrent real-time task is usually characterized at least by two parameters: the period T and the deadline D.

A recurrent task is said to be *periodic* if successive jobs of the task are required to be generated a T times units apart. A task is said *sporadic* if a lower bound, but no upper bound is specified between the generation of successive jobs (at least T times unit). Finally, an *aperiodic* task is one that occurs with no repetitions and which then does not have a specified T.

The deadline states within how many time units the task needs to be completed. The relationship between deadline and period can be of three types: (i) in the *implicit-deadline* case, the relative deadline of each task is equal to the task's period; (ii) in the *constrained-deadline* case, the relative deadline of each task is no larger than the task's period; (ii) in the *arbitrary-deadline* case, the relative deadline does not have to satisfy any constraint with regards to the period.

Algorithms A real-time scheduling algorithm is responsible for distributing the executing jobs of the system on its available processors during the execution interval of the task set. At each time unit within this interval, the scheduling algorithm chooses particular jobs among the ready ones to execute on the system's processors using a particular priority assignment.

This works focuses mainly on multiprocessor work-conserving scheduling, in which eligible jobs must be executed if there are available cores.

Real-time scheduling algorithms can be divided into (i) fixed task priorities (FTP) (ii) fixed-job priorities (FJP) and (iii) dynamic priorities algorithms.

Earliest Deadline First (EDF) assigns priorities to jobs based on their absolute deadlines where jobs with earlier absolute deadlines have higher priorities. Different jobs of the same task may be assigned different priorities, however, once assigned, the deadline will not change (FJP algorithm). This algorithm is an optimal scheduling algorithm on uniprocessors, although it loses its optimality for multiprocessors.

Deadline Monotonic (DM) and Rate Monotonic (RM) algorithms are instead fixed priority algorithms. Priorities are assigned to tasks based on their relative deadline (in the former) or their period (in the latter).

In dynamic priority algorithms, there are no restrictions in the manner in which priorities are assigned to jobs.

Schedulability The concept of schedulability was well defined by Baruah [8] and takes into account both the scheduling algorithm and the task set properties.

Definition 2.1.1. *Let A denote a scheduling algorithm. A task system T is said to be A-schedulable, if A meets all deadlines when scheduling each of the potentially infinite different collection of jobs that could be generated by the task system, upon the specified platform.*

To check the schedulability of a task set under a chosen scheduling algorithm, a schedulability test needs to be derived and applied to the task set.

Definition 2.1.2. *An A-schedulability test accepts as input the specifications of a task system and a multiprocessor platform and determines whether the task system is A-schedulable.*

Infinite schedulability tests can be derived. However, a schedulability test is called exact, only if it identifies all A-schedulable systems. The test is called sufficient if it identifies some A-schedulable systems.

A schedulability test can be used to compare scheduling algorithms. Indeed, a scheduling algorithm A dominates scheduling algorithm B if all task sets schedulable by algorithm B are also schedulable by algorithm A, but the opposite is not correct.

Sustainability The notion of *sustainability* [10, 28] formalizes the expectation that a system that is schedulable under its worst-case specifications should remain schedulable when its real behavior is "better" than worst-case.

A scheduling policy and/or a schedulability test for a scheduling policy is sustainable if any system deemed schedulable by the schedulability test remains schedulable when the parameters of one or more individual tasks(s) are changed in any, some, or all of the following ways: (i) decreased execution requirements; (ii) larger periods; (iii) smaller jitter; and (iv) larger relative deadlines.

Generally, there is often a trade-off between accuracy and sustainability: exact schedulability tests are usually not sustainable, but sustainable sufficient schedulability tests can often be designed.

Preemption policy The preemption strategy is a key aspect of real-time scheduling, as it could favor urgent tasks and affects schedulability.

Traditionally there are two opposite approaches: Fully Preemptive (FP) and Fully Non-Preemptive (FNP) scheduling strategies [8]. In FP scheduling a job can be interrupted at any time if another job with higher priority arrives, and be resumed to continue only when all higher priority jobs have completed their execution. On the other hand, in FNP scheduling the execution of a job cannotbe interrupted: as it starts it executes until completion.

A common misconception is that the former strategy is more effective in terms of schedulability than the latter one. However, the two are incomparable, there are task sets schedulable under FP that are not schedulable under FNP and vice versa, but they both have some disadvantages. In FP scheduling the risk is to produce many and unnecessary preemptions, whose cost is not negligible, affecting the schedulability of low priority tasks. On the contrary, in FNP scheduling there are no problems regarding preemption overhead, but it could produce long blocking delays that could still undermine schedulability.

To reduce those problems, preemption and migration-related overheads, while also constraining the amount of blocking, Limited Preemption (LP) has been proposed in the literature [78, 18, 31]. This model combines and generalizes the other

two: jobs can be preempted, but only in certain points, otherwise they execute Non-Preemptive Regions (NPRs). Moreover, given the chosen scenario, there exist two approaches to define the way a preemption occurs: the *eager* and the *lazy*. In the *eager* approach a high priority job can preempt the first lower priority job that reaches a preemption point; in the *lazy* approach a high priority job has to wait until the lowest priority job in execution reaches its preemption point to preempt it.

Multiprocessor scheduling strategy When dealing with multiprocessors a strategy on how to map tasks on cores should be chosen.

A *global* real-time scheduler allows migration of any job in the system between the processors during their execution. It is known that this policy leads to lower average response time and automatic load balancing while being easy to implement. However, it suffers from migration costs, overheads due to inter-core synchronization and lack of cache affinity.

The other approach to scheduling multiprocessor real-time systems is *partitioned* scheduling, in which each task is assigned statically to one processor. Partitioned scheduling has the advantages of being supported by the automotive industry (e.g. AUTOSAR), having isolation among cores, and, moreover, exploit established uniprocessor schedulability analysis techniques. The problems of this second method are the likely possibility to have unbalanced loads of the cores and the non-deterministic polynomial-time (NP) hardness of the allocation strategies.

Indeed the allocation problem can be reconducted to the bin packing problem, which is known to be NP-hard [41]. Therefore, heuristics based on a utilization factor are used, as *First Fit*, that fits a task into a processor by scanning from the beginning of available processors to the end, until the first processor that can accommodate it is found; *Best Fit*, which fits a task into a processor by looking for the processor whose left capacity is closer to the utilization of the task; or *Worst Fit*, that fits a task into a processor by looking for the processor with the greatest left capacity.

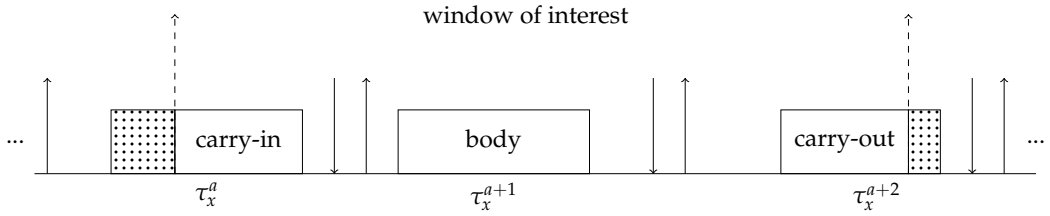


FIGURE 2.1: Carry-in, body and carry-out jobs.

Carry-in, body, carry-out Many schedulability tests rely on the concept of interval (or window) of interest to compute bounds on the workload (or interfering workload) of a task. When computing an upper bound workload over some identified interval of interest $[a, b]$ of length t, the contributions of jobs to this workload are typically considered in three separate categories: (i) **carry-in**, that is the contribution of at most one job, which has its release time earlier than a and its deadline in $[a, b)$; (ii) **body**, that is the contribution of jobs with both release times and deadlines in $[a, b)$; (iii) **carry-out**, that is the contribution of at most one job, which has its release time in $[a, b)$ and its deadline later than b. A graphical representation is depicted in Figure 2.1.

Note that, when considering constrained-deadline sporadic task systems, there can be at most one carry-in job and one carry-out job for each task.

Real-Time Task Models Tasks in a real-time system have specific properties and are represented via a task model.

In 1973 Liu and Layland [73] introduced the first periodic sequential task model. In their famous work, a task $\tau = (C, T)$ is described only by its worst-case execution time (WCET) C and period T, having implicit deadlines. Immediate extensions are the addition of a deadline D and the concept of sporadic tasks.

Several other task models exists [26], as the Multiframe [83], the Generalized Multiframe [14], the Elastic [32], the Mixed Criticality [124], the Splitted Task [29] and the Self-Suspending Task [39].

However, all these models assume that there is a single thread of execution within each task. They are quite simple and do not take into account factors as intrinsic parallelism or precedence constraints among sub-tasks.

Therefore, parallel models have been introduced, such as the Digraph Real-Time task model [108] and the DAG task model [11], object of this thesis.

2.2 Graph Theory

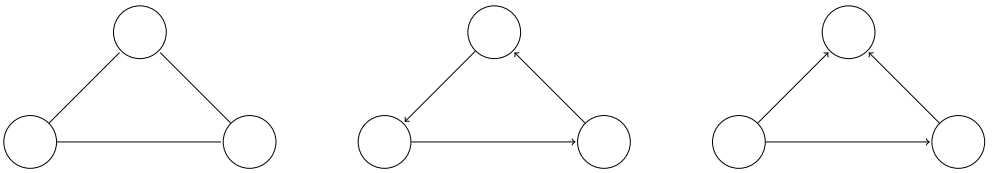
This thesis focuses on the concept of Directed Acyclic Graph (DAG), therefore some basic concepts of graphs [43, 41] are here introduced.

Definition 2.2.1. *A graph is a pair $G = (V, E)$ where V is a set of vertices (also called nodes) and $E \subseteq V \times V$ is a set of edges.*

Definition 2.2.2. *A directed graph is a graph in which edges have an orientation.*

Definition 2.2.3. *A graph $G = (V, E)$ is called acyclic if it does not contain any cycles.*

Figure 2.2 reports an example of a graph (Figure 2.2a), a directed graph (Figure 2.2b) and a directed acyclic graph (Figure 2.2c).



(A) Graph. (B) Directed graph. (C) Directed aclyclic graph.

FIGURE 2.2

Two other concepts worth mentioning are the concept of path in a graph and, sequentially, connected graph.

Definition 2.2.4. *A sequence $\lambda = (v_0, v_1, ..., v_k)$ of nodes $v_i \in V$ of a directed graph $G = (V, E)$ is called a path from v_0 to v_k if $(v_i, v_{i+1}) \in E$ holds for each $i \in 0, ..., k-1$.*

Definition 2.2.5. *A non-empy graph G = (V, E) is called connected if any two of its vertices are linked by a path.*

Several algorithms applied to DAGs can be found in literature[41]. Surely, the most relevant to this work are the depth-first search and the topological sort.

There are many ways of searching a graph, depending upon the way in which edges to search are selected. The depth-first search is a very useful and used one. To perform a depth-first search exploration, When selecting an edge to traverse, the

outgoing edge from the most recently reached vertex which still has unexplored edges is the one that has to be chosen.

The depth-first search of a DAG can be computed in linear time in the size of the DAG $\mathcal{O}(|V| + |E|)$ [113].

Definition 2.2.6. *A topological sort of a directed acyclic graph (DAG) $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (v_i, v_j), then v_i appears before v_j in the ordering.*

The topological sort of a DAG can be computed in linear time in the size of the DAG $\mathcal{O}(|V| + |E|)$ [41].

2.3 Modern Embedded Architectures

While uniprocessors and multiprocessors architectures are well known, not everyone could be familiar with the architecture of heterogeneous embedded platforms. An embedded system can be defined as a special-purpose computing system whose main features are reliability, low power consumption, high-performance, and contained costs. Embedded boards are those whose System on a Chip (SoC) is embedded as part of a complete device, often including electrical or electronic hardware and mechanical parts. Some examples are the Jetson Series by NVIDIA (e.g. Nano or Xavier AGX), the Ultrascale+ family by Xilinx (e.g. ZCU102 or ZCU 104), the Snapdragon 820 Automotive from Qualcomm, etc.

Modern embedded platforms feature a very complex SoC. A high-level representation of a general architecture is reported in Figure 2.3.

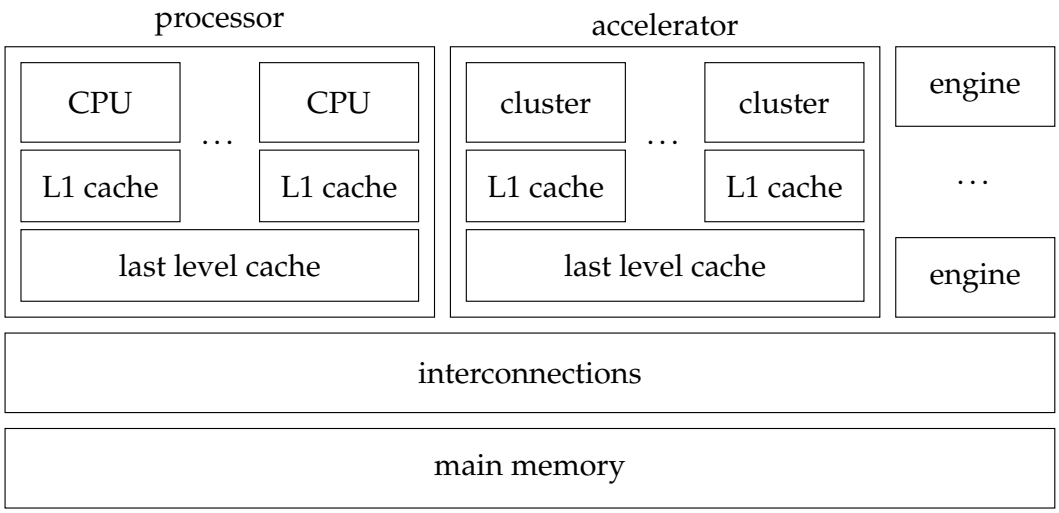


FIGURE 2.3: High level description of a heterogeneous SoC architecture.

The three main building blocks are (i) processing units, (ii) memory, and (iii) interconnection. The architecture features two principal components, namely the processor and the accelerator. The former is a multi-core, the latter a many-core in which, usually, cores are grouped in clusters. In the SoC there could also be one or several engines, typically specialized for specific purposes. GP-GPUs and FPGAs are the most commonly used accelerators. Deep Learning Accelerator (DLA) by NVIDIA, Neural Processing Unit (NPU) by Hilisicon, or Neural Processing Engine (NPE) by Qualcomm are examples of Artificial Intelligence (AI) specialized engines.

Memory and interconnections are shared resources.

The interconnection is shared among the processor, the accelerator, the various engines, and the memory.

Memory is at least split into two layers, namely caches and main memory. Usually, there exist also several layers of caches with different sharing policies. In a general case, as the one reported in Figure 2.3, each core (cluster) of the processor (accelerator) is assigned a private, small (some KiB) L1 cache. Then there is a slightly larger (some MiB) cache, called last level cache, that is shared among all the cores (clusters) of the component. Finally, there is the main memory, which is way wider (some GiB) and shared among all the processing units of the platform, including the processors, the accelerator, and the available engines.

When designing an application for these kinds of platforms, it is crucial to know the sizes and configuration of the memory levels, besides having a proper idea of the memory usage of the considered application. Accessing data is expensive in terms of latency, and retrieving data from L1 cache, last level cache, or main memory has costs that significantly differ. The latency of accessing the main memory can differ in orders of magnitude with respect to the one of accessing L1.

The problem of interference A paramount important aspect to consider when designing a real-time application that will run onto these platforms is the problem of interference. The contention for access to shared memory resources is one of the main predictability bottlenecks of modern multi-core and heterogeneous embedded systems.

It has been shown that the aforementioned problem on these kinds of SoC, for example NVIDIA Tegra K1 and Tegra X1 [36, 67], can lead to terrific degradation in the task latencies. That should be for sure considered when computing the worst-case execution time of a task. Moreover, memory-centric models, as the Predictable Execution Model (PREM) proposed by Pellizoni et al [91], should be better investigated and adopted.

To mitigate the contention in the memory hierarchy and over the bandwidth, some preliminary solutions have been proposed: software solutions as partitioning memory for the former, control and reservation systems for the latter. A clean way to introduce those solutions is through virtualization.

Chapter 3

DAG survey

This chapter tries to summarize the existing literature of the Directed Acyclic Graph (DAG) task model applied to real-time scheduling, focusing on the various schedulability tests proposed in the years. The whole literature of DAGs in real-time is clearly much more extensive. The effort, hereafter reported, aims to deep analyze the task model and its extensions, cover various scheduling policies, and point out the ways to check if a task, or task set, is feasible on a given system.

DAGs were firstly introduced in real-time literature in the context of global scheduling by Liu and Anderson [75]. The authors investigated the Global Earliest Deadline First (G-EDF) scheduling on multiprocessors of a multi-rate task set in which tasks' precedence constraints were modeled via a DAG. The following year, the same authors further investigated the problem for distributed systems [74], considering clustered scheduling and edges with assigned communication costs.

However in both their works, the task model adopted is the sporadic one, with a given deadline and period for each task and no notion of internal parallelism.

3.1 The Directed Acyclic Graph Task Model

The Directed Acyclic Graph task model has been introduced for the first time by Baruah et al. [11]. The sporadic DAG task model is a parallel task model in which each task τ_x is specified by a tuple (G_x, D_x, T_x), where $G_x = (V_x, E_x)$ is a vertex-weighted directed acyclic graph, and D_x and T_x are positive integers that represent, respectively, (relative) deadline and period of the task. An example of a DAG task is depicted in Figure 3.1.

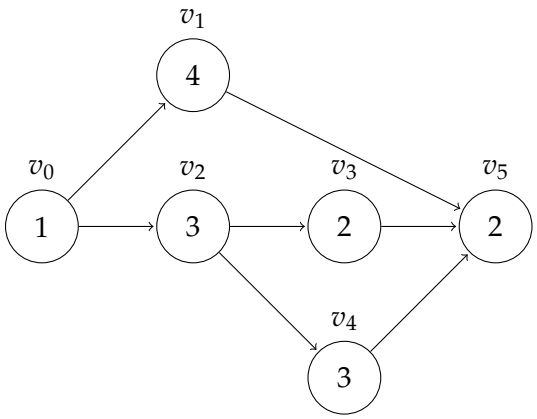


FIGURE 3.1: DAG task τ_x, with period $T_x = 16$ and deadline $D_x = 10$

Each vertex $v_i \in V_x$ of the DAG corresponds to a sequential job, and is characterized by a worst-case execution time (WCET) C_x. Each directed edge of the DAG represents a precedence constraint: if $(v, w) \in E_x$ then the job corresponding to vertex v must complete its execution before the job corresponding to vertex w can begin. Groups of jobs that are not constrained by precedence constraints in such a manner may execute in parallel, if there are processors available for them to do so.

The task τ_x releases a DAG-job at time-instant t when it becomes available for execution. When this happens, we assume that all jobs in $|V_x|$ become available for execution simultaneously, subject only to the precedence constraints. During any given run the task may release an unbounded sequence of DAG-jobs; all $|V_x|$ jobs that are released at some time-instant t must complete execution by time-instant $t + D_x$. A minimum interval of duration T_x must elapse between successive releases of DAG-jobs.

Properties Considering the topology of the model, several features can be noticed.

A *chain* λ in DAG task τ_x is a sequence of vertices $v_1, v_2, ..., v_k \in V_x$ such that (v_i, v_{i+1}) is an edge in G_x. The length of this chain is defined to be the sum of the WCETs of all the vertices in the chain: $\sum_{i=1}^{k} C_i$. We denote by L_x the length of the *longest chain* in the DAG G_x.

The *volume* of a DAG is the total WCET of each job of τ_x, namely $vol_x = \sum_{v_i \in V_x} C_i$.

The *density* of a DAG task is defined as $\delta_x = \frac{L_x}{D_x}$.

The *utilization* of a DAG task is defined as $U_x = \frac{vol_x}{T_x}$.

Example 3.1. *Let us consider the DAG task in Figure 3.1. For τ_x the longest chain is $\{v_0, v_2, v_4, v_5\}$ and $L_x = 9$. Its volume is $vol_x = 15$, its density $\delta_x = \frac{9}{10} = 0.9$, and its utilization $U_x = \frac{15}{16} = 0.94$.*

For each vertex $v_i \in V_x$ two sets $Ancst_i$ and $Desc_i$ can be defined. $Ancst_i$ is the set of ancestors subtasks $v_j \in V_x$ of v_i such that there exists a path starting from v_j that reaches v_i; $Desc_i$ is the set of descendants subtasks $v_k \in V_x$ of v_i such that there exists a path starting from v_i that reaches v_k; .

Task set of DAGs A DAG sporadic task system (or task set) Γ is a collection of n DAG tasks. For the task set Γ we define (i) its maximum density $\delta_{max}(\Gamma)$ to be the largest density of any task in Γ : $\delta_{max}(\Gamma) = max_{\tau_x \in \Gamma}\{\delta_x\}$; (ii) its utilization $U(\Gamma)$ to be the sum of the utilizations of all the tasks in τ : $U(\Gamma) = \sum_{\tau_x \in \Gamma} U_x$; and (iii) its hyper-period $HP(\Gamma)$ to be the least common multiple of the period parameters of all the tasks in Γ : $HP(\Gamma) = lcm_{\tau_x \in \Gamma}\{T_x\}$.

In a DAG task set Γ each task τ_x is assigned a priority P_x based on the schedulability algorithm chosen. Given the task set and the priorities four sets can be defined, namely $hp(\tau_x)$ ($hep(\tau_x)$) that comprises all the tasks with higher priority (higher or equal priority) w.r.t. τ_x and $lp(\tau_x)$ ($lep(\tau_x)$) that comprises all the tasks with lower priority (lower or equal priority) w.r.t. τ_x.

Schedulability A DAG task τ_x is said to be feasible if the subtasks of all of its jobs respect its deadline, that is when its Worst-Case Response Time (WCRT) R_x is less than its deadline $R_x \leq D_x$.

When considering the scheduling of the DAG, the concept of critical chain should be introduced. The *critical chain* λ_x^* of a task τ_x is the chain of vertices of τ_x that leads to its WCRT R_x. It is important to remember that the critical chain does not necessarily correspond to the longest chain.

The problem of testing the feasibility of a given DAG task system is highly intractable, NP-hard in the strong sense, even when there is a single DAG task. [117]. Therefore in the literature, several sufficient schedulability tests have been introduced.

Table 3.1 reports an overview of all the relevant works hereafter discussed, ordered by year, and categorizes them in terms of

- *workload*, if the method consider a single DAG τ or a task set Γ;
- *deadline*, implicit (I), constrained (C) or arbitrary (A);
- *model*, task model used in the work: DAG, Conditional DAG (C-DAG), Heterogeneous (or typed) DAG (H-DAG), Heterogeneous Conditional DAG (HC-DAG);
- *scheduling*, multiprocessors policy: global (G), partitioned (P) or federated (F);
- *preemption*,fully preemptive (FP), fully non preemptive (FNP), limited preemptive (LP);
- *algorithm*, earliest deadline first (EDF), deadline monotonic (DM), fixed task priority (FTP) or specific ones (Hier EDF, DAG-Fluid);
- *complexity*, polynomial (Poly), pseudo-polynomial (Ps-Poly).

A summary of the notation used in this thesis is reported in Table 3.2.

Work	Workload	Deadline	Model	Scheduling	Preemption	Algorithm	Complexity
Baruah '12 [11]	τ	C	DAG	G	N.A.	EDF	Ps-Poly
Bonifaci '13 [25]	Γ	A	DAG	G	FP	EDF, DM	Ps-Poly
Li '13 [70]	Γ	I	DAG	G	FP	EDF	-
Qamhieh '13 [96]	Γ	C	DAG	G	FP	EDF	-
Baruah '14 [4]	Γ	C	DAG	G	FP	EDF	Ps-Poly
Saifullah '14 [102]	Γ	I	DAG	G	FP, FNP	EDF	-
Li '14 [69]	Γ	I	DAG	F	FP	Greedy, EDF	-
Fonseca '15 [51]	Γ	C	C-DAG	-	-	-	-
Baruah '15 [9]	Γ	C	C-DAG	G	FP	EDF	Ps-Poly
Melani '15 [81]	Γ	C	DAG, C-DAG	G	FP	EDF, FTP	-
Baruah '15 [7]	Γ	C	DAG	F	FP	EDF	Poly
Baruah '15 [3]	Γ	A	DAG	F	FP	EDF	Poly
Parri '15 [89]	Γ	A	DAG	G	FP	EDF, DM	Poly, Ps-Poly
Fonseca '16 [50]	Γ	C	DAG	P	FP	FTP	-
Serrano '16 [107]	Γ	C	DAG	G	LP	FTP	Ps-Poly
Yang '16 [129]	Γ	A	DAG	G	LP	FTP	Ps-Poly
Pathan '17 [90]	Γ	C	DAG	G	FP	DM	-
Fonseca '17 [48]	Γ	C	DAG	G	FP	DM	-
Serrano '17 [106]	Γ	C	DAG	G	LP	FTP	Ps-Poly
Serrano '18 [105]	τ	C	H-DAG	G	-	-	-
Han '19 [60]	τ	C	H-DAG	G	-	-	Poly
Casini '18 [35, 34]	Γ	C	DAG	P	LP	FTP	-
He '19 [61]	Γ	C	DAG	G	FP	EDF, FTP	-
Fonseca '19 [49]	Γ	C, A	DAG	G	FP	EDF, FTP	-
Jiang '19 [64]	Γ	A	DAG	F	FP	EDF	Poly
Yang '19 [130]	Γ	C	DAG	P	FNP	Hier EDF	-
Nasri '19 [86]	Γ	C	DAG	G	LP	FTP, EDF	-
Guan '20 [55]	Γ	I	DAG	G	FP	DAG-Fluid	Poly
Chang '20 [38]	τ	C	H-DAG	G	-	-	-
Zahaf '20 [133]	Γ	C	HC-DAG	P	FP, FNP	EDF	-
Baruah '20 [5]	τ	C	DAG	P	FP	EDF	Poly
Casini '20 [33]	Γ	C	DAG	P	LP	FTP	-

TABLE 3.1: Summary of the considered works

task x	τ_x	
job a of task x	τ_x^a	
vertex i of task x	$\tau_{x,i}$	
relative deadline	D_x	
period	T_x	
worst-case response time	R_x	
number of preemption suffered by τ_x	pn_x	
priority of τ_x	P_x	
DAG	G_x	
edge set	E_x	
vertices set	V_x	
vertex (or node, or subtask) i	v_i	
WCET of v_i	C_i	
local offset of v_i or EST	O_i or EST_i	
local deadline of v_i or LFT	D_i or LFT_i	
Best Case Execution Time (BCET) of v_i	BC_i	
immediate predecessors of v_i	$pred_i$	
immediate successors of v_i	$succ_i$	
ancestors of v_i	$Ancst_i$	
descendants of v_i	$Desc_i$	
chain l of τ_x	$\lambda_{x,l}$	
set of chains of τ_x	Π_x	
length of longest chain in G_x	L_x or $L(G_x)$	$\sum_{i=1}^{k} C_i$
critical chain of τ_x	λ_x^*	
volume of G_x	vol_x or $vol(G_x)$	$\sum_{v_i \in V_x} C_i$
utilization	U_x	vol_x / T_x
density	δ_x	L_x / D_x
processor to which v_i is assigned to	p_i	
super-period of tasks τ_x and τ_y	$SP_{x,y}$	$lcm\{T_x, T_y\}$
task set	Γ	
hyper-period of task set Γ	$HP(\Gamma)$	$lcm_{\tau_x \in \Gamma}\{T_x\}$
higher priority tasks w.r.t. τ_x	$hp(\tau_x)$	
lower priority tasks w.r.t. τ_x	$lp(\tau_x)$	
higher-equal priority tasks w.r.t. τ_x	$hep(\tau_x)$	
lower-equal priority tasks w.r.t. τ_x	$lep(\tau_x)$	

TABLE 3.2: Notation used in this thesis. For the sake of clarity, a standardized set of indexing names is adopted throughout the entire thesis, i.e., $\{i, j, k\}$ denote vertices in a DAG, $\{x, y, z\}$ indicate tasks, and $\{a, b, c\}$ are used for jobs in superscription.

3.1.1 Global Policy, Fully Preemptive (G-FP)

Baruah et al. [11] introduced the sporadic DAG task model and focused on the schedulability of a single DAG task with constrained deadlines, in the context of G-EDF scheduling, with a fully preemptive (FP) policy.

The authors proved that the sporadic DAG feasibility problem is NP-hard in the strong sense. Moreover, they have shown that even identifying the worst-case behavior is not trivial. In fact, a sporadic DAG task might be infeasible on m processors even when its synchronous arrival sequence is.

They proposed two sufficient tests: a polynomial test, specific for constrained deadlines (Theorem 3.1.1), from which it is possible to compute the minimum number of needed processors, and a pseudo-polynomial test for arbitrary deadlines (Theorem 3.1.2).

Theorem 3.1.1: Baruah et al. [11]

Any (single) sporadic DAG task $\tau_x = (G_x, D_x, T_x)$ with $D_x < T_x$ satisfying

$$(m-1)\frac{L_x}{D_x} + 2\frac{vol_x}{T_x} \leq m \tag{3.1}$$

is EDF-schedulable on m unit-speed processors.

Theorem 3.1.2: Baruah et al. [11]

Any (single) sporadic DAG task $\tau_x = (G_x, D_x, T_x)$ satisfying the following properties

1. $L_x \leq \frac{2D_x}{5}$
2. $vol_x \leq \frac{2mT_x}{5}$

is EDF-schedulable on m unit-speed processors.

Bonifaci et al. [25] addressed the same scheduling problem, extending it to a DAG task set with arbitrary deadlines and considering both fully preemptive G-EDF and Global Deadline Monotonic (G-DM).

They proposed a polynomial sufficient test for the G-EDF case (Theorem 3.1.3).

Theorem 3.1.3: Bonifaci et al. [25]

Assume a sporadic DAG task system satisfies the following conditions:

1. $L_x \leq \frac{D_x}{3}, x = 1, 2, \ldots, n$
2. for each $x, x = 1, 2, \ldots, n$

$$\sum_{y:T_y \leq D_x} \frac{vol_y}{T_y} + \sum_{y:T_y > D_x} \frac{vol_y}{D_x} \leq \frac{(m + \frac{1}{2})}{3} \tag{3.2}$$

Then the system is EDF-schedulable on m unit-speed processors.

Moreover, they proposed two polynomial sufficient tests for G-DM, one for arbitrary deadlines (Theorem 3.1.4) and the other for constrained deadlines (Theorem 3.1.5), with slightly better guarantees.

Theorem 3.1.4: Bonifaci et al. [25]

Assume a sporadic DAG task system satisfies the following conditions:

1. $L_x \leq \frac{D_x}{5}, x = 1, 2, \ldots, n$
2. for each $x, x = 1, 2, \ldots, n$

$$\sum_{y:T_y \leq 2D_x} \frac{vol_y}{T_y} + \sum_{i:T_y > 2D_x} \frac{vol_y}{4D_x} \leq \frac{(m + \frac{1}{4})}{5} \tag{3.3}$$

Then the system is DM-schedulable on m unit-speed processors.

Theorem 3.1.5: Bonifaci et al. [25]

Assume a sporadic DAG task system satisfies the following conditions:

1. $L_x \leq \frac{D_x}{4}, x = 1, 2, \ldots, n$
2. for each $x, x = 1, 2, \ldots, n$

$$\sum_{y:T_y \leq 2D_x} \frac{vol_y}{T_y} + \sum_{i:T_y > 2D_x} \frac{vol_y}{D_x} \leq \frac{(m + \frac{1}{3})}{4} \tag{3.4}$$

3. $D_x \leq T_x, x = 1, 2, \ldots, n$

Then the system is DM-schedulable on m unit-speed processors.

Li et al. [70] focused on the schedulability of a DAG tasket with G-EDF, and they proposed a sufficient test for tasks with implicit deadlines (Theorem 3.1.6).

Theorem 3.1.6: Li et al. [70]

G-EDF can successfully schedule an implicit deadlines task set Γ of n DAG tasks, if the following conditions are satisfied:

1. $D_x = T_x, x = 1, 2, \ldots, n$
2. $U(\Gamma) \leq \frac{m}{(4 - \frac{2}{m})}$
3. $L_x \leq \frac{1}{(4 - \frac{2}{m})} T_x, x = 1, 2, \ldots, n$

Qamhieh et al. [96] later on, proposed another sufficient schedulability test for G-EDF of sporadic DAG task sets with constrained deadlines.

The authors analyzed DAG tasks by considering their internal structures and providing a tighter bound on the workload and interference analysis. Indeed, they showed that ignoring the structure of the DAGs results in a pessimistic analysis.

Their approach consists of assigning a local offset and local deadline for each subtask in the DAG.

Considering a DAG task τ_x, a *local offset* O_i of its vertex v_i is defined as the earliest possible release time w.r.t the activation of τ_x in which vertex v_i can be ready, and this is the length of the longest path from the starting vertex in τ_x to v_i. The calculation of the local offset of each vertex in the DAG is done assuming that the system has an infinite number of processors. This local offset is also known in the literature as Earliest Starting Time (EST). To compute all the local offset, a depth-first search algorithm, that executes in linear time, can be applied on each DAG task.

On the other hand, a *local deadline* D_i of vertex v_i is defined as the latest possible deadline of v_i with $D_i = D_x - R(v_i)$, where $R(v_i)$ is the length of the longest execution path from a successor of vertex v_i to the ending vertex in the DAG (without including the WCET of v_i). The local deadline is also known in the literature as Latest Finishing Time (LFT). If a vertex v_i misses its local deadline D_i at time t, the DAG τ_x will definitely miss its deadline even if $t \leq D_x$.

Moreover, for their schedulability test, they better analyzed interference between different tasks, taking into account the roles of body and carry-in jobs. The derived test is presented in Theorem 3.1.7.

Theorem 3.1.7: Qamhieh et al. [96]

A DAG task set Γ is G-EDF schedulable on m processors of unit speed if:

$$\begin{aligned} &\forall x \in \{1, \ldots, n\} \\ &\sum_{y=1}^{n} \sum_{v_i \in V_y} \max(0, \left(\left\lfloor \frac{D_x - D_i}{T_y} \right\rfloor + 1 \right) \times C_i) + \\ &+ \sum_{y=1, y \neq x}^{n} \sum_{v_i \in V_y} \min(C_i, \max(0, D_i)) \leq D_x \end{aligned} \tag{3.5}$$

Considering the left part of the inequality in Equation (3.5), the first member is a Demand Bound Function (DBF) [13] and represents the interference of body jobs; the second denotes the interference of carry-in jobs instead. The test is sustainable for sporadic DAG w.r.t. decreased execution requirements and later arrival times, but it is not w.r.t. larger relative deadlines.

Baruah [4] later improved the test for constrained deadline systems introduced by Bonifaci et al. [25], proposing a test that dominates the previous solution. To do so, the author exploits the concept of *work function*, introduced by Bonifaci et al. [25], and also later explained and generalized by Baruah et al. [8].

Let τ_x denote a sporadic DAG task, and s any positive real number ≤ 1. Let J denote any collection of jobs legally generated by the task τ_x. For an interval I, let $work(J, I, s)$ denote the amount of execution occurring within the interval I in the schedule $S_\infty(J, s)$, of jobs with deadlines that fall within I. For any positive integer t, let $work(J, t, s)$ denote the maximum value $work(J, I, s)$ can take, over any interval I of duration equal to t. Then, let $work(\tau_x, t, s)$ denote the maximum value of $work(J, t, s)$, over all job sequences J that may be generated by the sporadic DAG task τ_x . Finally, to extend the definition of the work function from individual tasks to a DAG task system Γ , let $work(\Gamma, t, s)$ denote the sum $\sum_{\tau_x in \Gamma} work(\tau_x, t, s)$.

Theorem 3.1.8: Baruah [4]

Let σ denote any constant < 1. Sporadic DAG task system Γ is G-EDF schedulable on m unit-speed processors if $\delta_{max}(\Gamma) \leq \sigma$, and

$$work(\Gamma, t, \sigma) \leq (m - (m-1)\sigma) \times t \tag{3.6}$$

for all values of $t \geq 0$.

In order to show that a given τ is EDF-schedulable upon m unit-speed processors it suffices (according to Theorem 3.1.8), to produce a witness to this fact: a value for σ such that Equation (3.6) holds for all $t \geq 0$. The schedulability test presented by Bonifaci et al. [25] essentially reduces to determining a specific value of σ, namely $\sigma = m/(2m-1)$. On the contrary, the pseudo-polynomial algorithm derived by Baruah correctly identifies all task systems for which any σ would cause Equation (3.6) to evaluate t to true for all $t \geq 0$. Therefore, this test dominated the one by Bonifaci et al. [25], which is also proven by the author.

Qamhieh and Midonnet [94] analyzed two DAG scheduling methods used in literature, i.e. (i) the parallel scheduling method and (ii) the stretching scheduling method. In the former parallel tasks are scheduled directly using common scheduling algorithms [11, 25, 70, 4]; in the latter DAG tasks are transformed into independent sequential tasks with intermediate offsets and deadlines that execute on multiprocessor systems [96].

The authors considered both methods and showed that are not comparable regarding schedulability in the case of G-DM and G-EDF scheduling algorithms, focusing on the implicit deadline case. They concluded that (i) no one dominates the other; (ii) the stretching method is more adapted to fixed task priority assignment algorithm such as DM; (iii) parallel scheduling method performs better in the case of preemptive G-EDF scheduling algorithm, which means is more convenient to the FTP assignment scheduling algorithms.

Melani et al. [81] derived efficient ways to compute an upper-bound on the response-time of each DAG task using different global scheduling algorithms.

A simple but pessimistic WCRT bound R_x for a DAG task G_x was already proposed by Graham [54] (Theorem 3.1.9).

Theorem 3.1.9: Graham [54]

A safe WCRT bound for a DAG G_x on a multiprocessor can be computed as:

$$R_x = L_x + \frac{1}{m}(vol_x - L_x) \tag{3.7}$$

where L_x is the length of the longest path in G_x; vol_x is the total execution time of all vertices in G_K; and m is the number of cores.

However, Graham bound does not take into account inter-task interference. The authors focused on finding an upper bound on the inter-task interference, considering carry-in, body, and carry-out, and used the Graham bound for intra-task interference.

They derived an upper-bound on the workload of an interfering task τ_y in a window of length t:

$$\mathcal{W}_y(t) = \left\lfloor \frac{t + R_y - vol_y/m}{T_y} \right\rfloor vol_y + \min\left(vol_y, m \cdot \left(\left(t + R_y - vol_y/m\right) \bmod T_y\right)\right). \tag{3.8}$$

where the first part of the equation represents the upper bound on carry-in and body jobs, the second the one on carry-out.

Moreover, they derived an upper-bound on the interfering workload of a task τ_y on a task τ_x with G-EDF:

$$\mathcal{I}_{y,x} = \left(\left\lfloor \frac{D_x - D_y}{T_y} \right\rfloor + 1\right) vol_y + \min\left(vol_y, m \cdot \max\left(0, D_x \bmod T_y - D_y + R_y\right)\right) \tag{3.9}$$

and again, the first term accounts for carry-in and body, the second for carry-out. Finally, they derived a schedulability test based on Response Time Analysis (RTA) both for G-EDF and global Fixed-Priority (G-FTP) algorithms (Theorem 3.1.10).

Theorem 3.1.10: Melani et al. [81]

Given a task set Γ globally scheduled on m cores, an upper-bound R_x^{ub} on the response-time of a task τ_x can be derived by the fixed-point iteration of the following expression, starting with $R_x^{ub} = L_x$:

$$R_x^{ub} \leftarrow L_x + \frac{1}{m}\left(vol_x - L_x\right) + \left\lfloor \frac{1}{m} \sum_{\forall y \neq x} \mathcal{X}_y^{ALG} \right\rfloor \tag{3.10}$$

With global FP:

$$\mathcal{X}_y^{ALG} = \mathcal{X}_y^{FTP} = \begin{cases} \mathcal{W}_y\left(R_x^{ub}\right), & \forall y < x \\ 0, & \text{otherwise} \end{cases} \tag{3.11}$$

When using global EDF:

$$\mathcal{X}_y^{ALG} = \mathcal{X}_y^{EDF} = \min\left\{\mathcal{W}_y\left(R_x^{ub}\right), \mathcal{I}_{y,x}\right\} \tag{3.12}$$

For any work-conserving scheduler

$$\mathcal{X}_y^{ALG} = \mathcal{W}_y\left(R_x^{ub}\right) \tag{3.13}$$

Parri et al. [89] proposed polynomial and pseudo-polynomial time schedulability tests, based on RTA, for G-EDF and G-DM scheduling of sporadic DAG-tasks with arbitrary deadlines and arbitrary vertex utilization.

The proposed method dominates the one proposed by Bonifaci [25], however, it is way more complicated and less trivial to apply.

Pathan et al. [90] were the first to propose a two-level preemptive G-FTP for constrained deadlines: a task-level scheduler first determines the highest-priority ready task and a subtask-level scheduler then selects its highest-priority subtask for execution.

In the task-level priority assignment phase, the fixed priorities of the tasks are assigned based on DM priority sorting. The set of tasks having higher fixed priorities

than that of task G_x is denoted by $hp(\tau_x)$. In the subtask-level priority assignment phase, the fixed priorities to the subtasks of G_x are assigned based on a topological order of the subtasks of G_x .

Then they proposed new techniques to compute both intra- and inter-task interference and derive a new schedulability test which reduces the pessimism of the analysis by Melani et al. [81].

The total intra-task interfering workload for a subtask v_i of τ_x, denoted by $\mathcal{W}_{x,i}^{intra}$, is due to the subtasks in set $\mathcal{S}_{x,i}$ and is computed as:

$$\mathcal{W}_{x,i}^{\text{intra}} = \sum_{v_j \in \mathcal{S}_{x,i}} \min \left\{ C_j, \max \left\{ 0, R_{x,j} - \max_{v_k \in Ancst_{x,i}} R_{x,k} \right\} \right\} \tag{3.14}$$

where $Ancst_{x,i}$ is the set of all the ancestors (or predecessors) of v_i and $\mathcal{S}_{x,i}$ comprises all the higher-priority subtask of v_i that are not its ancestors. This formulation is possible because the response times are computed in topological order, therefore all the ancestors will come before their successors.

The inter-task interfering workload $\mathcal{W}_{x,i}(t)$ of all the tasks with higher priority that τ_x ($hp(\tau_x)$), within the problem window t of v_i is:

$$\mathcal{W}_{x,i}^{\text{inter}}(t) = \sum_{\tau_y \in hp(\tau_x)} \left(\text{CR}_y\left(t_{\text{cin}}\right) + \left\lfloor \frac{\mathcal{X}_y(t)}{T_y} \right\rfloor \cdot W_y + W_y \right) \tag{3.15}$$

The first term inside the summation accounts for the carry-in; t_{cin} is the length of the interval inside the problem window where the carry-in job execute, $\text{CR}_y\left(t_{\text{cin}}\right)$ is the maximum inter-task interfering workload of the carry-in job of τ_y in t_{cin}. The second term accounts for the body job, $\mathcal{X}_y(t)$ is the length of the interval in which body jobs interfere; and the last term accounts for the carry-out.

Theorem 3.1.11: Pathan et al. [90]

The response time $R_{x,i}$ of subtask v_i of a DAG task τ_x can be found starting from $R_{x,i}^{(0)} = C_i$ and using the fixed-point recurrence:

$$R_{x,i}^{(t+1)} \leftarrow \max_{v_j \in Ancst_{x,i}} R_{x,j} + \frac{\mathcal{W}_{x,i}^{\text{intra}} + \mathcal{W}_{x,i}^{inter}(R_{x,i}^{(t)})}{m} + C_i \tag{3.16}$$

The first term on the right-hand side or Equation (3.16) is the latest time when subtask v_i becomes ready for execution, the second term represents the interference on subtask v_i within a problem window of size $R_{x,i}^{(t)}$ and finally, the third term C_i is the WCET of subtask v_i.

Theorem 3.1.12: Pathan et al. [90]

Given the response time $R_{x,i}$ of each subtask v_i of task τ_x, the response time R_x of the task can be computed based as

$$R_x = \max_{v_i \in V_x} \{R_{x,i}\} = R_{x,sink} \tag{3.17}$$

All the tasks in set $\Gamma = \{\tau_1, \tau_2, \ldots \tau_n\}$ meet their deadlines if $R_x \leq D_x$ for $x = 1, 2, \ldots n$.

Finally the authors theoretically proved that the response-time test in Equation (3.16) dominates the response-time test in Equation (3.10) with $\mathcal{X}_y^{ALG} = \mathcal{X}_y^{FTP}$.

Fonseca et al. [48] also presented an RTA technique based on the concept of problem window for the G-FTP scheduling of sporadic DAG tasks.

They proposed an algorithm to compute a tighter upper bound on the intra-task interference, with improved carry-in and carry-out workload computation, from which they derived a new schedulability test (Theorem 3.1.13).

The analysis is based on the concept of workload distribution, both for carry-in and carry-out, and takes into account the topology of the DAG task. A tricky part in the computation of the carry-out workload is the conversion of the DAG in a Nested Fork-Join (NFJ) DAG and then in a sequential-parallel decomposition binary tree (SPTree) [118]. Then they derived an algorithm to account for the whole interfering workload $\mathcal{W}_y(\Delta)$ for a time interval Δ, combining the carry-in and carry-out contributions.

Theorem 3.1.13: Fonseca et al. [48]

A task τ_x is schedulable under G-FTP iff $R_x \leq D_x$ where R_x is the smallest $\Delta > 0$ to satisfy

$$\Delta = L_x + \frac{1}{m}(vol_x - L_x) + \frac{1}{m}\sum_{\tau_y \in hp(\tau_x)} \mathcal{W}_y(\Delta) \tag{3.18}$$

The task set is declared schedulable if all tasks are schedulable. This can be checked by applying Equation (3.18) to each task $\tau_x \in \Gamma$, starting from the highest priority task and proceeding in decreasing order of priority.

The authors proved that the proposed test empirically dominates Equation (3.16) [90].

He et al. [61] propose a priority assignment algorithm to assign priorities to vertices, such that the response time bound is reduced as much as possible.

Therefore, the authors proposed prioritized list scheduling, which is work conserving and preemptive.

The work is based on the fact that the choice of eligible vertices for execution affects the actual response time. Intuitively, one should prioritize vertices along the longest path for execution in order to get a smaller response time. The critical path depends on how the DAG is actually scheduled, i.e., the critical path of a DAG may be different in different execution sequences of the DAG.

Theorem 3.1.14: He et al. [61]

The response time R_x of a DAG task τ_x with constrained deadline scheduled by prioritized list scheduling on a platform with m cores can be bounded by

$$R_x \leq \max_{\lambda \in \Pi_x}\{\sum_{v_i \in \lambda} C_i + \frac{\sum_{\cup_{v_i \in \lambda} I(v_i)} C_i}{m}\} \tag{3.19}$$

where Π_x is the set of all complete paths of the DAG G_x and $I(v_i)$ is the interference set of a vertex $v_i \in V_x$ defined as

$$I(v_i) = \{v_j \in V_x \backslash \{v_i\} \mid v_j \notin Ancest_i \wedge v_j \notin Desc_i \wedge prio(v_j) \leq prio(v_i)\}$$

Equation (3.19) dominates the classic bound in Equation (3.7) [54].

However, the number of paths in a DAG can be exponential in the size of the DAG. So it is impractical to enumerate all the paths to compute the response time

bound. Therefore, the authors used dynamic programming to solve the problem, deriving an algorithm with complexity $O(|V| + |E|)$.

In the task level, the scheduling algorithm is the same as Melani [81], which can be any global work-conserving scheduler, such as EDF, RM. In the vertex level, the vertices inside a task are scheduled by prioritized list scheduling. The authors proposed an algorithm that first priorities vertices based on the topological order, and later assign higher priorities to vertices belonging to longer paths.

Theorem 3.1.15: He et al. [61]

For a constrained deadlines DAG task set Γ scheduled by global prioritized list scheduling on a platform with m cores, a bound R_x on the response time of a task τ_x can be derived by the fixed-point iteration of the following expression, starting with $R_x = L_x$

$$R_x \leftarrow \max_{\lambda \in \Pi_x} \left\{ \sum_{v_i \in \lambda} C_i + \frac{\sum_{\cup_{v_i \in \lambda} I(v_i)} C_i}{m} \right\} + \frac{1}{m} \sum_{\forall y \neq x} \mathcal{X}_y^{ALG} \tag{3.20}$$

where $\mathcal{X}_y^{ALG}$ is the one from Melani et al. [81] (Theorem 3.1.10).

The authors showed that Theorem 3.1.15 dominates Theorem 3.1.10 [81] both theoretically and empirically.

Fonseca et al. [49] studied again the RTA problem for DAG task set on multiprocessor with global policy, and propose two techniques to derive less pessimistic upper-bounds on the workload produced by the carry-in and carry-out jobs of the interfering tasks. They used the same formulation as Theorem 3.1.13 [48] but they proposed new algorithms to compute $\mathcal{W}_y(\Delta)$, both for constrained and arbitrary deadlines.

They compared their solution with Melani [81] and Parri [89] and they empirically dominated the first method. However, the complexity of the method and the latency of the tests are not evaluated.

3.1.2 Global Policy, Limited Preemption (G-LP)

Saifullah et al. [102] considered the problem of G-EDF within the limited preemptive policy, which means that nodes are not preemptive, but the DAG task can be preempted at node boundaries.

They proposed a technique to decompose the DAG task. Upon decomposition, each node of a DAG becomes an individual sequential task with its own deadline and offset, and with a WCET equal to the node's execution requirement.

After the decomposition, they applied literature results both for the fully preemptive [6] and limited preemptive [12] G-EDF for constrained deadline sporadic sequential task set.

Serrano et al. [107] also focused on limited preemption, combining the results of Melani [81] and Thekkilakattil [114].

Within the limited preemption context, tasks are not only interfered by higher-priority tasks, but also by already started lower-priority tasks whose execution has not reached a preemption point yet, and so cannot be suspended. The higher priority

interference I_x^{hp} is computed as in Equation (3.11) [81]. The lower priority interference considering G-FTP scheduling with eager preemptions is:

$$I_x^{lp} = \Delta_x^m + pn_x \times \Delta_x^{m-1} \tag{3.21}$$

where pn_x is an upper-bound on the number of preemptions suffered by τ_x, and Δ_x^m and Δ_x^{m-1} are upper-bounds on the lower-priority interference on the first NPR and the p^{th} NPRs of task τ_x, respectively. The easiest way of deriving the lower priority interference is to account for the m and $m-1$ largest NPRs among all lower-priority tasks as:

$$\Delta_x^m = \sum \max_{\tau_y \in lp(\tau_x)}^{m} (\max_{\forall v_i \in V_y}^{m} C_i) \tag{3.22}$$

$$\Delta_x^{m-1} = \sum \max_{\tau_y \in lp(\tau_x)}^{m-1} (\max_{\forall v_i \in V_y}^{m-1} C_i) \tag{3.23}$$

Theorem 3.1.16: Serrano et al. [107]

Given a task set Γ scheduled on m cores with G-FP and limited preemption, an upper-bound R_x^{ub} on the response-time of a task τ_x can be derived in pseudo-polynomial time by the fixed-point iteration of the following expression, starting with $R_x^{ub} = L_x$:

$$R_x^{ub} \leftarrow L_x + \frac{1}{m}(vol_x - L_x) + \left\lfloor \frac{1}{m}(I_x^{lp} + I_x^{hp}) \right\rfloor \tag{3.24}$$

Despite its simplicity, this strategy is pessimistic because it considers that the largest m and $m-1$ NPRs can execute in parallel, regardless of the precedence constraints defined in the DAG. Therefore, the authors presented also a pseudo polynomial Integer Linear Programming (ILP) method to calculate the blocking impact of the largest Parallel NPRs which is less pessimistic and dominates the previous one.

Serrano et al. [106] further investigated the results for G-FTP with limited preemption, analyzing the eager and lazy cases, using the scheduling techniques derived in their previous work [107].

The authors concluded that LP-eager clearly outperforms LP-lazy, despite a higher number of priority inversions are considered in the RTA, and a high number of preemptions are observed at system deployment. Therefore, contrary to what has been demonstrated when considering sequential task-sets, the LP lazy scheduling approach has been proven to be a very inefficient scheduling strategy when DAG-based task-sets are considered, and so not suitable for parallel execution.

Nasri et al. [86] provided a schedulability analysis for global limited-preemptive earliest-deadline first (G-LP-EDF) or fixed-priority (G-LP-FTP) scheduling.

The analysis constructs a schedule-abstraction graph that abstracts all the possible orderings of job dispatch times resulting from the underlying scheduling policy, based on which the authors derived bounds on the best- and worst-case response time of each job. Because jobs experience release jitter and execution time variation, exponentially many execution scenarios exist, and the exact finishing time of each job cannot be known a priori. For this reason, they considered an interval

$[EFT_x, LFT_x]$ in which a job τ_x^a will finish. The authors proposed a new algorithm to compute EST and LST of the jobs.

The schedule-abstraction graph is built iteratively in two alternating phases: expansion and merging. The expansion phase expands (one of) the shortest path(s) λ in the graph by considering all jobs that can possibly be dispatched next in the job-dispatch sequence represented by λ. The merge phase slows down the growth of the graph by merging, whenever possible, the terminal vertices of paths that have the same set of dispatched jobs. The search ends when there is no vertex left to expand, that is, when all paths represent a valid schedule of all jobs considered, which implies that all possible schedules have been explored.

In conclusion, the proposed analysis dominates Serrano [107] and many other methods for sequential tasks. However, it does not scale to highly parallel DAG tasks or systems with a large number of cores (e.g., more than 64).

The code was made publicly available[1].

3.1.3 Federated Scheduling

Li et al. [69] analyzed for first the problem of federated scheduling of DAG tasks on multiprocessors.

Given a task set Γ, tasks are divided into two disjoint sets: Γ_{high} contains all high-utilization tasks ($\forall \tau_x \in \Gamma | U_x \geq 1$), and Γ_{low} contains all the remaining low-utilization tasks. Considering a high-utilization task τ_x, m_x dedicated cores are assigned to it, m_x is

$$m_x = \left\lceil \frac{vol_x - L_x}{D_x - L_x} \right\rceil \tag{3.25}$$

m_{high} denotes the total number of cores assigned to high-utilization tasks and is computed as $m_{high} = \sum_{\tau_x \in \Gamma_{high}} m_x$. The remaining cores $m_{low} = m - m_{high}$ are assigned to all the low-utilization tasks.

Theorem 3.1.17: Li et al. [69]

The federated scheduling algorithm admits the task set Γ, if

1. m_{low} is non-negative;
2. $m_{low} \geq 2 \sum_{\tau_x \in \Gamma_{low}} U_x$.

If the task set is feasible then (i) any work-conserving parallel scheduler can be used to schedule a high-utilization task τ_x on its assigned m_x cores; (ii) low-utilization tasks are treated and executed as sequential tasks and any multiprocessor scheduling algorithm can be used to schedule Γ_{low} on m_{low} cores.

Baruah [7] introduced a two-phase algorithm for the federated scheduling of any system of constrained-deadline sporadic DAG tasks.

In the first phase, the author developed an algorithm for each high-density task (i.e., $\forall \tau_x \in \Gamma | \delta_x \geq 1$), in which the number of processors to be devoted to this task is determined. Then the schedulability is checked with Graham's list scheduling algorithm [54].

In the second phase, the remaining low-density tasks are partitioned upon the remaining processors. Since any intra-task parallelism cannot be exploited upon a

[1] https://github.com/brandenburg/np-schedulability-analysis

single processor, the internal structure of the DAG can be ignored and each task can be represented in the simpler three-parameter sporadic model. An algorithm to partition the tasks on the cores based on DBF [13] is presented. During run-time, each shared processor is scheduled using preemptive uniprocessor EDF.

Baruah [3] later on the same year, extended his previous work also for arbitrary deadline systems. The main difference w.r.t the previous work is an extension of Graham's list scheduling algorithm with constraints in order to apply to arbitrary deadlines.

The method is efficiently implementable in time that is polynomial in the representation of the tasks in Γ and the number of processors m.

Jiang et al. [64] studied the analysis of G-EDF scheduling for DAG parallel tasks with arbitrary deadlines, focusing on the case of $D > T$.

Theorem 3.1.18: Jiang et al. [64]

Any DAG task τ_x is schedulable on m processors when $U_x \leq m$ if at least one of the following conditions holds.

$$\frac{vol_x\lceil U_x\rceil + (m-1)L_x}{m} \leq D_x \tag{3.26}$$

$$\frac{U_x \times L_x}{m - U_x} + \frac{vol_x + (m-1)L_x}{m} \leq D_x \tag{3.27}$$

The schedulability tests in Theorem 3.1.18 can be easily extended to the analysis of multiple DAG tasks under federated scheduling, where each heavy task $(U_x > 1)$ is assigned several dedicated processors and exclusively executes on them while all light tasks $(U_x \leq 1)$ share the remaining processors together as if they were sequential tasks.

Then the schedulability test for a DAG task set Γ has two parts: (i) assign dedicated processors to each heavy task according to Theorem 3.1.18 if there are enough processors, otherwise, return failure, and (ii) test whether all light tasks are schedulable on the remaining processors treating them as arbitrary-deadline sequential tasks under partitioned EDF (as in the work of Baruah [3]).

3.1.4 Partitioned Scheduling

Fonseca et al. [50] focused on the partitioned scheduling for fixed-priority (P-FP) scheduling of DAG tasks on multiprocessors, assuming the partitioning to be given, fully preemptive policy and constrained deadlines. The authors showed that a partitioned DAG task can be modeled as a set of self-suspending tasks and propose an algorithm to traverse a DAG and characterize the worst-case scheduling scenario.

In their setting, each vertex of a DAG task is assigned to a specific core. Then, different vertices of the DAG can run in parallel over the multiprocessor platform but they are not allowed to migrate. Therefore, each subtask is characterized not only by its WCET, but also the core to which it is assigned p_i; That is, $v_i = (C_i, p_i)$. They denoted by $\lambda_{x,l}$ a path l of task τ_x and by $|\Pi_x|$ the number of all the possible different paths $\lambda_{x,l} \in \Pi_x$.

Theorem 3.1.19: Fonseca et al. [50]

The worst-case response time of a path $\lambda_{x,l}$ of a partitioned DAG task τ_i is given by

$$R(\lambda_{x,l}) = len(\lambda_{x,l}) + I_x(\lambda_{x,l}) + \sum_{\forall \tau_y \in hp(\tau_x)} I_y(\lambda_{x,l}) \tag{3.28}$$

where $len(\lambda_{x,l})$ is the length of the path $\lambda_{x,l}$; $I_x(\lambda_{x,l})$ is the self-interference and $\sum_{\forall \tau_y \in hp(\tau_x)} I_j(\lambda_{x,l})$ is the inter-task interference.
Consequently, the worst-case response time of a DAG task τ_x is given by

$$R_x = \max_{l=1}^{|\Pi_x|} R(\lambda_{x,l}) \tag{3.29}$$

A DAG task τ_x is deemed schedulable if $R_x \leq D_x$.

A path λ can be modeled as a set of sporadic self-suspending tasks, one for each core reached by the path, i.e. $\forall p \in proc(\lambda)$. The idea is to treat the subtasks of λ assigned to the current core under analysis as execution regions and the response time of all the remaining subtasks as suspension regions. The problem of computing the response time of λ on a multicore platform becomes then equivalent to the analysis of $|proc(\lambda)|$ self-suspending tasks in a uniprocessor system. However, unlike previous works, the duration of the suspension regions is not known beforehand as they are in fact computations to be executed on different cores.

The authors proposed an algorithm that recursively divides a path into smaller ones, creating a tree of subpaths which represent self-suspending tasks. The resulting tree reflects the hierarchy of dependencies. When a leaf is reached, the corresponding task has no suspension regions (i.e., it is a sequential task), thus its WCRT does not depend on anything else other than the interfering workload on that core and can be computed immediately. The computed values are then backpropagated to the self-suspending tasks on the upper levels so that their suspension time is no longer unknown.

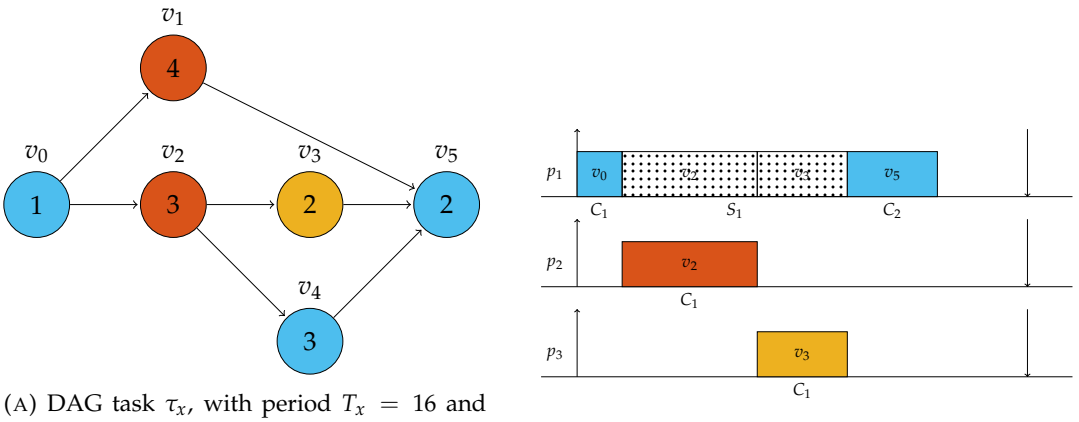
The WCRT of a single subtask can then be computed by adding the self-interfering workload to the traditional equation for fixed-priority sequential tasks in uniprocessors.

The authors then showed how three different state-of-the-art response time analyses [23, 88] for sporadic self-suspending tasks can be extended to cope with both the dependent suspension regions and the self-interfering workload.

Casini et al. [35, 34] focused again on P-FTP, analyzing the non-preemptive scenario, or better, the limited preemptive case for DAG tasks. They proposed both a new analysis that exploits self-suspending tasks and a partitioning algorithm.

A segmented self-suspending task τ_x^{ss} is characterized by an ordered sequence of N_x^S execution segments alternated by self-suspensions, both with bounded duration and represented by the tuple $\left\langle C_{x,1}, S_{x,1}, \ldots, S_{x,N_x^S-1}, C_{x,N_x^S} \right\rangle$, where $C_{x,i}$ denotes the WCET of the i -th execution segment of τ_x^{ss} and $S_{x,i}$ denotes the maximum duration of the i -th self-suspension of τ_x^{ss}.

They proposed an algorithm to convert a DAG into a set of segmented self- suspending tasks, based on Fonseca et al. [50]. The algorithm takes as input a path $\lambda_{x,l}$ that must begin and end with nodes allocated to the same processor p_i . It returns a self-suspending task modeling the execution of $\lambda_{x,l}$ on p_i, and an upper bound



(A) DAG task τ_x, with period $T_x = 16$ and deadline $D_x = 10$, partitioned on processors p_1 (light blue), p_2 (orange) and p_3 (yellow).

(B) Self suspended tasks generated by the path $\{v_0, v_2, v_3, v_5\}$.

FIGURE 3.2: Example of conversion of a path in the DAG into self-suspend tasks

on the total suspension time that the self-suspending task may incur. To better undestrand, let us consider Figure 3.2. Figure 3.2a represents a DAG partitioned on three processors: $\{v_0, v_4, v_5\}$ are assigned to processor p_1 (light blue), $\{v_1, v_2\}$ are assigned to processor p_2 (orange), and $\{v_2\}$ is assigned to processor p_3 (yellow). Figure 3.2b shows the analysis of the path $\{v_0, v_2, v_3, v_5\}$: for processor p_2 and p_3 two self-suspended task with a single execution segment are derived; for processor p_1 a single self-suspended task is obtain with two execution segments (represented by v_0 and v_5) and a single suspension regions that accounts for v_2 and v_3. To complete the analysis, the self interference and the high priority tasks interference needs to be accounted; moreover this procedure has to be performed for each path in the DAG.

The authors then proposed two safe response-time bounds for self-suspending non-preemptive tasks.

Theorem 3.1.20: Casini et al. [35]

The response-time of a self-suspending task τ_x^{ss} is bounded by

$$R_x = R'_x + C_{x,N_x^S} \tag{3.30}$$

where R'_x is given by the least positive fixed point of the following recursive equation:

$$R'_x = \sum_{i=1}^{N_x^S - 1} (C_{x,i} + S_{x,i}) + \sum_{b \in \mathcal{B}_x\left(N_x^S, R'_x, \overline{\mathbf{R}}\right)} b + I_x\left(R'_x\right) \tag{3.31}$$

that is the sum of (i) the execution of the task, (ii) its maximum blocking time and (iii) the maximum interference experienced.

Theorem 3.1.21: Casini et al. [35]

The response-time of the l-th segment of a self-suspending task τ_x^{ss} is bounded by

$$R_{x,l} = \sum_{i=1}^{l} C_{x,i} + \sum_{i=1}^{l-1} S_{x,i} + \sum_{b \in \mathcal{B}_x\left(l, r_{x,l}(\overline{\mathbf{R}}), \overline{\mathbf{R}}\right)} \Delta_x(b, \overline{\mathbf{R}}) \tag{3.32}$$

where $r_{x,l}(\overline{\mathbf{R}})$ is the latest release time of the l-th segment of τ_x^{ss} and $\Delta_x(b, \overline{\mathbf{R}})$ is the maximum time it can be delayed from its release up to the time it starts executing.

They showed that both Theorem 3.1.20, at self-suspended task level, and Theorem 3.1.21, at segment level, offer safe response-time bounds for each task τ_x^{ss} and its execution segments. Therefore the minimum of the two is still a safe response-time bound. They proposed an algorithm to combine the two to implement a schedulability test for segmented non-preemptive self-suspending tasks. The algorithm is then applied to study the response time of a path, including also the self-interference of a DAG task.

Finally, the authors presented an algorithm for partitioning parallel tasks upon the various cores of a multicore platform. The algorithm leverages the analysis presented in the previous section and is general enough to be combined with several partitioning heuristics such as First-Fit, Worst-Fit, and Best-Fit, and different orderings with which the tasks are selected.

In addition to that, the same authors at the same conference (RTSS 2018) [34] proposed solutions for bounding the worst-case memory space requirement the DAG task set analyzed running on multicore platforms with scratchpad memories. They introduced a feasibility test that verifies whether memories are large enough to contain the maximum memory backlog that may be generated by the system. Both closed-form bounds and more accurate algorithmic techniques are proposed.

Casini et al. [33] further extended their previous works [34], proposing a fine-grained analysis of the memory contention experienced by parallel tasks running on a multi-core platform.

The DAG task is then extended to include memory latencies. The edges also represent producer-consumer communications between nodes. Each edge $e_{i,j} = (m_{i,j}, \xi_{i,j})$ among vertices v_i and v_j, is associated with a weight $m_{i,j}$, and a worst-case memory access time $\xi_{i,j}$. Specifically, $m_{i,j}$ denotes the number of transactions needed to transfer from global memory the data produced by node $v_{x,i}$ and consumed by $v_{x,j}$, while $\xi_{i,j}$ denotes the maximum amount of time needed to perform $m_{i,j}$ requests in isolation, i.e., without contention generated by nodes running on the other cores. For each communication $e_{i,j} \in E_x$, the corresponding data buffer is allocated to a single DRAM bank.

The execution of the nodes follows a three-phase scheme. First, a copy-in phase is performed to load into the local memory all the data corresponding to global communications (stored in the global DRAM). Once the copy-in phase is completed, the node can execute, only accessing the local memory (i.e., it cannot experience memory contention). Finally, when the node execution is completed, a copy-out phase is performed to store in the global DRAM memory all the data related to global communications. A node completes after the termination of its copy-out phase.

For the sake of completeness, inter-task communication between two nodes $v_{x,i} \in V_x, v_{y,j} \in V_y$ belonging to different tasks are modeled with dummy nodes and edges. In particular, if $v_{x,i}$ is producing data for $v_{y,j}$, a dummy node $v_{x,k}$ is added to V_x, and a dummy edge $e(v_{x,i}, v_{x,k})$ is added to E_x to connect $v_{x,i}$ and $v_{x,k}$. Dummy nodes have zero execution time and they are never executed. Pre-fetching of instructions from global memory and communications between successive jobs of the same task can be similarly handled.

The WCET of each vertex is computed as $C^*_{x,i} = R^{IN}_{x,i} + C_{x,i} + C^{OUT}_{x,i}$, where $R^{IN}_{x,i}$ is the upper bound on the response time experienced by the copy-in phase of a node $v_{x,i}$, $C_{x,i}$ it's its original WCET and $C^{OUT}_{x,i}$ is the contention-free WCET of the copy-out phase. Once the inflated WCETs are available for each task and node, a response-time analysis for parallel tasks under non-preemptive scheduling can be applied [35].

3.1.5 Exact Tests

As already mentioned, testing the feasibility of a DAG task system is NP-hard in the strong sense, and there exist no methods to solve it in polynomial time. Exact tests are however valuable, even if they can be applied only on small examples due to their computational costs, to be compared with sufficient tests for understanding their goodness. There are not many works on finding exact results for DAG tasks, e no one for DAG task set. In the following, the most related ones are reported.

Burmyakov et al.[27] proposed an exact schedulability test for sporadic real-time tasks with constrained deadlines, for G-FP.

The authors employed a set of techniques that cut down the state space of exploration of the analysis. They extended the prior work by Bonifaci et al. [25] and made their C++ code publicly available[2].

The greatest limitation is that the proposed method can evaluate DAG tasks that have only one path.

Yalcinkaya et al. [128] proposed the first exact schedulability test for LP (and FNP) self-suspending constrained real-time tasks scheduled upon a uniprocessor or multiprocessor platform (under a G-FP scheduling policy). The authors mapped the schedulability problem to the reachability problem in timed automata (TA), using TA extensions available in UPPAAL[3].

The work also provides the first exact baseline against which sufficient schedulability tests for self-suspensions and limited-preemptive models can be compared.

Sun et al. [110] studied the exact scheduling of a single non-recurrent DAG task executed on a multi-core platform under the list scheduling algorithm in a non-preemptive manner.

The basic idea of the method is borrowed from the traditional scheduling theory in the operational research domain, which aims at formulating the RTA problem into an optimization problem. The authors implemented the analysis by using Satisfaction Modular Techniques (SMT), and formulate the WCRT analysis problem for a DAG task under the list scheduling algorithm upon the multi-core platform into an SMT program.

[2] www.cister.isep.ipp.pt/docs/CISTER-TR-150503
[3] https://www.uppaal.com/

The number of constraints involved in the SMT model is bounded by $O(|V||E|)$. The authors provedthat their SMT, using the proposed constraints, can precisely solve the WCRT of DAGs under the list scheduling upon m cores.

Baruah [5] made an effort at obtaining an exact algorithm for scheduling a DAG on multiprocessors when processor assignment is specified. The processors are assumed to be preemptive and each processor is considered separately.

The author derived an algorithm for representing the scheduling problem as an ILP, which can then be solved using standard ILP-solvers. He showed that solving even this simple problem turned out to be surprisingly challenging: it required the author to draw upon, and integrate, disparate ideas from Operations Research and real-time scheduling theory (i.e. DBF) to synthesize the ILP, and then fall back on results from real-time scheduling theory - the optimality of preemptive uniprocessor EDF - to synthesize the actual schedule using the individual jobs' start-times and completion times as determined by the solution to the ILP.

The proposed ILP has a computational complexity of $\mathcal{O}(n^3)$, which means that the ILP is of size polynomial in the number of jobs.

3.1.6 Other Approaches

Finally, not-standard approaches are here described. These methods propose their schedulability algorithms and therefore cannot be fairly compared with the rest of the literature. They pertain though to this thesis because they try to solve the problem in unconventional ways, offering new points of view.

Qamhieh et al. [95] studied the global scheduling of n synchronous periodic parallel real-time graphs with implicit deadlines on m identical processor system. The schedulability is studied on the hyper-period of each task set.

Least Laxity First job priority assignment is adopted to schedule each subtask in the graph according to their laxity while considering the global deadline and period without the need to assign them local ones.

Guo et al. [56] studied for first the energy-aware real-time scheduling of a set of sporadic DAG tasks with implicit deadlines. While meeting all real-time constraints, they tried to identify the best task allocation and execution pattern such that the average power consumption of the whole platform is minimized. The authors first adapted the decomposition-based framework for federated scheduling and propose an energy-sub-optimal scheduler. Then they derived an approximation algorithm to identify processors to be merged together for further improvements in energy efficiency.

Yang et al. [130] focused on the scheduling of a DAG task set under partitioned EDF scheduling. They proposed a new method that adopts a hierarchical scheduling approach which divides the overall scheduling problem into two parts: (i) scheduling DAG tasks onto virtual processors, (ii) scheduling virtual processors on physical processors. More specifically, each DAG task is assigned with several dedicated virtual processors (and at runtime executes on them exclusively). With proper characterization of the resource provided by virtual processors, each DAG can be analyzed independently as in federated scheduling. On the other hand, virtual processors

are scheduled onto the physical processors at runtime which effectively enables the processor sharing among different DAG tasks.

The rationale behind this work is that the hierarchical scheduling approach inherits the strengths of both federated scheduling and global scheduling, and thus achieves better schedulability.

Guan et al. [55] proposed DAG-Fluid, a real-time scheduling algorithm for DAG task sets. DAG-Fluid is a combination of a task decomposition method and the corresponding scheduling algorithm based on fluid scheduling.

For a task τ_x , DAG-Fluid performs differently according to the value of U_x . If $U_x \leq 1$ the task is said light, and DAG-Fluid transforms τ_x into a non-parallel task where all of its subtasks are forced to execute sequentially with a constant execution rate. If $U_x > 1$, DAG-Fluid first decomposes the heavy task into several consecutive segments and then assigns an execution rate to each segment individually. Once rates have been assigned both to light and heavy tasks the scheduling is trivial: whenever the task (or subtask) is ready, it starts executing immediately at the assigned constant execution rate.

The offline part of the algorithm has polynomial computational complexity, and the online part has linear computational complexity.

3.2 Conditional DAG Tasks

Conditional DAGs are the first proposed extension to the DAG task. The task model is enriched with new types of vertices, namely conditional vertices, that represent conditions in the execution of the task. In this way, constructs as `if-then-else` or `switches` can be represented. The model becomes more expressive but also more complex.

Fonseca et al. [51] analyzed for first the problem of conditional branches within the DAG model, for scheduling of constrained-deadlines task set. They presented a multi-DAG model where each task is characterized by a set of execution flows, each of which represents a different execution path throughout the task code and is modeled as a DAG of sub-tasks.

The authors proposed a two-step solution that computes a single synchronous DAG of servers for a task modeled by a multi-DAG and show that these servers can supply every execution flow of that task with the required CPU-budget so that the task can execute entirely, irrespective of the execution flow taken at run-time while satisfying its precedence constraints.

In the multi-DAG model each task τ_x is characterized by a 3-tuple (F_x, T_x, D_x). Due to the control structures within τ_x 's code (e.g., the "if-then-else" statements), two different jobs of τ_x may execute two different parts of the code, called execution flows, represented by F_x. F_x is the set of execution flows, namely a collection of DAGs.

For each execution flow $F_{l,m}$ of every task τ_x , a synchronous DAG of servers referred to as synchronous server graph (SSG) and denoted by $F_{l,m}^{SGG}$ is derived. Then an algorithm to merge all the SSGs $F_{l,m}^{SGG}$ created for a task τ_x , into a single synchronous DAG of servers, called "global synchronous server graph" (GSSG) and denoted by F_x^{GSSG} is introduced.

If a valid GSSG F_x^{GSSG} is deemed schedulable by a schedulability test of a scheduling algorithm A, so does the task τ_x from which it was derived. Therefore, the previous results for the DAG task model can be applied, once F_x^{GSSG} is computed.

Baruah et al. [9] formally introduced the conditional sporadic DAG task model, which from here will be called C-DAG, as an extension to the sporadic DAG task model [11] that is capable of modeling certain conditional control-flow constructs.

As with the traditional sporadic tasks, each conditional sporadic DAG task τ_x is specified as a 3-tuple (G_x, D_x, T_x), where $G_x = (V_x, E_x)$ is a DAG that, apart from regular vertices, has also conditional ones. Conditional vertices are special vertices in V_x that are defined in pairs. Let (c_1, c_2) be such a pair in the DAG $G_x = (V_x, E_x)$. Vertex c_1 represents a point in the code where a conditional expression is evaluated and, depending upon the outcome of this evaluation, control will subsequently flow along exactly one of several different possible paths in the code. It is required that all these different paths meet again at a common point in the code, represented by the vertex c_2 .

Edges (v_1, v_2) between pairs of vertices, neither of which are conditional vertices, represent precedence constraints exactly as in traditional sporadic DAG tasks, while edges involving conditional vertices represent conditional execution of code. More specifically, let (c_1, c_2) denote a defined pair of conditional vertices. After the vertex c_1 completes execution, exactly one of its successors becomes eligible to execute; it

is not known beforehand which successor may execute. Vertex c_2 begins to execute upon the completion of exactly one of its predecessors.

For each pair (c_1, c_2) of conditional vertices in G_x , we refer to the subgraph of G_x beginning at c_1 and ending at c_2 as a conditional construct in G_x . It is permitted for conditional constructs to be nested: a conditional construct may contain additional conditional constructs within it.

Let $\mathcal{J}_x$ denote all possible complete collections of jobs that comprise a C-DAG τ_x . Each $J \in \mathcal{J}_x$ denotes a collection of jobs obtained by completely executing through the DAG G_x once, taking into account the conditional branches within it. For each $J \in \mathcal{J}_x$, all the jobs have a common release time and a common deadline. Note that $|\mathcal{J}_x|$ may be exponential in the number of vertices in G_x.

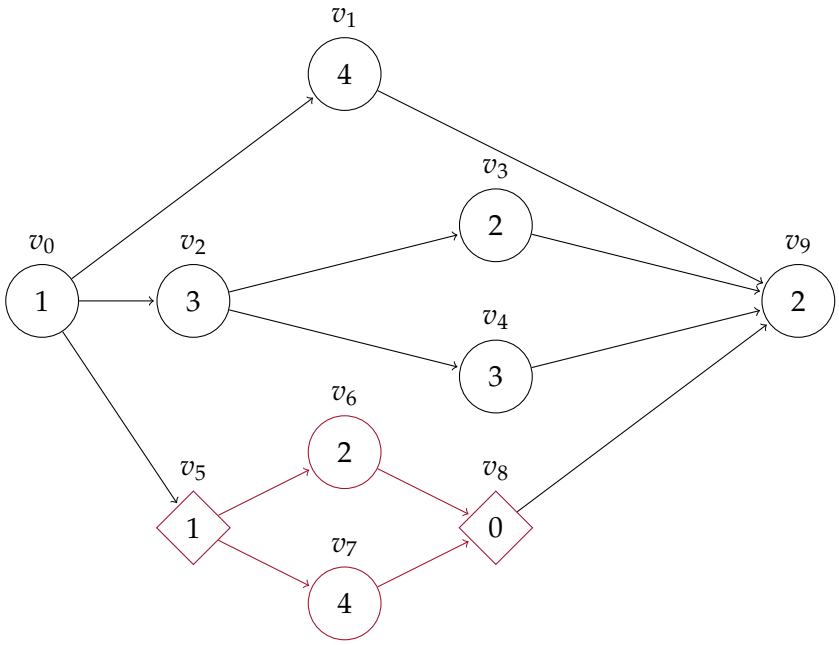


FIGURE 3.3: C-DAG τ_x, with period $T_x = 16$ and deadline $D_x = 10$. (v_5, v_8) is a pair of conditional vertices (diamonds), where v_5 is the entry point and v_8 is the exit point. The whole conditional construct (v_5, v_6, v_7, v_8) is depicted in red.

An example of C-DAG is depicted in Figure 3.3.

Besides the introduction of this new model, the authors also analyzed the schedulability of C-DAG task sets with constrained-deadline on multiprocessors.

Firstly, the authors proposed a method to compute the work for this new task model, in order to apply the results of Baruah [4] (Theorem 3.1.8).

Although its correctness, the method suffers from the same problem as the multi-DAG model of Fonseca et al. [51]: the number of distinct possible flows of control may be exponential in the size of the DAG; the overall algorithm would therefore take exponential time.

For this reason, the authors developed a novel transformation strategy that converts each conditional sporadic DAG task to a non-conditional one in polynomial time, and tests the system of transformed tasks for G-EDF schedulability using the pseudo-polynomial test provided by Bonifaci et al. [25].

Melani et al. [81] at the same time (ECRTS 2015) derived efficient ways to compute an upper-bound on the response-time of each C-DAG using different global

scheduling algorithms.

The method introduced in Section 3.1.1, was actually developed for C-DAGs. The analysis is exactly the same, except for the computation of: (i) the longest path L_k and (ii) the volume vol_k (called workload W_k for C-DAG) of the DAG. Indeed, when handling C-DAG, those two factors depend on the choices made on the conditional branches. The authors proposed algorithms to efficiently compute those values.

Experiments among randomly generated C-DAG workloads show that the proposed approach improves over previously proposed solutions. The code for the analysis of the proposed method and other several ones has been made public[4].

Pathan et al. [90] analyzed G-FP scheduling of DAG (as already introduced in Section 3.1.1) and their analysis can also be applied to C-DAGs. Their method reduces the pessimism of the analysis of Melani [81] also in the case of conditional DAG tasks.

Sun et al. [109] considered the problem of G-FP scheduling of C-DAG OpenMP tasks on multiprocessors. The innovative part of their model is that they studied the problem of how to bound their WCRT when non-well-nested branching structures are in the C-DAG.

A C-DAG contains a non-well-nested branching structure when there exists an edge between a regular vertex of one of the conditional construct and a vertex that does not belong to that conditional construct. For example, inserting $e(v_6, v_4)$ in the C-DAG tau_x of Figure 3.3 would create a non-well-nested branching structure. This possibility was not taken into account in the previous works.

Starting from the Graham bound [54] (Theorem 3.1.9), they proposed a method to compute the volume and the length of C-DAGs with non-well-nested branching structures.

[4] https://retis.sssup.it/~d.casini/resources/DAG_Generator/cptasks.zip

3.3 Heterogeneous DAG Tasks

Heterogeneous, or typed, DAGs are another extension of the DAG task model. The additional feature of this extended DAG task is a new property of the vertices. To take into account the fact that sub-tasks can execute on different types of cores or different engines, each vertex is further described by means of a tag that specify the kind of computational unit on which it will run onto.

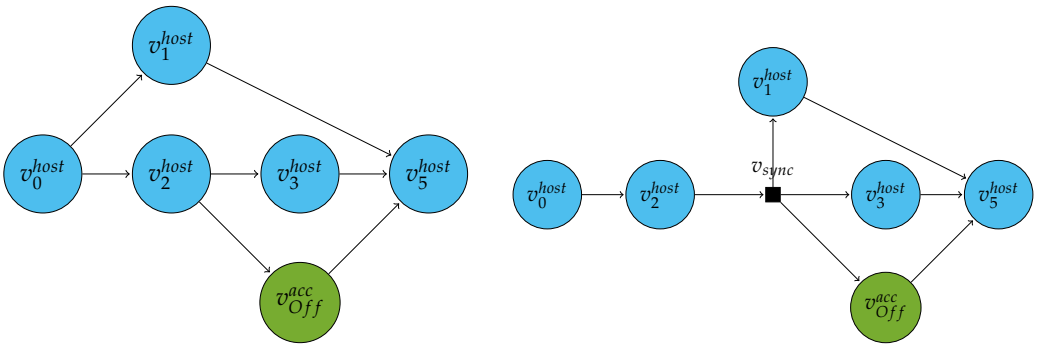
Yang et al. [129] considered for the first time an RTA for DAG-based real-time task systems implemented on heterogeneous multicore platforms.

The compute engines (CE) are clustered in pools, so that all the same kind of CE are grouped in the same pool. The authors focused on non-preemptive G-EDF scheduling algorithm within each CE pool. They introduced the offset-based independent task (obi-task) model and an algorithm to convert DAG tasks to obi-tasks.

Serrano and Quiñones [105] proposed a novel RTA for verifying the schedulability of a single DAG task supporting heterogeneous computing. The authors considered a parallel heterogeneous architecture composed of a host processor with m identical cores and a single accelerator device (e.g. a FPGA, GPU, etc.). Moreover, they considered a host-centric acceleration model in which the host offloads code and data to the accelerator device and collects results.

The task model is the same ad the DAG task model, but in the set of vertices V, besides traditional vertices that execute on the host, there is also a single node v_{Off} that represents the workload executed in the accelerator device, named offloaded node, with its WCET C_{Off}.

The proposed algorithm (i) identifies the sub-DAG that may potentially execute in parallel with v_{Off} , named $G^{Par} = (V^{Par}, E^{Par})$, and (ii) adds a synchronization point to guarantee that G^{Par} and v_{Off} actually execute in parallel. However, this strategy may impact the average performance of the tasks because: (i) the critical path can potentially enlarge, and (ii) the potential parallelism is reduced due to the synchronization point. An example of the model is depicted in Figure 3.4.



(A) Heterogeneous DAG

(B) Transformed heterogeneous DAG after the insertion of the synchronization point.

FIGURE 3.4: Example of the heterogeneous-DAG task proposed; before (Figure 3.4a) and after (Figure 3.4b) the insertion of a synchronization point (black square). Regular nodes that execute on the host are filled in light blue; the single node that executes on the accelerator is filled in green.

For the schedulaiblity test, they first computed the WCRT on the homogeneous cores $R^{hom}(\tau)$ using the Graham bound eq. (3.7) (Theorem 3.1.9). Then they derived a new RTA supporting heterogeneous computing:

Theorem 3.3.1: Serrano and Quiñones [105]

Consider an heterogeneous DAG task τ'. Depending on the execution scenario, its response time upper bound is computed as follows:

1. v_{Off} does not belong to the longest path.

$$R^{het}(\tau') = len(G') + \frac{1}{m}(vol(G') - len(G') - C_{Off}) \tag{3.33}$$

2. v_{Off} belongs to the longest path and $C_{Off} \geq R^{hom}(G^{Par})$

$$R^{het}(\tau') = len(G') + \frac{1}{m}\left(vol(G') - len(G') - vol\left(G^{Par}\right)\right) \tag{3.34}$$

3. v_{Off} belongs to the critical path and $C_{Off} \leq R^{hom}(G^{Par})$

$$\begin{aligned} R^{het}(\tau') =& len(G') - C_{Off} + len\left(G^{Par}\right) \\ &+ \frac{1}{m}\left(vol(G') - len(G') - len\left(G^{Par}\right)\right) \end{aligned} \tag{3.35}$$

Han et al. [60] studied the WCRT analysis of typed scheduling of a single parallel DAG task on heterogeneous multi-cores, where the workload of each vertex in the DAG is only allowed to execute on a particular type of cores.

The authors proposed two new WCRT bounds, considering and formally describing the typed DAG task model. A typed DAG is a DAG task in which each vertex has a type, that is the type of core on which the vertex should be executed. S is the set of core types, and for each $s \in S$ there are M_s cores of this type $(M_s \geq 1)$. The type function $\gamma : V \times S$ defines the type of each vertex, i.e., $\gamma(v) = s$, where $s \in S$, represents vertex v must be executed on cores of type s.

Then, they introduced the concept of scaled graph: the scaled graph $\hat{G}$ of G has the same topology (V and E) and type function γ as G, but vertices have different WCET

$$\forall v_i \in V : \hat{C}_i = C_i \times \left(1 - \frac{1}{M_{\gamma(v_i)}}\right) \tag{3.36}$$

Therefore they derived a schedulability test with overall time complexity of $O(|V| + |E|)$.

Theorem 3.3.2: Han et al. [60]

The worst-case response time of single parallel typed DAG task $\tau = (G, T, D)$ with implicit deadlines on heterogeneous multi-cores is bounded by

$$R(G) \leq L(\hat{G}) + \sum_{s \in S} \frac{vol_s(G)}{M_s} \tag{3.37}$$

where $\hat{G}$ is the scaled graph of G and $vol_s(G)$ is the volume of the DAG G computed on the core s.

The second bound the authors introduced, explores task graph structure information to improve the precision, but is computationally more expensive. The authors proved that the problem of computing the second bound is strongly NP-hard if the number of types in the system is a variable, and develop an efficient algorithm that has polynomial time complexity if the number of types is a constant.

Chang et al. [38] proposed a novel scheduling algorithm for the single typed DAG task on heterogeneous multi-cores, based on a criticality allocation strategy.

The criticality allocation strategy assigns each vertex v_j a varying criticality $Crit_j$ that depends on the remaining workload of the vertex. The vertex with higher criticality is more urgent to execute. Eventually, the order in which vertices are scheduled is determined by their criticality. Through this scheduling algorithm, the set of vertices that will block the vertices on each path can be more accurately determined. On this basis, a new scheduling algorithm is proposed to reduce the range of possible parallel execution vertices with the same type and different paths blocking each other. Secondly, based on this scheduling algorithm, a new RTA method is established to eliminate unnecessary blocking time.

Theorem 3.3.3: Chang et al. [38]

The worst-case response time of single parallel typed DAG task $\tau = (G, T, D)$ with constrained deadlines on heterogeneous multi-cores is bounded by

$$R(G) \leq L(G) + \sum_{k=1}^{n} \max_{s \in \Delta(Crit_k)} \inf(Crit_k, s) \tag{3.38}$$

where s is the core type, $n = L(G) - C_{src} - C_{sink}$, $\Delta(Crit_k)$ is the set of types of nodes in criticality set $Crit_k$ and $\inf(Crit_k, s)$ represents the time that nodes of type s block each other in $Crit_k$.

Experimental results show that the response time upper bound is significantly better than the bound proposed by Serrano et al [105] and Han et al [60] that are limited and pessimistic.

3.4 Heterogeneous Conditional DAG Task

Finally, the most expressive extension of the DAG task model is the one that combines the two previous ones: the Heterogeneous Conditional DAG (HC-DAG) task model. Even though this is the model that best describes the complexity of modern real applications, the literature in this context is very scarce and requires more attention.

Zahaf et al. [132, 133] proposed a novel real-time application model, called Heterogeneous Parallel Condition Directed Acyclic Graph Model (HPC-DAG), specifically conceived for heterogeneous platforms.

An HPC-DAG allows the system designer to specify alternative implementations of a software component for different processing engines, as well as conditional branches to model if-then-else statements.

The authors modeled a heterogeneous architecture as a set of execution engines, characterized by (i) its execution capabilities, and (ii) its scheduling policy (e.g. FTP or EDF), which can be preemptive or non-preemptive. Each engine has its own scheduler and a separate ready-queue. Given the importance of communication of real-time tasks on heterogeneous architectures, copy engines are treated as processing units, in which schedule communication tasks are scheduled.

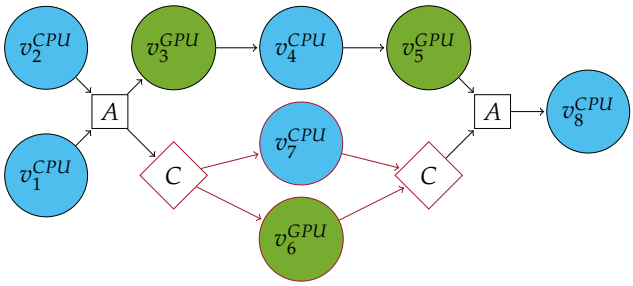
Even though the model is generic, the focus is this work is on sporadic task model with constrained deadlines under P-FP-EDF scheduling.

In the proposed model two kind of tasks need to be clarified: (i) the specification task τ_x and (ii) the concrete task $\bar{\tau}_x$. The specification task τ_x is an extension of the C-DAG (introduced in Section 3.2). In addition, each vertex v_i of the task τ_x has a tag that represents the engines where it is eligible to execute onto. Besides regular and conditional nodes, in the specification task there exist also alternative nodes, which represent alternative implementations of parts of the graph/task. A concrete task $\bar{\tau}_x$ is an instance of a specification task where all alternatives have been removed by making implementation choices. Informally, we could say that the specification task represents the HPC-DAG model, while the concrete task is a C-DAG with tags for specific engines. An example is given in Figure 3.5.

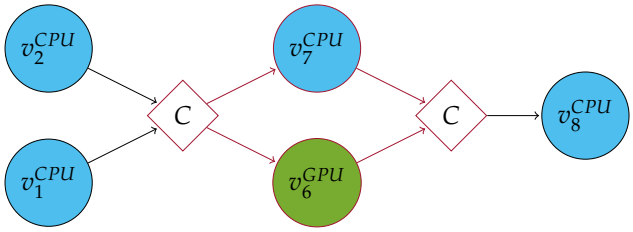
Given a specification task, one of the possible concrete tasks needs to be selected before proceeding to the allocation and scheduling of the sub-tasks on the computing engines. Since the number of combinations can be very large, the authors proposed a heuristic algorithm based on a greedy strategy. Moreover, to reduce the complexity of dealing with precedence constraints directly, they imposed intermediate offsets and deadlines on each sub-task.

The proposed algorithm tries to allocate one single task specification at a time: it generates all concrete tasks and, for each one of them, it assigns the intermediate deadlines and offsets accounting for the slack distributions. The algorithm gives priority to single-engine allocations to reduce preemption costs. Moreover, it can be customized with four parameters:

- Sorting order of the concrete task sets: (i) sorting according to concrete tasks total execution time (ii) sorting according to less allocation on scarce resources.
- Distribution of the slack when assigning intermediate deadlines and offsets: (i) fair distribution, that assigns slack as the ratio of the original slack by the number of sub-tasks in the path, (ii) proportional distribution: assigns slack according to the contribution of the sub-task WCET in the path.



(A) Specification task τ_x



(B) Concrete task $\bar{\tau}_x$

FIGURE 3.5: Example of HPC-DAG task. The alternative nodes in the specification task in Figure 3.5a are depicted with a rectangle, while tags (in this case CPU and GPU) are represented with filling colors (resp. light blue and green). The task shown in Figure 3.5b is one of the possible concrete tasks for the given specification task.

- Allocation strategy: (i) best-fit, (ii) worst-fit.
- Elimination strategy of sub-tasks (when needed): (i) random selection, (ii) parallel selection, selecting a subtask that is parallel to the critical path.

The authors showed that the HPC-DAG dominates the corresponding results for the C-DAG [81].

3.5 Methods evaluation

Besides the survey part, this chapter offers also a contribution to the community with the implementation of some of the introduced works. Several methods have been implemented in C++ and the code has been made publicly available at https://github.com/mive93/DAG-scheduling, so that other researchers can use it as a library, improve it, extend it or simply reproduce the experiments.

Task set generation In the literature different ways to randomly generate DAG task set for multiprocessors have been adopted:

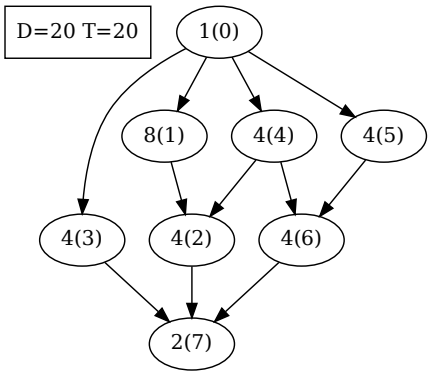
- Erdös-Rényi. method [45], which generates graphs.
- Uunifast-Discard [42] , that efficiently generates task utilization values for task sets with a chosen number of tasks and total utilization.
- method proposed by Melani et al. [81] to generate DAG and C-DAG.
- YARTISS, an open-source simulation tool written in Java [37].

For our implementation, we selected the method proposed by Melani et al. [81], because it allows one to create not only DAG but also conditional DAGs, and it has been extended to generate Heterogeneous DAG also.

To improve usability with user-defined DAGs, the implemented library can also read and write DAGs in DOT (graph description language). An example is shown in Figure 3.6.

```
digraph Task {
i [shape=box, label="D=20 T=20"];
0 [label="1(0)"];
1 [label="8(1)"];
2 [label="4(2)"];
3 [label="4(3)"];
4 [label="4(4)"];
5 [label="4(5)"];
6 [label="4(6)"];
7 [label="2(7)"];
0 -> 1;
0 -> 3;
0 -> 4;
0 -> 5;
1 -> 2;
2 -> 7;
3 -> 7;
4 -> 6;
4 -> 2;
5 -> 6;
6 -> 7;
}
```

(A) DAG written in DOT format.



(B) Graphic conversion of the DOT.

FIGURE 3.6: Example of DAG in DOT format.

Selected Methods The selected methods are:

- G-FP: Graham1969 [54], Baruah2012 [11], Bonifaci2013 [25], Li2013 [70], Qamhieh2013 [96], Melani2015 [81], Pathan2017 [90], Fonseca2017 [48], He2019 [61], Fonseca2019 [49], Han2019 [60].
- G-LP: Serrano2016 [107], Nasri2019 [86].
- P-FP: Fonseca2016 [50].

- P-LP: Casini2018 [34].

All the methods have been implemented from zero except from Melani2015 [81] and Nasri2019 [86] whose code was available and has been integrated into the developed library.

3.5.1 Comparison

In the following, three different kinds of test have been reported: (i) schedulability test varying the total utilization of a task set with multiple tasks, (ii) schedulability test varying the number of processors assigned to a task set with multiple tasks, and (iii) schedulability test varying the total utilization of a single task. For the generation of the tasks, the parameters have been set in the same way as the original implementation by Melani [81]. Moreover, the computational time of every single test has been measured in microseconds.

All the experiments have been performed on an Intel i7-7700HQ CPU @ 2.80GHz.

Varying U for a task set (task) In this test, the utilization of the task set (task) is varied from $U = 0$ to $U = 8$, with an increasing step of 0.25. For each step, 500 different task set (task) are generated and tested, for a total of 16k tests. The number of cores is always fixed to $m = 8$.

Varying m for a task set (task) In this test, the number of cores assigned to the task set (task) is varied from $m = 2$ to $m = 30$, with an increasing step of 1. For each step, 500 different task set (task) are generated and tested, for a total of 14k tests. The total utilization is always fixed to $U = 2$.

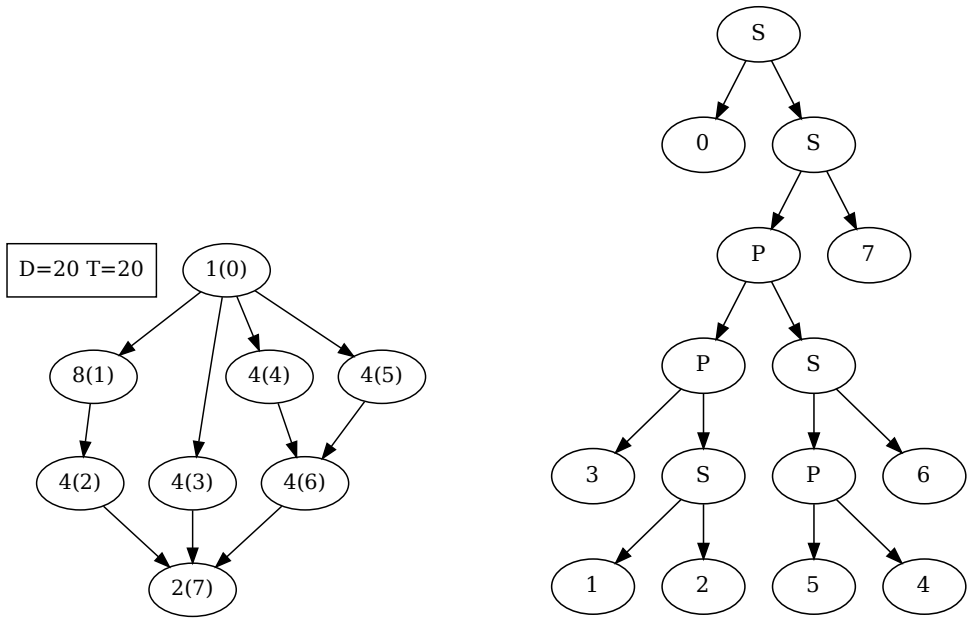
Allocation strategy in partitioned scheduling For the partitioned scenario, cores are assigned to nodes with the Worst-Fit strategy, ordering all the vertices by decreasing utilization first and decreasing density second. The First-Fit and Best-Fit strategies have also been considered, but the Worst-Fit was the one leading to be best results, reason why it has been selected.

G-FP for DAG, FTP and implicit deadlines Figure 3.8 reports the results for the comparison among the G-FP-FTP scheduling for DAGs with implicit deadlines. From these charts, we can notice that the best methods in terms of schedulability are Fonseca2017 and Fonseca2019, which dominate all the other approaches when considering multiple tasks. The algorithms are very similar, and we can notice that the improvements introduced in the version of 2019 are almost negligible. The execution time of these algorithms is also comparable and it takes up to $\sim 4ms$. These methods are also the most complex one among all the considered. To compute the carry-out workload, they require two conversions of the DAG: first, the DAG needs to be converted into an NFJ DAG and then this needs to be decomposed in an SPTree.

Figure 3.7 reports the DAG of Figure 3.6b converted, first in a NFJ DAG (Figure 3.7a) and then decomposed in a SPTree (Figure 3.7b).

When considering a single DAG, the performance of Fonseca2019, Fonseca2017, and Melani2015 are the same, because the methods differ only in the computation of high-priorities tasks interference, which is obviously missing in the single task experiments.

He2019 has similar results with respect to Melani2015, even though it dominates it in every test, both for multiple and single tasks. This method is the one with more



(A) Conversion from the DAG of Figure 3.6b to a NFJ DAG.

(B) Conversion from the NFJ DAG of Figure 3.7a to a SPTree.

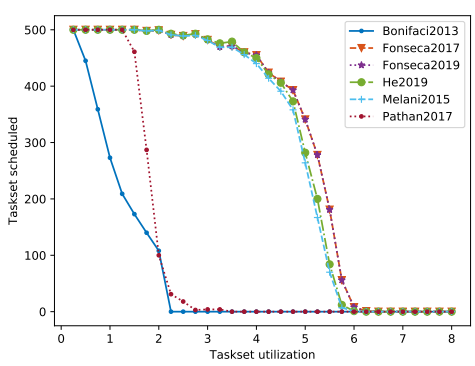
FIGURE 3.7: Decomposition needed for Fonseca2017 and Fonseca2019.

variability in the execution times, reaching even more than 20 seconds for some runs (as can be seen in Figure 3.8d).

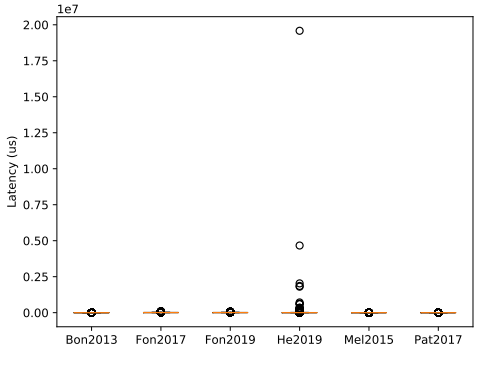
Pathan2017 was supposed to always dominate Melani2015. For the single task test, this is true, and the method slightly dominates He2019. However, when considering multiple tasks, Pathan2017 performance is poor and is greatly dominated by both He2019 and Melani2015. The problem can be found in Equation (3.15): in this test the authors account for the whole workload of a task τ_y for every subtask of τ_x which is very pessimistic. Melani also accounted for the whole workload of a task τ_y in Equation (3.9), but only once for task.

G-FP for DAG, EDF and implicit deadlines Figure 3.9 reports the results for the comparison among the G-FP-EDF scheduling for DAGs with implicit deadlines. It is here reported the implicit case and not the constrained because the former is more comprehensive, including also Li2013 which is thought for implicit DAGs only. From these charts, we can notice that Melani2015 dominates all the other methods, both for multiple and single DAGs. This method is also the one with the highest computational time, reaching in some cases $\sim 14ms$.

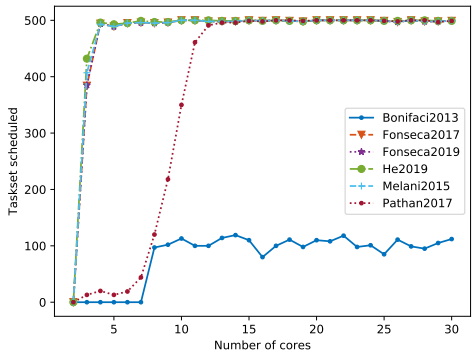
G-FP for DAG, FTP and arbitrary deadlines Figure 3.10 reports the results for the comparison among the G-FP-FTP scheduling for DAGs with arbitrary deadlines. Fonseca2019 dominates both Bonifaci2013 and Graham1969 when considering multiple tasks; while it has the same performance as Graham1969 for a single task. The method in some cases reaches a computational time of $\sim 40ms$.



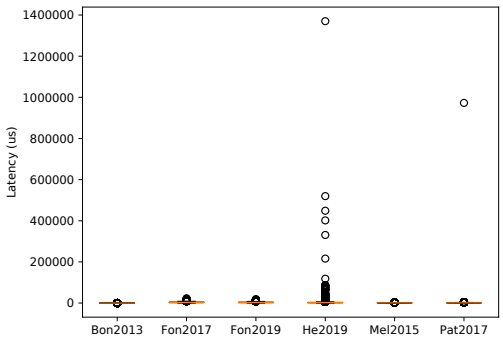
(A) Task set, varying utilization.



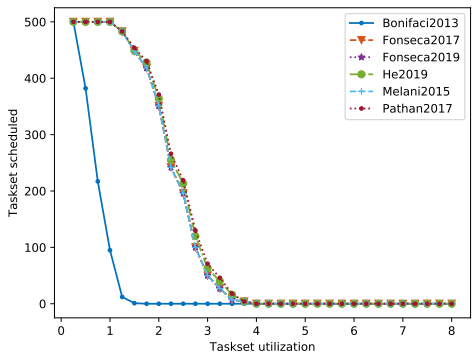
(B) Execution times.



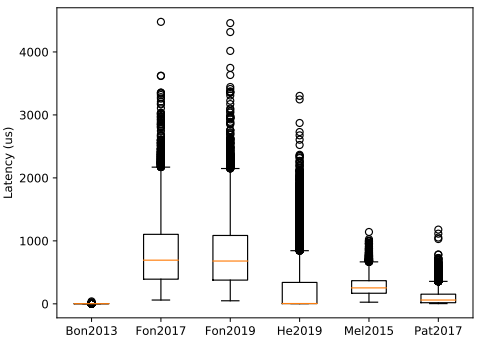
(C) Task set, varying number of processors.



(D) Execution times.



(E) Single task, varying utilization.

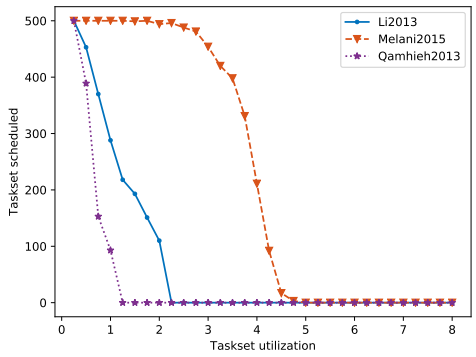


(F) Execution times.

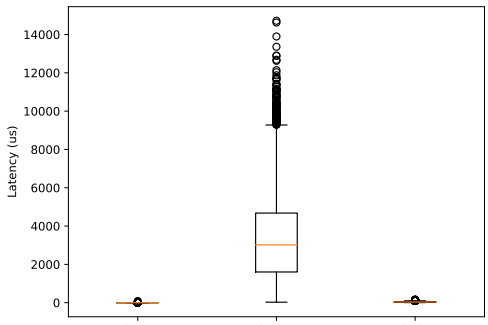
FIGURE 3.8: Methods comparison for DAGs with FTP scheduling and implicit deadlines under G-FP scheduling.

G-FP for C-DAG, FTP and constrained deadlines Figure 3.11 reports the results for the comparison among the G-FP-FTP scheduling for conditional DAGs with constrained deadlines. Similar results with respect to the DAG case can be seen in the plots. Again, Pathan2017 dominates Melani2015 only when a single task is considered, but it is definitely dominated in the multiple tasks tests.

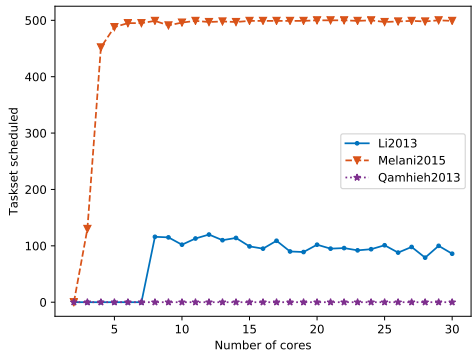
G-LP for DAG, FTP and constrained deadlines In the context of global scheduling, two methods have been developed for limited preemption, namely Serrano2016 and Narsi2019. The result of their comparison can be found in Figure 3.12.



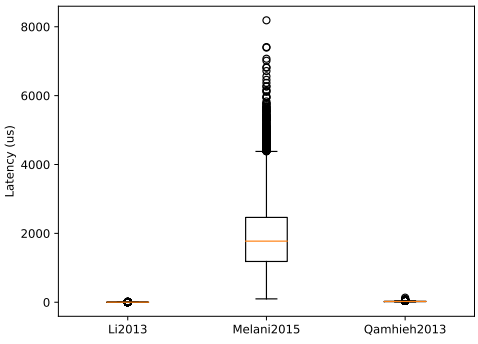
(A) Task set, varying utilization.



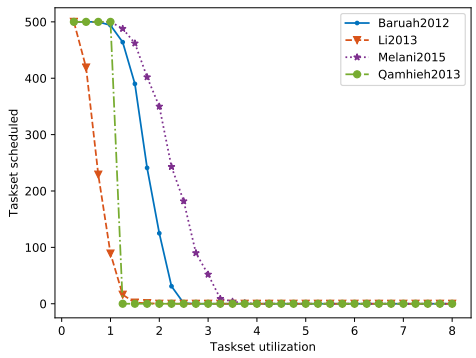
(B) Execution times.



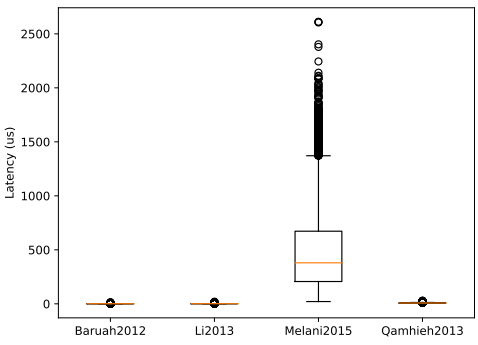
(C) Task set, varying number of processors.



(D) Execution times.



(E) Single task, varying utilization.

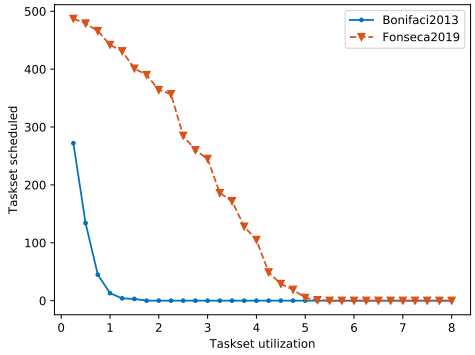


(F) Execution times.

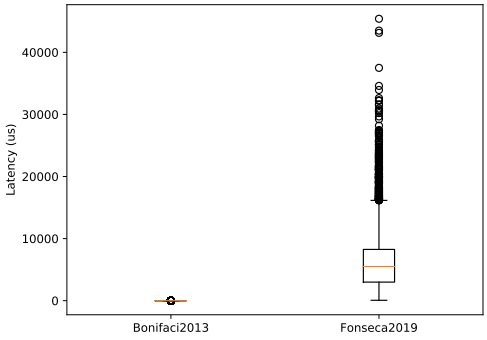
FIGURE 3.9: Methods comparison for DAGs with EDF scheduling and implicit deadlines under G-FP scheduling.

In these charts it is clear that Nasri2019 dominates Serrano2016, having significantly better results in each performed test. However, given the greatest complexity of the Nasri2019 method, its computational time is also the highest, reaching 500*ms* for a single test.

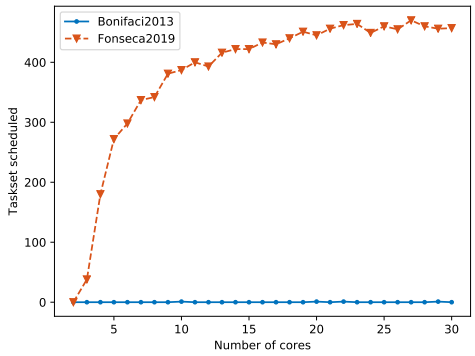
P for DAG, FTP and implicit deadlines Regarding partitioned scheduling, few works can be found in the literature. Indeed, the most relevant ones are Fonseca2016, for fully preemptive policy, and Casini2018 for the limited preemptive policy. Figure 3.13 shows the comparison of these methods, where it can be noticed that Casini2018



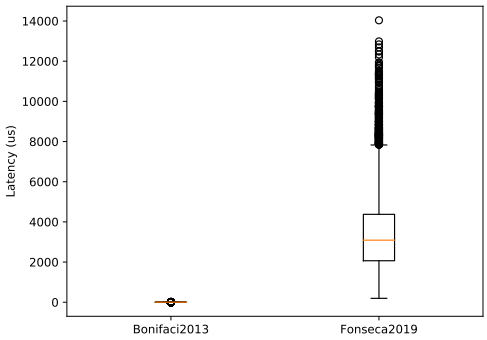
(A) Task set, varying utilization.



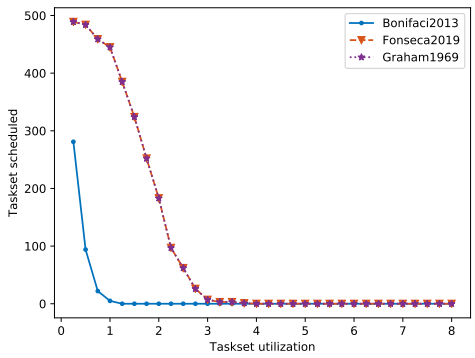
(B) Execution times.



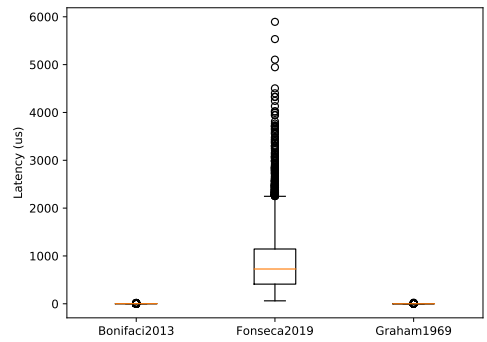
(C) Task set, varying number of processors.



(D) Execution times.



(E) Single task, varying utilization.

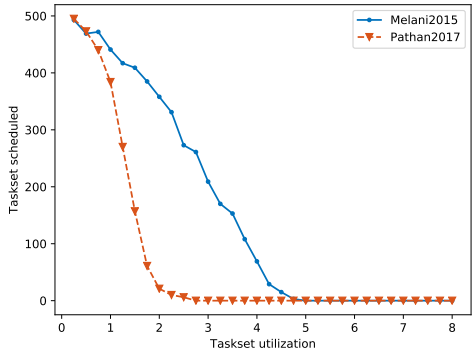


(F) Execution times.

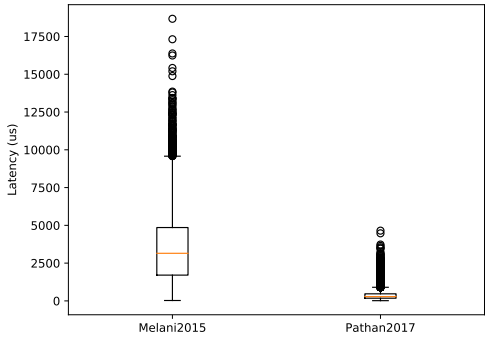
FIGURE 3.10: Methods comparison for DAGs with FTP scheduling and arbitrary deadlines under G-FP scheduling.

dominates the other method in each test. However, Casini2018 is also the method with the highest computational time, reaching even $\sim 800ms$ for a single run.

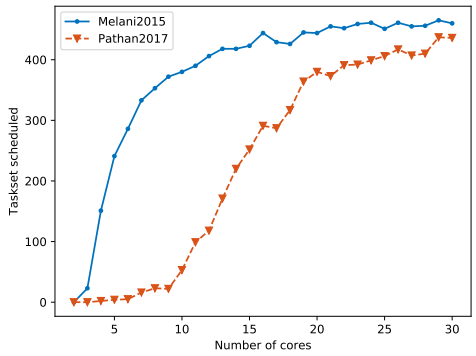
State-Of-The-Art methods for DAG, FTP and constrained deadlines Finally, the best methods in terms of schedulability have been considered all together. Figure 3.14 reports the comparison among Casini2018 (P-LP), Fonseca2016 (P-FP), Fonseca2019 (G-FP), Melani2015 (G-FP), Nasri2019 (G-LP). Nasri2019 dominates each other method for every performed test, while the poorest performance is given by Fonseca2016, which is dominated by all the others. For these tests it can be also



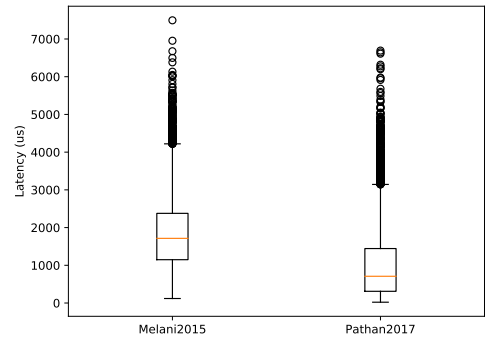
(A) Task set, varying utilization.



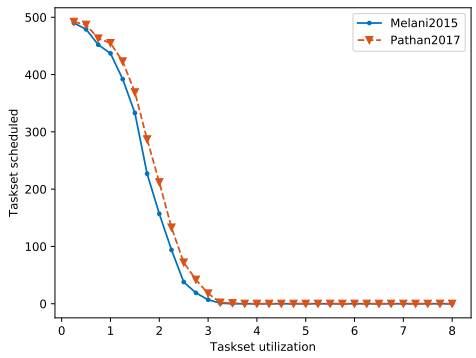
(B) Execution times.



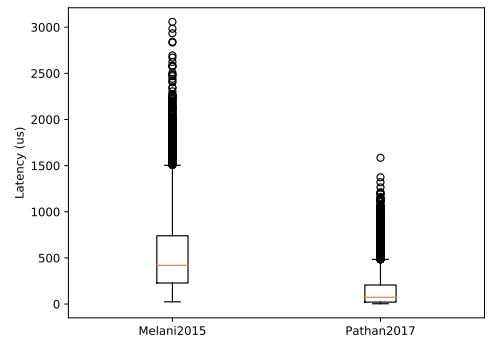
(C) Task set, varying number of processors.



(D) Execution times.



(E) Single task, varying utilization.



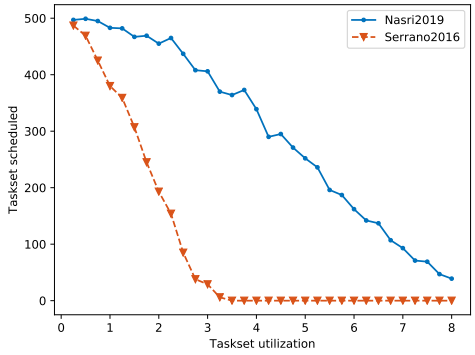
(F) Execution times.

FIGURE 3.11: Methods comparison for Conditional DAGs with FTP scheduling and constrained deadlines under G-FP scheduling.

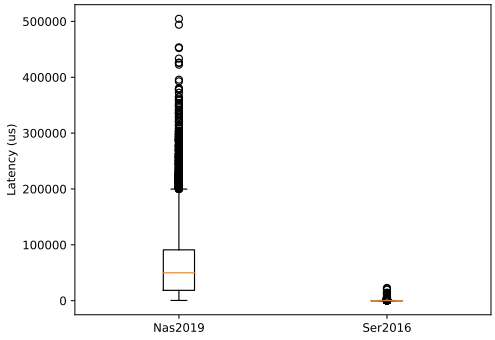
noticed that Nasri2019 and Casini2018 are the ones with the highest computational time, reaching more than 500 and 200 ms respectively. Again, for single task test, Fonseca2019 and Melani2015 have the same exact resutls.

3.6 Conclusions

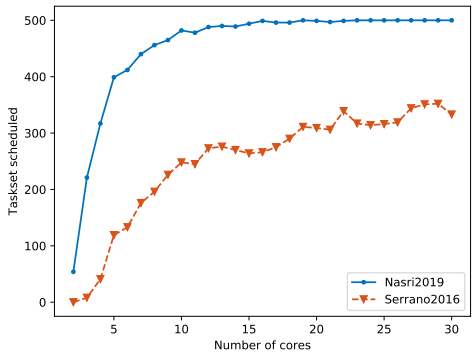
Global scheduling When considering global scheduling for a task set of DAG tasks the three SOTA methods are Nasri2019, Fonseca2019, and Melani2015. Nasri2019 outperforms every method analyzed and it is for sure the best method for G-LP.



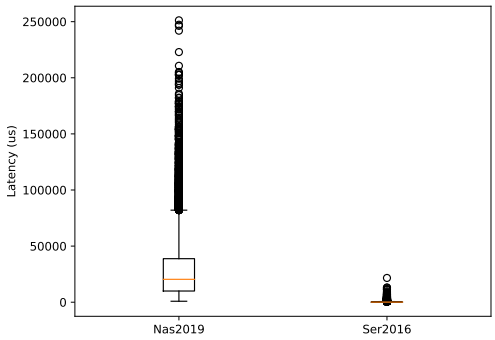
(A) Task set, varying utilization.



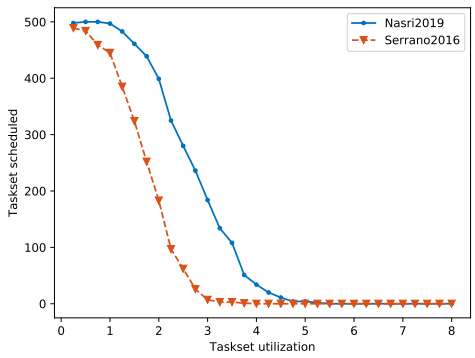
(B) Execution times.



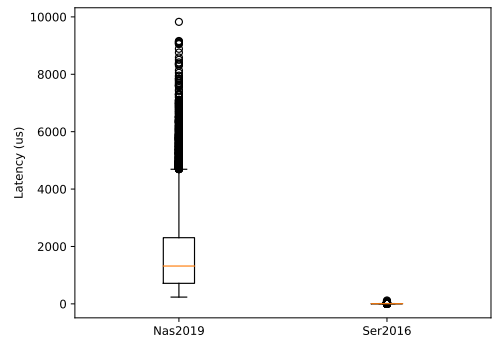
(C) Task set, varying number of processors.



(D) Execution times.



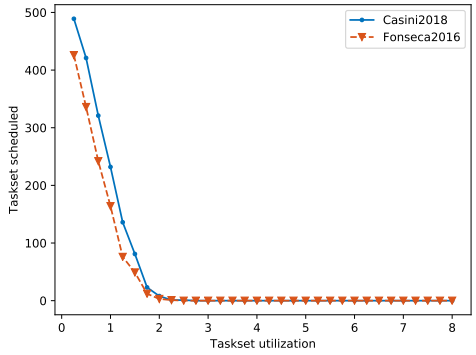
(E) Single task, varying utilization.



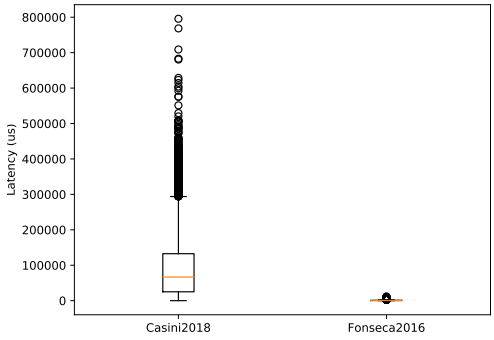
(F) Execution times.

FIGURE 3.12: Methods comparison for DAGs with FTP scheduling and constrained deadlines under G-LP.

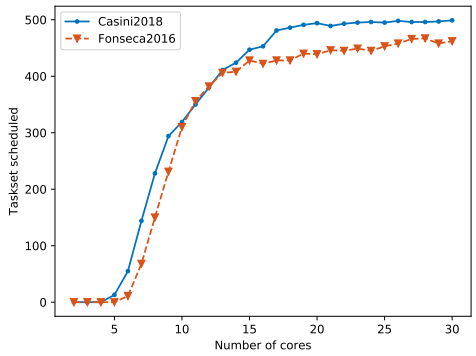
However, it is also the method with the highest computational time, due to the expensive exploration in the possibilities space. Indeed, it can not be used for online purposes, to handle a dynamic task set, but only for offline static ones. Moreover, the actual performance could be even worse: for the performed tests the number of nodes in the random generated DAGs was limited, varying from 4 to 37 nodes; in the original paper [86] the authors state that execution times for tests become intractable for DAGs with more than 64 nodes, making the method not suitable for those cases. Fonseca2019 is the method to adopt when using the DAG task model with G-FP-FTP scheduling, being the one that dominates all the other G-FP solutions



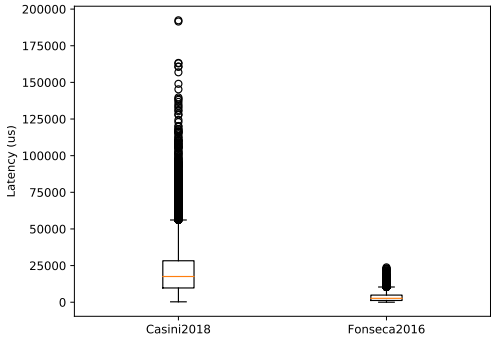
(A) Task set, varying utilization.



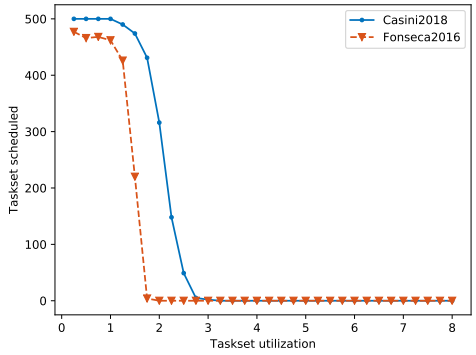
(B) Execution times.



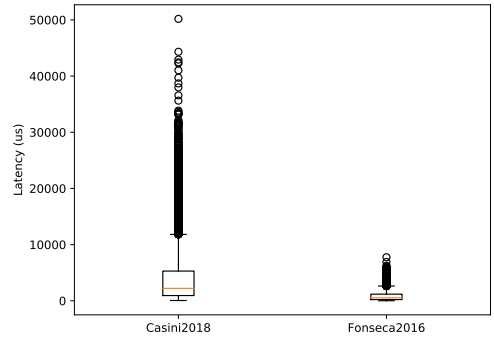
(C) Task set, varying number of processors.



(D) Execution times.



(E) Single task, varying utilization.

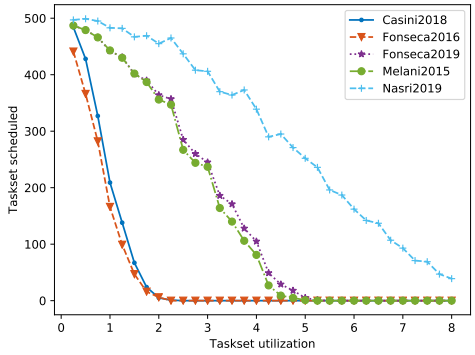


(F) Execution times.

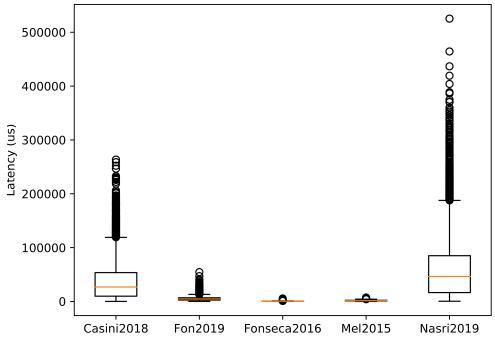
FIGURE 3.13: Methods comparison for DAGs with FTP scheduling and implicit deadlines under partitioned scheduling.

both for constrained and arbitrary deadlines, especially when considering multiple tasks. Melani2015 is the best solution for multiple and single tasks for the DAG task model when using EDF. When considering G-FP for C-DAGs, it is still the best solution for EDF (single and multiple tasks) and for multiple tasks when using FTP. However, When using G-FP-FTP for a single task, Melani is dominated by Pathan2017.

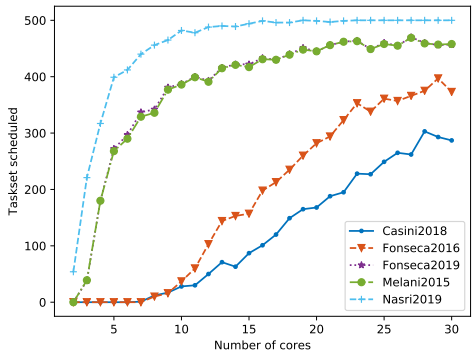
Partitioned scheduling When considering partitioned scheduling the two SOTA methods are Fonseca2016 (P-FP) and Casini2018 (P-LP). Casini2018 is the best method



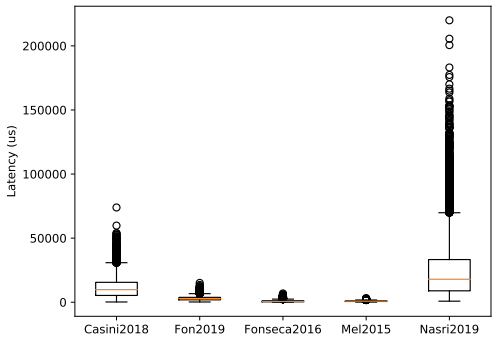
(A) Task set, varying utilization.



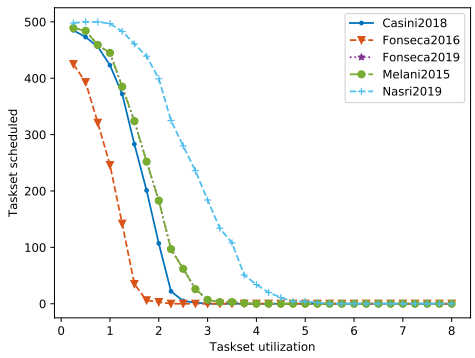
(B) Execution times.



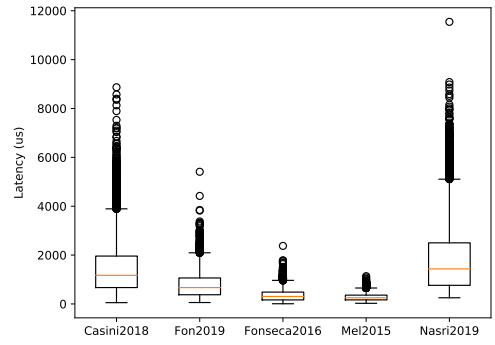
(C) Task set, varying number of processors.



(D) Execution times.



(E) Single task, varying utilization.



(F) Execution times.

FIGURE 3.14: SOTA methods comparison for DAGs with FTP scheduling and constrained deadlines.

among the partitioned, but it has a higher computational time. Nonetheless, we have still to remember that both those methods become too expensive in terms of execution time when the DAG has several nodes: indeed the analyses are based on the complete exploration of all the possible paths in a graph, which is known to be of exponential complexity.

Global vs Partitioned In conclusion, what is better when scheduling DAGs, partitioned or global scheduling? The results of these tests show that Nasri2019 is the

best method available, and it is a global scheduling algorithm. However, a clear answer is missing. Schedulability tests are not sufficient to answer this delicate and important question: one should also compare actual response times, end-to-end latencies and account for migration overheads. Moreover, additional tests varying all the DAG generator parameters should be performed to have a clearer overview.

However, we can notice that the method with the best trade-off among computational time, schedulability, and easy-adoption is Melani2015. It has medium schedulability ratio performance, however, it has several advantages: (i) it has a very small computational time ($\sim 4ms$ in the worst case among all the performed tests) which make it possible to use it as an online method; (ii) it can be applied both for EDF, FTP and any work-conserving scheduling algorithms; (iii) it can be easily extended for G-LP (i.e. Serrano2016); (iv) it is also valid for C-DAGs; (v) it can handle DAGs with hundreds of nodes in feasible time (e.g. the worst-case computational time for task sets having among 5 to 30 tasks with 50 to 300 nodes is $\sim 700ms$); (vi) it is easy to implement.

Research directions Finally, this work led to a list of open questions for the real-time community that are hereafter summarized.

As already mentioned, understanding whether partitioned or global scheduling is more suitable for DAGs would greatly help to improve real-time systems.

To get there, there are sub-problems to tackle first: (i) it would be interesting to see how a method like Nasri2019 would perform in a partitioned scenario; (ii) it is crucial to find a good allocation strategy for partitioned methods, better than Worst-Fit, given that the performance of partitioned approaches highly depends on that initial step; (iii) a deeper investigation on the scheduling algorithms should be carried on. Regarding the last point, many works under analysis focused on FTP, however, it is not clear whether it is better or worse than EFP, not mentioning other strategies like hierarchical scheduling.

All these open problems are left to future investigations.

Chapter 4

Latency Aware DAGs

This chapter does not focus anymore on the schedulability of the DAG task model, but rather on the computation of end-to-end latencies such as *data age* and *reaction time* on chains of DAGs. When dealing with industrial or robotics application, once the task set schedulability has been checked, the interest is moved to the actual end-to-end latency, usually, from sensors to actuators.

A method to convert a general multi-rate task set into a DAG in which scheduling and end-to-end latency constraints are met is introduced, as well as methods to compute *data age* and *reaction time* on DAGs. The results hereafter reported were originally presented in *"Latency-Aware Generation of Single-Rate DAGs from Multi-Rate Task Sets"* [122] (RTAS 2020).

4.1 End-to-End Latency and DAGs

Modern automotive and avionics real-time embedded systems are composed of applications including sensors, control algorithms and actuators to regulate the state of a system in its environment within given timing constraints. Task chains are commonly adopted to model a sequence of steps performed along the control path. Complex data dependencies may exist between task chains with different activation rates, making it very hard to find reliable upper bounds on the end-to-end latency of critical effect chains [58].

This problem is exacerbated by the adoption of even more complex task models based on DAG to capture the parallel activation of multiple jobs executing on heterogeneous multi-core platforms. A recent example in the automotive domain is given in the WATERS industrial challenge [59], focusing on the minimization of the end-to-end latency of critical effect chains of an autonomous driving system involving several sensors. The application is modeled in Figure 4.1, with three sensors providing input to multiple task chains. Nodes represent tasks with different activation periods, while edges represent the exchange of data between tasks, forming effect chains. Reaction to input stimuli and freshness of data are key factors to consider when deploying the application on a selected computing platform. *Data age* quantifies for how long an input data affects an output of a task chain, i.e., it is the maximum delay between a valid sensor input until the *last* output related to that input in the chain. *Data age* constraints are commonly found in control systems, where the age of the data can directly influence the quality of the control. In the considered application, key effect chains to optimize for *data age* are connected to the processing of camera frames and LiDAR point clouds: the older the input, the less precise is the localization of the ego vehicle and the detection of obstacles.

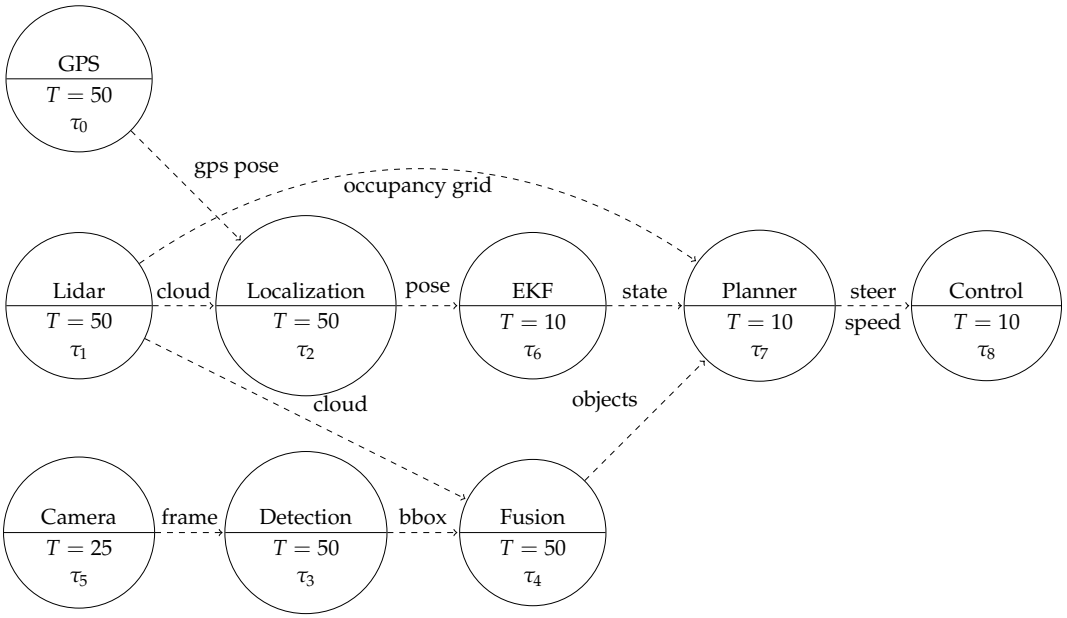


FIGURE 4.1: An example of high-utilization automotive application with tasks at different periods.

Another key metric to optimize is the *reaction latency*, a parameter that measures the reactivity of the system to a change in the input. It is defined as the maximum delay between a valid sensor input until the *first* output of the event chain that reflects such an input. It measures how much time it takes for a new event to propagate through a chain. In the considered autonomous driving example, key *reaction times* to optimize are the detection of an obstacle in the driving path, and the related actuation on the steering and breaking system to safely avoid it in due time.

This chapter aims to consider such systems composed of DAG tasks having multiple rates with given constraints on age and reaction latencies. Starting from a high-level representation, a method is presented to create a single-rate DAG that fulfills the given restrictions, optimizing schedulability and end-to-end delays. To do so, a set of DAG candidates is generated and evaluated by a constrained cost function designed to pick the best DAG meeting the given requirements.

4.1.1 Previous Works

End-to-end latency A task chain is a sequence of communicating tasks in which every task receives data from its predecessor. In literature, two types of task chains can be found: periodic chains and event-driven chains [125]. In the former, each task is activated independently at a given rate, and it communicates with its successor by means of shared variables; in the latter, task executions are triggered by an event issued from a preceding task. The propagation delays of a task chain affect the responsiveness, performance and stability of an application.

We hereafter focus on the periodic model, which is the most common in the automotive domain [58]. Di Natale et al. [87] proposed a method to evaluate the worst-case latency of mixed chains of real-time tasks and Controller Area Network (CAN) messages. Zeng et al. [135, 134] computed the probability distribution, via statistical analysis, of end-to-end latencies for CAN message chains.

Feiertag et al. [47] were the first to define *data age* and *reaction time* and to propose a framework to calculate end-to-end latencies in automotive systems, where

each task operates according to the read-execute-write semantic, also known as the implicit communication model of AUTOSAR [2].

Becker et al. [17, 15] presented a method to compute worst- and best-case *data age* for periodic tasks with implicit deadlines using implicit, explicit and Logical Execution Time (LET) communication models. The analysis is based on Read Interval (RI) and Data Interval (DI), which respectively are the interval in which a task can possibly read its input data in order to complete its execution before the deadline, and the interval for which the output data of a task can be available to the successor task in the chain. Multiple Data Propagation Trees are constructed in order to compute the *data age*. A method is also described to constrain the maximum latency by inserting job-level dependencies. A tool, called MECHAniSer [16], is presented to compute latency values for a given task set.

Regarding the LET model, Biondi et al. [22] and Martinez et al. [79, 80] addressed the problem of computing end-to-end latency bounds on multi-cores, improving the results of Becker et al. [15]. Our paper does not focus on the LET model, but it aims at deriving better latency bounds for the implicit model.

There exist other works that aim at selecting the best periods or deadlines to minimize data age in simpler task models. Xiong et al. [126], do this on a single core platform, without considering task chains. Golomb et al. [53] propose a method to find the best period to bound data freshness of task chains, assuming the task set given in input be already schedulable. Adapting these solutions to our setting is not trivial, because we assume periods and deadline to be given.

Multi-rate DAG Saito et al. [103] present a framework developed for the Robot Operating System (ROS) to handle automotive applications with multi-rate tasks. The model assumes an event-driven data-flow system in which a node starts when the predecessor nodes are completed. In order to handle multi-rate tasks, a synchronization system is adopted consisting of two kinds of additional nodes: synch driver nodes and synch nodes. The synch driver node is used to adjust the publishing period of the sensors, buffering the data of the highest rate one, in order to have a node with a unique rate for all the sensors. Synch nodes are then inserted before the tasks to handle buffered data. In this way, a single-rate DAG is obtained and scheduled using a fixed-priority algorithm based on the HLBS scheduler [111].

Forget et al. [52] faced the same problem for autopilot applications, considering periodic tasks modeled as nodes in a DAG with two kinds of edges: simple and extended. Simple edges are precedence constraints between tasks having the same rate, while extended edges are data dependencies between tasks having different rates. To handle extended edges, a method is proposed to generate multiple conversions from extended edges through simple precedence constraints between jobs, selecting a permutation that guarantees EDF schedulability.

Another conversion method from a multi-rate DAG to a single-rate one has been proposed by Saidi et al. [101] for a similar DAG model. The output DAG has a period equal to the hyper-period of the input task set. The nodes are the job instances activated in a hyper-period for each task. Edges are precedence constraints between jobs, which are inserted based on the ratio between the periods of the communicating tasks. A multi-core heuristic is proposed to schedule the DAG, while minimizing a cost function related to task schedulability.

Converting the original task set to a DAG is a very convenient approach that allows seamlessly inserting explicit precedence constraints to control end-to-end latency. To our knowledge, most of the other methods in the literature perform similar conversions to impose such precedence constraints for limiting latency. While the

work of Becker [17] may appear different, as it does not explicitly consider DAGs, it ends up implementing a similar approach by inserting precedence constraints between different jobs. In Section 4.6, we will highlight the differences between the presented methods and our approach.

4.2 System Model

This section shows how to convert a *Multi-Rate Task set with Constraints* into a *Single-Rate Directed Acyclic Graph* (DAG), in order to analyze schedulability and end-to-end latency of task-chains.

Multi-Rate Task set with Constraints The input to the proposed method is a task set Γ, modeling an application like the one in Figure 4.1, composed of N periodic tasks τ_x arriving at time $t = 0$. Each task τ_x is described by the tuple (C_x, BC_x, T_x, D_x), where: $C_x \in \mathbb{R}$ is the WCET of the task; $BC_x \in \mathbb{R}$ is the Best Case Execution Time (BCET); $T_x \in \mathbb{N}$ is the period; $D_x \in \mathbb{R}$ represents the relative deadline.

The exchange of data between two tasks is modeled with as *data edge*, a directed (dashed) edge between the producer and the consumer of the data. Moreover, precedence constraints may be specified between two tasks (τ_x, τ_y), stating that a job $\tau_{y,b}$ cannot start until all the jobs of τ_x released in τ_y's period completed their execution. For this reason, precedence constraints can be inserted only between tasks having the same period, corresponding to job level precedence constraints.

To constrain the latency of data propagation in task-chains, upper bounds on *data age* and on *reaction time* can be given.

Our approach is based on a global non-preemptive list scheduling approach. Such a policy allows different instances of the same task to run on different cores, while preventing a job to be migrated during its execution, mitigating the preemption overhead.

Directed Acyclic Graph The output of the proposed method is a *single-rate Directed Acyclic Graph (DAG)*. Such a model is based on the parallel DAG model proposed by Baruah et al. [11]. In this work, we use a similar model with a semantic difference, i.e., a DAG represents a full application, with each vertex representing a task instance, which we call a job. In detail, the DAG is specified by a 3-tuple (V, E, HP) where: V represents the set of nodes, namely the jobs of the tasks of Γ, and $n = |V|$; E is the set of edges describing job-level precedence constraints; HP is the period of the DAG, namely the hyper-period of the tasks involved: $HP = lcm_{\forall \tau_x \in \Gamma}\{T_x\}$.

In this model, the communication between jobs utilizes buffers in shared memory, which can be accessed by all the cores. The time to write/read a shared buffer is included in the execution time of each task. We adopt the implicit communication model defined in AUTOSAR [2], solving mutual exclusion via double-buffering. Each task complies with a read-execute-write semantic, i.e., it reads a private copy before the execution, and it writes a private copy at the end of the execution [58].

4.3 DAG Matrices Operations

A DAG ca be represented as an adjacency matrix $\mathbf{T} \in \mathbb{B}^{n \times n}$, in which $T_{i,j} = 1$ iff there exists an edge $e(v_j, v_i)$[1]. Given this Boolean formulation of the DAG, Boolean algebra can be applied. Therefore, the Boolean matrix product is defined as:

$$\mathbf{C} = \mathbf{AB}, \quad \mathbf{A} \in \mathbb{B}^{n \times m}, \mathbf{B} \in \mathbb{B}^{m \times n}, \mathbf{C} \in \mathbb{B}^{n \times n} \tag{4.1}$$

for which the cells of $\mathbf{C}$ evaluate to

$$c_{i,j} = \bigvee_{k=0}^{m-1} a_{i,k} \wedge b_{k,j} \tag{4.2}$$

Cell-wise Boolean operations are denoted as $\wedge$ and $\vee$ for *and* and *or*, respectively. Additionally, a maximum matrix multiplication is used in this work to combine Boolean matrices with real matrices. It is defined as

$$\begin{gathered} \mathbf{C} = \text{maxProduct}(\mathbf{A}, \mathbf{B}), \\ \mathbf{A} \in \mathbb{B}^{n \times m}, \mathbf{B} \in \mathbb{R}^{m \times n}, \mathbf{C} \in \mathbb{R}^{n \times n} \end{gathered} \tag{4.3}$$

where the cells of $\mathbf{C}$ are calculated as

$$c_{i,j} = \max_{k \in \{0,\ldots,m-1\}} \{a_{i,k} b_{k,j}\}. \tag{4.4}$$

Transitive Closure The proposed scheduling method and the related end-to-end latency computation make use of the mathematical principles of graph theory [20]. One principle is the transitive closure [93] of a DAG, defined as

$$\mathbf{D} = \bigvee_{k=1}^{n} \mathbf{T}^k \tag{4.5}$$

where the exponentiation of a Boolean matrix is calculated through the Boolean matrix product defined in Equation (4.1). The transitive closure of a DAG describes the set of descendants of each node, where $d_{i,j} = 1$ if there exists a path from v_j to v_i, i.e., v_i is a descendant of v_j. Consequently, v_j is an ascendant of v_i. The transpose of the descendants matrix, $\mathbf{D}^{\mathrm{T}}$, therefore represents the ascendants matrix.

Computing the power of k of an adjacency matrix of a graph means calculating the nodes reachable through any k-step walk from every node v_i, which is a general result in graph theory (Lemma 2.5 in Biggs [20]). Instead of computing the descendants matrix via Equation (4.5), we can adopt a simpler formulation. By introducing a self-loop to every node, the power of k of the adjacency matrix calculates not only the reachable nodes of any k-step walk, but it also includes the reachable nodes through all shorter walks. Therefore,

$$\mathbf{D} = (\mathbf{T} \vee \mathbf{I})^n \wedge \neg \mathbf{I} \tag{4.6}$$

[1] We chose the column-row approach over the commonly used row-column approach to perform state and value propagation, described later in this section, by left-multiplying the transition matrix to a column state vector.

where $\mathbf{I} \in \mathbb{B}^{n \times n}$ is the identity matrix, and $\neg\mathbf{I}$ is the Boolean complement of $\mathbf{I}$. Given that $\mathbf{T}$ is an acyclic transition matrix, $\mathbf{T}^k$ has no element on the main diagonal $\forall k \in \mathbb{N}_{>0}$. Therefore, the elements introduced on the main diagonal are set back to zero.

State and Value Propagation To use the DAG matrix $\mathbf{T}$ for the analysis of a DAG, two propagation methods are useful. The first is a Boolean state propagation and the second is a maximum value propagation. Let $\mathbf{x}_k \in \mathbb{B}^{n \times 1}$ denote a state describing which node of the DAG is visited at iteration k. Then, the state of the DAG in iteration $k+1$ can be calculated using the Boolean matrix multiplication as:

$$\mathbf{x}_{k+1} = \mathbf{T}\mathbf{x}_k \tag{4.7}$$

In this way $\mathbf{x}_{k+1}$ will contain 1 for the nodes that are reached with one step-walk from the ones in state $\mathbf{x}_k$, 0 for the others.

Similarly, a value can be propagated through the DAG. Let $\mathbf{v}_k \in \mathbb{R}^{n \times 1}$ denote a value for each node of the DAG at iteration k. This value can be propagated through the paths of the DAG by using

$$\mathbf{v}_{k+1} = \text{maxProduct}(\mathbf{T}, \mathbf{v}_k), \tag{4.8}$$

where the vector $\mathbf{v}_{k+1}$ describes the value $\mathbf{v}_k$ in the next iteration.

In this work, we are interested in propagating execution times along the DAG. Given that in a DAG more paths can converge to the same node, we will propagate the maximum value among converging paths. In the case of propagating execution times through the DAG, we can define a value function $\mathbf{v}$ as

$$\mathbf{v} = \text{maxProduct}(\mathbf{T}, \mathbf{v} + \mathbf{c}), \tag{4.9}$$

with $\mathbf{c}$ being the execution time of each node (C or BC). In this equation, the value of a node is equal to the maximum of its predecessors' values plus its execution time. The fixed-point $\mathbf{v}^*$ solving Equation (4.9) can be found by iterating

$$\mathbf{v}_{k+1} = \text{maxProduct}(\mathbf{T}, \mathbf{v}_k + \mathbf{c}) \tag{4.10}$$

until it converges to $\mathbf{v}^*$ when $\mathbf{v}_{k+1} = \mathbf{v}_k$. Convergence is guaranteed to happen after at most n iterations, because the graph is acyclic and, therefore, all its paths are composed of n or fewer nodes.

4.4 DAG Generation

In this section, we explain how to convert a task set of periodic tasks with constraints to a set of potential *single-rate* DAGs. The explanation and mathematical derivations are augmented with an example to illustrate the conversion.

Example 4.1. *We consider an application modeled as a Multi-Rate task set* $\Gamma = \{\tau_0 = (7, 5, 10, 10), \tau_1 = (13, 10, 30, 30), \tau_2 = (10, 8, 30, 30)\}$, *with a constraint on the maximum data age of chain* $\{\tau_0, \tau_1, \tau_2\}$ *to be smaller than 50. The Multi-Rate task set is represented in Figure 4.2.*

A set of DAGs is generated using a 4-Stage DAG Generation. The set is subsequently pruned to accelerate the analysis in the next sections.

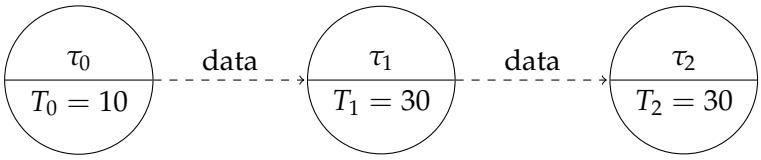
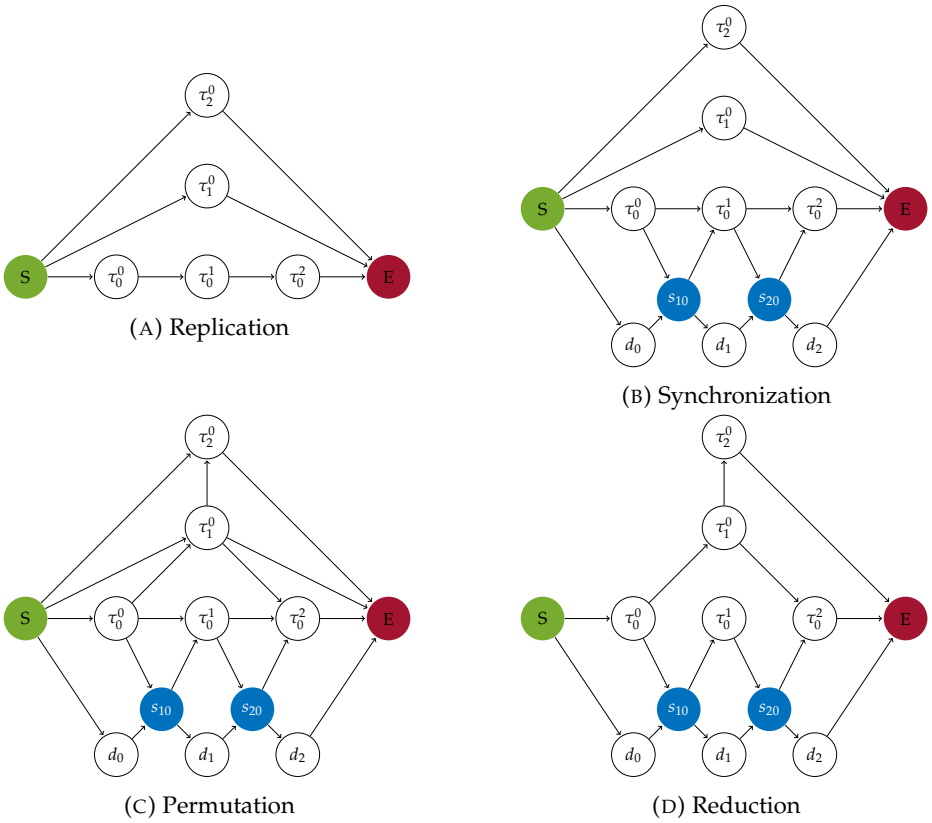


FIGURE 4.2: The simple task set defined in Example 4.1.

4.4.1 4-Stage DAG Generation



(A) Replication

(B) Synchronization

(C) Permutation

(D) Reduction

FIGURE 4.3: The 4-Stage DAG generation depicted.

We aim at generating a set of DAGs that have the potential to meet all the constraints. The DAG generation can be split into four stages:

1. The respective jobs of the tasks are created.
2. The jobs are synchronized to meet their respective deadlines.
3. The job-level precedence edges are added to address the *data edges*.
4. The DAGs are simplified by removing redundant edges.

The four steps for the example are depicted in Figure 4.3. We hereafter detail each step.

Replication Each task has to execute a number of jobs within one hyper-period. For a task τ_x, the number of jobs is $\frac{HP}{T_x}$. Since jobs are just instances of the same task, they should always run sequentially, therefore job-level precedence edges are added between successive jobs τ_x^a and τ_x^{a+1} where $a \in \{0, \ldots, \frac{HP}{T_x} - 1\}$. Additionally,

the start node of the DAG is connected to each first job of each task, and each last job is connected to the end node. The resulting DAG for the example is shown in Figure 4.3a. To synchronize the jobs in the following step, each job gets an offset and deadline. For τ_x^a, the offset is aT_x and the deadline is $aT_x + D_x$.

Synchronization To be sure that tasks' instances maintain their original period and deadlines in the DAG, a synchronization mechanism has to be applied. In this way, we can enforce a job to start after its offset and to finish before its deadline. To accomplish this, we add additional nodes for synchronization purposes, as in Figure 4.3b. Firstly, we add a synchronization node σ_t, with $C = BC = 0$, for each unique value t in the list of offsets and deadlines of all jobs. Secondly, we add dummy nodes δ between each two consecutive synchronization nodes σ_t and $\sigma_{t'}$, with $C = BC = t' - t$, i.e., the difference in the timestamps of the corresponding synchronization nodes. The source and sink of the DAG are synchronization nodes too, with a timestamp of 0 and HP, respectively.

To enforce the jobs to execute in a time-window within its offset and deadline, an edge to the job is added from the synchronization node of the corresponding offset, and another one from the job to the synchronization node corresponding to its deadline.

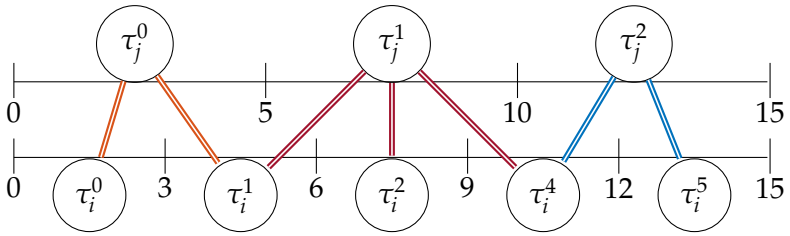


FIGURE 4.4: Example of jobs of non-harmonic tasks limited to their super-period. Doubled arches indicate possible interaction between jobs.

Permutation The various instances of tasks with different periods may be scheduled in multiple ways. We would like to enforce a suitable execution order between such instances, in order to minimize the latency of a given set of task chains. Thus, we convert the original multi-rate task set into several single-rate DAGs, each representing a possible activation pattern of the considered tasks. To do so, we include additional precedence edges to the DAG obtained at the previous step.

Consider two tasks τ_x and τ_y with periods T_x and T_y, assuming $T_y \geq T_x$ without loss of generality. Let $SP_{x,y}$ be the super-period of tasks τ_x and τ_y, defined as the least common multiple of their periods, i.e., $SP_{x,y} = lcm(T_x, T_y)$. Note there are $\frac{HP}{SP_{x,y}} - 1$ super-periods in the hyper-period HP of the whole task set.

There exist multiple ways to insert precedence edges between jobs of τ_x and τ_y in each super-period of length $SP_{x,y}$. Each possible edge assignment that complies with the multi-rate task specification is called "job arrangement".

To find all the possible permutations, two cases must be considered: harmonic and non-harmonic periods. In the former case, there exists $q \in \mathbb{N}$ for which $q = \frac{T_y}{T_x}$ and $SP_{x,y} = T_y$. Therefore, finding all the permutations between one job of τ_y and q of τ_x allows finding all the job arrangements in their super-period. The non-harmonic case is slightly more complicated. For two non-harmonic tasks, $q \in \mathbb{N}$ can be computed as $q = \lceil \frac{T_y}{T_x} \rceil$, but $SP_{x,y} \neq T_y$. In this case, one job of τ_y can be arranged

with q or $q+1$ jobs of τ_x, because of the non-harmonicity. To better understand the problem, let us consider an example in which $T_x = 3$ and $T_y = 5$, as in Figure 4.4.

When periods are harmonic, a job of τ_x always interact with exactly one job of τ_y (and respectively, τ_y interacts with exactly q jobs of τ_y) . However, for non-harmonic periods, a job of τ_x can interact with 1 or 2 (at most) jobs of τ_y, as shown in Figure 4.4. For this reason, in the non-harmonic scenario some jobs of τ_y will interact with q (in the example $\lceil \frac{5}{3} \rceil = 2$, as for $\tau_{y,0}$ and $\tau_{y,2}$) jobs of τ_x, while others with $q+1$ (in this case 3, as for $\tau_{y,1}$).

In general, a job τ_y^a can interact with all the jobs between τ_x^b and τ_x^c, where b and c can be obtained as:

$$b \in \mathbb{N} \mid O(\tau_x^b) \leq O(\tau_y^a) \wedge O(\tau_x^b) + T_x > O(\tau_y^a) \tag{4.11}$$

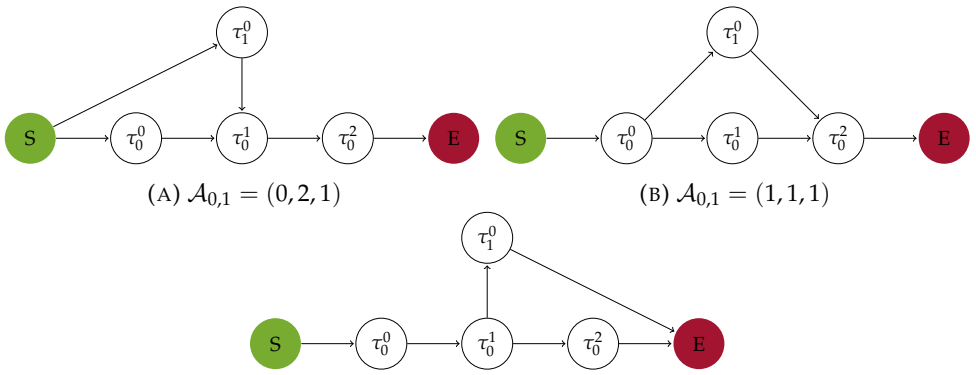
$$c \in \mathbb{N} \mid O(\tau_x^c) < O(\tau_y^a) + T_y \wedge O(\tau_x^c) + T_x \geq O(\tau_y^a) + T_y \tag{4.12}$$

where $O(\tau_x^b)$ stands for the offset of the job τ_x^b.

Once the interacting job of τ_x and τ_y have been associated, this case can be traced back to the harmonic one.

Now, let us consider a job τ_y^s and all the possible arrangements with Q jobs of τ_x (which is either q or $q+1$), denoted as $\mathcal{A}_{x,y}(s) = (pre_s, post_s, \psi_s)$ in the super-period $SP_{x,y}$. In this tuple, pre_s (resp. $post_s$) denotes the number of jobs of τ_x executing before (resp. after) each job of τ_y. ψ_s denotes the number of jobs of τ_x that can execute in parallel to the job of τ_y. This parameter is critical for the data update *variability*, that is defined as the difference between the maximum and minimum number of data updates. Since pre_s, $post_s$, and ψ_s comprise all the jobs of τ_x interacting with τ_y^s, it follows that

$$pre_s + post_s + \psi_s = Q. \tag{4.13}$$



(A) $\mathcal{A}_{0,1} = (0, 2, 1)$ (B) $\mathcal{A}_{0,1} = (1, 1, 1)$

(C) $\mathcal{A}_{0,1} = (2, 0, 1)$

FIGURE 4.5: Arrangement permutations with one parallel job of τ_x; there are three permutations because $Q = 3$ and $\psi_0 = 1$, therefore $Q + 1 - \psi = 3$. The permutation (b) corresponds to the example in Figure 4.3c.

Three example arrangements for two tasks, τ_0 with $T_0 = 10$ and τ_1 with $T_1 = 30$, are shown in Figure 4.5. In all three arrangements, the job of τ_1 is parallel to one job of τ_0 ($\psi_0 = 1$). The number of permutations of arrangements with each τ_y^s can be calculated as:

$$perm(\mathcal{A}_{x,y}(s)) = \sum_{\psi=\{0\ldots Q\}} (Q + 1 - \psi), \tag{4.14}$$

while the permutations can be found combining all the possible edges between the jobs of the two tasks.

For the harmonic case, this value is also the total number of permutations of a super-period:

$$perm_{SP_{x,y}} = perm(\mathcal{A}_{x,y}(s)). \tag{4.15}$$

On the other hand, for the non-harmonic case, the number of permutations for the super-period is obtained as:

$$perm_{SP_{x,y}} = \prod_{\forall \tau_y^s \in \{0 \dots \frac{SP_{x,y}}{T_y}\}} perm(\mathcal{A}_{x,y}(s)). \tag{4.16}$$

Finally, considering all the super-periods contained in a hyper-period, the total number of permutations can be given by:

$$perm_{total} = \prod_{\forall x,y} perm_{SP_{x,y}}^{\frac{HP}{SP_{x,y}}}, \tag{4.17}$$

where $x \neq y$ and τ_x and τ_y are consecutive tasks in a given task chain. Each combination of arrangement permutations generates a new DAG that can be analyzed. Therefore, it is critical to keep the number of possible permutations as small as possible. A reduction of the exploration space is discussed in Section 4.4.2.

Figure 4.3c shows one of the obtained DAG, whose simplified[2] adjacency matrix $\mathbf{T}$ and transitive closure matrix $\mathbf{D}$ (obtained with Equation (4.6)) are the following:

$$\mathbf{T} = \begin{array}{c} S \\ \tau_0^0 \\ \tau_0^1 \\ \tau_0^2 \\ \tau_1^0 \\ \tau_2^0 \\ E \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \quad \mathbf{D} = \begin{array}{c} S \\ \tau_0^0 \\ \tau_0^1 \\ \tau_0^2 \\ \tau_1^0 \\ \tau_2^0 \\ E \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Reduction While constructing the DAGs, it is possible to end up generating redundant edges. There is a redundant edge between two nodes when there exist both a direct edge and a non-direct path. Redundant edges can be removed using a technique called transitive reduction, firstly proposed by Aho et al. [1]. The transitive reduction of a DAG uniquely describes the sub-graph of this DAG with the fewest possible edges, while maintaining the same reachability relation.

The transitive reduction of a DAG can be calculated in different ways. Since in this work we need the transitive reduction as well as the transitive closure of the DAG, we compute the transitive reduction using

$$\mathbf{T}_r = \mathbf{T} \wedge \neg(\mathbf{T} \cdot \mathbf{D}), \tag{4.18}$$

where $(\mathbf{T} \cdot \mathbf{D})$ has 1 in (j, i) if the node j can reach the node i in more than one step, 0 otherwise. Applying Equation (4.18) means removing direct edges $e(v_j, v_i)$ in $\mathbf{T}$ that are redundant because a non-direct path already exists between node j and node i.

[2] Without synchronization and dummy nodes, removed for a clearer representation, but used in the actual algorithm.

In Figure 4.3d, the obtained DAG with reduced edges is presented. For that example, the matrix $\mathbf{T} \cdot \mathbf{D}$ and $\mathbf{T_r}$ (obtained with Equation (4.18)) are the following[3]:

$$\mathbf{T} \cdot \mathbf{D} = \begin{matrix} S \\ \tau_0^0 \\ \tau_0^1 \\ \tau_0^2 \\ \tau_1^0 \\ \tau_2^0 \\ E \end{matrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \mathbf{T_r} = \begin{matrix} S \\ \tau_0^0 \\ \tau_0^1 \\ \tau_0^2 \\ \tau_1^0 \\ \tau_2^0 \\ E \end{matrix} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

The interaction of the different *data edges* during the Permutation stage can result in DAGs that are inherently not schedulable. These DAGs can be removed to speed up the analysis. Then, two factors are further inspected: potential cycles in the generated DAG, and length of the longest chain.

DAGs containing cycles need to be removed, as they are inconsistent with the task semantics and they could not be feasibly scheduled. Finding cycles in a graph is a common problem which can be solved with several approaches. In this work, we use state propagation, described in Section 4.3. We adopt a state vector $\mathbf{x}$, whose elements indicate whether a path exists (1) or not (0). Initially, $\mathbf{x}_0 = \mathbf{1}$ to consider the potential paths from all the nodes. Then, we apply state propagation in Equation (4.7), multiplying the state vector with the adjacency matrix $\mathbf{T}$. This means stepping from a node to its successor: if it has any, the resulting vector will have a 1 in the corresponding position, otherwise it will have a 0. Repeating this operation means going through all the possible paths. Since the graph is acyclic and it has n nodes, there should be no path with a length greater than n. In other words, the resulting vector should have all 0's after at most n steps, indicating that all the paths have ended, i.e., there are no more nodes to step into. If this is not the case, it means the DAG contains cycles, and it can be discarded.

Finally, the longest chain in the DAG corresponds to the chain with the longest execution time. This chain can be explicitly found by calculating the fixed-point of Equation (4.9) with $\mathbf{c} = \mathbf{C}$, the WCET of each node. If any value in $\mathbf{v}^* + \mathbf{C}$ is bigger than the hyper-period HP, it means that there exists a path whose sum of WCETs exceeds the hyper-period, which makes the DAG not schedulable. Also these DAGs are discarded.

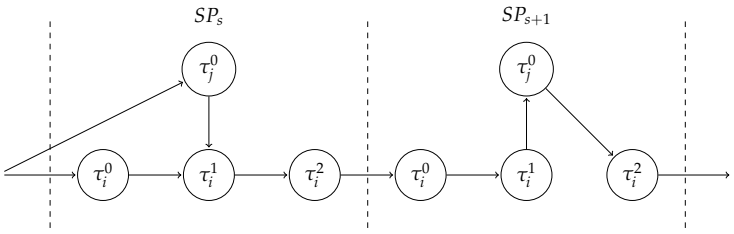
In the example, the vectors of $\mathbf{C}$ and $\mathbf{v}^*$ are the following:

$$\mathbf{C} = \begin{bmatrix} 0 \\ 7 \\ 7 \\ 7 \\ 10 \\ 13 \\ 0 \end{bmatrix} \quad \mathbf{v}^* = \begin{bmatrix} 0 \\ 7 \\ 14 \\ 24 \\ 17 \\ 30 \\ 30 \end{bmatrix}$$

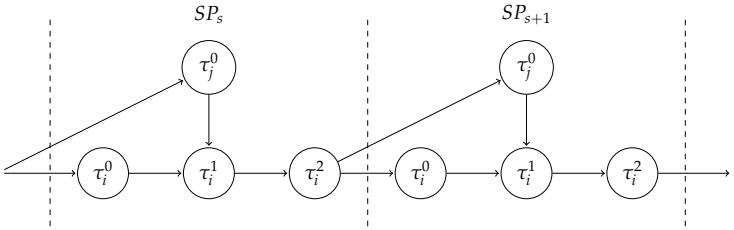
[3]Given the previously mentioned simplification, consecutive jobs of the same task have precedence constraints between them, rather than having edges to and from synchronization nodes. In the example, there is an edge from τ_0^0 to τ_0^1 and one from τ_0^1 to τ_0^2.

4.4.2 Permutation Space Reduction

The worst-case number of permutations, and thus the total number of DAGs created, is given in Equation (4.17), i.e., it is scaling exponentially with the size of the task set. Therefore, a reduction of the permutation space is essential to keep the approach computationally tractable for larger task sets. To reduce the permutation space, we inspect the inter-super-period-arrangement.



(A) Heterogeneous arrangement example with $\mathcal{A}_{x,y}(s) = (0,2,1)$ and $\mathcal{A}_{x,y}(s+1) = (2,1,0)$



(B) Homogeneous arrangement example with $\mathcal{A}_{x,y}(s) = \mathcal{A}_{x,y}(s+1) = (0,2,1)$

FIGURE 4.6: Examples showing heterogeneous and homogeneous arrangements.

Given the previously adopted tasks τ_0 and τ_1, Figure 4.6 shows two possible arrangements, omitting synchronization nodes for simplicity.

Let us consider a couple of harmonic tasks τ_x, a τ_y in their super-period $SP_{x,y}$. For a given parallelism ψ_s, relative to job τ_y^s, the execution order of the parallel jobs is not defined. Therefore, a bounded number of jobs of τ_x, denoted as $prePar_s \in \{0, \ldots, \psi_s\}$, can execute before τ_y^s. Consequently $\psi_s - prePar_s$ jobs of τ_x will execute after τ_y^s. The probability distribution of $prePar_s$ is not relevant since the only values that affect the latency variability are, by definition, the extremes, i.e.,

$$\max(prePar_s) = \psi_s \text{ and } \min(prePar_s) = 0 \tag{4.19}$$

Based on these definitions, the number of jobs of τ_x between two consecutive jobs τ_y^s and τ_y^{s+1} (in the following super-period) is given by

$$n_{s,s+1} = post_s + (\psi_s - prePar_s) + pre_{s+1} + prePar_{s+1} \tag{4.20}$$

The upper and lower bound of this value are given by

$$\max(n_{s,s+1}) = post_s + \psi_s + pre_{s+1} + \psi_{s+1} \tag{4.21}$$

$$\min(n_{s,s+1}) = post_s + pre_{s+1} \tag{4.22}$$

The *variability* of the data updates, i.e., the difference between the maximum and the minimum data updates in between, can be formalized as:

$$Var_{x,y} = \max_s\{\max(n_{s,s+1})\} - \min_t\{\min(n_{t,t+1})\}, \tag{4.23}$$

using t in the minimum to highlight that the maximum and minimum do not need to consider the same job of τ_y, and thus the same arrangement. However, in a homogeneous arrangement, $\mathcal{A}_{x,y}(s) = \mathcal{A}_{x,y}(s+1) = (pre, post, \psi)$ and $s = t$. Therefore,

$$Var_{x,y,hom} = 2\psi_s \tag{4.24}$$

Comparing to heterogeneous arrangements, in which $\mathcal{A}_{x,y}(s) \neq \mathcal{A}_{x,y}(s+1), \forall s$, two observations can be made. On the one hand, a higher value of ψ_s for a job τ_y^s increases schedulability of the related super-period, since it allows for more parallelism and shortens the longest path. Given that the schedulability of all the super-periods determines the schedulability of the hyper-period, the value of ψ_s is crucial. On the other hand, from an application side, the data update *variability* should be as low as possible to constrain end-to-end latency.

To reduce the permutation space while investigating all the permutations that optimize latency, we chose to sacrifice optimality w.r.t. schedulability. Homogeneous arrangements are better at this compromise. To show it, we prove that

$$Var_{x,y,het} > Var_{x,y,hom} \tag{4.25}$$

Proposition: Given two tasks τ_x and τ_y with periods $gT_x = T_y, \quad hT_y = HP, \quad g, h \in \mathbb{N}_+$, a heterogeneous arrangement results in a strictly higher *variability* than a homogeneous arrangement.

Proof: In a heterogeneous arrangement, $\mathcal{A}_{x,y}(s) \neq \mathcal{A}_{x,y}(s+1)$, which means that $(pre_s, post_s, \psi_s) \neq (pre_{s+1}, post_{s+1}, \psi_{s+1})$. Let us define $\alpha_s \in \mathbb{Z}$ (resp. $\beta_s \in \mathbb{Z}$) as the difference between the jobs of τ_x that execute before (resp. after) τ_y^{s+1} and the jobs of τ_x that execute before (resp. after) τ_y^s[4]

$$\alpha_s = pre_{s+1} - pre_s \tag{4.26}$$
$$\beta_s = post_{s+1} - post_s \tag{4.27}$$

Consequently, considering that $pre_s + post_s + \psi_s = pre_{s+1} + post_{s+1} + \psi_{s+1} = Q$, we derive

$$\psi_{s+1} = \psi_s - \alpha_s - \beta_s \tag{4.28}$$

With this definition, Equation (4.20) provides

$$\begin{aligned} n_{s,s+1} &= post_s + (\psi_s - prePar_s) + pre_{s+1} + prePar_{s+1} \\ n_{s,s+1} &= post_s + (\psi_s - prePar_s) + pre_s + \alpha_s + prePar_{s+1} \\ &= Q + \alpha_s - prePar_s + prePar_{s+1}. \end{aligned}$$

Then,

$$\begin{aligned} \max(n_{s,s+1}) &= Q + \alpha_s + \psi_{s+1} \\ &= Q + \psi_s - \beta_s \\ \min(n_{s,s+1}) &= Q + \alpha_s - \psi_s \end{aligned}$$

[4]Remember that there is only one job of τ_y^s in each super-period $SP_{x,y}$. Therefore, τ_y^{s+1} refers to the next super-period.

The *variability* in Equation (4.23) can then be simplified to

$$\begin{aligned} Var_{x,y,het} &= \max_s\{\max(n_{s,s+1})\} - \min_t\{\min(n_{t,t+1})\} \\ &= \max_s\{\psi_s - \beta_s\} - \min_t\{\alpha_t - \psi_t\} \\ &= \max_s\{\psi_s - \beta_s\} + \max_t\{\psi_t - \alpha_t\} \\ &= 2\psi_s + \max_s\{-\beta_s\} + \max_t\{-\alpha_t\}. \end{aligned}$$

As the full arrangement is the same in each hyper-period, the super-period arrangement is cyclic. Since α_s and β_s denote the change of the arrangement, the cyclicity of $\mathcal{A}$ requires

$$\sum_{s\in\{0,\ldots,\frac{HP}{T_y}\}} \alpha_s = \sum_{s\in\{0,\ldots,\frac{HP}{T_y}\}} \beta_s = 0. \tag{4.29}$$

Therefore, $\exists\alpha_s < 0$ and $\exists\beta_s < 0$ such that

$$Var_{x,y,het} > 2\psi_s \tag{4.30}$$

Since $Var_{x,y,het} > 2\psi_s$, it then follows $Var_{x,y,het} > Var_{x,y,hom}$, proving the proposition. □

We can therefore omit heterogeneous arrangements without affecting the resulting end-to-end latency, since no such arrangement can provide a better compromise with respect to *variability*. By discarding the heterogeneous arrangements in the permutations, the value of $perm_{total}$ in Equation (4.17) can be reduced to

$$perm_{total} = \prod_{\forall x,y} perm_{SP_{x,y}}, \tag{4.31}$$

where $x \neq y$ and τ_x and τ_y are consecutive tasks in a given task chain, and the full hyper-period arrangement is defined by a unique super-period arrangement. This is valid both for harmonic and non-harmonic tasks.

4.4.3 Computational Complexity

The computational cost of the overall method can be summarized as $\mathcal{O}(perm_\psi \times perm_\mathcal{A} \times n^4)$. The first term $perm_\psi$ represents all the permutations for all the possible ψ values. From Equation (4.31), it can be expressed as

$$perm_\psi = \prod_{(e_x,e_y)\in E} \frac{\max(T_x, T_y)}{\min(T_x, T_y)}. \tag{4.32}$$

The second term $perm_\mathcal{A}$ represents all the arrangement permutations for a fixed ψ. From Equation (4.14), it can be expressed as

$$perm_\mathcal{A} = \prod_{(e_x,e_y)\in E} \left(\frac{\max(T_x, T_y)}{\min(T_x, T_y)} - \psi + 1\right). \tag{4.33}$$

Lastly, $\mathcal{O}(n^4)$ is the maximum cost of all the math operations applied on the obtained DAGs, which are matrix-vector multiplication $\mathcal{O}(n^2)$, matrix multiplication $\mathcal{O}(n^3)$ and matrix exponentiation $\mathcal{O}(n^4)$. Let us define R as the maximum ratio between

periods of the task set, i.e., $R = \frac{max(T_x)}{min(T_y)} \forall x, y \in \{0 \dots N-1\}$. The computational cost of the method can then be expressed as:

$$\mathcal{O}(R^{|E|}R^{|E|}(RN)^4) = \mathcal{O}(R^{2|E|}(RN)^4). \tag{4.34}$$

The complexity is thus exponential in the number of edges $|E|$. Such a high cost is mainly determined by the need to take into account all the permutations at once. However, this is also the reason why the proposed conversion method allows better controlling end-to-end latencies, jointly optimizing data and reaction times of all the task chains given in input. This is achieved by picking up the best configuration out of all the permutations generated by means of a cost function.

4.5 End-to-End Latency and Schedulability

In this section, a method to calculate an upper bound on *data age* and *reaction time* is proposed. As explained in the introduction, data age defines the maximum time a data produced by the first task of the chain can influence the last one. Reaction time is the maximum interval between the acquisition of a stimulus in the first task of a chain and the moment the first instance of the last task in the chain reacts to it.

We first define a set of additional timing attributes, that will be used to compute the end-to-end latency. The schedulability of the DAG is verified by deriving a static schedule. If more than one generated DAG meets the latency and schedulability constraints, we select the DAG that maximizes a weighted sum of the end-to-end latencies, taking into account all the tasks chains in input.

For each job, we define the following timing attributes: Earliest Finishing Time (EFT), Latest Finishing Time (LFT), Earliest Starting Time (EST) and Latest Starting Time (LST). The earliest a node can start is the maximum of all its predecessors' earliest finishing times. Similarly, the latest a node can finish is the minimum of its successors' latest starting times. These values can be iteratively calculated using the operators defined in Section 4.3, initializing $EST_j = 0$ and $EFT_j = HP$ for all nodes j:

$$\begin{aligned} EST_i &= \max_{\forall j} \{(EST_j + BC_j)\mathbf{T}_{j,i}\} \\ LFT_i &= \min_{\forall j} \{(LFT_j - C_j)\mathbf{T}_{i,j}\} \\ EFT_i &= EST_i + BC_i \\ LST_i &= LFT_i - C_i. \end{aligned}$$

Table 4.1 reports the timing attributes computed for Example 4.1.

job	EST	LST	EFT	LFT
τ_0^0	0	0	5	7
τ_0^1	10	13	15	20
τ_0^2	20	23	25	30
τ_1^0	5	7	15	20
τ_2^0	15	20	23	30

TABLE 4.1: Timing attribute for the for Example 4.1.

4.5.1 Task Chain Propagation

In a DAG $\mathcal{G}$, a node j is defined to *react* to node i if there exists a direct or indirect edge from node i to node j. A node k *reacting* to node j also *reacts* to node i. Further, a node k *reacts* to the chain (i, j) if node j *reacts* to node i and node k reacts to node j.

Extending this definition to tasks and jobs:

- τ_y^b *reacts* to $\tau_y^a, \forall b > a$;
- Consequently, if τ_y^a *reacts* to τ_x^c, it follows that τ_y^b reacts to τ_x^c.

Given a task chain $(\tau_x, \ldots, \tau_z)$, the *reactions* of jobs of task τ_z to each job of τ_x can be found. Consider a job τ_x^a of the first task in the chain. The *first* (resp. *last*) *reaction* to τ_x^a is defined as the first (resp. last) job of the last task τ_z that reacts to τ_x^a. The *reaction time* (resp. *data age*) is then defined as the maximum interval between a stimulus in a job τ_x^a and the finishing time of the first (resp. last) reaction, taken over all instances τ_x^a, for all $a \in [0, \frac{HP}{T_x}]$. Since the structure of the DAG repeats after each hyper-period, it is sufficient to consider only the first hyper-period.

Algorithm 1: findReactions

Input: $\mathcal{C} = \{\tau_{start}, ..., \tau_{end}\}$
Output: $1^{st}reactions, lastreactions$

```
1  forall a ∈ {0, ..., HP/T_start + 1} do
2      fr_job = τ^a_start;
3      lr_job = null;
4      forall τ_x ∈ C \ τ_start do
5          b = 0;
6          while τ^b_x does not react to fr_job do
7              b++;
8          fr_job = τ^b_x;
9          if b > 0 then
10             lr_job = τ^{b-1}_x;
11     if a ≤ HP/T_start then
12         1st reactions.insert(τ^a_start, fr_job);
13     if (b ≠ null) and (a > 0) and (1st reactions(τ^{a-1}_start) ≠ fr_job) then
14         lastreactions.insert(τ^{a-1}_start, lr_job);
15 return 1st reactions, lastreactions;
```

A method to compute the *first* and *last reactions* is shown in Algorithm 1. The algorithm considers every job of the starting task of the chain in one hyper-period, plus an additional job (to cover the last reactions). The first reacting job (fr_job) is set to τ_{start}^a. Then, for each task in the chain, we find the first job τ_x^b that reacts to fr_job, and we use it to update fr_job. This can happen either in the same hyper-period of fr_job, or in the next one. The preceding job τ_x^{b-1} is instead used to update lr_job, which keeps track of the last reaction to τ_{start}^{a-1}. Once the whole chain has been considered, $1^{st}reactions$ and $lastreactions$ are updated. The latter is updated only if b is not null and if the first reaction to τ_{start}^a is different from the first reaction to τ_{start}^{a-1}. *Reaction time* and *data age* can then be simply derived as

$$RT = \max_{\tau_x^a \in 1^{st}reactions} \{LFT_{1^{st}reactions} - EST_{\tau_x^a}\} \tag{4.35}$$

$$DA = \max_{\tau_x^a \in lastreactions} \{LFT_{lastreaction} - EST_{\tau_x^a}\}, \tag{4.36}$$

i.e., reaction time (resp. data age) is the difference between the first (resp. last) moment some data is used by a job of the last task in the chain (LFT) and the first moment the same data is read from the job of the first task in the chain (EST). Since the schedule repeats identically after each hyper-period, it is sufficient to consider all the jobs of the first task in the first hyper-period.

We hereafter prove that Algorithm 1 correctly finds the first and last reactions. The algorithm considers all the jobs of the starting task in the chain (line 1). For each of the starting jobs, it iterates over all the other tasks in the chain, always looking for the first and last reacting job (lines 4-14). Let us consider two consecutive tasks in the chain τ_x and τ_y and only one job a of τ_x.

- To find the maximum reaction time, the jobs of τ_y that are said to react to τ_x^a are those that are definitely executing after τ_x^a, i.e., they belong to τ_x^a's descendants, or their EST is greater than the LFT of τ_x^a. Since the DAG is schedulable, a reacting job can always be found (and the loop at line 6 is not infinite) either in the same hyper-period of τ_x^a, or in the next one. Once a job τ_x^b is found to react to τ_x^a, it becomes the starting job to find the first reaction between τ_y and the next task in the chain.

- The maximum data age of τ_x^a is strictly related to the first reaction to τ_x^{a+1}. Indeed, the first reaction to τ_x^{a+1} assures that the data from τ_x^a are no longer used: the last time they were used was by the job preceding the one that surely reacts to τ_x^{a+1}. Thus, when finding the first reaction τ_x^{a+1}, the last reaction of τ_x^a can be found (line 14).

In the example DAG in Figure 4.3d, *data age* is 30, while *reaction time* is 50. The chains leading to these values are $\{\tau_0^0, \tau_1^0, \tau_2^0\}$ for *data age* and $\{\tau_0^1, \tau_1^{0'}, \tau_2^{0'}\}$ for *reaction time*, where a prime indicates that the job is in the next hyper-period. However, in this case it can be noticed that the data age is smaller than the reaction time, which is counterintuitive. This happens because the reaction time is computed for each job of the first task of the chain, even if the data produced by that job could be later overwritten by a more recent one, before other tasks in the chain can actually react to it. This occurs when the data age is smaller, as in the example. Therefore when the data age is smaller than the reaction time, the actual reaction time will be always bounded by the data age value.

4.5.2 Schedulability

To build a feasible schedule for a given number of cores, we apply a list-scheduling heuristic for non-preemptive DAG, very similar to the Heterogeneous Earliest Finishing Time (HEFT) algorithm presented by Topcuoglu et al. [116]. We decided to use a (node-level) limited preemptive scheduling for (i) avoiding job-level migrations, (ii) reducing cache-related preemption delays, and (iii) minimizing the input-output delay and jitter [30].

The list-scheduling algorithm is summarized in Algorithm 2. Jobs are sorted in increasing LFT order (line 3). Given p homogeneous processors, a job is scheduled at time t only if it is ready and a processor is available. A job enters the ready queue (line 8) at time t only if (i) its EST is greater than or equal than t, (ii) all its predecessors in the DAG have been executed, and (iii) its LFT is the smallest between all the remaining jobs' LFT. The ready queue is sorted in increasing LFT order (line 9). A

ready job is scheduled if a processor is available and if its execution time, starting from the current t, does not exceed its LFT (lines 13,16,17). If this last condition is not met, the algorithm declares the DAG not schedulable (lines 13,14). An example of the schedule obtained for Example 4.1 is shown in Figure 4.7.

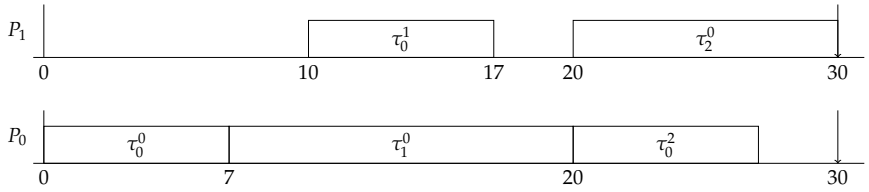


FIGURE 4.7: Schedule produced for the DAG in Figure 4.3d with 2 cores

Algorithm 2: isDAGSchedulable

Input: $V, pred, p, EST, LFT, C$
Output: *true* if the DAG is schedulable on p processors, *false* otherwise
Data: Ready queue of jobs $rq = \{\}$, procExec vector

1 $pq_i = \{\}, \forall i = 1, \ldots, p$;
2 $nodes = \{v_0, \ldots, v_{n-1}\}, n = |V|$;
3 sort($nodes$) `sort by ascending value of LFT`
4 **for** $t = 0, 1, \ldots, HP$ **do**
5 **forall** $node \in nodes$ **do**
6 **if** $EST[node] > t$ ***and*** *all* $pred[node]$ *have finished* ***and*** $LFT[node] \leq$ *all other nodes LFTs* **then**
7 $nodes = nodes \setminus node$;
8 $rq.push(node)$;
9 sort(rq) `sort by ascending value of LFT`
10 **for** $i = 1, ..., p$ **do**
11 **if** $rq \neq \{\}$ ***and*** $procExec_i == 0$ **then**
12 $readyJob = rq.pop()$;
13 **if** $t + C[readyJob] > LFT[readyJob]$ **then**
14 **return** *false*;
15 **else**
16 $procExec_i = C[readyJob]$;
17 $pq_i.push(readyjob)$;
18 **if** $procExec_i > 0$ **then**
19 $procExec_i = procExec_i - 1$;
20 **return** *true*;

4.6 Evaluation

To evaluate our approach, we first use simulation to validate end-to-end latency bounds as well as schedulability and then compare the proposed method with the state-of-the-art using a realistic automotive benchmark.

4.6.1 Evaluation via Simulation

To validate that the DAGs generated with the method presented in this paper comply with the constraints, we developed a simulation tool. The tool uses the DAG to schedule the individual tasks, which tracks the data propagation through the task chains under analysis. The execution time of each task is identically and independently sampled from the BC to C interval. The schedule is generated according to EDF, and the deadline is set equal to LFT.

We simulated the best DAG, in terms of schedulability and end-to-end latency, produced for the application introduced in Figure 4.1. The task set specification and constraint are detailed in Table 4.2. The latency computed for the given chains is reported in Table 4.3. The best DAG resulting from the conversion is the one depicted in Figure 4.8.

Using the simulation tool, the DAG is simulated for 10^9 ms, which leads to the following results. Two distributions of *reaction time* and *data age* of two task chains are shown in Figure 4.9 and Figure 4.10. In Figure 4.9 the *reaction time* plot is showing only one distributions, bounded by the data age. In the DAG, the camera jobs are serialized to the detection job, leading to only one distribution for the *data age*, because the detection job always receives the freshest camera frame. A similar distribution for the *reaction time* can be seen for the task chain in Figure 4.10, as the task chain, is extended with the planner and control task. The *data age*, however, shows several distributions. This is due to the higher rate of the planner and control task with respect to the fusion task. Nevertheless, the *data age* of the data corresponding to each control output is always based on the freshest camera frame, which can be seen by comparing the distribution shapes. The simulation showed that all the calculated upper bounds for *data age* and *reaction time* for the four task chains are not exceeded.

Task set Γ	
$\tau_i = (C_i, BC_i, P_i, D_i)$	Task
$\tau_0 = (7, 5, 50, 50)$	GPS
$\tau_1 = (12, 10, 50, 50)$	Lidar
$\tau_2 = (28, 22, 50, 50)$	Localization
$\tau_3 = (28, 25, 50, 50)$	Detection
$\tau_4 = (25, 18.9, 50, 50)$	Fusion
$\tau_5 = (2, 1.8, 25, 25)$	Camera
$\tau_6 = (6.5, 3, 10, 10)$	EKF
$\tau_7 = (5, 3.2, 10, 10)$	Planner
$\tau_8 = (4.5, 1.8, 10, 10)$	Control
Task chains	
chain $\{\tau_{start}, \ldots, \tau_{end}\}$	(Age, Reaction)
$\{\tau_5, \tau_3, \tau_4\}$	$(120, 120)$
$\{\tau_0, \tau_2, \tau_6, \tau_7, \tau_8\}$	$(120, 150)$
$\{\tau_1, \tau_2, \tau_6, \tau_7, \tau_8\}$	$(120, 150)$
$\{\tau_5, \tau_3, \tau_4, \tau_7, \tau_8\}$	$(150, 150)$
Scheduling constraints	
6 processors	

TABLE 4.2: Periodic task set and constraints used for the simulation, referring to the application of Figure 4.1.

chain $\{\tau_x, \ldots, \tau_y\}$	(Age, Reaction)
$\{\tau_5, \tau_3, \tau_4\}$	$(75, 98.2)$
$\{\tau_0, \tau_2, \tau_6, \tau_7, \tau_8\}$	$(105, 65)$
$\{\tau_1, \tau_2, \tau_6, \tau_7, \tau_8\}$	$(105, 65)$
$\{\tau_5, \tau_3, \tau_4, \tau_7, \tau_8\}$	$(125, 108.2)$

TABLE 4.3: Maximum *data age* and *reaction time* for task chains of the best DAG produced for the task set described by Table 4.2

4.6.2 Evaluation via Benchmark

To further analyze the performance of the proposed method the detailed automotive benchmark proposed by BOSCH for the WATERS challenge in 2015 [68] has been adopted. Multi-rate periodic task sets and cause-effect chains are randomly generated while conforming with the characterization. Task periods are selected with given distribution, out of the periods found in automotive applications [1, 5, 10, 20, 50, 100, 200, 1000]ms. Cause-effect chains are generated to include tasks of either 1, 2, or 3 different period wherein tasks of the same period can appear 2 to 5 times. To obtain a higher utilization, the individual task execution times are generated based on UUniFast [21]. For the experiments 1000 task set composed of 5 tasks and 15 chains have been taken into account, with a utilization equal to 1.5, considering 2 cores available.

	permutation	admissible (%)	schedulable (%)
min	0.00	0.00	0.00
avg	1.830.48	62.10	61.60
max	18.148.00	100.00	100.00

TABLE 4.4: Statistics about DAG permutations, admissible and schedulable DAGs on 1000 different task set.

Table 4.4 reports some statistics about the DAG obtained from the 1000 multi-rate periodic task sets. From the initially generated permutations the 40% is on average removed due to cycles or a non-schedulable longest chain. However, between the admissible generated DAGs[5], almost the totality is also schedulable on 2 cores.

4.6.3 Comparison with state-of-the-art

Qualitative Saidi et al. [101] present a method to convert a parallel multi-rate task set with precedence and data edges into a single-rate DAG. However, the end-to-end latency is not considered, and only one possible DAG is generated. Therefore, no guarantee is given on *reaction time* or *data age*. Moreover, there are no synchronization methods to force task instances to execute within their periods, potentially leading to a wrong implementation of the system.

Foget et al. [52] present another conversion method, producing different DAGs. However, neither this work takes into account latency. Different solutions are created just for schedulability reasons, picking the version that makes the DAG schedulable with EDF.

[5]DAGs that have a correct structure (i.e., no cycles and no path greater than the hyper-period) but that may still be unschedulable.

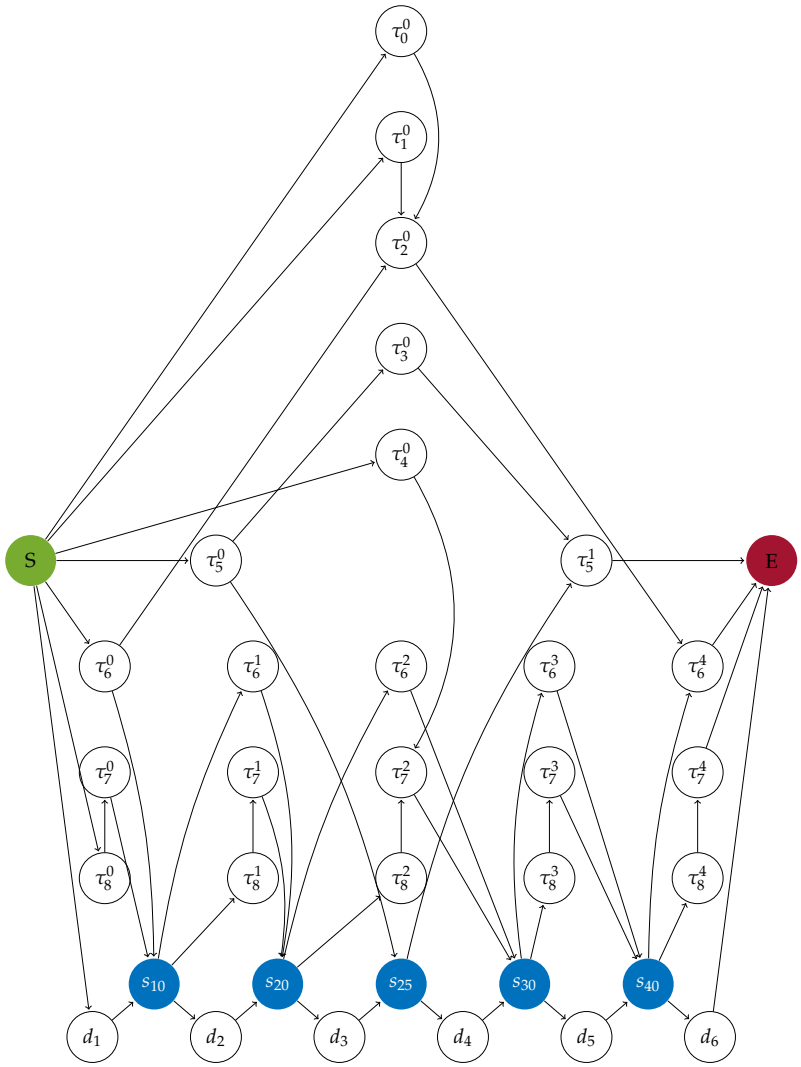


FIGURE 4.8: The DAG generated for the task set and constraint definitions in Table 4.2.

Focusing on data latency, the most related approach is the one introduced by Becker et al. [17], where the focus is on the computation of *data age*. In their work, they can compute *data age* for a given chain of periodic tasks, given a communication model (i.e. implicit, explicit or LET). The method allows generating job-level dependencies to meet latency requirements. However, *data age* is the only parameter under their analysis. Moreover, they can optimize end-to-end latency for only a single chain. Once job-level dependencies are inserted, all the other chains are affected. Finally, their work assumes the input task set is already schedulable. Our work has several improvements over their approach, i.e., (i) the model is more general and can jointly optimize the latency of multiple chains, (ii) we consider not only *data age*, but also *reaction time*, and (iii) our task allocation and scheduling algorithms also consider the schedulability of the system.

Quantitative To show that our method dominates the state-of-the-art, we implemented the solutions proposed by Saidi et al. [101], Forget et al. [52], Becker et

[6]Since no method is proposed by Becker et al [17] to check schedulability, we applied our method to derive the schedulable task sets.

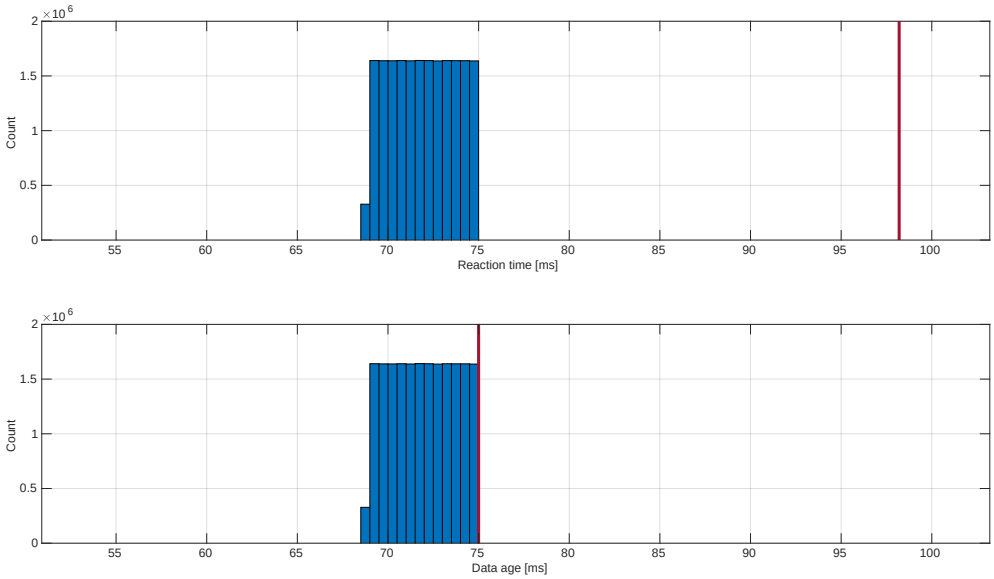


FIGURE 4.9: *Reaction time* and *data age* of the chain {Camera, Detection, Fusion}, or $\{\tau_5, \tau_3, \tau_4\}$, evaluated in simulation with the red lines showing the calculated maximum.

	Forget [52]	Saidi [101]	Becker [17]	Verucchi [122]
schedulable task set(%)	46.9	21.8	90.5[6]	**90.5**
1st lowest data age (%)	45.89	17.85	77.81	**96.82**
2nd lowest data age (%)	2.79	0.09	13.55	3.15
3rd lowest data age (%)	3.03	1.46	6.38	0.03
4th lowest data age (%)	0.00	4.58	2.25	0.00

TABLE 4.5: Schedulability and *data age* results on 1000 task set compliant to Kramer et al. [68] of 5 tasks and 15 chains, with utilization equals to 1.5.

al. [17], and tested them on the previously presented automotive benchmark by BOSCH. Table 4.5 shows the results for the 1000 task sets considered, and all the 15000 task chains, while Table 4.6 offers a comparison of the running times of the considered methods for larger task sets, i.e., composed of 10 tasks with 15 task chains.

The proposed method not only dominates the others in term of schedulability, but also in terms of *data age*. Given that Forget et al. [52] and Saidi et al. [101] do not propose a method to compute end-to-end latency, we adopted our algorithm for this scope. Considering *data age*, our method produces a DAG that leads to the lowest end-to-end latency bound in 96.82% of cases. There are some cases in which Becker et al. [17] method obtains a tighter latency, since it optimizes a single chain. However, the limitation of that approach is that it is not able to optimize all the given chains for a task set, while our method optimizes them all. Therefore, we are willing to sacrifice the latency of some chains for a more balanced improvement of all chains.

On the other hand, when optimizing a single chain, our method allows finding a better solution than with the method presented in Becker et al [17]. As an example, consider the task chain in Example 1. Using the method by Becker et al., a minimum *data age* of 40 can be achieved, inserting a precedence constraint between τ_1^0 and τ_2^0.

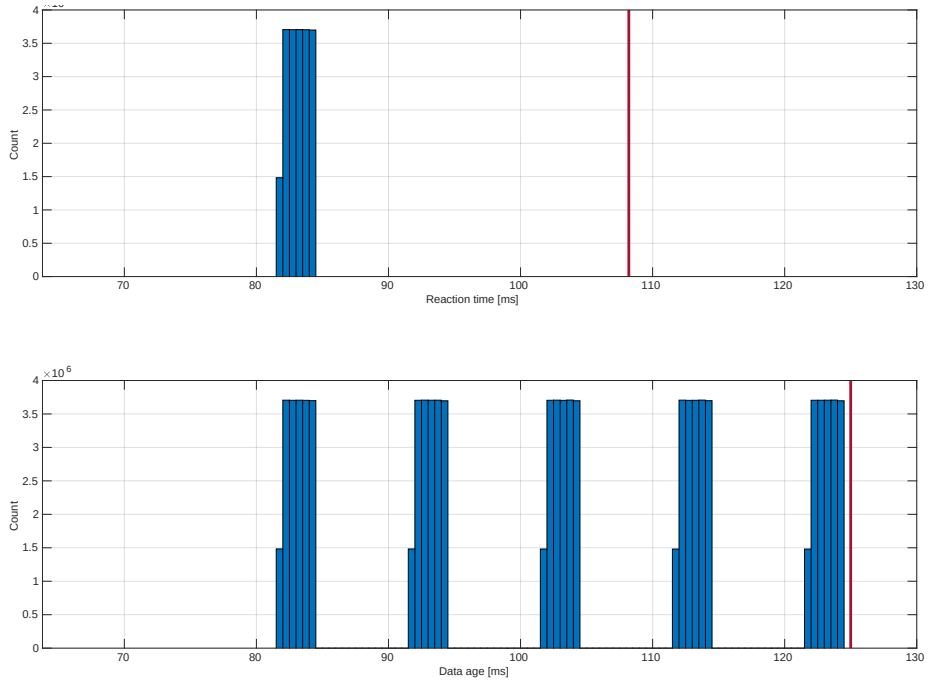


FIGURE 4.10: *Reaction time* and *data age* of the chain {Camera, Detection, Fusion, Planner, Control}, or $\{\tau_5, \tau_3, \tau_4, \tau_7, \tau_8\}$, evaluated in simulation with the red lines showing the calculated maximum.

	Forget [52]	Saidi [101]	Becker [17]	Verucchi [122]
min [ms]	0.002	0.002	0.078	0.002
avg [ms]	0.571	0.022	3.001	21.410
max [ms]	4.422	0.116	16.614	433.033

TABLE 4.6: Execution times in milliseconds on an Intel i7-7700HQ CPU @2.80GHz.

Instead, our method allows achieving a *data age* of 30, picking a DAG with additional precedence constraints.

As can be expected, the improved performance of the proposed algorithm is obtained by paying a somewhat higher computational cost. Table 4.6 shows that our method is on average about 7 times slower than Becker et al. [17]. We believe such a slow-down is acceptable for an offline analysis performed at system design time, as it allows obtaining the best solution for even complex task systems within a reasonable time.

Chapter 5

Real-World Real-Time Applications

With respect to the previous ones, this chapter is more application-oriented. In the following, three real-world applications of industry 4.0, smart city, and self-driving cars are described.

This chapter offers the missing link between the theory and the modern real-time applications. Indeed, the three presented applications are compliant with the definition of modern real applications given in Chapter 1: they have a considerable computational demand and high parallelism, due to the exploitation of neural networks; they work on many data, obtained from one or more sensors; they run onto heterogeneous embedded boards and they are subject to real-time constraints.

Working on those applications made it possible to derive the content of Chapter 4 and made us clear that the DAG task model needs extension such as the concepts of conditionality and heterogeneity. As a contribution to the real-time community, those three use cases are hereafter detailed and modeled as DAGs, and the code is made available so that can it be used as a reference for other researchers.

5.1 Industry 4.0: Defect Detection

Visual detection is a pervasively-used technique that consists of finding instances of semantic objects of some predefined categories within digital images or video streams. The problem, better known as Object Detection (OD), has indeed the objective to develop computational models and methods that provide to the computer the answer to a basic and needed question: "What objects are where?". In practice, it is the composition of two separate subtasks: (i) a classification problem and (ii) a regression problem. Given a user-defined set of categories $\mathcal{C}$ and a frame, for each object in the frame a probability of belonging to one of the $\mathcal{C}$ classes should be predicted; as well as the location of that object in the frame, in terms of bounding box (BB).

Autonomous driving and Smart IoT (Internet of Things) systems for transportation and urban surveillance are typically required to detect road users, such as other cars, bicycles, and pedestrians [19, 112]; and there are many other application domains, ranging from robotics [92], to avionics [65], where different vision tasks are required upon object detection, such as segmentation, image captioning, or object tracking.

In the context of industrial automation applications, object detection may be applied to detect items to be manipulated, or defects to be signalled [71]. The latter case was the subject of an investigation carried out in a project in collaboration with Tetra Pak. The goal of the project was not only to identify the best method to detect

defects in their products but also to point out the most appropriate board to adopt for that task.

5.1.1 Object Detection Deep Neural Networks

Since 2012, Deep Neural Networks (DNNs) have surpassed the accuracy of classical methods, becoming the state-of-the-art technology for vision task [137]. Nowadays, there exists plenty of DNNs that tackle the problem of object detection, and more and more are proposed every new month.

Conflicting Goals Selecting the best solution out of all the existing ones is surely a nontrivial task. Not only this field is evolving fast and methods become quickly deprecated, but one should be also aware of conflicting goals when designing a solution:

i *Performance* - A good object detector is characterized by two main performance figures: *latency*, i.e. the time needed for a single frame to be processed, and *accuracy*, i.e. the quality of the output given the input. These represent a well-known bi-dimensional trade-off: improving one often worsens the other.

ii *Power* - When targeting embedded platforms, one-stage detectors (which performs classification and localization together) are preferable to two-stage ones (which split the classification and localization tasks) because of their faster performance, which leaves room for further functionalities on the limited resources of the computing platform. Some of these constraints, like the platform's physical characteristics or price, are independent from performance indicators like the ones above. The *power absorption*, instead, directly correlates with the attainable precision and latency, hence compelling the system design to a third trade-off dimension.

Strategies to Optimise Inference Regarding fast inference on-device computation, three major axes have been identified [40] to maximize performances or contain power consumption.

i *Network model design* - Reducing the number of parameters in the DNN model is a very common approach to reduce memory and execution latency while aiming at preserving high accuracy. Some examples include MobileNets [62], Single-Shot Detectors (SSD) [77], Yolo [98], and SqueezeNet [63], with a rapidly evolving state of the art.

ii *Model Compression* - DNN models can also be compressed, sacrificing accuracy compared with the original model, to achieve faster performance. There are several popular model compression methods, like parameter quantization, parameter pruning, and knowledge distillation. In our work, we adopted quantization, such as half floating-point precision (FP16) or INT8 inference, as it is a popular compression method supported on many modern embedded platforms.

iii *Platforms* - Classic x86 CPU architectures have long been dominating the high-end segment of the industrial automation domain for central control systems, as they offer the best sequential performances (throughput and response time) and easiest programmability. General-Purpose computing on Graphics Processing

Units (GPGPUs) provide order-of-magnitude improvements in parallel throughput and in power usage, at a reasonable programming cost. FPGA platforms share a similar ambition, requiring a higher programming cost, but providing a more flexible communication paradigm with simpler computational units.

Contributions We propose a platform-optimized setup for hosting OD-DNN workload for 3 hardware families and 6 platforms: an industrial PC (Intel i7-7700), NVIDIA (TX2, Xavier AGX, and Nano), and Xilinx (Zynq Ultrascale+ ZCU102 and ZCU104), with frameworks ONNX Runtime [82], tkDNN with TensorRT[1] and DNNDK v3.0[2] framework and the Vitis-AI[3] respectively. Details on the boards and relative frameworks can be found in Table 5.1.

TABLE 5.1: Details of the considered boards. N.U. stands for Not Used for the implementation.

	Xilinx XCZU7EV	**Xilinx XCZU9EG**
CPU	Arm Cortex-A53 (v8) 4 cores @1.2GHz	Arm Cortex-A53 (v8) 4 cores @1.2GHz
GPU	Mali-400 [N.U]	Mali-400 [N.U]
Memory	2 GiB DDR4 64-bit SODIMM w/ ECC	4 GiB DDR4 64-bit SODIMM w/ ECC
Power	≈25 W	≈30 W
Board	Zynq UltraScale+ ZCU104	Zynq UltraScale+ ZCU102
DNN Accelerators	2× DPUv1.4@250MHz	3× DPUv1.4@330MHz
Data types	INT8	INT8
Operating system	Debian Buster 10.0	Debian Buster 10.0
Framework used	DNNDK v3.0	DNNDK v3.0
Module price	≈$1,000	≈$900
Release year	2014	2014
	Intel i7-7700	**Nvidia Jetson Nano**
CPU	Intel i7-7700 4 cores @3.60GHz	128-core Maxwell @ 921 MHz
GPU	–	Mali-400 [N.U]
Memory	16 GiB RAM	4 GiB LPDDR4, 25.6 GiB/s
Power	65 W	5W / 10W
Board	Industrial PC	Jetson Nano
DNN Accelerators	–	–
Data types	FP32	FP32, FP16
Operating system	Windows 10 Enterprise LTSC 1809	Ubuntu 18.04.4 LTS, Jetpack 4.4
Framework used	ONNX Runtime	tkDNN with TensorRT
Module price	≈$500	≈$150
Release year	2017	2019
	Nvidia Jetson TX2	**Nvidia Jetson Xavier AGX**
CPU	4-core Arm Cortex-A57 @ 2 GHz, 2-core Denver2 @ 2 GHz	8-core Arm Carmel v.8.2 @ 2.26 GHz
GPU	256-core Pascal @ 1.3 GHz	512-core Volta @ 1.37 GHz
Memory	8 GiB 128-bit LPDDR4, 58.3 GiB/s	16 GiB 256-bit LPDDR4, 137 GiB/s
Tensor cores	–	64
Power	7.5W / 15W	10W / 15W / 30W
Board	Jetson TX2	Jetson Xavier AGX
DNN Accelerators	–	2× Deep Learning Accelerators [N.U.]
Data Types	FP32, FP16	FP32, FP16, INT8
Operating system	Ubuntu 18.04.2 LTS, Jetpack 4.4	Ubuntu 18.04.3 LTS, Jetpack 4.3
Framework used	tkDNN with TensorRT	tkDNN with TensorRT
Module price	≈$400	≈$1,000
Release year	2017	2018

[1] https://developer.nvidia.com/tensorrt

[2] https://www.xilinx.com/products/design-tools/ai-inference/edge-ai-platform.html#dnndk

[3] https://github.com/Xilinx/Vitis-AI

In our work, we have considered the embedded networks that, at the time of writing, are the best-performing ones along with the considered metrics, i.e., YOLOv3 and YOLOv3-tiny [97], YOLOv4 and YOLOv4-tiny [24], Mobilenetv2-SSDLite [104], Centernet-Resnet101 and Centernet-DLA34 [136]. We have re-implemented the OD-DNNs to perform at best on the chosen hardware, with free and open-source access to the code for NVIDIA[4] and Xilinx platforms[5].

We have delivered a systematic and fair comparison of the 7 OD-DNNs, in terms of mean Average Precision (mAP), latency, throughput, power consumption and cost on the 6 embedded boards. The setup is uniform on input size, training dataset, threshold for bounding boxes' (BBs) confidence, while the platform setup and OD-DNN implementation are designed to be the best available.

5.1.2 Lessons Learned

The complete results of this project can be found in the paper [121]. Hereafter, some insights and lessons learned relevant to this thesis are reported.

A fair comparison requires effort To properly compare ODCNNs performances, network configurations must be as uniform as possible. It would be unfair to compare, e.g., the latency if input sizes are different, or the accuracy when input image resolution used for training is different. Hence, we decided to choose one of the most widely used datasets in object detection, i.e. COCO [72]. Input size and training set were chosen after the newer networks (CenterNet, YOLOv4), using COCO 2017 with input size 512x512. This input size allows discriminating more clearly the differences between the latencies, while achieving good accuracy.

For CenterNet and YOLOv4 networks, we used the weights from the SOTA, already fitting our requirements. On the other hand, we trained YOLOv3, YOLOv3-tiny and MobileNetV2-SSDLite, performing a single, full-precision training per network, and then exporting the obtained weights for the different frameworks.

End-to-end latency is not inference latency For each of the considered OD-DNNs, the execution time can be divided in: (i) pre-processing to convert the image in the NN input, (ii) NN inference, (iii) post-processing to convert the output of a NN into BBs. The end-to-end (e2e) latency is the time elapsed between feeding an image to the detector, and obtaining the BBs.

In each board, the pre-processing and the post-processing is performed in full-precision (FP32), while the inference can be quantized (FP16 or INT8). Pre-/post-processing are performed on the CPU, except for NVIDIA boards, where pre-processing of every network and post-processing of CNet(D34) and CNet(R101) have been optimized on GPU, implementing a CUDA version of the corresponding (slower) OpenCV functions.

We performed the tests on 5k images, and we computed maximum, minimum and average e2e latency over those 5k latency records.

Pre-processing, inference, and post-processing latency are separated in Figure 5.1 using the worst-case. On Intel and NVIDIA platforms, the longest phase is clearly the inference. For the Xilinx boards, instead, pre- and post-processing are longer. This can be firstly ascribed to the A53 being the slowest performing CPU of the considered platforms. Furthermore, the load of the post-processing phase has been

[4] https://github.com/ceccocats/tkDNN
[5] https://git.hipert.unimore.it/gbrilli/dpunn

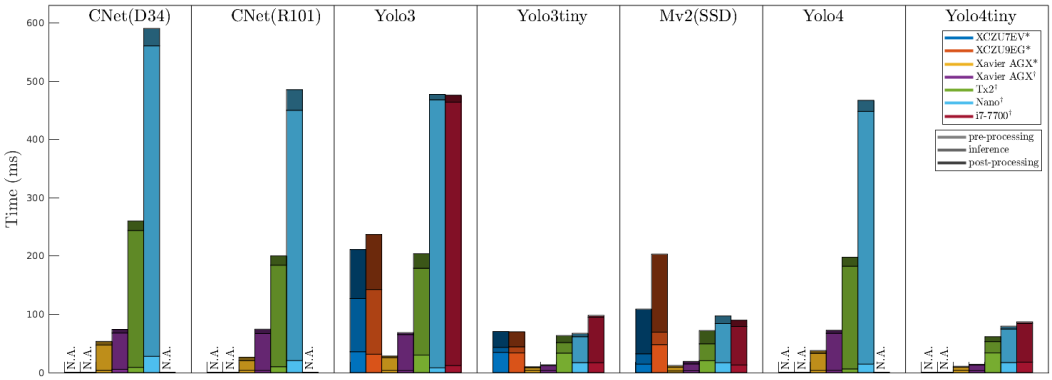


FIGURE 5.1: Worst-case execution time divided in pre-processing (normal), inference (dark) and post-processing (darker) w.r.t. the e2e latency. ∗ stands for INT8, † for FP32.

augmented so to include the final normalization steps of the NN, which could not be executed on the DPU because of missing API (i.e. sigmoid for Yolo3 and Yolo3tiny) or unacceptably slow implementation (i.e. softmax for Mv2(SSD).

There is not a single winner All the tests have been performed on the COCO 2017 validation tests, counting 5K images. For each test the following common metrics have been considered: (i) *mAP 0.5:0.95*, being the "de facto" metric for object detection [100] [76], which tells how good a method is; (ii) best-, average- and worst-case *latency* of the processing of a frame, in ms; (ii) *efficiency* as Frame Per Seconds (FPS) over the power consumption (W). The power usage has been sampled at 40 Hz on the NVIDIA and Xilinx boards using *powerapp* tool[6], at 1Hz on the i7-7700 using Open Hardware Monitor 0.9.2.

For the NVIDIA platforms, all supported data types were considered: FP32, FP16 for TX2, Xavier AGX and Nano, INT8 for Xavier AGX only. XCZU9EG and XCZU7EV supported only INT8, and only FP32 is supported by the i7-7700. The INT8 quantizations have been obtained on 1000 images of the COCO2017 training set, both on Xilinx and NVIDIA boards.

The collected results of the analysis have been reported in Fig. 5.2, having the trade-off between mAP-latency above and mAP-efficiency below, where grey, dashed lines suggests Pareto-optimality curves for each board. For the former trade-off, the best performance can be found on the top-left corner (i.e. high mAP and low latency); for the latter the best can be found in the top-right (i.e. high mAP and high efficiency).

Some clear conclusions can be made. Regarding networks, YOLOv4 is the one achieving the highest accuracy, while YOLOv3-tiny is the best network in terms of latency and power consumption, but also the one achieving the smallest mAP. YOLOv3 is dominated by other models and it is the most power greedy. About platforms, Xavier AGX is the clear winner in almost all considered aspects, achieving the best power efficiency, as well as the highest mAP. Among the NVIDIA boards, the AGX Xavier dominates the TX2, which dominates the Jetson Nano. The Xilinx platforms have a very stable power consumption for all considered networks, and dominate the i7-7700 in terms of efficiency and inference latency. The i7-7700 is the least efficient board, but it is also the one with better sequential performance,

[6] https://git.hipert.unimore.it/tetra-pak/dl-arch/powerapp

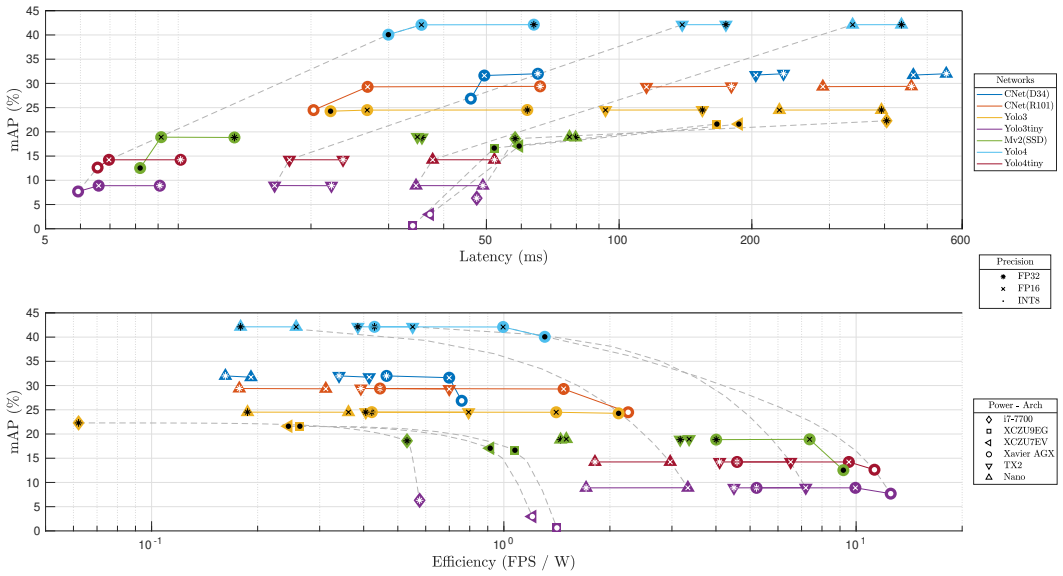


FIGURE 5.2: Comparison of the different networks on the selected platforms considering average-case latency and power.

leading to a smaller post-processing latency variance. Moreover, it has comparable performance w.r.t the Jetson Nano.

However, there is not a single winner among the models, nor among the platforms. It all depends on the requirements and constraints of the applications: the budget for the platform, the throughput required, the maximum power consumption allowed and so on. The terrific complexity from numerous degrees of freedom should be therefore guided from the requirements of the final application.

5.1.3 Defect Detection as a DAG

Once a model is chosen an application for defect detection can be developed. The DNN should be trained on the specific sets of defects to be detected, but this should be made just once.

The application takes as input a video stream, of a camera, and the detection task is applied frame by frame. The heterogeneous conditional DAG of the task is depicted in Figure 5.3.

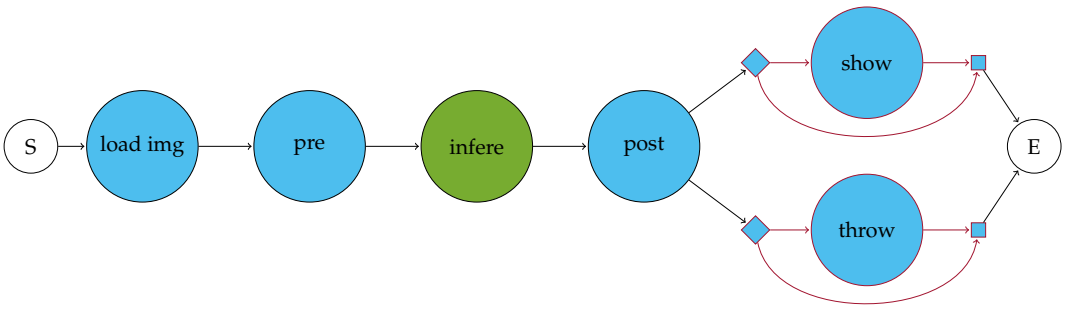


FIGURE 5.3: Defect detection task modeled as a HC-DAG

At first, the frame is loaded by subtask *load img*, then object detection pre-processing is applied by *pre*. The data is then offloaded onto the GPU and the inference is computed (*infere*). The results of the inference are then copied back to the CPU and

post-processed by *post*. Finally, there are two conditional constructs: if the product presents defects, then a message is sent to throw away the current item (*throw*); if the visualization is enabled, the task also shows on screen the detected BBs (*show*).

5.2 Smart City: CLASS

In the context of smart cities, the use of combined data-in-motion and data-at-rest analysis provides efficient methods to exploit the massive amount of data generated from heterogeneous and geographically distributed sources including pedestrians, traffic, (autonomous) connected vehicles, city infrastructures, buildings, IoT devices, etc. Certainly, exposing city information to a dynamic, distributed, powerful, scalable, and user-friendly big data system is expected to enable the implementation of a wide range of new services and opportunities provided by analytics tools. However, there exist several challenges, not only related to size and heterogeneity of data but also from its geographical dispersion, making it difficult to be properly and efficiently combined, analyzed, and consumed by a single system.

The CLASS project [44], funded by the European Union's Horizon 2020 Programme, faces these challenges and proposes a novel software platform that aims to facilitate the design of advanced big-data analytics workflows, incorporating data-in-motion and data-at-rest analytics methods, and efficiently collect, store and process vast amounts of geographically-distributed data sources.

The CLASS software stack covers the whole compute continuum, from edge to cloud, and relies on a well-organized distributed infrastructure. The final goal is not only to develop a smart city framework, but also to implement four smart city applications, i.e. (i) digital traffic sign, (ii) smart parking, (iii) air pollution estimation, and (iv) obstacle detection and tracking, with a real use-case in the Modena Automotive Smart Area (MASA). MASA is a 1 Km^2 area in the city of Modena (Italy), equipped with a sensing, communication, and computation infrastructure.

The University of Modena and Reggio Emilia is one of the partners of the class CLASS consortium, in charge of the edge software stack, with edges being both pole-mounted cameras and smart (or connected) vehicles. Hereafter, the focus will be on the application that runs on the camera streams to compute obstacle detection and tracking.

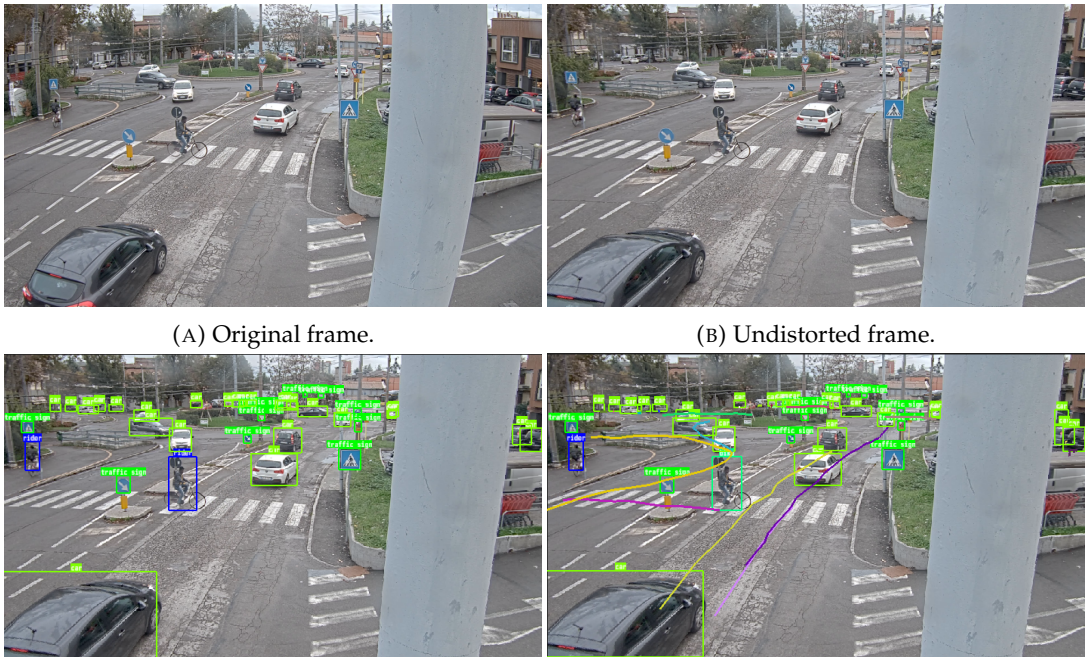
5.2.1 Obstacle Detection and Tracking

From cameras located in the streets, objects can be detected, classified, and tracked. All the cameras belonging to the infrastructure are supposed to perform those tasks in real-time and send the extrapolated information to a data aggregator. This aggregator will then de-duplicate repeated information and send messages to the connected cars, which will receive only objects that are relevant to their surroundings.

Geo-localization To perform the above operations, it is necessary to know the real-world position of each object detected from the cameras. To do so, an extrinsic calibration of each camera in the MASA has been performed, in order to have a mapping for each pixel in the camera frame to its GPS position on a geo-referenced map.

Real-time requirements Data is *constantly* being produced and processed and it is extremely important to guarantee that the results are meaningful by the time they are computed. This is especially relevant for the *Obstacle Detection and Tracking* use case since alerts must raise within a time interval that is useful for the driver to react. A reasonable metric, considered in the scope of the CLASS project, is to get updated results at a rate between 10 and 100 milliseconds. Assuming that the maximum speed of a vehicle within the city is 60 km/h, vehicles will advance between 0.17

and 1.7 meters. This level of granularity is enough to implement the proposed use-cases.



(A) Original frame. (B) Undistorted frame.

(C) Object detection. (D) Object tracking.

FIGURE 5.4: Results on various steps of the application.

The flow of the application We have implemented an application called `class-edge` and we have released the code open source[7].

`class-edge` takes in input *N_streams* streams and for each of them perform four main steps:

- First, frames are retrieved via the Real-Time Streaming Protocol (RTSP). Given that the image is taken from a camera, it is very likely that it is affected by distortion. To correct that, undistortion, based on the intrinsic calibration of the camera, is applied. Figure 5.4a shows an original frame while Figure 5.4b shows the output of undistortion.
- Object detection is then performed on the undistorted image. For this projects we picked the tkDNN implementation of Yolov4 [24] (already introduced in Section 5.1). As already mentioned, this task is divided into three parts, i.e. pre-processing, inference, and post-processing. An example is given by Figure 5.4c.
- The detection gives in output a list of bounding boxes (BBs). For each of them, a single point is picked to represent the whole object, namely the center of the bottom side of the BB. This pixel is converted first, into a GPS position, and then in meters. This is the format required from the next step: the tracker. Indeed, to track and predict the position of the detected objects an Extended Kalman Filter (EKF) [66] on the real-world position of the object has been applied. Objects between frames are matched together only if the class corresponds and their distance is under a user-defined threshold. Tracking is not

[7] https://github.com/mive93/class-edge

only used to have a more robust detection, but also to have a history of the objects. The idea of history is given by the lines in Figure 5.4d.

- Finally the information can be sent both to the data aggregator, in an anonymized form, and to the optional viewer.

5.2.2 `class-edge` as a DAG

Let us now pick $N\_streams = 2$. The heterogeneous conditional DAG of the task is depicted in Figure 5.5, and its implicit deadline is $D = 100ms$.

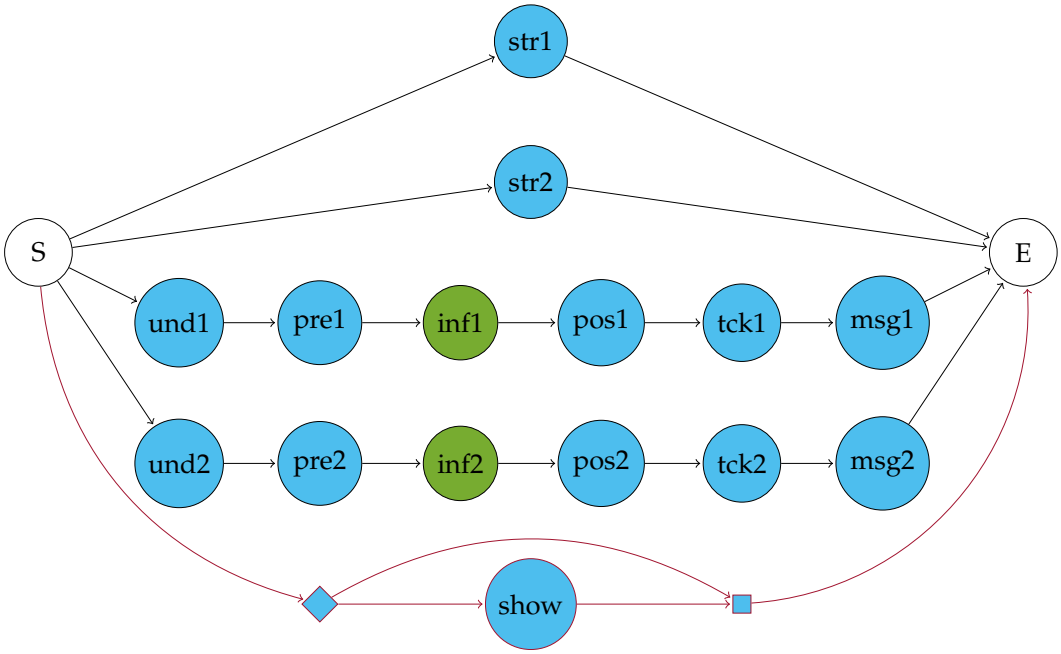


FIGURE 5.5: class-edge represented as a HC-DAG.

The process is actually composed of 5 threads: 2 threads, one for stream, that retrieve the stream ad store the image (*str1* | 2); 2 threads that execute the flow of undistortion (*und1* | 2), detection (*pre1* | 2, *inf1* | 2, *pos1* | 2), tracking (*tck1* | 2) and message sending (*msg1* | 2); and an optional thread for visualization purposes (*show*).

The reason behind this split is to control the end-to-end latency of the application. Having all the steps in sequential order would cause the missing of the 100 ms deadline while having high data age and reaction time latencies. Indeed, the first bottleneck of this application is to retrieve and read the frame from the stream, especially because the minimum resolution of the considered stream in our infrastructure is HD (1920x1080). The second bottleneck is the complete undistort-detect-track-send flow. Having these two operations in sequence leads to always working on old data, while separating them improves the performance of the system.

To better understand the problem, some experimental results are reported. The experiments have been carried on an Intel i9-9900KF (@3.60GHz) coupled with an NVIDIA RTX 2080Ti. Two streams were considered: (i) stream [1] with an image resolution of 1920x1080 and rate of 25 FPS, (ii) stream [2] with image resolution 3072x1728 and rate of 30 FPS. A better idea of the timing and the data exchange of the application is depicted in Figure 5.6.

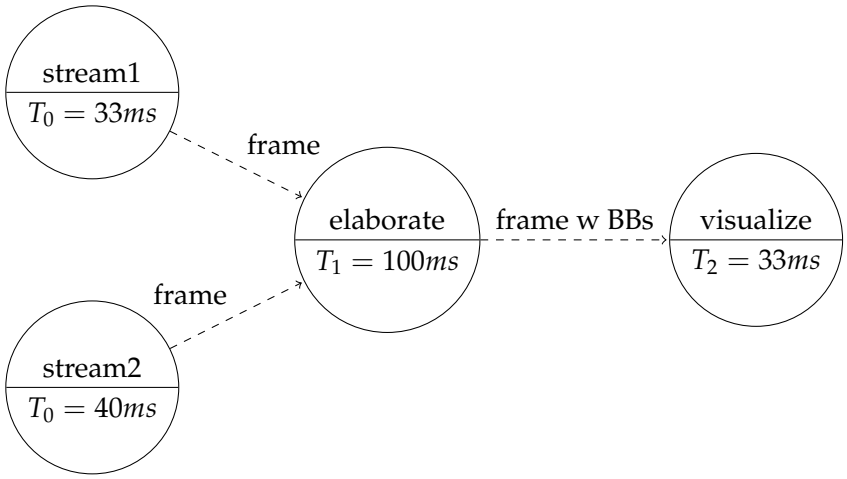


FIGURE 5.6: `class-edge` timing and data exchange details.

Figure 5.7 reports the minimum, average and maximum latencies of subtask str1|2 (Figure 5.5) over 5k frames, split in three further phases: (i)frame acquisition, form the RTSP stream, (ii) frame resize to a smaller resolution (i.e. 960x540) and frame copy to a shared buffer. From the chart, two main observations can be made. The first is that the most expensive part is the capture of the stream, which is affected by the original resolution and it's higher for stream [2]. The second is that, even if in average str1|2 take between 15 and 20 milliseconds to execute, the same operation can take up to 160 milliseconds. Given these results, we decided to set 1920x1080 as an upper bound resolution for the RTSP stream, changing the setting directly on the cameras, so that the maximum execution is always less than 100 milliseconds.

A similar chart for the undistort-detect-track-send flow is reported in Figure 5.8. Additionally to the already described phases, the copy of the frame from the shared buffer and the optional viewer feeding have been profiled. From this chart, we can evince that still there are some differences in the streams, not related to the resolution but on the scene itself instead. Stream [1] comes from a camera that points on a roundabout: it's a dynamic scenario and even though there are many objects, the trackers don't last long. On the other hand, stream [2] comes from a camera that points to a parking lot: it's a static view, there are many objects and their trackers are always alive, keeping their information (with a limited history). For this reason, differences in terms of latency can be found in the tracking and viewer phases, which are proportional to the stored trajectories, while the other phases have similar duration. In any case, the total time of the flow is always less than 100 milliseconds.

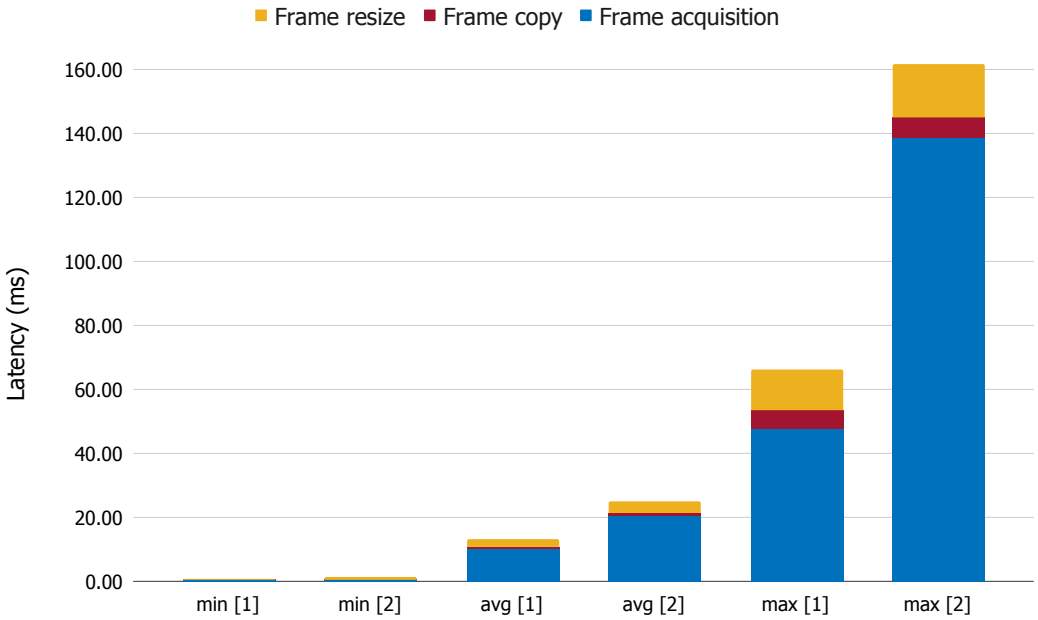


FIGURE 5.7: Minimum, average and maximum latencies of subtask str1|2 of Figure 5.5, split in three further phases: frame acquisition, frame resize and frame copy.

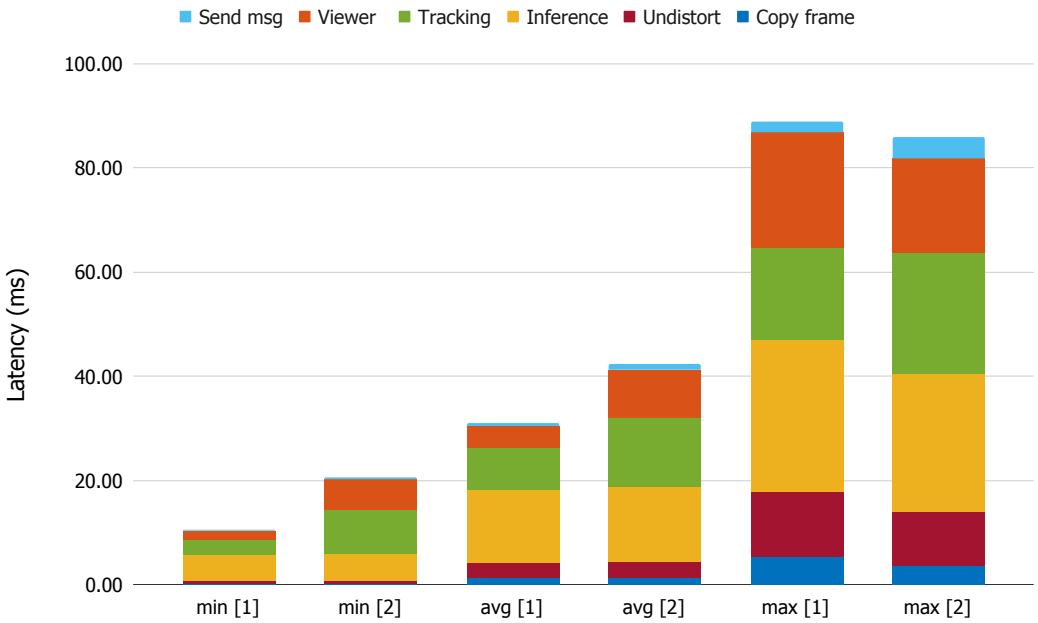


FIGURE 5.8: Minimum, average and maximum latencies of undistort-detect-track-send flow.

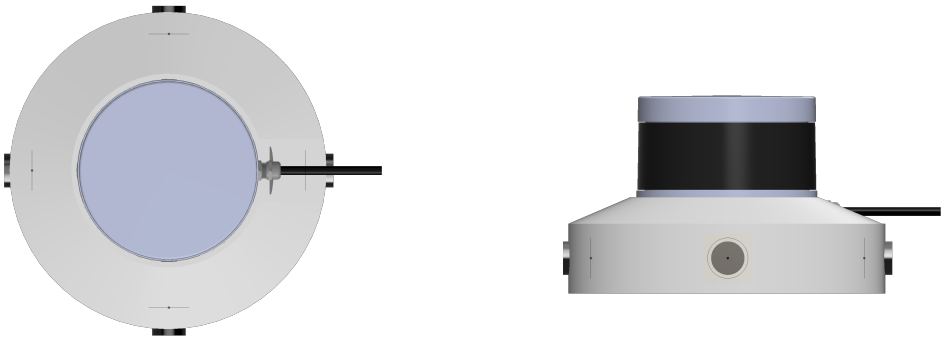
5.3 Automotive: Sensor Fusion

3D object detection and classification are crucial tasks for perception in Autonomous Driving (AD). To promptly and correctly react to environmental changes and avoid hazards, it is of paramount importance to perform those operations with high accuracy and in real-time. One of the most widely adopted strategies to improve detection precision is to fuse information from different sensors, like e.g. cameras and LiDAR. However, sensor fusion is a computationally intensive task, that may be difficult to execute in real-time on embedded platforms.

Currently, cameras and laser scanners (LiDARs) are the most used sensors for detection. Cameras are used to obtain textures and colors of targets, which are valuable features for classification of e.g. traffic lights and traffic signs. However, cameras have difficulties in obtaining depth information and are greatly affected by lighting conditions. On the other hand, LiDARs can more easily acquire distance and three-dimensional information, even at long range. They are robust and not affected by illumination conditions, but they lack color information, complicating the classification task. Therefore, fusing data from the two mentioned sensors has become a trend in the AD field, being a good compromise to obtain good classification and 3D detection. The main problem of this approach is its high computational cost, which prevents its adoption in real-time sensitive scenarios.

5.3.1 Online Clustering and LiDAR-Camera Fusion

We present a new approach for LiDAR and camera fusion, that can be suitable to execute within the tight timing requirements of an autonomous driving system. The proposed method is based on a new clustering algorithm developed for the LiDAR point cloud, a new technique for the alignment of the sensors, and an optimization of the YOLO-v3[97] neural network. Missing details can be found in the paper [123].



(A) Top view. (B) Side view.

FIGURE 5.9: Support for cameras and LiDAR.

Application settings The self-driving prototype used for this work mounts a Velodyne ULTRA Puck VLP-32C LiDAR and 4 Sekonix cameras SF332X-10X with 120° FOV mounted on the support shown in Fig. 5.9. The LiDAR is located in the center of the support, surrounded by the four synchronized cameras covering 360°. The support is then placed on the roof of the car.

The adopted computing platform is an NVIDIA AGX Xavier, whose specifics can be found in Table 5.1.

LiDAR LiDARs produce a 3D point cloud, which is processed to place 3D bounding boxes (BBs) around the objects via clustering methods. Several types of clustering techniques exist[127, 99]: centroid-based, distribution-based, hierarchical and density-based clustering. This latter category is the most suitable for processing LiDAR point clouds. The most widely adopted algorithm in this class is DBSCAN (Density-Based Spatial Clustering of Applications with Noise)[46].

To improve existing LiDAR clustering methods for achieving real-time performance on the considered embedded computing platforms, three expedients have been adopted: (i) light cloud preprocessing; (ii)parallelization; and (iii) limited neighbor exploration.

The preprocessing task aims at removing from the sampled LiDAR point cloud all points corresponding to the ground or higher than the expected height of the objects to be detected. This allows significantly decreasing the computational complexity of later clustering steps. Additionally, the height dimension is dropped and the clustering algorithm is performed only on two dimensions, i.e. latitude and longitude. This choice follows by assuming objects are never placed one on top of another.

The search of the neighboring points is then parallelized on a GPU, creating a thread for each point. Moreover, the exploration is bounded to a user-defined threshold, checking at most $2W$ points for each object in the cloud. This is a reasonable choice for a LiDAR sensor because of the nature of the output data, where cloud points are semi-ordered by horizontal angle. Exploiting this feature, the search for neighboring points can be limited to adjacent angles. The proposed Window-Based Lidar Clustering (WBLC) receives a 2D point cloud as input, and produces the identified clusters as output. The algorithm is divided into two parts: the searching for neighbors and the merging of clusters.

Cameras From camera frames, objects are detected and classified with tkDNN version YOLO-v3[97] (already introduced in Section 5.1)). Moreover, training from scratch has been performed on the BDD100K Berkeley Dataset[131] (with 10 classes) to better recognize road objects.

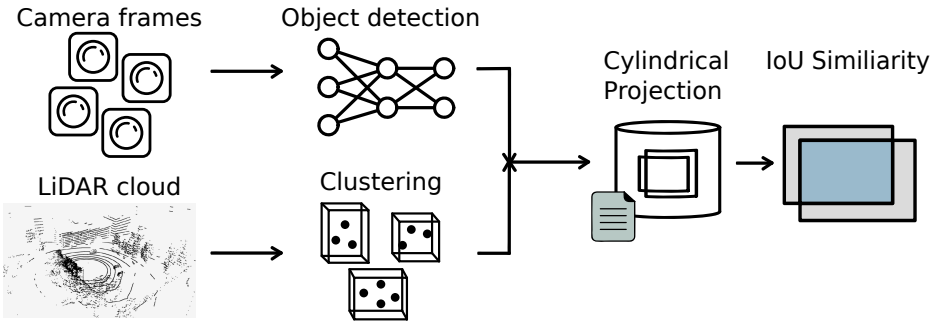


FIGURE 5.10: Fusion scheme.

Fusion To perform sensor fusion a new method to align cameras and LiDAR has been proposed. The alignment of the sensor is an offline procedure that has to be executed once, following the initial intrinsic calibration of the camera. It is inspired

by the method proposed by Velas et al.[119], which adopts a green flat panel with four holes, however, instead of projecting the LiDAR cloud on the camera's plane, it projects the cameras in the LiDAR cylindrical representation. It consists of five phases (i) feature extraction from both sensors, (ii) cylindrical projection of cameras (iii) cylinders alignment (iv) evaluation, and (v) correction. Once this procedure is concluded, a calibration file is produced.

Camera frame BBs and clusters given by WBLC are projected thanks to the pre-computed calibration file and sequentially merged. The Intersection over Union (IoU) metric is adopted to check if a BB and a cluster match the same object. If the IoU is greater than a user-defined threshold, they are deemed to represent the same object. In this case, clusters are labeled using the corresponding BDD100K class (e.g., pedestrian, car, bus, traffic light, etc). Clusters that do not match any BB (e.g., trees, bins, etc.) are labeled with the "unknown" class.

A complete overview of the fusion approach is depicted in Figure 5.10, while Table 5.2 reports the results of the tested application on the AGX Xavier in terms of minimum, average and maximum execution time over 5k frames.

	Clustering	Detection	Fusion
min (ms)	1,98	26,07	0,18
avg (ms)	2,96	28,86	0,57
max (ms)	7,44	38,50	1,20

TABLE 5.2: Execution times on the AGX Xavier, statistics computed over 5K frames.

5.3.2 Sensor Fusion as a DAG

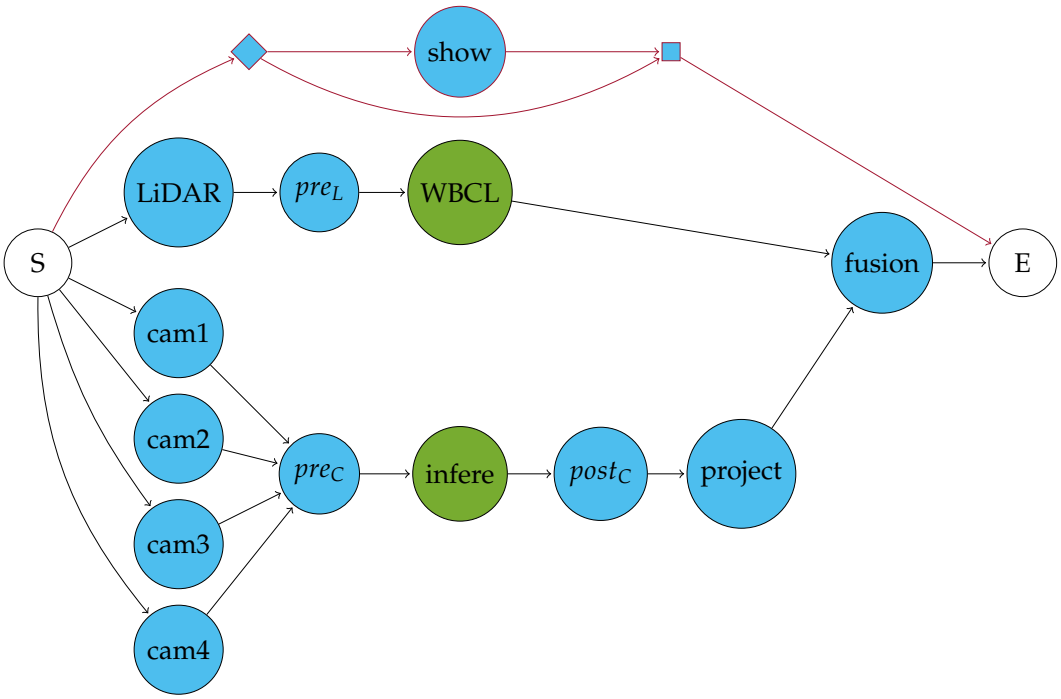


FIGURE 5.11: Sensor fusion represented as a HC-DAG.

This application can be seen as a sporadic DAG task with $T = 100ms$, and it is depicted as so in Figure 5.11. The DAG task is both heterogeneous, because multi-processor CPU and GPU are used, and conditional, given that the visualization is optional. To better understand the flow:

- the subtask *LiDAR* retrieves the point cloud from the sensor, that is then pre-processed by subtask pre_L and then offloaded onto the GPU to permorm the *WBCL* subtask.
- there are four subtask (*cam1* to *cam4*) that collect the frame from the four different cameras. Then the subtask pre_C pre-process them to obtain the format required by YOLO-v3. The pre-processed frames are offloaded onto the GPU and the *inference* is computed. Finally the subtask $post_C$ applies the post-processing and *project* projects the 2D BBs into the cylindrical representation.
- the subtask *fusion* fuses the output of the clustering and the detection, once the previous subtasks have completed.
- the subtask *show* is in charge of the visualization and it is optional.

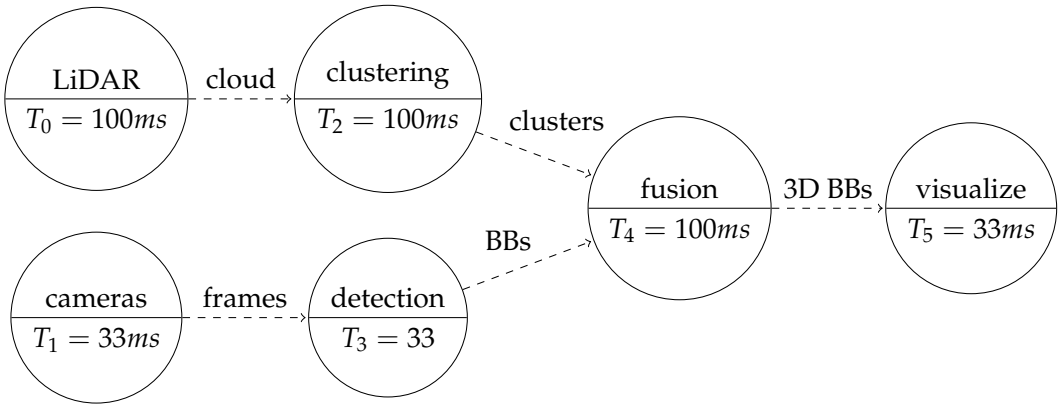


FIGURE 5.12: Sensor fusion timing and data exchange details.

Application as a Multi-Rate Task set This same application could also be modeled as a multi-rate task set, which is depicted in Figure 5.12. The periods of the sensors are given, $T_0 = 100ms$ for the LiDAR and $T_1 = 33ms$ for the cameras; while the period of the visualization is bounded to $T_5 = 33ms$. The task *clustering* and *detection* inherit the period from the LiDAR and cameras respectively, but the fusion period needs to be $T_4 = 100ms$ in order to have the data from the LiDAR.

Chapter 6

Conclusions and Open Problems

6.1 Conclusions

As introduced in Chapter 1, in the context of real-time system on new-generation heterogeneous embedded boards, an effort was necessary towards proper models for modern real-time applications. This was the focus of this thesis, carried out in the three central chapters, that represent separate but synergistic efforts in that direction.

Chapter 3 reported a well-organized survey of the real-time DAG literature, analyzing different DAG models (i.e. DAG, C-DAG, H-DAG, HC-DAG), for global, partitioned and federated scheduling, considering different scheduling algorithms (e.g. EDF, FTP), preemption policies (i.e. FP, LP, FNP) and deadlines types (i.e. constrained, implicit and arbitrary). Besides the survey, a library comprehensive of all the SOTA methods has been implemented which made a fair evaluation of the different methods possible.

Two goals have been accomplished. First, it emerged that the most suitable model for new-generation real-time system is the Heterogeneous Conditional-DAG, which comprises the concepts of parallelism, conditionality, and heterogeneity in a single model. Second, a collection point for a clear and fair comparison among methods related to the DAG task model and its extensions has been achieved, thanks to the joint efforts of translating each method into a common convention and implementing them in a single library.

The hope is that the real-time community will use this thesis as a point of reference, to start working on this subject and improve the state-of-the-art.

Chapter 4 focused instead on the problem of end-to-end latency of applications that contemplates input sensors (as most of all the robotics applications), multi-rate tasks, and data exchanges among them. To constrain both latencies and schedulability, a novel method has been proposed to convert these multi-rate task sets into a DAG that respects the given restrictions.

The introduced solution was proved to dominate the existing approaches, correctly compute data age and reaction time and it is right now the state-of-the-art to compute latencies on DAGs.

This method, and more in general the aspects evaluated in this chapter, should be carefully analyzed when designing real-time applications that involve sensors and actuators, in order to deploy predictable and safe software.

Chapter 5 introduced three real-world examples of real-time applications spacing from industry 4.0, to smart cities and self-driving cars. The detailed applications run

on new-generation embedded boards, make use of neural networks, and have been modeled via HC-DAGs.

The application point-of-view was needed to highlight the importance of both the HC-DAG task model and the end-to-end latency problem. This is what companies are looking for, what modern applications require, and what makes the contributions of the previous chapters worthy.

All the examples presented are correlated with the corresponding code, so that they can be exploited also by other researchers as real use-cases.

Open Code In the real-time community, unlike other communities, it is not common use to share the implementation of the proposed solutions and make the code available. In Chapter 3 more than 40 methods have been deeply analyzed. Among those, only three of them [81, 27, 86] have the code publicly available, and one [48] was privately shared[1]. Indeed, many papers compare their results with them, rather than the state of the art at the time of their writing, which could also be seen as unfair. The problem is that implementing solutions proposed by others is not trivial. The algorithms can be very complex and if some little, but important, details are missing it is easy to develop a method that is not exactly the same thought by the authors.

It is then of paramount importance for the research to have available implementations of the proposed and published solutions. Not only to be able to reproduce the results and compare to them but also to let other researchers start from there, improve what exists, and also find errors.

The aim of the researcher is to go towards innovation, together, and we, as a community, should facilitate others to start from our work and make it better. This is also the reason why in this thesis, every application was presented along with its corresponding code.

6.2 Open Problems

Working on the assessment of Chapter 3 and being involved in several projects like the ones presented in Chapter 5 made it clear that in the context of the real-time models, there are still many open problems. To correctly and fully address modern real-time applications, many further steps need to be done.

Exact tests An input requested from the real-time community is the development of exact tests for DAG task models. As shown in Section 3.1.5, the existing tests for DAGs are only a few. Moreover, there are no solutions for DAG task sets, conditional or heterogeneous DAGs.

It is known that the problem of schedulability of even one single DAG is NP-hard in the strong sense, therefore there is no way that polynomial algorithms will be produced (unless eventually it will be shown that P=NP).

However, having exact solutions even for small examples would be very important to understand the goodness of every sufficient test. Indeed, we can now compare methods and see if one dominates the other, but we are missing a ground-truth saying how far we are from the exact solution.

[1] to Casini et al. [35].

The Heterogeneous Conditional DAG As shown in Chapter 5, modern applications cannot be represented by the DAG task model only. The use of accelerators, additional engines, and the optional parts in the code of the tasks are aspects more and more frequent in real-world tasks. The DAG model is not expressive enough to represent their entirety.

The most suitable model is therefore the conditional heterogeneous DAG. However, as shown in the survey chapter, there is only one work [133] that focuses on that. Here again, the real-time community is called to contribute.

Memory, Energy, I/O and Bandwidth One of the major drawbacks of this thesis is that it does not contemplates crucial aspects as memory, energy consumption, input/output (I/O), and bandwidth. In modern embedded real-time systems those aspects can't be ignored anymore. Memory and bandwidth are shared among several heterogeneous computing engines, not on one but on many levels. Applications require high performance, high computation, and MiB (and even GiB) of data are moved from sensors, among engines, and to actuators. The models used to represent tasks and applications need then to be aware of those aspects.

However, in the literature very few works take into account, for example, memory [33, 34] or energy [56]. In hard real-time systems, especially, this can lead to serious problems. The interference caused by the competition on those shared resources has already been proven to be heavy and to seriously affect predictability.

Regarding memory, a solution to mitigate contention and control accesses was proposed by Pellizzoni et al [91] with a Predictable Execution Model (PREM) (also knows as the three-phase model). Each task in PREM is divided into three phases: (i) read, (ii) execute, and (iii) write. Thanks to that, the computation and the data fetching can be separately treated and memory accesses can be scheduled as well. A natural union with the DAG model, as already suggested [35, 133] or considered [33] in other works, would be to have the three phases for each node in the DAG. This extension would clearly complicate the schedulability analysis but would start solving the problem of memory contention.

Naturally, this solution introduces another huge problem which is not only the modeling of memory accesses but also the handling of the memory for real-time systems. Indeed, to enhance these new-generation embedded board with predictability, techniques such as cache partitioning need to be adopted and implemented. This is another decisive issue to be addressed in our community, as many researchers and start-ups like Minerva[2] are starting to do, exploiting the power of virtualization.

[2] http://minervasys.tech/

Bibliography

[1] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. "The transitive reduction of a directed graph". In: *SIAM Journal on Computing* 1.2 (1972), pp. 131–137.

[2] *AUTOSAR - Specification of Timing Extensions*. Tech. rep. 2014.

[3] Sanjoy Baruah. "Federated scheduling of sporadic DAG task systems". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2015, pp. 179–186.

[4] Sanjoy Baruah. "Improved multiprocessor global schedulability analysis of sporadic DAG task systems". In: *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. IEEE. 2014, pp. 97–105.

[5] Sanjoy Baruah. "Scheduling DAGs When Processor Assignments Are Specified". In: *Proceedings of the 28th International Conference on Real-Time Networks and Systems*. 2020, pp. 111–116.

[6] Sanjoy Baruah. "Techniques for multiprocessor global schedulability analysis". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. IEEE. 2007, pp. 119–128.

[7] Sanjoy Baruah. "The federated scheduling of constrained-deadline sporadic DAG task systems". In: *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2015, pp. 1323–1328.

[8] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Jan. 1, 2015. published.

[9] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. "The global EDF scheduling of systems of conditional sporadic DAG tasks". In: *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*. IEEE. 2015, pp. 222–231.

[10] Sanjoy Baruah and Alan Burns. "Sustainable scheduling analysis". In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE. 2006, pp. 159–168.

[11] Sanjoy Baruah et al. "A generalized parallel task model for recurrent real-time processes". In: *2012 IEEE 33rd Real-Time Systems Symposium (RTSS 2012)*. IEEE. 2012, pp. 63–72.

[12] Sanjoy K Baruah. "The non-preemptive scheduling of periodic tasks upon multiprocessors". In: *Real-Time Systems* 32.1-2 (2006), pp. 9–20.

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

[14] Sanjoy K. Baruah et al. "Generalized Multiframe Tasks". In: *Real-Time Systems* 17 (1999), pp. 5–22.

[15] Matthias Becker et al. "End-to-end timing analysis of cause-effect chains in automotive embedded systems". In: *Journal of Systems Architecture* 80 (2017), pp. 104–113.

[16] Matthias Becker et al. "Mechaniser-a timing analysis and synthesis tool for multi-rate effect chains with job-level dependencies". In: *7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems WATERS*. Vol. 16. 05. 2016.

[17] Matthias Becker et al. "Synthesizing job-level dependencies for automotive multi-rate effect chains". In: *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE. 2016, pp. 159–169.

[18] Marko Bertogna and Sanjoy Baruah. "Limited preemption edf scheduling of sporadic task systems". In: *IEEE Transactions on Industrial Informatics* 6.4 (Jan. 1, 2010), pp. 579–591. published.

[19] Marko Bertogna et al. "Adaptive coordination in autonomous driving: motivations and perspectives". In: *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE. 2017, pp. 15–17.

[20] Norman Biggs, Norman Linstead Biggs, and Biggs Norman. *Algebraic graph theory*. Vol. 67. Cambridge university press, 1993.

[21] Enrico Bini and Giorgio C Buttazzo. "Measuring the performance of schedulability tests". In: *Real-Time Systems* 30.1-2 (2005), pp. 129–154.

[22] Alessandro Biondi and Marco Di Natale. "Achieving Predictable Multicore Execution of Automotive Applications Using the LET Paradigm". In: *Proceedings of the 24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2018)*. Porto, Portugal, 2018.

[23] Konstantinos Bletsas. *Worst-case and best-case timing analysis for real-time embedded systems with limited parallelism*. Citeseer, 2007.

[24] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection". In: *arXiv preprint arXiv:2004.10934* (2020).

[25] Vincenzo Bonifaci et al. "Feasibility analysis in the sporadic dag task model". In: *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. IEEE. 2013, pp. 225–233.

[26] P Burgio, R Cavicchioli, and M Verucchi. "Real-Time Heterogeneous Platforms". In: *Heterogeneous Computing Architectures*. CRC Press, 2019, pp. 233–259.

[27] Artem Burmyakov, Enrico Bini, and Eduardo Tovar. "An exact schedulability test for global FP using state space pruning". In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. 2015, pp. 225–234.

[28] Alan Burns, Sanjoy K Baruah, et al. "Sustainability in Real-time Scheduling." In: *JCSE* 2.1 (2008), pp. 74–97.

[29] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. Springer US, 2004. ISBN: 9780387231372. URL: https://books.google.it/books?id=fpJAZM6FK2sC.

[30] Giorgio Buttazzo and Anton Cervin. "Comparative assessment and evaluation of jitter control methods". In: *Proceedings of the 15th conference on Real-Time and Network Systems*. 2007, pp. 163–172.

[31] Giorgio C. Buttazzo, Marko Bertogna, and Gang Yao. "Limited Preemptive Scheduling for Real-Time Systems. A Survey". In: *IEEE Transactions on Industrial Informatics* 9 (2013), pp. 3–15.

[32] Giorgio C. Buttazzo, Giuseppe Lipari, and Luca Abeni. "Elastic Task Model for Adaptive Rate Control". In: *RTSS*. 1998.

[33] Daniel Casini et al. "A Holistic Memory Contention Analysis for Parallel Real-Time Tasks under Partitioned Scheduling". In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2020, pp. 239–252.

[34] Daniel Casini et al. "Memory feasibility analysis of parallel tasks running on scratchpad-based architectures". In: *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2018, pp. 312–324.

[35] Daniel Casini et al. "Partitioned fixed-priority scheduling of parallel tasks without preemptions". In: *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2018, pp. 421–433.

[36] Roberto Cavicchioli, Nicola Capodieci, and Marko Bertogna. "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms". In: *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2017, pp. 1–10.

[37] Younès Chandarli et al. "Yartiss: A tool to visualize, test, compare and evaluate real-time scheduling algorithms". In: 2012.

[38] Shuangshuang Chang et al. "Real-Time scheduling and analysis of parallel tasks on heterogeneous multi-cores". In: *Journal of Systems Architecture* 105 (2020), p. 101704.

[39] Jian-Jia Chen et al. "Many suspensions, many problems: a review of self-suspending tasks in real-time systems". In: *Real-Time Systems* 55.1 (2019), pp. 144–207.

[40] Jiasi Chen and Xukan Ran. "Deep learning with edge computing: A review". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1655–1674.

[41] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[42] Robert I Davis and Alan Burns. "Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems". In: *Real-Time Systems* 47.1 (2011), pp. 1–40.

[43] Reinhard Diestel. "Graph Theory, volume 173 of". In: *Graduate texts in mathematics* (2012), p. 7.

[44] *Edge and Cloud Computation: A Highly Distributed Software for Big Data Analytics (CLASS)*. `https://class-project.eu/`. [Online; accessed October 2020]. 2020.

[45] P Erdös and A Rényi. "On Random Graphs I". In: *Publicationes Mathematicae Debrecen* 6 (1959), pp. 290–297.

[46] Martin Ester et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *Kdd*. Vol. 96. 34. 1996, pp. 226–231.

[47] Nico Feiertag et al. "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics". In: *IEEE Real-Time Systems Symposium: 30/11/2009-03/12/2009*. IEEE Communications Society. 2009.

[48] 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*. 2017, pp. 28–37.

[49] José Fonseca, Geoffrey Nelissen, and Vincent Nélis. "Schedulability analysis of DAG tasks with arbitrary deadlines under global fixed-priority scheduling". In: *Real-Time Systems* 55.2 (2019), pp. 387–432.

[50] José Fonseca et al. "Response time analysis of sporadic dag tasks under partitioned scheduling". In: *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2016, pp. 1–10.

[51] José Carlos Fonseca et al. "A multi-dag model for real-time parallel applications with conditional execution". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM. 2015, pp. 1925–1932.

[52] Julien Forget et al. "Scheduling dependent periodic tasks without synchronization mechanisms". In: *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2010, pp. 301–310.

[53] Dagaen Golomb et al. "Data Freshness Over-Engineering: Formulation and Results". In: *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE. 2018, pp. 174–183.

[54] Ronald L Graham. "Bounds on multiprocessor timing anomalies". In: *SIAM J. Appl. Math.* 17 (1969), pp. 263–269.

[55] Fei Guan, Jiaqing Qiao, and Yu Han. "DAG-Fluid: A Real-Time Scheduling Algorithm for DAGs". In: *IEEE Transactions on Computers* (2020).

[56] Zhishan Guo et al. "Energy-efficient multi-core scheduling for real-time DAG tasks". In: *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[57] David S Hall and BS Hall. *Team DAD Technical Paper*. Tech. rep. Technical Report, 2005.

[58] Arne Hamann et al. *WATERS Industrial Challenge 2017*. 2017.

[59] Arne Hamann et al. *WATERS Industrial Challenge 2019*. 2017.

[60] Meiling Han et al. "Response time bounds for typed dag parallel tasks on heterogeneous multi-cores". In: *IEEE Transactions on Parallel and Distributed Systems* 30.11 (2019), pp. 2567–2581.

[61] Qingqiang He, Nan Guan, Zhishan Guo, et al. "Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores". In: *IEEE Transactions on Parallel and Distributed Systems* 30.10 (2019), pp. 2283–2295.

[62] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv:1704.04861* (2017).

[63] Forrest N Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size". In: *arXiv preprint arXiv:1602.07360* (2016).

[64] Xu Jiang et al. "Analyzing GEDF Scheduling for Parallel Real-Time Tasks with Arbitrary Deadlines". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1537–1542.

[65] Sunggoo Jung et al. "Perception, guidance, and navigation for indoor autonomous drone racing using deep learning". In: *IEEE Robotics and Automation Letters* 3.3 (2018), pp. 2539–2544.

[66] Rudolph Emil Kalman. "A new approach to linear filtering and prediction problems". In: (1960).

[67] Tomasz Kloda et al. "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems". In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2019, pp. 1–14.

[68] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. "Real world automotive benchmarks for free". In: *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2015.

[69] Jing Li et al. "Analysis of federated and global scheduling for parallel real-time tasks". In: *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. IEEE. 2014, pp. 85–96.

[70] Jing Li et al. "Outstanding paper award: Analysis of global edf for parallel tasks". In: *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. IEEE. 2013, pp. 3–13.

[71] Liangzhi Li, Kaoru Ota, and Mianxiong Dong. "Deep learning for smart industry: Efficient manufacture inspection system with fog computing". In: *IEEE Transactions on Industrial Informatics* 14.10 (2018), pp. 4665–4673.

[72] Tsung-Yi Lin et al. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[73] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20 (1973), pp. 46–61.

[74] Cong Liu and James H Anderson. "Supporting graph-based real-time applications in distributed systems". In: *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*. Vol. 1. IEEE. 2011, pp. 143–152.

[75] Cong Liu and James H Anderson. "Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss". In: *2010 31st IEEE Real-Time Systems Symposium*. IEEE. 2010, pp. 3–13.

[76] Li Liu et al. "Deep learning for generic object detection: A survey". In: *International Journal of Computer Vision* 128.2 (2020), pp. 261–318.

[77] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[78] J. J. Lukkien, W. F. Verhaegh, and R. J. Bril. "Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with Deferred Preemption Revisited". In: *19th Euromicro Conference on Real-Time Systems (ECRTS'07)(ECRTS)*. Vol. 00. June 2007, pp. 269–279. DOI: 10.1109/ECRTS.2007.38. URL: doi.ieeecomputersociety.org/10.1109/ECRTS.2007.38.

[79] J. Martinez, I. Sañudo, and M. Bertogna. "Analytical Characterization of End-to-End Communication Delays With Logical Execution Time". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2244–2254. DOI: 10.1109/TCAD.2018.2857398.

[80] Jorge Martinez, Ignacio Sañudo, and Marko Bertogna. "End-to-end latency characterization of task communication models for automotive systems". In: *Real-Time Systems* 56 (July 2020). DOI: 10.1007/s11241-020-09350-3.

[81] Alessandra Melani et al. "Response-time analysis of conditional dag tasks in multiprocessor systems". In: *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*. IEEE. 2015, pp. 211–221.

[82] Microsoft. *ONNX Runtime*. https://microsoft.github.io/onnxruntime/.

[83] Aloysius K. Mok and Deji Chen. "A multiframe model for real-time tasks". In: *RTSS*. 1996.

[84] Gordon Moore. "The future of integrated electronics". In: *Fairchild Semiconductor internal publication* 2 (1964).

[85] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965.

[86] Mitra Nasri, Geoffrey Nelissen, and Björn B Brandenburg. "Response-time analysis of limited-preemptive parallel DAG tasks under global scheduling". In: *31st Conference on Real-Time Systems*. 2019, pp. 21–1.

[87] Marco Di Natale et al. *Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice*. Springer Publishing Company, Incorporated, 2012. ISBN: 1461403138, 9781461403135.

[88] Geoffrey Nelissen et al. "Timing analysis of fixed priority self-suspending sporadic tasks". In: *2015 27th Euromicro Conference on Real-Time Systems*. IEEE. 2015, pp. 80–89.

[89] Andrea Parri, Alessandro Biondi, and Mauro Marinoni. "Response time analysis for G-EDF and G-DM scheduling of sporadic DAG-tasks with arbitrary deadline". In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. 2015, pp. 205–214.

[90] Risat Pathan, Petros Voudouris, and Per Stenström. "Scheduling parallel real-time recurrent tasks on multicore platforms". In: *IEEE Transactions on Parallel and Distributed Systems* 29.4 (2017), pp. 915–928.

[91] Rodolfo Pellizzoni et al. "A predictable execution model for COTS-based embedded systems". In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2011, pp. 269–279.

[92] M. Pfeiffer et al. "From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 1527–1533.

[93] Paul Purdom. "A transitive closure algorithm". In: *BIT Numerical Mathematics* 10.1 (1970), pp. 76–94.

[94] Manar Qamhieh and Serge Midonnet. "An experimental analysis of DAG scheduling methods in hard real-time multiprocessor systems". In: *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*. 2014, pp. 284–290.

[95] Manar Qamhieh, Serge Midonnet, and Laurent George. "Dynamic scheduling algorithm for parallel real-time graph tasks". In: *ACM SIGBED Review* 9.4 (2012), pp. 25–28.

[96] Manar Qamhieh et al. "Global EDF scheduling of directed acyclic graphs on multiprocessor systems". In: *Proceedings of the 21st International conference on Real-Time Networks and Systems*. 2013, pp. 287–296.

[97] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018).

[98] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[99] Lior Rokach and Oded Maimon. "Clustering methods". In: *Data mining and knowledge discovery handbook*. Springer, 2005, pp. 321–352.

[100] Olga Russakovsky et al. "Imagenet large scale visual recognition challenge". In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[101] Salah Eddine Saidi, Nicolas Pernet, and Yves Sorel. "Automatic parallelization of multi-rate fmi-based co-simulation on multi-core". In: *Proceedings of the Symposium on Theory of Modeling & Simulation*. Society for Computer Simulation International. 2017, p. 5.

[102] Abusayeed Saifullah et al. "Parallel real-time scheduling of DAGs". In: *IEEE Transactions on Parallel and Distributed Systems* 25.12 (2014), pp. 3242–3252.

[103] Yukihiro Saito et al. "ROSCH: Real-Time Scheduling Framework for ROS". In: *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE. 2018, pp. 52–58.

[104] Mark Sandler et al. "Mobilenetv2: Inverted residuals and linear bottlenecks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[105] Maria A Serrano and Eduardo Quiñones. "Response-time analysis of DAG tasks supporting heterogeneous computing". In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6.

[106] Maria A Serrano et al. "An Analysis of Lazy and Eager Limited Preemption Approaches under DAG-based Global Fixed Priority Scheduling". In: *Real-Time Distributed Computing (ISORC), 2017 IEEE 20th International Symposium on*. IEEE. 2017, pp. 193–202.

[107] Maria A Serrano et al. "Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions". In: *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium. 2016, pp. 1066–1071.

[108] Martin Stigge et al. "The Digraph Real-Time Task Model". In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium* (2011), pp. 71–80.

[109] Jinghao Sun et al. "Calculating Response-Time Bounds for OpenMP Task Systems with Conditional Branches". In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2019, pp. 169–181.

[110] Jinghao Sun et al. "On Computing Exact WCRT for DAG Tasks". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–6.

[111] Yuhei Suzuki, Takuya Azumi, Shinpei Kato, et al. "HLBS: Heterogeneous laxity-based scheduling algorithm for DAG-based real-time computing". In: *2016 IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*. IEEE. 2016, pp. 83–88.

[112] Emil Talpes et al. "Compute Solution for Tesla's Full Self-Driving Computer". In: *IEEE Micro* 40.2 (2020), pp. 25–35.

[113] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[114] Abhilash Thekkilakattil et al. "Multiprocessor fixed priority scheduling with limited preemptions". In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. 2015, pp. 13–22.

[115] Sebastian Thrun et al. "Stanley: The robot that won the DARPA Grand Challenge". In: *Journal of field Robotics* 23.9 (2006), pp. 661–692.

[116] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing". In: *IEEE transactions on parallel and distributed systems* 13.3 (2002), pp. 260–274.

[117] Jeffrey D. Ullman. "NP-complete scheduling problems". In: *Journal of Computer and System sciences* 10.3 (1975), pp. 384–393.

[118] Jacobo Valdes, Robert E Tarjan, and Eugene L Lawler. "The recognition of series parallel digraphs". In: *Proceedings of the eleventh annual ACM symposium on Theory of computing*. 1979, pp. 1–12.

[119] Martin Vel'as et al. "Calibration of rgb camera with velodyne lidar". In: (2014).

[120] Micaela Verucchi et al. "A Systematic Assessment of Embedded Neural Networks for Object Detection". In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE. 2020, pp. 937–944.

[121] Micaela Verucchi et al. "Embedded Neural Networks for Object Detection: A Systematic Assessment". In: ().

[122] Micaela Verucchi et al. "Latency-Aware Generation of Single-Rate DAGs from Multi-Rate Task Sets". In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2020, pp. 226–238.

[123] Micaela Verucchi et al. "Real-Time clustering and LiDAR-camera fusion on embedded platforms for self-driving cars". In: *2020 4th IEEE International Conference on Robotic Computing (IRC)*. IEEE. 2020.

[124] S. Vestal. "Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance". In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. 2007, pp. 239–243. DOI: 10.1109/RTSS.2007.47.

[125] AS Vincentelli et al. "Optimizing end-to-end latencies by adaptation of the activation events in distributed automotive systems". In: *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. IEEE. 2007, pp. 293–302.

[126] Ming Xiong et al. "Deferrable scheduling for maintaining real-time data freshness: Algorithms, analysis, and results". In: *IEEE Transactions on Computers* 57.7 (2008), pp. 952–964.

[127] Rui Xu and Donald C Wunsch. "Survey of clustering algorithms". In: (2005).

[128] Beyazit Yalcinkaya, Mitra Nasri, and Björn B Brandenburg. "An exact schedulability test for non-preemptive self-suspending real-time tasks". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2019, pp. 1228–1233.

[129] Kecheng Yang, Ming Yang, and James H Anderson. "Reducing response-time bounds for DAG-based task systems on heterogeneous multicore platforms". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 349–358.

[130] Tao Yang, Qingxu Deng, and Lei Sun. "Building real-time parallel task systems on multi-cores: A hierarchical scheduling approach". In: *Journal of Systems Architecture* 92 (2019), pp. 1–11.

[131] Fisher Yu et al. "Bdd100k: A diverse driving video database with scalable annotation tooling". In: *arXiv preprint arXiv:1805.04687* (2018).

[132] Houssam-Eddine Zahaf et al. "A C-DAG task model for scheduling complex real-time tasks on heterogeneous platforms: preemption matters". In: *arXiv preprint arXiv:1901.02450* (2019).

[133] Houssam-Eddine Zahaf et al. "The HPC-DAG Task Model for Heterogeneous Real-Time Systems". In: *IEEE Transactions on Computers* (2020), pp. 1–1.

[134] Haibo Zeng. "Probabilistic Timing Analysis of Distributed Real-time Automotive Systems". PhD thesis. EECS Department, University of California, Berkeley, 2008.

[135] Haibo Zeng et al. "Statistical analysis of Controller Area Network message response times". In: *IEEE Fourth International Symposium on Industrial Embedded Systems, SIES 2009, Ecole Polytechnique Federale de Lausanne, Switzerland, July 8-10, 2009*. 2009, pp. 1–10.

[136] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. "Objects as points". In: *arXiv preprint arXiv:1904.07850* (2019).

[137] Zhengxia Zou et al. "Object detection in 20 years: A survey". In: *arXiv preprint arXiv:1905.05055* (2019).