



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH



Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

MASTER THESIS

Scheduling strategies for
time-sensitive distributed applications
on edge computing

Author:

Eudald SABATÉ CREIXELL

Supervisors:

Dr. Eduardo QUIÑONES

Dr. María A. SERRANO

Tutor:

Dr. Miquel MORETÓ

*A thesis submitted in fulfillment of the requirements
for the Master in Innovation and Research in Informatics,
Computer Networks and Distributed Systems specialization*

in the

Facultat d'Informàtica de Barcelona (FIB)
at Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

June 22, 2020

Declaration of Authorship

I, Eudald Sabaté Creixell, declare that this thesis titled, “Scheduling strategies for time-sensitive distributed applications on edge computing” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at the [Universitat Politècnica de Catalunya \(UPC\)](#) - BarcelonaTech.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at [Universitat Politècnica de Catalunya \(UPC\)](#) - BarcelonaTech or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Abstract

Edge computing is a distributed computing paradigm that shifts the computation capabilities close to the data sources. This new paradigm, coupled with the use of parallel embedded processor architectures, is becoming a very promising solution for time-sensitive distributed applications used in Internet of Things and large Cyber-Physical Systems (e.g., those used in smart cities) to alleviate the pressure on centralized solutions. However, the distribution and heterogeneity nature of the edge computing complicates the response-time analysis on these type of applications. This thesis addresses this challenge by proposing a new Directed Acyclic Graph (DAG)-task based system model to characterize: (1) the distribution nature of applications executed on the edge; and (2) the heterogeneous computation and network communication capabilities of edge computing platforms. Based on this system model, this work presents five different scheduling strategies: four sub-optimal but tractable heuristics and an optimal but costly approach based on a mixed integer linear programming (MILP), that minimize the overall response time of distributed time-sensitive applications. To address both issues, and as a proof of concept, we use COMPSs, a framework composed of a task-based programming model and a runtime used to program and efficiently distribute time-sensitive applications across the compute continuum. However, COMPSs is agnostic of time-sensitive applications, hence in this work we extend it to consider the dynamic scheduling based on the proposed scheduling strategies. Our results show that our scheduling heuristics outperform current scheduling solutions, while providing an average and upper-bound execution time comparable to the optimal one provided by the MILP allocation approach.

Acknowledgements

First of all, I want to deeply thank my advisors María and Eduardo for their support, guidance and mentoring through the development of this thesis. I want to specially thank them for providing me with the opportunity to participate in two European Projects, CLASS and ELASTIC.

I also want to thank the rest of the people of the PPC group at BSC who always offered help whenever I needed.

Furthermore, I would like to acknowledge BSC for financially supporting my master studies.

Last but not least, I would like to thank my friends and family for their support during my studies. I specially want to thank my mother, my brother and my father for all the help and unconditional support in the more critical moments throughout my life.

Contents

Declaration of Authorship	i
Acknowledgements	iii
Contents	iv
List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Motivation	1
1.2 Contributions	2
1.3 Publications and Impact	3
1.4 Structure of the Thesis	3
2 State of the Art	5
2.1 Timing Guarantees for Time-Sensitive Applications	5
2.1.1 Timing Analysis	6
2.1.2 Shared-Memory Parallel Programming	6
2.1.3 Example	8
2.2 Distributed Systems	8
2.3 Edge Computing	10
3 Background: the COMPSs Framework	11
3.1 Overview	11
3.2 Task-based Programming Model	12
3.3 Runtime System: DAG and Task Scheduler	13
3.4 Advancing the Scheduling Capabilities of COMPSs	15

4	System Model	17
4.1	The Digraph Compute Continuum Model	17
4.2	The DAG Task Model	18
4.3	Network Communications	19
4.4	Redefining <i>Volume</i> and <i>Workload</i> Concepts	19
4.5	Putting it all Together: a Real Example	21
5	Task Scheduling Strategies	23
5.1	MILP-based Optimal Task Scheduling	23
5.2	Task Scheduling Heuristics	26
5.2.1	Heuristics Based on Successors	27
5.2.2	Heuristics Based on Processing Time	29
5.3	Benefits of Static Task Scheduling Strategies in Distributed Environments	30
5.4	Tasks Deadlines	31
5.5	Example	31
6	A New Scheduling Component for COMPSs	33
6.1	Analysis Phase	33
6.1.1	Example	37
6.2	Scheduling Strategies Implemented in COMPSs	37
6.2.1	MILP-based Optimal Task Scheduling	37
6.2.2	Task Scheduling Heuristics	39
6.3	Task Monitoring	40
6.4	A Reactive Scheduler	41
7	Evaluation	42
7.1	Experimental Setup	42
7.1.1	Compute Continuum Configuration	42
7.1.2	Applications	43
7.2	Performance and Accuracy	44
7.2.1	Classical Applications	44
7.2.2	Object Detection and Tracking	46
7.3	Reactive Scheduling: Static Allocation in Dynamic Environments	48

7.4 MILP Complexity	49
8 Conclusions and Future Work	50
Bibliography	52

List of Figures

2.1	Time-sensitive DAG task example. Nodes are labeled with their corresponding WCET.	8
2.2	Scheduling of DAG from Figure 2.1 scheduled in a 3-core system.	8
3.1	Object detection and tracking (ODT) COMPSs example.	12
3.2	DAG representing the application of Figure 3.1.	14
4.1	Compute Continuum model example and values.	21
4.2	DAG task model values of the DAG in Figure 3.2.	22
5.1	MILP task scheduling for the ODT application in 3.1.	31
5.2	LPT task scheduling for the ODT application in 3.1.	32
6.1	iPerf output for Bandwidth retrieval of Ethernet IEEE 802.3.	34
6.2	<i>dot</i> file for the DAG of the ODT application in 3.2.	36
6.3	<i>prv</i> file for retrieving the tasks execution times.	37
6.4	Logging file extract to retrieve the payload size produced by task 1 from 4.2.	37
6.5	Input file generated by the profiling mechanism for ODT application.	38
6.6	MILP output for the ODT application input in 6.5.	39
7.1	Boxplot of the execution time of each COMPSs applications under different scheduling strategies (Ethernet).	45
7.2	Object detection and tracking COMPSs application.	47
7.3	Execution time (multiple iterations) and R^{ub} of the object detection and tracking, in a dynamic compute continuum environment (from 4 to 3 computing resources).	48

List of Tables

7.1	Number of nodes N , and communication and computation volumes, vol^{comm} and vol^{comp} , for each application.	48
7.2	MILP and best scheduling heuristic execution times in seconds.	49

Chapter 1

Introduction

1.1 Motivation

The digitalization process undergone in multiple application domains is challenging the development of Internet of the Things (IoT) and large cyber-physical systems (CPS) with a variety of requirements. Among these requirements, high-performance and real-time guarantees are two important ones: the former refers to the capability of processing a considerable number of data sources retrieving and sending information collected about their surroundings; while the latter refers to the requirement of operating within a given time budget to accomplish certain functionalities. Smart cities are a prominent example that is facing these two requirements. The advent of connected and autonomous cars, featuring advanced functionalities based on knowledge coming from the city, imposes the need of accomplishing high-performance and real-time requirements.

Edge computing [1] is a new computing paradigm that effectively addresses these two challenges. On one side, it shifts the computation as close as possible to where the data is originated, allowing to manage the increasing number of data sources and alleviating the pressure on centralized solutions. Moreover, the use of powerful parallel embedded processor architectures at the edge side (e.g., NVIDIA Jetson AGX [2], Xilinx Versal [3], Kalray MPPA [4]) can also help alleviating this pressure due to their huge computing capabilities, while providing lower network costs and more energy efficient solutions. On the other side, recent works have demonstrated that the parallel programming models used to program these edge devices, e.g., CUDA [5, 6] and OpenMP [7, 8], can provide the real-time guarantees needed on the final resource.

Distributed systems play an immense role in edge computing, as the latter is distributed by nature. Distributed systems not only support the shifting of both the computation and storage closer to the origin introduced by edge computing, but also contributing with reliability, efficiency and scalability. Moreover, the vast amount of geographically-distributed data produced by edge devices needs to be processed in a distributed manner as centralized systems became a bottleneck for data analytics. Hence, distributed systems also help alleviating the processing of Big Data analytics providing a high-quality aggregate performance.

Despite the clear benefits of edge computing and distributed systems, the development of time-sensitive applications remains challenging mainly for two reasons. First, the edge computing environment composed of heterogeneous interconnected edge devices, a.k.a. the *compute continuum*, complicates the development and deployment of these applications. Second, due to the distribution and heterogeneous nature of the compute continuum, the response time analysis of such applications is fairly complex. In this thesis we aim to address these two challenges.

To do so, and as a proof of concept, we use COMPSs, a framework composed of a task-based programming model and a runtime [9, 10], used to program and efficiently distribute time-sensitive applications across the compute continuum. However, COMPSs is agnostic of time-sensitive applications, not supporting any response time analysis. In this work, we extend the COMPSs framework to consider a dynamic scheduling based on different scheduling heuristics that we also present in this work.

1.2 Contributions

The contributions presented in this thesis are the following:

1. We first propose a novel task based system model to describe the execution and timing behavior of time-sensitive distributed applications on edge computing environments. It consists of two components: a *Digraph compute continuum model* that characterizes the edge computing resources and the communication links between them; and a *Directed Acyclic Graph (DAG) task model* that captures the parallel and distributed nature of applications, when executing on the

compute continuum. Our DAG based model is an extension of the DAG task model widely used for scheduling analysis in shared-memory processors [11].

2. Secondly, based on the proposed system model, we present an optimal but costly *mixed integer linear programming* (MILP) scheduling strategy, and four different sub-optimal but tractable scheduling heuristics. Our scheduling strategies have a common twofold objective: (1) to minimize the overall execution time and (2) to provide an upper bound response time of time-sensitive distributed applications.
3. Finally, we also enhanced the Comp Superscalar (COMPSs) framework, developed at Barcelona Supercomputing Center, to provide a dynamic scheduling based on the proposed scheduling heuristics. This allows, not only to provide timing guarantees to COMPSs applications, but also to reduce the average execution time of applications, as we demonstrate in the evaluation.

1.3 Publications and Impact

The contributions presented in this thesis are currently being used in two different European Projects: CLASS [12] and ELASTIC [13], in which the different allocation strategies implemented in the COMPSs framework are used to provide real-time guarantees to the time-sensitive distributed applications executed throughout the edge compute continuum. Moreover, the application developed and used in the evaluation chapter for object detection and tracking are been used in the use-case of the CLASS project.

The work performed for this thesis has also been submitted to the International Conference on Computer Aided Design (ICCAD) 2020 and it is currently pending on the notification of acceptance. Part of this work was also presented in the ACACES Summer School Poster Session 2019 [14] and in the 6th BSC Severo Ochoa Doctoral Symposium in 2019 [15].

1.4 Structure of the Thesis

The rest of the document is organized as follows. Chapter 2 presents the state of the art regarding distributed time-sensitive applications executed on distributed real-time

systems and timing analysis techniques applied to provide timing requirements. Chapter 3 introduces COMPSs, the framework considered to evaluate this work. Chapter 4 introduces the system model that we consider to support the analysis and execution of time-sensitive applications on edge computing (Contribution 1 in Section 1.2). Chapter 5 presents our MILP and scheduling heuristic strategies developed (Contribution 2 in Section 1.2). Chapter 6 presents the enhancements performed in the current COMPSs framework (Contribution 3 in Section 1.2). Chapter 7 provides the evaluation and results of the aforementioned scheduling strategies. Finally, Chapter 8 draws conclusions and presents future work.

Chapter 2

State of the Art

This chapter presents the state of the art regarding distributed time-sensitive applications and the characteristics of the environments in which they execute. Moreover, the different challenges present in these systems are tackled.

2.1 Timing Guarantees for Time-Sensitive Applications

A system defined as a *real-time system* is the one that operates under any timing constraint [16]. On these systems, and by inference for the time-sensitive applications running in them, timing guarantees are of high importance as the context in which they operate are generally critical environments. In these environments, for the result to be valid it not only needs to be correct but it also needs to be provided within a certain time window, namely a deadline [17].

Time-sensitive applications are assigned this global end-to-end deadline in order to be valid, which is usually used as a metric indicator for the *Quality of Service* (QoS) of the application specified in the *Service Level Agreement* (SLA) [18, 19].

Car navigation, smart cities or industrial automation are clear examples of time-sensitive applications in which missing the end-to-end deadline might lead to a catastrophic failure of the system or endanger people lives.

Generally, time-sensitive applications take advantage of being executed in *application specific integrated circuits* (ASIC) or, more recently, general-purpose processors such as *graphics processing units* (GPUs) or along with re-configurable fabrics such as *field programmable gate arrays* (FPGAs). Moreover, the appearance of powerful

and parallel embedded processor architectures such as NVIDIA Jetson AGX [2], Xilinx Versal [3] or Kalray MPPA [4] can be used to meet the performance and timing requirements of these applications.

2.1.1 Timing Analysis

Time-sensitive applications are generally defined as a set of tasks, that is, asynchronous concurrent functions executed by the computing resources composing the system. This is critical for achieving the timing constraints needed by these applications as deadlines can be assigned at a task level, hence allowing to better predict their behavior.

In order to assess whether a deadline will be met or missed, the concept of *worst case execution time* (WCET) is also introduced. WCET can be estimated either by using timing models of the system or by collecting measurements and adding a safety margin over the highest execution time observed, which is a highly common industrial practice that relies on software profiling reinforced by the use of safety margins [20].

The timing analysis of these particular tasks composing time-sensitive applications is performed by estimating the WCET of each task. Given the WCET obtained for each of them, a scheduling is obtained by using any of the many scheduling algorithms on real-time systems such that their execution is completed before the deadline provided by the scheduling.

The precedence constraints between these tasks are represented by a Directed Acyclic Graph (DAG), which allows to analyze their schedulability and to derive a (worst-case) response time analysis of parallel applications. The DAG allows to represent all the tasks composing an application, in which each node represents *task*) annotated with its WCET, and the edges between two nodes represent the precedence constraints and/or data transfers among them.

2.1.2 Shared-Memory Parallel Programming

The majority of the real-time systems are designed as parallel concurrent processing systems, which allows for the system to easily react to events. The scheduling of concurrent activities is critical for achieving real-time constraints. Hence, a programming model that exploits the massively parallel recent architectures is needed, such as

parallel programming, which allows to effectively process vast amounts of data simultaneously. OpenMP [21] and NVIDIA CUDA [22] parallel programming models used to program real-time devices have demonstrated that can provide real-time guarantees on the final resources [7, 8, 5, 23, 6].

However, the use of parallel programming models increases the complexity of the timing analysis and schedulability of time-sensitive applications executed in real-time systems due to the fact that the tasks of the application are executed on heterogeneous systems, providing different WCET for the same task based on where it is executed [5, 23]. Moreover, unless parallel programming is not managed carefully, it can also introduce overhead, thus affecting the final real-time performance [24].

The DAG task model was introduced with the appearance of shared memory processors and multi-core processors under different scheduling algorithms [11, 25, 26] to better express the parallelism offered by these architectures, thus allowing a further exploitation of parallelism within workflows. The use of DAGs has also enabled multi-core processors to schedule periodic parallel tasks with implicit deadlines [11].

Recently with the introduction of heterogeneous architectures due to the increasing demands of modern cyber-physical embedded systems, it is a common trend to combine high-performance multi-core CPU hosts with a certain number of application-specific accelerators. The DAG model evolved as it needed to take into account different implementations of each task on heterogeneous platforms [27]. Furthermore, the use of the DAG model allowed to improve the impact of the response time upper-bound by offloading computation to those accelerators, providing a more accurate response time upper-bound [28].

As systems evolved from single-core to multi-core, and in time to heterogeneous and distributed real-time systems, the DAG task model becomes more complex due to the fact that distributed systems are heterogeneous by nature, and so, the WCET annotated for each node is transformed into a WCET for each implementation of that node in each device that is able to execute it. Regarding the edges composing the DAG, they need to provide information on the amount of data exchanged between nodes (or tasks), as in distributed systems there exists the need to characterize the data transfers and communication between the underlying devices composing them.

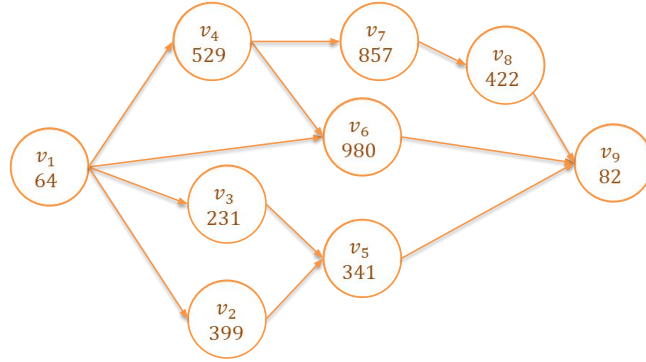


FIGURE 2.1: Time-sensitive DAG task example. Nodes are labeled with their corresponding WCET.

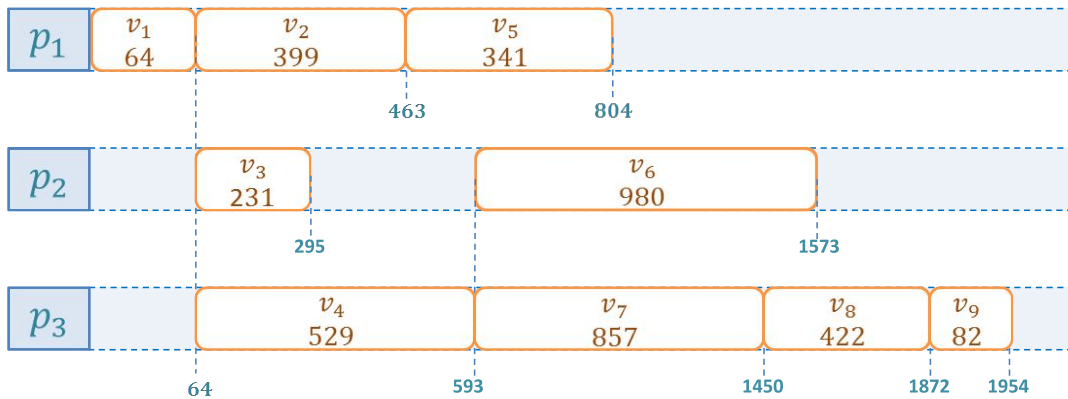


FIGURE 2.2: Scheduling of DAG from Figure 2.1 scheduled in a 3-core system.

2.1.3 Example

Figure 2.1 depicts an example of a DAG characterizing a parallel time-sensitive application, composed by 9 tasks each labeled with their corresponding WCET, and 12 edges representing the precedence constraints among them.

Considering a real-time multi-core system of three cores, we schedule the DAG from Figure 2.1, thus obtaining the scheduling shown in Figure 2.2.

2.2 Distributed Systems

In the context of smart cities and industrial automation, a vast amount of Internet of Things (IoT) and Cyber-Physical Systems (CPS) devices with real-time requirements for the time-sensitive applications are needed which need to be distributed in order to cover the maximum amount of area. These distributed systems are heterogeneous by nature due to the fact that are composed by an immense amount of different devices

and hence challenging any centralized solution, hence also providing reliability to avoid system failures. Moreover, the vast amount of geographically-distributed data collected by the different devices needs to be processed in a distributed manner [29] in order to improve the efficiency in the data analytics process and to achieve their timing requirements.

Apache Spark [30], a cluster computing framework for real-time large-scale data processing, has been widely considered as it provides an efficient solution to parallelize and distribute the computation of Big Data analytics among the distributed system [29, 31].

However, real-time distributed systems introduce a new challenge: *deadline assignment*. This problem refers to the process of assigning local deadlines to each of the real-time tasks composing a distributed application, with the objective of being able to meet a global end-to-end deadline, which is tightly related with the QoS assigned to the time-sensitive distributed applications. Deadline assignment became a challenge due to the fact that an increasing number of workflows are concurrently running on a distributed system even though several algorithms to approach the deadline assignment problem have been presented [18, 19, 32].

Vehicular systems and automotive systems are also an example of heterogeneous distributed real-time systems. In these scenarios, the scheduling not only of real-time tasks composing the distributed application is critical, but also the communications that take place between the devices composing the distributed system should be taken into account [33]. In order to model those systems and to provide an optimal scheduling, *integer linear programming* (ILP) is widely considered [34]. ILP formulations allow to solve the allocation problems of parallel and distributed functions providing an optimal but costly solution, while it can also be tuned in order to assign deadlines and activation times to tasks such that tasks partitioned onto different virtual processors can be analyzed separately [35].

Even though ILP provides the optimal solution, it comes as a high processing cost in terms of time. Hence, much literature exists regarding heuristic approaches and scheduling algorithms that provide task allocation and scheduling [36, 37, 38].

From the High-Performance Computing (HPC) domain, task based schedulers have been widely proposed for distributed environments. The representation of the

workflow as a DAG, coupled together with scheduling algorithms that benefit from the information provided by the DAG itself, allow to improve the performance of distributed systems [39, 40].

2.3 Edge Computing

Edge computing [1] is a new distributed computing paradigm that shifts the computation as close as possible to where the data is originated, allowing to manage the increasing number of data sources and alleviating the pressure on centralized solutions, which makes him a perfect candidate for contexts such as the smart cities and industrial automation. Moreover, the use of powerful parallel embedded processor architectures at the edge side (e.g., NVIDIA Jetson AGX [2], Xilinx Versal [3], Kalray MPPA [4]) can also help alleviating this pressure due to their huge computing capabilities, while providing lower network costs and more energy efficient solutions.

However, the challenges from distributed systems described in Section 2.2 also apply in edge computing. Furthermore, the compute continuum composed by heterogeneous devices increases the complexity of both the development and deployment of the time-sensitive applications. This also complicates the response time analysis of such applications as network connections need to be considered [36, 37, 38].

Chapter 3

Background: the COMPSs

Framework

This chapter presents background information on task-based programming models and DAG representation. In particular, we present COMPSs [9, 10, 41], a task-based programming model and runtime framework, used in the *high-performance computing* (HPC) domain for the development of parallel applications and their execution over distributed infrastructures, such as clusters, clouds and containerized platforms. In this work, without loss of generality, COMPSs has been selected and adapted for the implementation and performance evaluation of the proposed scheduling techniques for time-sensitive distributed applications over edge computing platforms. However, it should be stressed that the proposed solutions are not limited to the COMPSs environment, but can be generally tested with any suitable task-based programming model.

3.1 Overview

The main objective of the COMPSs framework is to facilitate the parallelization and execution of sequential source codes (written in C/C++, Python or Java) in distributed computing environments. The application is agnostic of the underlying distributed infrastructure, i.e., they do not include any detail that could tie them to a particular platform, boosting portability among diverse infrastructures. Clearly, this is a very convenient property in heterogeneous environments such as the edge computing paradigm.

```
1 @task (returns = numpy.ndarray)
2 def get_frame():
3     return get_next_frame_from_video()

5 @task (frame = IN, returns = list)
6 def get_objects_from_frame(frame):
7     return yolo.detect(frame)

9 @task (list_objects = IN, returns = list)
10 def tracker(list_objects):
11     return tracker.track(list_objects)

13 @task (list_objects = IN, frame = IN)
14 def collect_and_display(list_objects,
15                          frame):
16     for obj in list_objects:
17         display(obj, frame)

18 ### Main function ###
19 while (true):
20     frame = get_frame()
21     list_obj = get_objects_from_frame(frame)
22     for i in range(0, 2):
23         list_obj[i] = tracker(list_obj)
24     collect_and_display(list_obj, frame)
```

FIGURE 3.1: Object detection and tracking (ODT) COMPSs example.

The COMPSs framework is composed of a task-based programming model which aims to ease the development of parallel applications, and a runtime system that exploits the inherent parallelism of applications, defined in the following sections.

3.2 Task-based Programming Model

The COMPSs programmer is responsible of identifying the portions of code, named COMPSs tasks, that can be distributed, by simply annotating the sequential source code. Data dependencies and their directionality (i.e., in, out or inout) must be also identified. Upon them, the runtime determines the order in which COMPSs tasks are executed and also the data transfers across the distributed system. A COMPSs task with an in or inout data dependency cannot start its execution until the COMPSs task with an out or inout dependency over the same data element is completed.

Figure 3.1 shows an example of a COMPSs application for object detection and tracking (ODT) written in Python (PyCOMPSs [41]). COMPSs tasks are identified with a standard Python decorator `@task`, at lines 1, 4, 7, and 10. The `returns` argument specifies the data type of the value returned by the function (if any). The

IN defines the input data directionality of parameters. The main code starts at line 15, where a loop iterates while detecting and tracking objects from an input video, e.g., from a street camera of the city. Each iteration operates over a video frame, first getting it using the COMPSs task at line 1. Then, the object detection is computed using YOLO [42], a well-know real-time object detection system (COMPSs task at line 4). At line 18, the application processes in parallel different detected objects to track them (COMPSs task at line 7). Finally, the updated list of tracked objects, `list_obj`, is merged and objects detected and tracked are displayed at line 20 (COMPSs task at line 10).

3.3 Runtime System: DAG and Task Scheduler

The available computing resources composing the compute continuum are identified in the XML configuration files `resources.xml` and `project.xml` by the COMPSs programmer. Each computing resource represents a COMPSs worker, which is the runtime entity in charge of executing the COMPSs tasks. The `resources.xml` file serves as a list of all configured and available workers in the environment, whereas the `project.xml` represents the subset of resources to be used for one specific application.

COMPSs is based on the *master-workers* paradigm, in which the COMPSs master is the component in charge of executing the main code of the application through means of the `runcompss` command line interface. Moreover, it is also the component in charge of detecting the COMPSs tasks at runtime, scheduling and spawning them asynchronously to the defined set of distributed and interconnected computing resources that execute them in parallel (as soon as all its data dependencies are honored). The data elements marked as `in` and `inout` are transferred to the compute resource in which the task will execute. This data transfers are identified in the COMPSs framework by tags, which represent the different input/output parameters that tasks either receive or produce, respectively. These tags allow to avoid unnecessary transfers if the aforementioned parameters are already available in a particular worker, that is, if the parameter has been previously transferred due to another task executed in the same worker requesting the same parameter. The tags representing the different transfers are updated accordingly in case the task updates the value

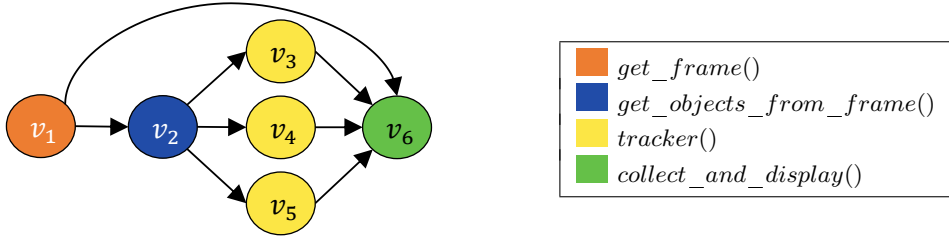


FIGURE 3.2: DAG representing the application of Figure 3.1.

of the parameters received, thus facilitating the master to manage whenever a given worker already contains the transfer required by the task it will execute.

The task-based programming model of COMPSs is supported by its runtime system, which manages several aspects of the application execution. For this, the runtime maintains the internal representation of the parallelism of the COMPSs application as a DAG. Each node corresponds to a COMPSs task and edges represent data dependencies. As an example, Figure 3.2 shows the DAG representation of one iteration of the COMPSs workflow presented in Figure 3.1. Each instantiated COMPSs task is represented by a different node, with colors identifying the different task functionalities.

Based on this DAG, the runtime can automatically detect data dependencies between COMPSs tasks. As soon as a task becomes ready, the COMPSs scheduler is then in charge of offloading its execution onto one of the available computing resources and transferring the input parameters before starting the execution. Concretely, upon receiving a dependency-free task, denoted as a *ready task*, the COMPSs task scheduler selects a resource to host its execution taking into account several parameters: (1) capabilities and status of the set of available resources, and (2) constraints of the invoked task.

Algorithm 1 shows a simple overview of the COMPSs scheduler to distribute and execute tasks. The way COMPSs tasks and resources to allocate them are selected, by functions *Get_Ready_Task* and *Get_Resource*, respectively, determine the different scheduling strategies that can be implemented. When an available resource is selected, the input parameters are transferred if the worker selected has not received them yet (*Transfer_Data*) and the task can initiate the execution (*Execute_Task*). Note that the data transfer between dependent COMPSs tasks does not start until the descendant one is scheduled and assigned to a computing resource to host its execution,

Algorithm 1 Pseudo code of a distributed tasks scheduler.

```
1: function SCHEDULE_TASKS
2:   while Not_Empty(ready_queue) do
3:     task ← Get_Ready_Task(ready_queue)
4:     resource ← Get_Resource()
5:     Transfer_Data(task, resource)
6:     Execute_Task(task, resource)
7:   end while
8: end function
```

thus limiting the overlapping of communication with computation.

COMPSs already provides different schedulers to be used based on the family of schedulers that consider only ready tasks:

- *First In, First Out* (FIFO): a very simple scheduler that selects ready tasks based on the order of entrance in the ready queue, that is, a task that first is the one with the highest priority.
- *Last In, First Out* (LIFO): scheduler that selects ready tasks based on order of arrival, however the last task entering in the ready queue is the one with highest priority.

Both schedulers select the next task to execute on the `Get_Ready_Task(ready_queue)` method based on the priority given by the order of entrance to the ready tasks, as explained above. Furthermore, both schedulers use the *First Idle* approach to select the resource to host the execution of the previously selected tasks in `Get_Resource()`, which selects the first available computing resource.

3.4 Advancing the Scheduling Capabilities of COMPSs

Despite the expressiveness of the task-based programming model to develop distributed applications, and the capabilities of the runtime to be independent of the available infrastructure, COMPSs is agnostic of time-sensitive applications as it does not take further advantage of the DAG constructed neither includes relevant information required in order to provide an upper-bound response time of the applications being executed.

We enhance the scheduling capabilities of COMPSs to effectively address this issue by providing new scheduling algorithms in COMPSs (see Chapter 6) based on

the system model (see Chapter 4) and the scheduling strategies proposed in this thesis (see Chapter 5).

Chapter 4

System Model

This chapter presents a novel system model to support the analysis and execution of time-sensitive applications on edge computing. The proposed model consists of two components: (1) a *Digraph compute continuum model*, representing the heterogeneous computing edge devices and the communication network that connects them, and (2) a *DAG task model*, describing the parallel structure of the time-sensitive distributed applications.

4.1 The Digraph Compute Continuum Model

The compute continuum is represented as a *Digraph* (directed graph) $G^{cc} = (V^{cc}, E^{cc})$, being $V^{cc} = \{p_1, p_2, \dots, p_M\}$ the set of nodes and $E^{cc} \subseteq V^{cc} \times V^{cc}$ the set of edges. Each node $p_i \in V^{cc}$ represents a computing resource, being $|V^{cc}| = M$ the total number of resources. Each edge $(p_k, p_l) \in E^{cc}$ corresponds to the communication link between two computing resources, p_k and p_l . In order to properly characterize the communication time, each communication link (p_k, p_l) is represented with the transport bandwidth $bw_{k,l}$, the maximum frame size $mfs_{k,l}$, and the size of the headers included in the frames $h_{k,l}$. If two resources p_k and p_l are not connected, then $bw_{k,l}$, $mfs_{k,l}$ and $h_{k,l}$ are equal to -1 . Also $bw_{k,k} = \infty$, $mfs_{k,k} = 0$ and $h_{k,k} = 0$ represent that there are no data transfers if two tasks are allocated to the same computing resource p_k .

4.2 The DAG Task Model

Our task model is based on the DAG task system model used to express parallel workloads and analyze their schedulability in shared memory architectures [11]. A time-sensitive distributed application is represented as a DAG $G = (V, E)$, being $V = \{v_1, v_2, \dots, v_N\}$ the set of nodes and $E \subseteq V \times V$ the set of edges. A node $v_i \in V$ represents a *distributed task* corresponding to a unit of computation (i.e., a COMPSs task in our case) in the compute continuum, being $|V| = N$ the total number of distributed tasks. A task v_i is characterized by a set of execution time upper bounds $\{C_{i_1}, C_{i_2}, \dots, C_{i_M}\}$, each corresponding to the potential task execution in a given computing resource $p_k, 1 \leq k \leq M$. If a task v_i cannot be executed in p_k , due to either an incompatibility or a design decision, then $C_{i,k} = -1$. It should be noted that a task may not only have different C_i 's, but also different implementations, to allow its execution in edge devices with different processor architectures. The COMPSs framework fully supports this feature by means of the COMPSs `@implements` decorator, which allows the definition of multiple implementations of the same task, e.g., using CUDA, OpenMP, or a simply sequential implementation, etc. Tasks, exploiting or not parallelism, are executed in a dedicated computing resource (given by the scheduling strategy) in isolation.

An edge $(v_i, v_j) \in E$ represents a data dependency between tasks v_i and v_j : if $(v_i, v_j) \in E$ then node v_i must complete before node v_j can begin its execution. Moreover, $(v_i, v_j) \in E$ is characterized by the total size of the data associated to the dependency, denoted by $z_{i,j}$, and the identifier of the data element, denoted by $zid_{i,j}$, which are non-null strings in the case that j requires the transfer from i , "0" otherwise. If v_j executes in a computing resource different than v_i and the same computing resource has not yet received a data element tagged with the same $zid_{i,j}$, then this data must be transmitted through the communication network, i.e., $z_{i,j}$ represents the payload.

Without loss of generality, the DAG is assumed to have exactly one source node v_{source} , i.e., a node with no incoming edges, and one sink node v_{sink} , i.e., a node with no outgoing edges. If this is not the case, a dummy source/sink node can be added to the DAG, with $C_{source,k} = 0$, and $C_{sink,k} = 0, \forall p_k \in V^{cc}$, and with zero data transfer

sizes to/from all the original source/sink nodes.

4.3 Network Communications

In the edge computing paradigm, the data transfer time is a key factor due to its potentially high impact on the timing behavior of the application. Equation 4.1 estimates the *data transfer time* due to the payload among two dependent tasks v_i and v_j :

$$T_{i,j}^{transf} = \begin{cases} \left(\left\lceil \frac{z_{i,j}}{mfs_{k,l}} \right\rceil \times h_{k,l} + z_{i,j} \right) \times \frac{1}{bw_{k,l}}, & \text{if } k \neq l \\ 0, & \text{otherwise} \end{cases} \quad (4.1)$$

where $z_{i,j}$ is the payload; and $bw_{k,l}$, $mfs_{k,l}$, and $h_{k,l}$ characterize the communication link that connects the p_k and p_l computing resources where v_i and v_j are executed. Notice that, if two computing resources p_k and p_l are not connected, $bw_{k,l} = -1$, then tasks v_i and v_j , such that $(v_i, v_j) \in E$, cannot be executed in p_k and p_l , respectively. This is controlled by the scheduling strategies (see Section 5). Also if v_i and v_j , are executed in the same computing resource p_k , $bw_{k,k} = \infty$, and then $T_{i,j}^{transf} = 0$. Furthermore, if there exists another task v_t such that $zid_{i,j} = zid_{i,t}$ and v_t executes before than v_j , then $T_{i,j}^{transf} = 0$ as well meaning that v_t already requested the same parameter, thus it is already presented in the computing resource p_l .

4.4 Redefining *Volume* and *Workload* Concepts

In the classical DAG-based system model targeting shared memory architectures, the volume of a DAG task, denoted as $vol(G)$, is defined as the sum of the worst-case execution time (WCET) C_i , of all the nodes of G . This value corresponds to the worst case response time of the DAG task on a dedicated single-core platform [7]. In our system model, this definition is not valid anymore since the response time of an application depends on the scheduling decisions: (1) the execution time $C_{i,k}$ of each node v_i may be different for each computing resource p_k ; and (2) the data transfer times $T_{j,i}^{transf}$ depend on the communication link existing between the computing resources where v_j and v_i are executed.

Therefore, the scheduling of nodes to computing resources has to be known to compute the volume of a DAG in our system model. To do so, we define $Y_{i,k}, \forall v_i \in V, \forall p_k \in V^{cc}$ as the *static scheduling function* of nodes v_i to computing resources p_k :

$$Y_{i,k} = \begin{cases} 1, & \text{if the node } v_i \text{ is allocated to the} \\ & \text{resource } p_k = p_l \\ 0, & \forall p_k \in V^{cc}, k \neq l \end{cases} \quad (4.2)$$

Notice that $\forall v_i \in V, \exists! p_k \in V^{cc} (Y_{i,k} = 1)$ – uniqueness property.

Since there two factors impacting on the response time of an application, i.e., the actual execution time on the computing nodes and the data transfer times, we define two different volumes:

Definition 1. *The computation volume of the DAG $G = (V, E)$ executed on the Digraph compute continuum $G^{cc} = (V^{cc}, E^{cc})$, is the sum of the $C_{i,k}$ of each node $v_i \in V$, when $Y_{i,k} = 1$ for a given computing resource $p_k \in V^{cc}$:*

$$vol^{comp}(G, G^{cc}, Y) = \sum_{i=1}^N \left(\sum_{k=1}^M C_{i,k} \cdot Y_{i,k} \right) \quad (4.3)$$

Definition 2. *The communication volume of a DAG $G = (V, E)$ executed on a Digraph compute continuum $G^{cc} = (V^{cc}, E^{cc})$, is the sum of the data transfer times $T_{i,j}^{transf}$ for each $(v_i, v_j) \in E$, when $Y_{i,k} = 1$ and $Y_{j,l} = 1$ for the computing resources $p_k, p_l \in V^{cc}$:*

$$\begin{aligned} vol^{comm}(G, G^{cc}, Y) &= \sum_{\forall (v_i, v_j) \in E} T_{i,j}^{transf} = \\ &= \sum_{\forall (v_i, v_j) \in E} \left(\frac{\left\lceil \frac{z_{i,j}}{mfs} \right\rceil \cdot h + z_{i,j}}{\sum_{k=1}^M \sum_{l=1}^M bw_{k,l} \cdot Y_{i,k} \cdot Y_{j,l}} \right) \end{aligned} \quad (4.4)$$

The computation and communication volumes are combined to compute the *workload* of a distributed application.

Definition 3. *The workload $W(G, G^{cc}, Y)$ is the worst case execution time upper-bound of the DAG task $G = (V, E)$ executed on the Digraph compute continuum $G^{cc} = (V^{cc}, E^{cc})$, when considering the static scheduling strategy given by Y :*

$$W(G, G^{cc}, Y) = vol^{comp}(G, G^{cc}, Y) + vol^{comm}(G, G^{cc}, Y) \quad (4.5)$$

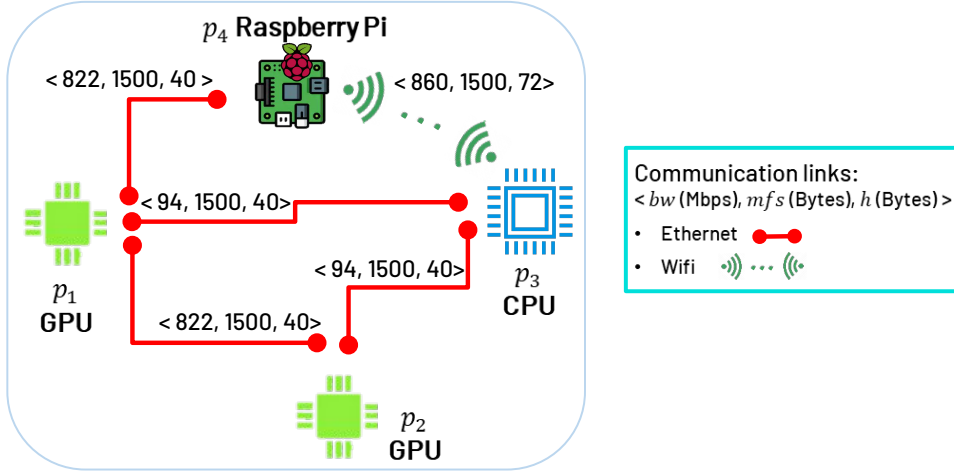


FIGURE 4.1: Compute Continuum model example and values.

4.5 Putting it all Together: a Real Example

Figure 4.1 shows an example of our *Digraph compute continuum model*, composed of four interconnected edge computing resources: two GPUs p_1 and p_2 , a multi-core p_3 , and a Raspberry Pi p_4 . The values that characterize the communication links correspond to Ethernet IEEE 802.3 [43] and WiFi IEEE 802.11ac [44] connections. The values for the non-existing link between the Raspberry Pi and the GPUs p_2 are $\langle -1, -1, -1 \rangle$.

The tables in Figure 4.2 provide the DAG task model values for the object detection and tracking application represented in Figure 3.2, when executed in the compute continuum of Figure 4.1. Concretely, the values provided are $C_{i,k}, \forall v_i \in V, \forall p_k$ in milliseconds, and the payload $z_{i,j}, \forall (v_i, v_j) \in E$ in Bytes. Notice that, for instance, the task obtaining the video frame can only be executed in one GPU, where the video streaming is processed.

For this system, an optimal scheduling minimizing the response time is $Y_{1,1} = Y_{2,2} = Y_{3,1} = Y_{4,2} = Y_{5,1} = Y_{6,3} = 1$ (corresponding to underlined values in the $C_{i,k}$ table of Figure 4.2). The Raspberry Pi is not used due to the high execution times, e.g., $C_{2,4} = 1814 \text{ ms}$ (vs. $C_{2,3} = 140 \text{ ms}$), or the incompatibility of executing tasks there, e.g., $C_{5,4} = -1$. The most time consuming data transfers correspond to the frame transmission, i.e., $z_{1,2}$ and $z_{1,6}$, being the rest almost negligible with respect to the tasks execution times. As an example, the transfer time of payload $z_{1,2}$ between nodes v_1 and v_2 , sent via an Ethernet connection between p_1 and p_2



























	Execution time $C_{i,k}$ (ms)				Edge	Payload $z_{i,j}$ (Bytes)
						
v_1 	<u>711</u>	-1	-1	-1	v_1  → v_2 	5932739
v_2 	256	<u>160</u>	140	1814	v_1  → v_6 	5932739
v_3 	<u>186</u>	211	-1	-1	v_2  → v_3 	416
v_4 	196	<u>176</u>	-1	-1	v_2  → v_4 	416
v_5 	<u>189</u>	229	-1	-1	v_2  → v_5 	416
v_6 	133	187	<u>97</u>	778	v_3  → v_6 	2175
					v_4  → v_6 	2175
					v_5  → v_6 	2175

FIGURE 4.2: DAG task model values of the DAG in Figure 3.2.

is $T_{1,2}^{transf} = \left(\lceil \frac{5932739}{1500} \rceil \times 40 + 5932739 \right) (bytes) \times \frac{1}{822(Mbps)} = 59.28 ms$. Overall, $vol^{comp} = 1520.71 ms$, $vol^{comm} = 580.62 ms$ and $W = 2101.33 ms$. and the response time upper bound for one iteration of the object detection and tracking application equals to 1406.700 ms.

Chapter 5

Task Scheduling Strategies

In this chapter we present our five different scheduling strategies based on the system model presented in Chapter 4: an optimal scheduling strategy based on *mixed integer linear programming* (MILP), and four novel tasks scheduling heuristics. The proposed scheduling strategies handle the computation and communication factors of a distributed application $G = (V, E)$ executed on a compute continuum $G^{cc} = (V^{cc}, E^{cc})$, with the objective of minimizing the overall end-to-end response time. On one hand, the MILP-based strategy optimizes the allocation of nodes $v_i \in V$ to those edge computing resources $p_k \in V^{cc}$ in which the execution time upper bound $C_{i,k}$ and the data transfers time $T_{i,j}^{transf}$ among resources are minimized. On the other hand, the heuristics prioritize the selection of nodes and resources with a similar objective, but making local decisions to speed up the scheduling process.

5.1 MILP-based Optimal Task Scheduling

Given a time-sensitive distributed application $G = (V, E)$, and a set of interconnected edge computing nodes $G^{cc} = (V^{cc}, E^{cc})$, the *objective function* of the MILP is to minimize the time interval between the starting time of the source node $v_{source} \in E$ and the completion time of the sink node $v_{sink} \in E$.

Input parameters. The input parameters considered in the MILP are the following:

1. $succ_{i,j} \in (0, 1), 1 \leq i \leq N, 1 \leq j \leq N$, a binary variable representing the edges in E . It equals to 1 if $(v_i, v_j) \in E$, 0 otherwise.

2. $z_{i,j} \in \mathbb{R}, z_{i,j} \geq 0, 1 \leq i \leq N, 1 \leq j \leq N$, the data transfer size of edges $(v_i, v_j) \in E$.
3. $zid_{i,j}$ are strings, $1 \leq i \leq N, 1 \leq j \leq N$, representing the data transfer identifiers of edges $(v_i, v_j) \in E$.
4. *source*, the index of $v_{source} \in V$.
5. *sink*, the index of $v_{sink} \in V$.
6. $ibw_{k,l} = \frac{1}{bw_{k,l}}, 1 \leq k \leq M, 1 \leq l \leq M$, the inverse of the transport bandwidth $bw_{k,l}$ of the communication link $(p_k, p_l) \in E^{cc}$. Notice that, $ibw_{k,k} = 0, \forall bw_{k,k} = \infty$, and $ibw_{k,l} = -1$ if $bw_{k,l} = -1$.
7. $mfs \in \mathbb{R}$, the maximum frame size¹.
8. $h \in \mathbb{R}$, the size of the headers².
9. $C_{i,k} \in \mathbb{R}, 1 \leq i \leq N, 1 \leq k \leq M$, the execution time upper bound of $v_i \in V$ when executing in the edge computing resource $p_k \in V^{cc}$; $C_{i,k} = -1$ if v_i cannot execute in p_k .

Problem variables. The decision variables considered in the MILP are:

1. $Y_{i,k} \in (0,1), \forall v_i \in V, \forall p_k \in V^{cc}$, a binary variable to represent the *optimal allocation function*. It equals to 1 if task v_i executes on p_k , 0 otherwise (see Section 4.4).
2. $t_i \in \mathbb{R}, t_i \geq 0, \forall v_i \in V$, the starting time of v_i .
3. $a_{i,j} \in (0,1), \forall v_i, v_j \in V$, an auxiliary binary variable that equals 1 if v_i is executed before v_j , 0 otherwise.
4. $dup_{i,j} \in (0,1), \forall v_i, v_j \in V$, a binary variable that is used to represent whether a transfer for a specific parameter has to be done or not. It equals to 1 if there exists a task v_j executed in the same resource as $v_k \in V$ such that $succ_{i,j} == succ_{i,k} == 1$ and both v_j and v_k receive the same output parameter from v_i , and $t_j \leq t_k$, 0 otherwise.

¹Given the time complexity of MILP, when different communication technologies are involved, the worst case is considered, i.e., $mfs = \min(mfs_{k,l}), \forall (p_k, p_l) \in E^{cc}$.

²Similarly to mfs , we consider $h = \max(h_{k,l}), \forall (p_k, p_l) \in E^{cc}$.

Initial assumption. v_{source} starts the execution at time instant 0: $t_{source} = 0$.

Constraints. The constraints are the following ones:

1. Each task $v_i \in V$ can be executed only by a single computing resource $p_k \in V^{cc}$:

$$\sum_{k=1}^M Y_{i,k} = 1, \forall v_i \in V$$

2. Each task $v_i \in V$ can execute in a computing resource $p_k \in V^{cc}$ if there exists implementation, i.e., if $C_{i,k} \neq -1$:

$$\sum_{k=1}^M C_{i,k} \cdot Y_{i,k} \geq 0, \forall v_i \in V$$

3. The starting time of a successor task $v_j \in V$ is greater or equal than the completion time of all its predecessor tasks $v_i \in V$ plus the corresponding data transfer time given by $T_{i,j}^{transf}$ (see Equation 4.1)³:

$$\begin{aligned} & succ_{i,j} \cdot \left(t_i + \left(\sum_{k=1}^M C_{i,k} \cdot Y_{i,k} \right) + \right. \\ & \left. + \left(\left(\left\lceil \frac{z_{i,j}}{mfs} \right\rceil \cdot h + z_{i,j} \right) \cdot \sum_{k=1}^M \sum_{l=1}^M ibw_{k,l} \cdot Y_{i,k} \cdot Y_{j,l} \right) \right) \leq t_j, \forall v_i \in V, \forall v_j \in V \end{aligned}$$

4. The execution of tasks within the same computing resource cannot overlap, i.e., if two tasks v_i and v_j are executed in $p_k \in V^{cc}$ then, either v_i finishes before the v_j starts, or vice versa:

$$Y_{i,k} = 1 \wedge Y_{j,k} = 1 \Rightarrow (t_i + C_{i,k} \leq t_j \vee t_j + C_{j,k} \leq t_i), \forall v_i \in V, \forall v_j \in V, \forall p_k \in V^{cc}$$

5. Two dependent tasks $(v_i, v_j) \in E$ ($succ_{i,j} = 1$) cannot be allocated in computing resources p_k and p_l , respectively, if they are not connected (i.e., if $ibw_{k,l} = -1$):

$$succ_{i,j} \cdot Y_{i,k} \cdot Y_{j,l} \cdot ibw_{k,l} \geq 0, \forall v_i \in V, \forall v_j \in V, \forall p_k \in V^{cc}, \forall p_l \in V^{cc}$$

³Notice that the inverse of the bandwidth ($ibw_{k,l}$) is used because the MILP implementations does not support dividing by a decision variable (i.e., $Y_{i,k}$ and $Y_{j,l}$).

6. For each task v_i that only has one direct descendant v_j such that $(v_i, v_j) \in E$ ($succ_{i,j} = 1$) and the data transfer identifier is not null ($zid_{i,j}! = "0"$), the data transfer needs to be accounted:

$$dup_{i,j} == 1 \text{ if } \sum_{k=1}^N (zid_{i,k}! = "0") == 1, \forall v_i \in V, \forall v_j \in V$$

7. Given three tasks v_i, v_j and v_l such that $(v_i, v_j) \in E$ ($succ_{i,j} = 1$) and $(v_i, v_l) \in E$ ($succ_{i,l} = 1$), that both v_j and v_l receive the same input parameter from v_i identified by the same non-null tag ($zid_{i,j} = zid_{i,l}$) and both tasks are executed by the same computing resource $p_k \in V^{cc}$, the data transfer has to be accounted only the first time this transfer takes place:

$$\begin{aligned} dup_{i,j} &\leq 2 - Y_{j,k} \cdot Y_{l,k} - (t_j \geq t_l), \\ dup_{i,k} &\geq Y_{j,k} \cdot Y_{l,k} + (t_j \geq t_l) - 1 \end{aligned}$$

Constrains (4) and (7) include the quadratic function of a decision variable, making the problem non linear, and constrain (5) includes logical functions *and* and *or*, to facilitate the explanation. Well know techniques are applied to linearize these constrains [45, 46].

Objective function. The objective function aims to minimize the execution time upper bound of the distributed application. It is equivalent to minimize the starting time plus execution time upper bound of v_{sink} . The MILP objective function also represents a valid response time upper bound of the real-time distributed workflow:

$$R^{ub} = \min \left(t_{sink} + \sum_{k=1}^M C_{sink,k} \cdot Y_{sink,k} \right) \quad (5.1)$$

5.2 Task Scheduling Heuristics

Given the time complexity of the MILP strategy (see Section 7), we propose four heuristics that have been inspired on existing approaches aiming to minimize the end-to-end response time of parallel applications for shared memory processor architectures

[8, 47, 48]. The four proposed schemes take into account two sets of *priority* rules, described in the following two subsections.

5.2.1 Heuristics Based on Successors

In the first two proposed heuristics, the internal structure of the DAG G prioritizes the *next ready task* to be allocated. A task $v_i \in V$ is ready if all its direct predecessor nodes $v_j \in V : (v_j, v_i) \in E$ have been completed.

- *Largest Number of Successors in Next Level* (LNSNL). This heuristic selects the task v_i with the largest number of direct successors, with the objective of increasing the number of nodes that become ready when v_i completes.
- *Largest Number of Successors* (LNS). This heuristic selects the task v_i with the largest number of successors, with the objective of prioritizing the execution of those portions of the DAG with the highest number of nodes, and so potentially, the largest impact on the execution time of the application.

Once v_i is selected, the LNSNL and the LNS approaches apply a *Best Fit* (BF) strategy to select the computing resource $p_k \in V^{cc}$ where the completion time of v_i is minimized. The *completion time* is computed considering (1) the start time of v_i , that depends on the last idle time of each p_k and the transfer times $T_{j,i}^{transf}, \forall (v_j, v_i) \in E$, and (2) the execution time $C_{i,k}$ on each computing resource p_k .

Algorithm 2 presents the pseudo source-code of the LNSNL and LNS scheduling heuristics. The input parameters are the distributed application G and the compute continuum G^{cc} models (line 1). The algorithm starts by initializing M , as the number of available computing resources, and N , as the number of tasks. The ready queue Q is initialized with the source node v_{source} of G (line 3) and the set of allocated tasks A is initialized to empty (line 4). An array L of size M , initialized to 0, is used to store the last idle time of each resource (line 5). For each node in G , the number of successors that each heuristic will consider is computed (line 6): in case of *LNS*, all (recursive) successors of each task are accounted, whereas in case of *LNSNL*, only the direct successors are considered.

A loop iterates (between lines 7-25) until all nodes in G have been allocated. At each iteration, a new ready task $v_i \in Q$ is selected (*nextNode*) based on the maximum

Algorithm 2 LNS/LNSNL task scheduling heuristic

```

1: procedure LNS_LNSNL ( $G = (V, E), G^{cc} = (V^{cc}, E^{cc})$ )
2:    $M \leftarrow |V^{cc}|; N \leftarrow |V|$ 
3:    $Q \leftarrow \{v_{source}\}$  // Ready Queue
4:    $A \leftarrow \emptyset$  // Set of pairs (task, computing resource)
5:    $L[] \leftarrow \text{ARRAY}(M, 0)$  // Last idle time of each computing resource
6:    $nSucc[] \leftarrow \text{COMPUTESUCCESSORS}(E)$ 
7:   while  $\sim \text{EMPTY}(Q)$  do
8:      $maxSucc \leftarrow 0$ 
9:     for each  $v_i \in Q$  do // Selects a ready task
10:      if  $nSucc[i] \geq maxSucc$  then
11:         $nextNode \leftarrow v_i$ 
12:         $maxSucc \leftarrow nSucc[i]$ 
13:      end if
14:    end for
15:     $minCT \leftarrow \infty$ 
16:    for each  $p_k \in V^{cc}$  do // Selects computing resource
17:       $time \leftarrow \text{COMPLETIONTIME}(G, nextNode, G^{cc}, p_k, L)$ 
18:      if  $minCT > time$  then
19:         $minCT = time$ 
20:         $bestRes = p_k$ 
21:      end if
22:    end for
23:     $L[bestRes] \leftarrow minCT + 1$ 
24:     $Q \leftarrow \text{UPDATEREADYQUEUE}(Q, A, G, nextNode)$ 
25:     $A \leftarrow A \cup \{(nextNode, bestRes)\}$ 
26:  end while
27:   $R^{ub} = \max_{k=1}^M L[p_k]$ 
28:  return  $R^{ub}, A$ 
29: end procedure

```

number of successors criterion (lines 9-14), which determines the priority rule that distinguishes LNS and LNSNL. The selected node is then allocated to the resource $p_k \in V^{cc}$ that minimizes its completion time, named *bestRes* (lines 15-22). The completion time is computed by the procedure `COMPLETIONTIME` considering the last idle time of each resource L , the task execution times $C_{i,k}$ on each resource, and the transfer times $T_{j,i}^{trans}$ of the data required from *nextNode* predecessors. However, $T_{j,i}^{trans} = 0$ in the case in which the identifier of the transfer $zid_{i,j}$ has already been used in the same computing resource p_k due to a prior task also requesting it, and hence, it will already be present.

Once *nextNode* has been allocated to the resource *bestRes*, the last idle time of that resource $L[bestRes]$ is updated (line 23). The procedure `UPDATEREADYQUEUE` removes *nextNode* from the ready queue and inserts the successors without any pending predecessor (line 24). Moreover, the set of allocated tasks A is also updated (line 25). The response time upper bound R^{ub} is computed by selecting the maximum of the idle times for each resource in L (line 27).

5.2.2 Heuristics Based on Processing Time

In the second set of heuristics, the minimum completion time of all ready tasks is first computed. This pre-selects a computing resource $p_k \in V^{cc}$ to execute each ready task. Then, among all the ready tasks, the next task $v_i \in V$ to execute is selected based on:

- *Shortest Processing Time* (SPT). It selects the task v_i with the shortest completion time, i.e., prioritizing the smallest nodes (in terms of $C_{i,k}$) in the fastest computing resources.
- *Longest Processing Time* (LPT). It selects the task v_i with the longest completion time, with the objective of prioritizing the biggest nodes in the fastest computing resources.

Algorithm 3 SPT task scheduling heuristic

```

1: procedure SPT ( $G = (V, E), G^{cc} = (V^{cc}, E^{cc})$ )
2:    $M \leftarrow |V^{cc}|; N \leftarrow |V|; Q \leftarrow \{v_{source}\}; A \leftarrow \emptyset; L[\ ] \leftarrow \text{ARRAY}(M, 0)$ 
3:   while  $\sim \text{EMPTY}(Q)$  do
4:      $minCT \leftarrow \text{ARRAY}(|Q|, 0); bestRes \leftarrow \text{ARRAY}(|Q|, 0)$ 
5:     for each  $v_i \in Q$  do // Selects a ready task
6:        $minCT[i] \leftarrow \infty$ 
7:       for each  $p_k \in V^{cc}$  do
8:          $time \leftarrow \text{COMPLETIONTIME}(G, v_i, G^{cc}, p_k, L)$ 
9:         if  $minCT[i] > time$  then
10:           $minCT[i] = time; bestRes[i] = p_k$ 
11:        end if
12:      end for
13:    end for
14:     $nextNode = \min_{v_i \in Q} minCT[i]$ 
15:     $L[bestRes[nextNode]] \leftarrow minCT[nextNode] + 1$ 
16:     $Q \leftarrow \text{UPDATEQUEUE}(Q, A, G, nextNode)$ 
17:     $A \leftarrow A \cup \{(nextNode, bestRes[nextNode])\}$ 
18:  end while
19:   $R^{ub} = \max_{k=1}^M L[p_k]$ 
20:  return  $R^{ub}, A$ 
21: end procedure

```

Algorithm 3 shows the pseudo source-code of the SPT scheduling heuristic. The line 2 is equivalent to Algorithm 2, and a loop (lines 3-18) similarly iterates until all nodes in G have been allocated. At each iteration, an inner loop iterates (lines 5-13) over the ready tasks $v_i \in Q$, to compute the minimum completion time of v_i on all the compute resources $p_k \in V^{cc}$, and select the one that minimizes this time (loop between lines 7-12). Among all the ready tasks, the one with the minimum completion time, $nextNode$ (line 14) is selected. Then, the last idle time of the computing resource $bestRes[nextNode]$ where $nextNode$ executes is updated (line 15). The ready queue

Q and the set of allocated tasks A are also updated accordingly (lines 16 and 17). Finally, R^{ub} is computed (line 19).

The LPT scheduling heuristic is equivalent to Algorithm 3, with the difference that, instead of the minimum, the maximum of the minimum completion time of all ready tasks is considered (line 14).

5.3 Benefits of Static Task Scheduling Strategies in Distributed Environments

The main motivation of the proposed static task scheduling approaches is the ability to provide a response time upper bound for time-sensitive distributed applications. Interestingly, as we show in the evaluation chapter, the use of such strategies also benefits the average application response time. Common dynamic scheduling strategies assign a task to a given computing resource as soon as the task becomes ready and the resource becomes idle. Only at this point in time the destination resource is known, and the data transfer of input data dependencies can be initiated. When the transfer is completed, the new scheduled task can start its execution. In the proposed static scheduling strategy instead, the data transfer can be initiated as soon as all the predecessor tasks complete, because the destination resource in which successor tasks execute is known a priori. As a result, the overlapping of computation and communication operations can be improved, so that the data input parameters of the new scheduled task are already available at the destination computing resource when the task becomes ready to execute.

However, the use of static task scheduling strategies in dynamic environments, such as the edge computing one, may seem counter-intuitive. In edge computing environments, the setup may vary in terms of available computing resources and connectivity. As we explain in next subsection, the proposed heuristics allow to quickly react to changes in the compute continuum model, and re-allocate the time-sensitive distributed applications to the new compute continuum model. Section 7.3 presents a simple experiment that shows the ability of dynamically re-configuring the system.

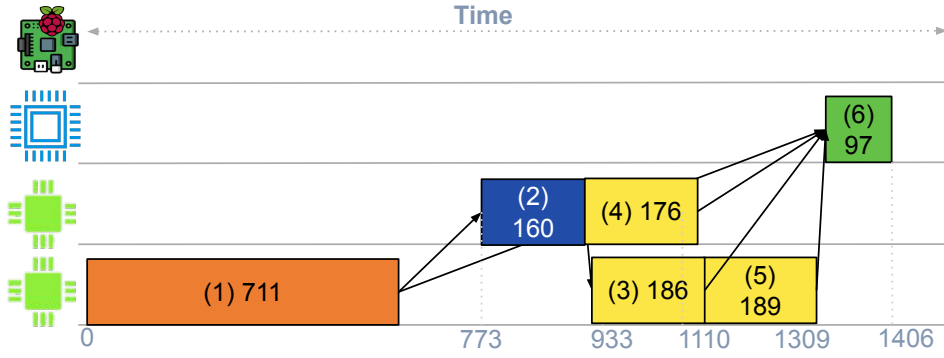


FIGURE 5.1: MILP task scheduling for the ODT application in 3.1.

5.4 Tasks Deadlines

Deadlines can be assigned to tasks by using any of the five scheduling approaches presented in Sections 5.1 and 5.2 and taking advantage of the provided end times (or start times) of the scheduling. These deadlines represent assigned points in time to each task upon which that particular task should have finished their execution.

Apart from the deadlines assigned to the tasks composing the time-sensitive application, the deadline of the last task of the DAG, that is, the sink node can be used to represent the global end-to-end deadline, the R^{ub} , used to improve the QoS of these applications.

The deadlines can be further used to define policies based on a certain threshold of deadlines misses, and upon the violation of this threshold a certain action is triggered as presented in section 6.3.

5.5 Example

Taking the application described in 3.1, Figure 5.1 represents the actual scheduling obtained by the MILP-based optimal task scheduling considering the compute continuum setup and the DAG task model values described in Figures 4.1 and 4.2.

The arrows in Figure 5.1 represent the data transfers between tasks, and each Y-axis represents the computing resources in the compute continuum. Note that from task v_2 there is only one dashed arrow to task v_3 , and not to any of the other descendants (neither v_4 nor v_5). That is because for v_4 no transfer is needed due to the fact that it is executing on the same resource as v_2 , whereas for task v_5 no data

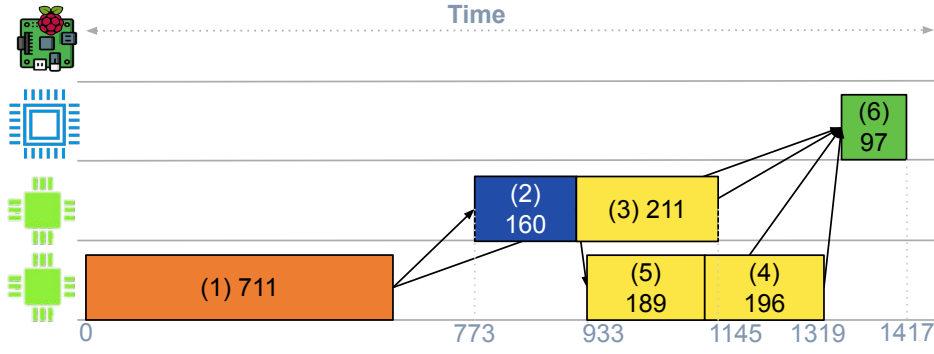


FIGURE 5.2: LPT task scheduling for the ODT application in 3.1.

transfer is needed as the same parameter has already been transferred for task v_3 . The R^{ub} obtained is represented by the last node to execute (namely the *sink* node), which in this case is 1406 milliseconds.

Figure 5.2 displays the task scheduling provided by the LPT heuristic for the same application. However, if comparing the task scheduling obtained by the MILP with the one provided by the LPT it can be seen that the scheduling decisions of the latter for the three yellow tasks (v_3 , v_4 and v_5) is the inverse of the one provided by the MILP. This decision is the one causing that the start of the execution of task v_6 is delayed, hence obtaining a different R^{ub} for both allocations, with a difference of 11 milliseconds more in the solution provided by LPT.

Both the scheduling decisions and the response time upper bound R^{ub} obtained for the rest of the scheduling strategies, that is LNS, LNSNL and SPT, are the same as the ones provided by the MILP in 5.1.

Taking Figure 5.1 and 5.2 as examples for the deadlines assigned to tasks mentioned in Section 5.4, task 1 should finish its execution before the given end time assigned by the scheduling strategies, that is, 711 milliseconds after the workflow's start. In the same sense, task 2 should finish its execution in both strategies as the deadline assigned to the task is 933 milliseconds, that is, its start time plus its execution time ($773 + 160 = 933$ ms).

Chapter 6

A New Scheduling Component for COMPSs

This chapter presents the modifications that have been applied in order to enhance the current COMPSs framework to support the proposed scheduling approaches, thus enhancing the execution of distributed time-sensitive applications. Section 6.1 presents the profiling mechanism for extracting both the application DAG and the Digraph compute continuum; Section 6.2 details the actual modifications in the COMPSs framework; Section 6.3 presents the component developed in order to monitor tasks execution at runtime; and Section 6.4 presents how these modifications allow COMPSs to react based on external events, such as connection/disconnection of computing resources.

6.1 Analysis Phase

The first step is to provide an enhanced profiling mechanism in order to retrieve both the application DAG and the compute continuum Digraph with the needed information, as described in the system model (see Chapter 4).

Digraph Compute Continuum Model. The profiling of the compute continuum setup is performed in order to obtain the characterization of the communication links between the different computing resources. This profiling process includes:

1. The actual available connections existing among them and the real bandwidth bw .

```
1 _____
2 Client connecting to 192.168.50.102, TCP port
   5001
3 CP window size: 85.0 KByte (default)
4 _____
5 [  3] local 192.168.50.103 port 59150 connected
   with 192.168.50.102 port 5001
6 [ ID] Interval      Transfer    Bandwidth
7 [  3] 0.0– 9.8 sec  1.00 GBytes  822 Mbits/sec
```

FIGURE 6.1: iPerf output for Bandwidth retrieval of Ethernet IEEE 802.3.

2. The maximum frame size mfs according to the existing technology used in the communication.
3. The header size h also based on to the existing technology used in the communication.

The available connections existing among them is represented by the bw matrix. Given two resources p_k and p_l such that there exists no a communication link between them, then $bw_{k,l} = -1$. In the rest of the cases, the real bandwidth bw is gathered by using *iPerf* [49], a tool for retrieving measurements of the maximum achievable bandwidth on networks. iPerf requires two computing resources, one acting as a server while the other as a client ¹.

The bandwidth bw between for each communication link of the compute continuum digraph is computed as the minimum observed bandwidth over a given number of executions. To setup one of the resources as a server, the following command is used: *iperf -s*, whereas the other resource uses the following command to connect to the already running server: *iperf -c \$SERVER_IP*, thus obtaining the real bandwidth as shown in Figure 6.1.

Both mfs and h are already defined values based on the technology used. For instance, for the Ethernet IEEE 802.3 [43] and the Wifi IEEE 802.11ac [44] standards, these values are 40 and 72 bytes, respectively (as shown in Figure 4.1).

Similar to the computation times, a percentage network margin parameter is also introduced in the communications time computation in order to safely guarantee the timing requirements. Concretely, this margin is added to the data transfer time

¹Given two computing resources p_k and p_l such that $k \neq l$ and are connected, we consider that $bw_{k,l} = bw_{l,k}$.

$T_{i,j}^{transf}$ (see Equation 4.1), in case the data transfer is needed (because the two dependent tasks execute in different computing resources).

DAG Task Model. On the other hand, the information regarding the COMPSs application, represented as a DAG, includes:

1. The structure of the DAG representing dependencies among tasks, i.e, the set of edges E , or similarly, the adjacency matrix $succ_{i,j}$.
2. The execution time upper bound for each COMPSs task executed in all the (supported) computing resources, $C_{i,k}$.
3. The identifiers of the data dependencies $zid_{i,j}$.
4. The payload of all data dependencies $z_{i,j}$.
5. Source node *source*.
6. Sink node *sink*.

The structure of the DAG is already provided by COMPSs which dynamically builds it at runtime. To retrieve this information another flag is used in *runcomps* which is the *-graph* (or *-g*). This flag generates a *dot* file [50] containing all the nodes composing the application and the edges representing the dependencies among them. By parsing this file using a Bash script [51] that retrieves all the dependencies among nodes and the identifiers of these dependencies, we are able to generate both the adjacency matrix $succ_{i,j}$ and tags identifying data transfers variables $zid_{i,j}$. Furthermore, both *source* and *sink* nodes are also retrieved from the *dot* file. Figure 6.2 shows the content of the dot file extracted from COMPSs that represents the DAG of the ODT application in 3.2, outlining the information used to obtain the adjacency matrix $succ_{i,j}$ and the data identifiers matrix $zid_{i,j}$.

The upper bound execution time $C_{i,k}$ is gathered by executing all tasks composing the application in the different computing resources (if the implementation is supported) while using all the logging and tracing functions provided by COMPSs. This is achieved by using the flags *-debug* (or *-d*) and *-tracing* (or *-t*) in the *runcomps* command line interface. The former activates the logging function in a verbose mode,

```
1 digraph {
2   subgraph dependence_graph {
3     1 [...];
4     1 -> 2 [label="d1v1"];
5     2 [...];
6     3 [...];
7     2 -> 3 [label="d3v2"];
8     4 [...];
9     2 -> 4 [label="d3v2"];
10    5 [...];
11    2 -> 5 [label="d3v2"];
12    6 [...];
13    1 -> 6 [label="d1v1"];
14    3 -> 6 [label="d5v2"];
15    4 -> 6 [label="d7v2"];
16    5 -> 6 [label="d9v2"];
17  }
18  subgraph legend {
19    key [label=<
20      <tr><td align="right">get_frame</td></tr>
21      <tr><td align="right">get_objects_from_frame
22        </td></tr>
23      <tr><td align="right">tracker </td></tr>
24      <tr><td align="right">collect_and_display </td>
25        <</tr>
26      </table>
27    >]
28  }
```

FIGURE 6.2: *dot* file for the DAG of the ODT application in 3.2.

while the latter activates Extrae [52], a dynamic instrumentation package to trace programs. The maximum observed execution time, over a given number of executions, for each task is then obtained by parsing the *prv* files generated by Extrae. The execution time upper bound of COMPSs tasks, $C_{i,k}, \forall v_i \in V, \forall p_k \in V^c$, is computed as the maximum observed time plus a 50% of safety margin. This is a common industrial practice that relies on software profiling reinforced by the use of safety margins [20].

Figure 6.3 presents an extract of a *prv* file generated by Extrae. In order to retrieve the execution times of the tasks composing the application, we parse the file by obtaining the custom Extrae event generated by COMPSs, 8000002. This particular event is used to indicate the start and end events of a given task, which can be obtained by the last field in the *prv* line being the task 2 in the example. The other important field is the time in which the event takes place, that is 6868968000 nanoseconds. The last step is to retrieve the end time of the task which is also represented by the same Extrae event, thus by subtracting them we obtain the observed time in this particular

```

1  2:2:1:3:2:6868968000:8000002:2
2  2:2:1:3:2:7034151000:8000000:0
3  2:2:1:3:2:7029162000:8000002:0

```

FIGURE 6.3: *prv* file for retrieving the tasks execution times.

```

1  New output value generated d1v1.IT with size 5932739

```

FIGURE 6.4: Logging file extract to retrieve the payload size produced by task 1 from 4.2.

execution for task 2, which is 160 milliseconds as in Figure 5.1 and 5.2.

Finally, the size of the data items involved in the dependencies between tasks are not accounted in the current COMPSs framework. Hence, we simply enhanced the logging information to retrieve this data, thus being able to generate the $z_{i,j}$ variable. If more than one item is involved, the Bash script that collects these results adds sizes of the different data elements as long as the parameters are not unique, that is, their zid are different as seen in Figure 6.4, which presents the logging file used to retrieve the size of the output produced by task 1 in Figure 4.2.

6.1.1 Example

Figure 6.5 shows the file for the example for the *Object Detection and Tracking* (ODT) application. The total number of tasks composing the application is described by n , whereas m contains the number of computing resources. $maxBw$ contains the maximum observed bandwidth value and it is used in the MILP model to serialize the Constrain 5.1.

6.2 Scheduling Strategies Implemented in COMPSs

The COMPSs framework has been enhanced in order to incorporate the different task scheduling strategies presented in Chapter 5.

6.2.1 MILP-based Optimal Task Scheduling

The MILP scheduling approach has been implemented with the IBM ILOG CPLEX Optimization Studio [53]. To run the model with the input file collected in the analysis

```

1  n = 6;
2  source = 1;
3  sink = 6;
4  succ = [
5      [0 1 0 0 0 1]
6      [0 0 1 1 1 0]
7      [0 0 0 0 0 1]
8      [0 0 0 0 0 1]
9      [0 0 0 0 0 1]
10     [0 0 0 0 0 0]
11 ];
12 z = [
13     [0 5.932739 0 0 0 5.932739]
14     [0 0 .000416 .000416 .000416 0]
15     [0 0 0 0 0 .002175]
16     [0 0 0 0 0 .002175]
17     [0 0 0 0 0 .002175]
18     [0 0 0 0 0 0]
19 ];
20 zid = [
21     ["0" "d1v1" "0" "0" "0" "d1v1"]
22     ["0" "0" "d3v2" "d3v2" "d3v2" "0"]
23     ["0" "0" "0" "0" "0" "d5v2"]
24     ["0" "0" "0" "0" "0" "d7v2"]
25     ["0" "0" "0" "0" "0" "d9v2"]
26     ["0" "0" "0" "0" "0" "0"]
27 ];
28 m = 4;
29 ibw = [
30     [0 822 94 822]
31     [822 0 94 -1]
32     [94 94 0 860]
33     [822 -1 860 0]
34 ];
35 maxBw = 822;
36 MFS = 1500;
37 H = 74;
38 C = [
39     [711 -1 -1 -1]
40     [256.130 160.243 140.258 1814.440]
41     [186.319 211.785 -1 -1]
42     [196.292 176.137 -1 -1]
43     [189.445 229.690 -1 -1]
44     [133.888 187.595 97.274 778.488]
45 ];
46 networkMargin = 0.05;
47 heuristic = {LNSNL, LNS, SPT, LPT};

```

FIGURE 6.5: Input file generated by the profiling mechanism for ODT application.

phase, the following command is used: `oplrun optimal_mapping.mod input_file`, where `optimal_mapping.mod` is the model file containing the MILP defined in Section 5.1. The MILP execution provides the scheduling obtained by showing the task-to-resource mapping and the order of tasks execution in the computing resources, as shown in Figure 6.6 obtained for the input file depicted in Figure 6.5. This output is

```
1 Task 1 (C = 711.291) executes on computing resource 1
   starting at 0
2 Task 2 (C = 160.243) executes on computing resource 2
   starting at 773.610330388
3 Task 3 (C = 186.319) executes on computing resource 1
   starting at 933.857976958
4 Task 4 (C = 176.137) executes on computing resource 2
   starting at 933.856976958
5 Task 5 (C = 189.445) executes on computing resource 1
   starting at 1120.176976958
6 Task 6 (C = 97.274) executes on computing resource 3
   starting at 1309.700195905
```

FIGURE 6.6: MILP output for the ODT application input in 6.5.

then parsed by another bash script and used in COMPSs to guide the allocation of tasks to resources. However, in this case, the dynamic re-configuration is not feasible due to the time complexity of the MILP (see Section 7.4).

6.2.2 Task Scheduling Heuristics

A new scheduler has been developed, which is the component in COMPSs in charge of receiving and dealing with the actual COMPSs tasks by allocating them to the different resources in the compute continuum accordingly to the selected scheduling heuristic. This new scheduler is not related with the family of ready schedulers present in COMPSs, whose behavior is depicted in 1, thus not only scheduling ready tasks. *runcompss* already provides a flag that allows to choose among the different available schedulers, allowing to select the new scheduler by adding `-scheduler="es.bsc.compss.scheduler.HeuristicScheduler"`. Moreover, this new scheduler component enhances the COMPSs runtime as it is able to anticipate (whenever possible) the data transfers of two dependent COMPSs tasks allocated in different computing resources. This is achieved by the creation of a new class, *TransferValueAction*, that advances the transfer of the data element belonging to a predecessor task to the computing resource selected to host the new task, whenever the predecessor finishes its execution, as described in Section 5.3. These heuristics require the input file presented in previous section with the information gathered in the profiling of both the COMPSs application and the compute continuum in order to provide the sub-optimal scheduling. To do so, we have used the *runcompss* flag `-scheduler_config_file` that provides the path to the configuration file upon which the scheduler receives not only

the location of the input file, but also other information such as the heuristic scheduling strategy to use.

Upon receiving the input file and the desired task scheduling strategy, the scheduler loads the content of the file and triggers the computation of the selected heuristic, which provides both a new task scheduling and the response time upper-bound R^{ub} . Moreover, this scheduling allows to obtain the start and end times in between the tasks are supposed to execute, thus benefiting the monitoring of deadlines of each task

6.3 Task Monitoring

In a real-case scenario there are other situations which generates events that trigger the re-computation of the task scheduling, for example, missing a certain amount of deadlines assigned to tasks.

However, COMPSs does not take into account the timings required to measure whether these assigned deadlines are met. In order to allow COMPSs to be aware of these times we have also enhanced the already implemented COMPSs *Task Monitor*. The Task Monitor includes methods that allow to gather at runtime the exact start and end execution times of a given task. Furthermore, it enables the enhanced scheduler to easily retrieve these timings given a specific task, thus allowing the scheduler to compare if the current times violate the assigned deadlines.

This modifications allow to enhance the execution model even more, specially in the case of the cloud in which *scaling in* (dynamically removing cloud resources) and *scaling out* (dynamically adding resources) can be performed based on the amount of these deadlines that are missed. For example, Prometheus [54], an open-source systems monitoring and alerting toolkit, is used to publish the amount of deadlines missed in a given COMPSs workflow, which provides the cloud an easy retrieval of these metrics to decide whenever to trigger the scale in or scale out policies in order to guarantee a certain QoS. As new resources are dynamically added or removed, the scheduling needs to be re-adapted to take them into account, described in the next section.

6.4 A Reactive Scheduler

The procedure that has been described so far refers to a pure static scheduling mechanism. First of all, the application and the compute continuum setup are analyzed and modeled prior to the actual execution. Then, the scheduler receives this information to compute the actual task-to-resource allocation and applies it in the COMPSs execution.

However, the compute continuum setup can be updated at runtime when new computing resources become up and running, or whenever a connection established between resources disappears. These events trigger a re-computation of the scheduling heuristic, which imposes a new response time upper bound R^{ub} , and thus implying a new scheduling, based on the new compute continuum setup.

The scheduler is informed through messages that contain the list of updated computing resources detected by another COMPSs component whenever there is a change in the resources in the compute continuum. Hence, the scheduler is able to react by updating the profiling information at runtime based on the list of computing resources received, and re-schedules the workflow considering these changes. In the case of removing a resource, the scheduler disables the information belonging to that one in particular, and hence it is not considered when the re-scheduling is applied. On the other hand, in the case of adding a resource the scheduler just enables the information so it is considered again by the scheduling heuristic.

Chapter 7

Evaluation

In this chapter we evaluate the response time upper bound provided by the heuristics and the MILP formulation, and also the average and maximum observed execution times compared with the baseline COMPSs scheduling strategies.

7.1 Experimental Setup

The five proposed scheduling strategies considered in the evaluation are presented in Chapter 5 and implemented in COMPSs as explained in Chapter 6, and the results are compared with the dynamic scheduling strategies presented in the pseudo code described in Algorithm 1.

7.1.1 Compute Continuum Configuration

We considered four computing resources: a Raspberry Pi 3, featuring a 4-core ARMv7 Processor; a NVIDIA Jetson TX2 featuring a 4-core ARMv8 host processor and a NVIDIA Pascal GPU with 256 NVIDIA CUDA cores; a NVIDIA Jetson AGX Xavier featuring a 8-core ARMv8 host processor and a 512-Core Volta GPU; and a 4-core Intel(R) i7-4600U processor. We evaluate two different configurations for the communications: (1) all resources connected through Ethernet IEEE 802.3 (labeled as *Ethernet*), and (2) the NVIDIA GPUs connected through Ethernet, and the Intel multi-core and Raspberry Pi through Wifi IEEE 802.11ac (labeled as *Hybrid Ethernet + Wifi*).

7.1.2 Applications

For the evaluation of the proposed scheduling heuristics, seven well-known HPC applications are first considered:

1. *Matrix Multiplication (Matmul)* (56 nodes), a simple application for matrix multiplication.
2. *Cholesky Factorization* (38 nodes), commonly used for efficient linear equation solvers, Monte Carlo simulations, or kalman filters acceleration (used in vehicle navigation systems to track pedestrians or bicyclists [55]), it processes a matrix of real floating-point numbers using low-level functions from the LAPACK library [56].
3. *QR Factorization* (36 nodes), that decomposes a matrix into a product of an orthogonal and an upper triangular matrix.
4. *Max Norm* (13), a simple application that computes the maximum number in a list of 16000 elements.
5. *Principal Component Analysis (PCA)* (26 nodes), applicable in machine learning and data mining fields among others, to reduce the number of variables in a dataset.
6. *Terasort* (27 nodes), a popular application that sorts one terabyte of randomly distributed data by also using the MapReduce procedure.
7. *Map/Reduce Matrix Multiplication (M/R Matmul)* (22 nodes), a matrix multiplication that is computed by applying the MapReduce procedure.

A real use case application for *Object detection and tracking (ODT)*, described in Figure 3.1, has also been considered. As input, a video of 2 minutes and 21 seconds of duration is used, with a total number of 150 frames. To increase complexity, we consider a DAG with 5 iterations (each processing a frame).

All applications are executed 50 times to compute the average and the maximum observed execution times. The execution time upper bound of COMPSs tasks, $C_{i,k}, \forall v_i \in V, \forall p_k \in V^{cc}$, is computed as the maximum observed time plus a 50% of

safety margin. This is a common industrial practice that relies on software profiling reinforced by the use of safety margins [20]. Similarly, a 5% of safety margin is used for the data transfer times. With respect to the payload $z_{i,j}, \forall (v_i, v_j) \in E$, when it is not a fixed value (e.g., if it depends on the number of detected objects), the maximum observed payload is used as a safe upper bound.

7.2 Performance and Accuracy

This section presents the evaluation of the proposed scheduling strategies in terms of the observed execution times (average and maximum) and the response time upper bound.

7.2.1 Classical Applications

Figure 7.1 shows the boxplots for the classical HPC applications on the compute continuum (*Ethernet* configuration). It shows the distribution of the execution time for the proposed scheduling strategies (*MILP*, *LNSNL*, *LNS*, *LPT* and *SPT*, and for the baseline COMPSs scheduler (*FIFO* or *LIFO*) that performs best in terms of average execution time. Moreover, the R^{ub} provided by the scheduling strategies is also depicted.

As shown, all the proposed strategies clearly outperform the baseline COMPSs schedulers in terms of best, average and maximum observed execution times. The reason is that the proposed algorithms improve scheduling by taking into account the timing information of the system model. Moreover, as seen in Section 5.3, they anticipate data transfers, enabling the overlap of computation and communication tasks. In terms of average execution time, MILP is 55%, 76%, 37%, 67%, 26%, 45% and 57%, faster than the baseline COMPSs scheduler for the Matrix Multiplication, Cholesky, QR, Max Norm, PCA, Terasort and MapReduce Matrix Multiplication applications, respectively. Similarly, the best heuristic for each application is faster by 54% (SPT), 75% (LNSNL), 37% (LNS), 62% (LNSNL), 24% (LNSNL), 40% (LPT) and 52% (SPT).

Interestingly, due to the static nature of our solutions, the execution time variation of our scheduling strategies is much smaller than the baseline COMPSs schedulers,

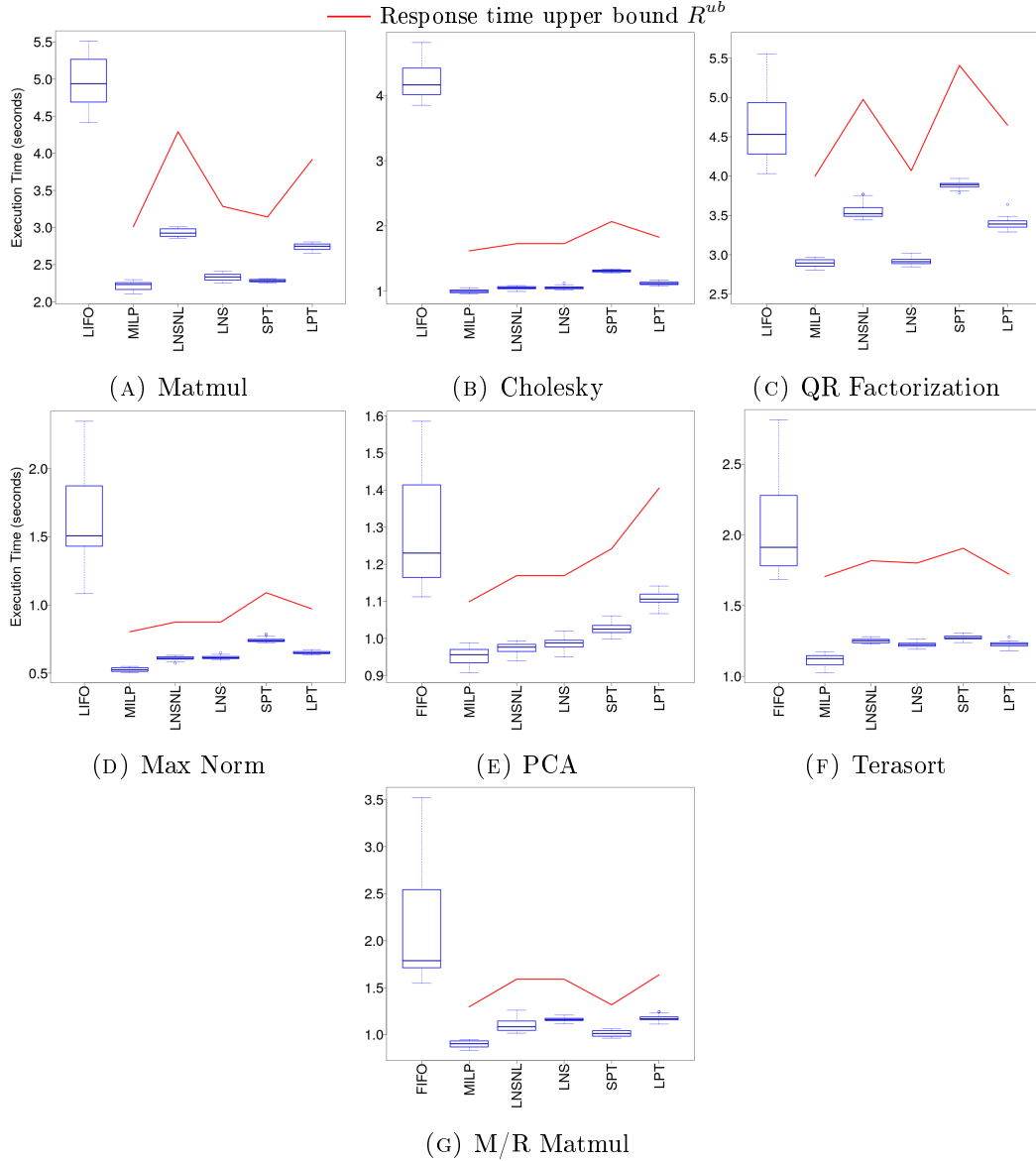


FIGURE 7.1: Boxplot of the execution time of each COMPSs applications under different scheduling strategies (Ethernet).

resulting in a more stable execution. This trend is clearly observed for all the applications.

If we now focus on the accuracy of the MILP solution and the scheduling heuristics, the R^{ub} for the MILP is 31%, 53%, 34%, 45%, 11%, 45% and 36% higher than the maximum observed execution time for the different HPC applications, respectively. A similar trend is also observed in the best scheduling heuristics for each application, being 35% (SPT), 26% (LNSNL), 31% (LNS), 38% (LNSNL), 17% (LNSNL), 34% (LPT) and 24% (SPT) higher. This overestimation allows to safely provide timing guarantees to the applications. In all the cases, the difference between R^{ub} and the

maximum observed execution time is smaller when the variability of the execution time is higher; this is clearly observed if we compare the PCA (higher variability, smaller difference) and the Cholesky (smaller variability, higher difference).

In terms of the R^{ub} estimation, the MILP solution outperforms the scheduling heuristics. The R^{ub} provided by the best heuristic for each application results in an increment of 4.33% (SPT), 6.84% (LNSNL), 1.74% (LNS), 8.78% (LNSNL), 6.38% (LNSNL), 0.97% (LPT), and 1.64% (SPT) compared to the R^{ub} obtained by MILP. Notice that the MILP strategy does not provide the optimal result for the Matmul, Merge/Reduce Matmul, QR and Terasort applications due to time restrictions, while the heuristics provide a much faster solution (see discussion in Section 7.4).

It is worth mentioning that no heuristic clearly outperforms the others, as the performance of each one depends on the application (e.g., the shape of the DAG or the execution time of all tasks). A clear example is shown for the SPT strategy, which is the best heuristic for the Matmul and M/R Matmul applications (and provides a similar R^{ub} as the MILP) but the worst by far for QR and Cholesky. The reason is that in both the QR and the Cholesky applications a bad scheduling decision makes a very long task to execute while the rest of resources are idle because all the shortest tasks have been already executed, while in the rest of heuristics the long tasks are executed in parallel with other tasks.

7.2.2 Object Detection and Tracking

Figure 7.2a shows the boxplots for the *ODT* application on the compute continuum (Ethernet configuration). Similar outcomes have been observed with respect to the classical HPC applications. Firstly, our task scheduling strategies clearly outperform the COMPSs baseline scheduler. In this case in which we have a more complex system, the differences are even more evident. The best scheduling heuristic for this application is LNS. Considering average execution times, MILP and LNS are 80.8% and 76.3% faster than the LIFO COMPSs scheduler, respectively. This is extremely important when considering applications with timing requirements, e.g., in the situation of an alert when a pedestrian is detected. Furthermore, the execution time variation of our scheduling strategies is again much smaller than the one achieved with the COMPSs scheduler. The R^{ub} of the LNS heuristic increments 10.8% w.r.t. to the MILP solution.

Figure 7.2b shows an interesting comparison when executing the application on the two different compute continuum setups, i.e., *Ethernet* and *Hybrid Ethernet + Wifi*, for the LNS strategy. Even though the execution for the Hybrid setup outperforms the Ethernet setup (in terms of minimum observed and average execution times), the variability introduced when using Wifi connections is much higher, leading to higher values for the maximum observed time.

Overall, we conclude that the proposed scheduling strategies significantly outperform the COMPSs baseline scheduling strategies. The scheduling heuristics provide comparable results to the ones obtained with the optimal but costly MILP approach. The selection of a heuristic depends on the actual setup of the system. Moreover, due to the static nature of our scheduling strategies, the execution time variability of distributed applications is significantly reduced, resulting in a more stable execution that also allows a dynamic reconfiguration, as discussed in Section 7.3.

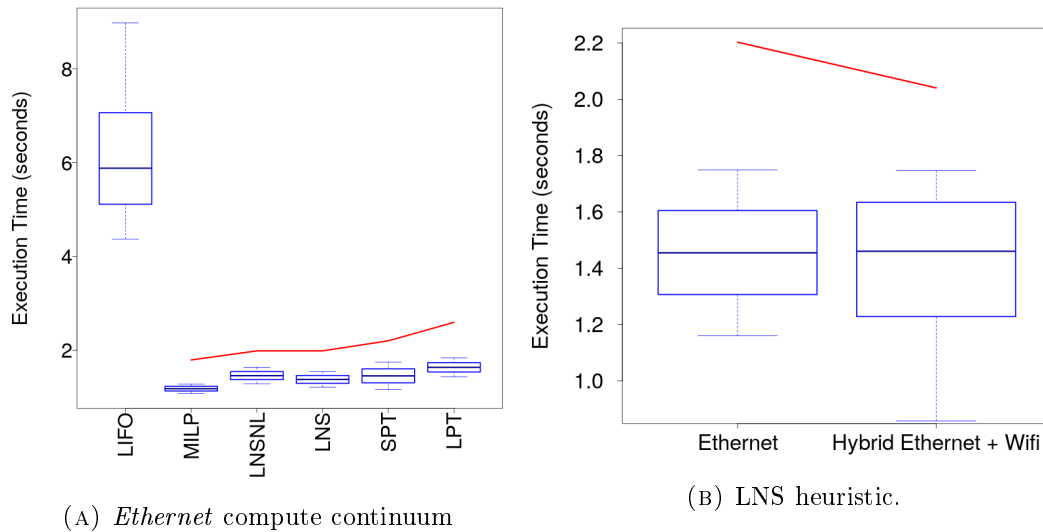


FIGURE 7.2: Object detection and tracking COMPSs application.

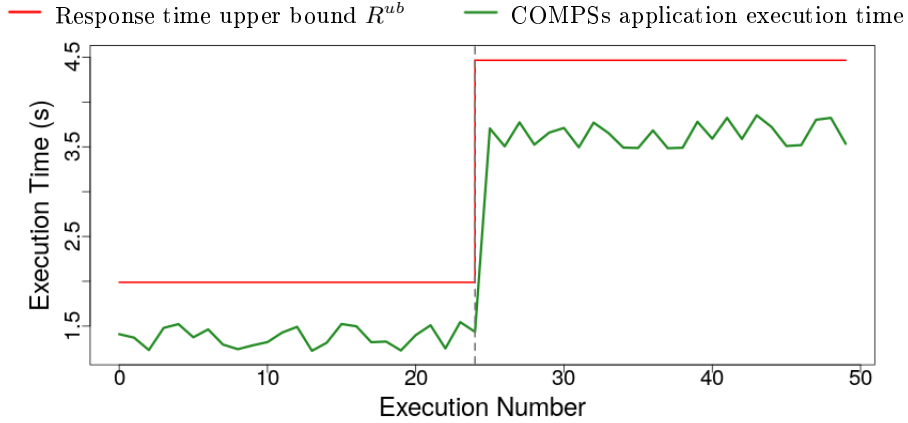


FIGURE 7.3: Execution time (multiple iterations) and R^{ub} of the object detection and tracking, in a dynamic compute continuum environment (from 4 to 3 computing resources).

7.3 Reactive Scheduling: Static Allocation in Dynamic Environments

As described in Section 5.3 and in Chapter 6, the proposed scheduling heuristics allow to quickly react to changes in the compute continuum model, appropriately re-allocating the tasks and providing a new response-time upper bound. This is demonstrated in Figure 7.3, where the evolution of the execution time of 50 executions of the object detection and tracking application is depicted. During the first 25 executions, the *Ethernet* compute continuum configuration described in Section 7.1.1 is considered. Then, we disconnect the Xavier GPUs, leaving only 3 available edge devices. This forces the re-scheduling of tasks and a new R^{ub} is provided at the 26th execution.

COMPSs		MILP		Best Heuristic	
App	N	vol^{comm}	vol^{comp}	vol^{comm}	vol^{comp}
MMul	56	6192.1182	10219.114	6539.6154	10219.114
Cho	38	1047.4992	3019.097	1222.0824	2930.571
QR	36	3102.5515	7210.551	4115.4598	12420.82
Max1	13	25.2588	1817.138	1.4789	2022.802
PCA	26	30.2914	3329.288	32.739	3413.293
Tera	26	3.0182	3899.114	3.4084	5617.271
M/RMul	22	1899.1662	3611.704	2115.0076	3724.256
ODT	32	3791.1277	4228.118	4751.2155	4469.456

TABLE 7.1: Number of nodes N , and communication and computation volumes, vol^{comm} and vol^{comp} , for each application.

Application	MILP	Best Heuristic
MMul	31692.60	1.90
Cho	133.2	0.31
QR	10966.8	1.36
Max1	1.2	0.22
PCA	34203.0	1.27
Tera	25800.5	0.44
M/RMul	26821.8	1.49
ODT	25803.0	0.44

TABLE 7.2: MILP and best scheduling heuristic execution times in seconds.

Table 7.1 characterizes the six COMPSs applications by showing the number of nodes N , and the communication and computation volumes in seconds (vol^{comm} and vol^{comp} respectively). Concretely, the table computes the volumes for the MILP and the best heuristics (SPT, SPT, LNSNL, LNSNL, LNSNL, LNS, SPT and LPT for *MMul*, *M/RMul*, *PCA*, *Max1*, *Cho*, *QR*, *ODT* and *Tera*, respectively).

7.4 MILP Complexity

The execution time to solve the MILP formulation on a 4-core Intel(R) Xeon(TM) CPU 5148 @ 2.33GHz processor is shown in seconds in table 7.2. Due to memory limitations, only the following applications obtain the optimal solution: Cholesky, Max Norm 1, PCA and ODT (the gap to explore the complete solutions space for the rest is 24%, 18%, 3% and 14% for Matmul, Merge/Reduce Matmul, QR and Terasort applications, respectively). The reasons for this is that it depends on the shape of the DAG, taking into account both the number of nodes and edges. Moreover, for the ODT application most of the combinations are limited due to the restrictions on the implementations of where tasks can execute, which reduces a lot the search space.

The same table 7.2 also presents the execution time spent on the dynamic allocation of tasks implemented in COMPSs for the scheduling heuristic obtaining the best average response time of each application. As it can be seen, the differences between the MILP and the heuristics is enormous. Taking this into account and the differences between the MILP and the heuristics solutions, it is more than reasonable to propose the use of the task scheduling heuristics.

Chapter 8

Conclusions and Future Work

The advent of the new edge computing paradigm, coupled with advanced parallel embedded processor architectures, provides an unprecedented level of computation to effectively process large amounts of data coming from distributed data sources. This new paradigm however, challenges the development and deployment of distributed time-sensitive applications due to the heterogeneous nature of the devices composing the compute continuum in edge computing, as well as the real-time analysis techniques to guarantee the response time of distributed time-sensitive functionalities, as requested in application domains such as smart cities and connected cars.

In this thesis we first propose the use of a task-based model to develop and execute *distributed time-sensitive applications* in edge computing domains. In particular, we use the COMPSs framework and extend its scheduling component to target such systems, as its current implementation is agnostic of time-sensitive applications executed. A novel system model, based on the classical DAG-based scheduling model, is proposed to characterize the applications. Also a Digraph model is used to characterize the compute continuum where both, the computation times and the data transfer times under the edge computing paradigm, are considered. A set of task scheduling strategies is proposed, based on an optimal solution provided by the MILP formulation and on a set of sub-optimal but tractable heuristics. The purpose is to minimize the execution time of the applications, while guaranteeing their timing requirements by means of a response time upper-bound. Our scheduling strategies are evaluated using a real framework, being the heuristics the preferred option due to the complexity of the MILP. Moreover, a time-sensitive object detection and tracking application has been developed in the context of this work, also being considered in the evaluation

performed. Results also reveal that, we are not only able to provide timing guarantees, but also to reduce the execution time of distributed time-sensitive applications.

Even though we provide a reactive scheduler which is able to detect changes in the compute continuum at runtime, and hence re-compute the schedule to take into consideration either the appearing resources or the disconnected ones, it still remains as future work to further exploit the benefit of using the proposed scheduling strategies, as that will allow to define policies for updating the compute continuum dynamically based on the monitoring applied at a task level at runtime, detecting the amount of deadlines missed in order to improve the QoS of the time-sensitive applications. Moreover, other metrics apart from deadline misses could be used. In edge computing, the devices are generally constrained both due to energy and computing capabilities limitations. Hence, by coupling new components able to monitor and provide alerts whenever these metrics surpass a given threshold, will allow the scheduler to react to these events by applying the defined policies and triggering a re-computation accordingly.

Bibliography

- [1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, “Edge computing: Vision and challenges,” *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.
- [2] NVIDIA, “Jetson AGX Xavier Developer Kit.” <https://developer.nvidia.com/embedded/buy/jetson-agx-xavier-devkit>.
- [3] K. Vissers, “Versal: The Xilinx Adaptive Compute Acceleration Platform (ACAP),” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 83–83, ACM, 2019.
- [4] B. D. De Dinechin, D. Van Amstel, M. Poulhiès, and G. Lager, “Time-critical computing on a single-chip massively parallel processor,” in *Proceedings of the conference on Design, Automation & Test in Europe (DATE)*, March 2014.
- [5] K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar, “Wcet measurement-based and extreme value theory characterisation of cuda kernels,” in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, p. 279, ACM, 2014.
- [6] A. Horga, S. Chattopadhyay, P. Eles, and Z. Peng, “Measurement based execution time analysis of gpgpu programs via se+ ga,” in *2018 21st Euromicro Conference on Digital System Design (DSD)*, pp. 30–37, IEEE, 2018.
- [7] M. A. Serrano, A. Melani, R. Vargas, A. Marongiu, M. Bertogna, and E. Quinones, “Timing characterization of OpenMP4 tasking model,” in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2015 International Conference on*, pp. 157–166, IEEE, 2015.
- [8] A. Melani, M. A. Serrano, M. Bertogna, I. Cerutti, E. Quinones, and G. Buttazzo, “A static scheduling approach to enable safety-critical OpenMP applications,” in

- Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pp. 659–665, IEEE, 2017.
- [9] F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Alvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia, “Servicess: An interoperable programming framework for the cloud,” *Journal of Grid Computing*, vol. 12, pp. 67–91, 2014.
- [10] R. M. Badia, J. Conejero, C. Diaz, J. Ejarque, D. Lezzi, F. Lordan, C. Ramon-Cortes, and R. Sirvent, “Comp superscalar, an interoperable programming framework,” *SoftwareX*, vol. 3, pp. 32–36, 2015.
- [11] A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill, “Multi-core real-time scheduling for generalized parallel task models,” *Real-Time Systems*, vol. 49, no. 4, pp. 404–435, 2013.
- [12] E. Quiñones, M. Bertogna, E. Hadad, A. J. Ferrer, L. Chiantore, and A. Reboa, “Big Data Analytics for Smart Cities: The H2020 CLASS Project,” in *Proceedings of the 11th ACM International Systems and Storage Conference, SYSTOR ’18*, (New York, NY, USA), p. 130, Association for Computing Machinery, 2018.
- [13] “ELASTIC: A Software Architecture for Extreme-Scale Big-Data Analytics in Fog Computing ECosystems,” 2019. URL: <https://elastic-project.eu/>.
- [14] K. De Bosschere, ed., *ACACES 2019, Advanced Computer Architecture and Compilation for high-performance and Embedded Systems: Poster abstracts*. Fabreschi Printing, 2019.
- [15] “6th Barcelona Supercomputing Center Severo Ochoa Doctoral Symposium,” 2019. URL: <https://www.bsc.es/education/predoctoral-phd/doctoral-symposium/6th-bsc-so-doctoral-symposium>.
- [16] K. Kavi, R. Akl, and A. Hurson, *Real-Time Systems: An Introduction and the State-of-the-Art*. 2009.
- [17] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, “Supporting time-sensitive applications on a commodity os,” *SIGOPS Oper. Syst. Rev.*, p. 165–180, 2003.

- [18] D. Marinca, P. Minet, and L. George, “Analysis of deadline assignment methods in distributed real-time systems,” *Computer Communications*, vol. 27, no. 15, pp. 1412–1423, 2004.
- [19] X. Liu, D. Wang, D. Yuan, and Y. Yang, “A novel deadline assignment strategy for a large batch of parallel tasks with soft deadlines in the cloud,” in *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pp. 51–58, IEEE, 2013.
- [20] E. Mezzetti and T. Vardanega, “On the industrial fitness of WCET analysis,” *11th International Workshop on Worst-Case Execution Time analysis*, 2011.
- [21] ARB, “Openmp 5.0 specification,” 2018.
- [22] NVIDIA, “CUDA C Programming Guide.” <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, October 2018.
- [23] A. Betts and A. Donaldson, “Estimating the wcet of gpu-accelerated applications using hybrid analysis,” in *2013 25th Euromicro Conference on Real-Time Systems*, pp. 193–202, IEEE, 2013.
- [24] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, “A real-time scheduling service for parallel tasks,” in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 261–272, 2013.
- [25] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah, “Analysis of federated and global scheduling for parallel real-time tasks,” in *2014 26th Euromicro Conference on Real-Time Systems*, pp. 85–96, IEEE, 2014.
- [26] M. A. Serrano, A. Melani, S. Kehr, M. Bertogna, and E. Quiñones, “An analysis of lazy and eager limited preemption approaches under DAG-based global fixed priority scheduling,” in *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 193–202, IEEE, 2017.
- [27] H.-E. Zahaf, N. Capodici, R. Cavicchioli, M. Bertogna, and G. Lipari, “A C-DAG task model for scheduling complex real-time tasks on heterogeneous platforms: preemption matters,” *arXiv preprint arXiv:1901.02450*, 2019.

- [28] M. A. Serrano and E. Quiñones, “Response-time analysis of dag tasks supporting heterogeneous computing,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.
- [29] A. J. Howard, T. Lee, S. Mahar, P. Intrevado, and D. Myung-Kyung Woodbridge, “Distributed data analytics framework for smart transportation,” in *2018 IEEE 20th International Conference on High Performance Computing and Communications (HPCC/SmartCity/DSS)*, pp. 1374–1380, 2018.
- [30] A. d. S. Veith and M. D. de Assuncao, *Apache Spark*, pp. 77–81. Springer International Publishing, 2019.
- [31] N. Maleki, M. Loni, M. Daneshtalab, M. Conti, and H. Fotouhi, “Sofa: A spark-oriented fog architecture,” in *IECON 2019 - 45th Annual Conference of the IEEE Industrial Electronics Society*, pp. 2792–2799, 2019.
- [32] J. J. Gutiérrez and H. Pérez, “Theory and practice of edf scheduling in distributed real-time systems,” in *Ada-Europe International Conference on Reliable Software Technologies*, pp. 123–137, Springer, 2018.
- [33] R. Lange, A. C. Bonatto, F. Vasques, and R. S. de Oliveira, “Timing Analysis of hybrid FlexRay, CAN-FD and CAN vehicular networks,” in *42nd Annual Conference of the IEEE Industrial Electronics Society (IECON)*, pp. 4725–4730, Oct 2016.
- [34] Q. Zhu, H. Zeng, W. Zheng, M. D. Natale, and A. Sangiovanni-Vincentelli, “Optimization of task allocation and priority assignment in hard real-time distributed systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 11, no. 4, 2013.
- [35] Y. Wu, Z. Gao, and G. Dai, “Deadline and activation time assignment for partitioned real-time application on multiprocessor reservations,” *Journal of Systems Architecture*, vol. 60, no. 3, pp. 247–257, 2014.
- [36] G. Xie, R. Li, and K. Li, “Heterogeneity-driven end-to-end synchronized scheduling for precedence constrained tasks and messages on networked embedded systems,” *Journal of Parallel and Distributed Computing*, vol. 83, pp. 1 – 12, 2015.

- [37] D. Zhang, Y. Ma, C. Zheng, Y. Zhang, X. S. Hu, and D. Wang, “Cooperative-competitive task allocation in edge computing for delay-sensitive social sensing,” in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 243–259, 2018.
- [38] H. R. Faragardi, S. Dehnavi, M. Kargahi, A. V. Papadopoulos, and T. Nolte, “A time-predictable fog-integrated cloud framework: One step forward in the deployment of a smart factory,” in *2018 Real-Time and Embedded Systems and Technologies (RTEST)*, pp. 54–62, 2018.
- [39] H. Alrammah, Y. Gu, C. Wu, and S. Ju, “Scheduling for energy efficiency and throughput maximization in a faulty cloud environment,” in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 561–569, IEEE, 2017.
- [40] Q. Wu and Y. Gu, “Supporting distributed application workflows in heterogeneous computing environments,” in *2008 14th IEEE International Conference on Parallel and Distributed Systems*, pp. 3–10, IEEE, 2008.
- [41] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta, “Pycompss: Parallel computational workflows in python,” *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 66–82, 2017.
- [42] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016.
- [43] “IEEE Standard for Ethernet,” *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, pp. 1–5600, Aug 2018.
- [44] “IEEE Standard for 802.11,” *IEEE Std 802.11ac-2013 (Amendment to IEEE Std 802.11-2012, as amended by IEEE Std 802.11ae-2012, IEEE Std 802.11aa-2012, and IEEE Std 802.11ad-2012)*, pp. 1–425, 2013.
- [45] M. F. Hasan and I. Karimi, “Piecewise linear relaxation of bilinear programs using bivariate partitioning,” *AIChE journal*, vol. 56, no. 7, pp. 1880–1893, 2010.

- [46] O. Keren, I. Levin, and R. Stankovič, “Linearization of logical functions defined by a set of orthogonal terms. ii. algorithmic aspects,” *Automation and Remote Control*, vol. 72, no. 4, p. 818, 2011.
- [47] M. Pinedo, “Scheduling: theory, algorithms, and systems,” 2012.
- [48] K. E. Raheb, C. T. Kiranoudis, P. P. Repoussis, and C. D. Tarantilis, “Production scheduling with complex precedence constraints in parallel machines,” *Computing and Informatics*, vol. 24, no. 3, pp. 297–319, 2012.
- [49] A. Tirumala, F. Qin, J. M. Dugan, J. A. Ferguson, and K. A. Gibbs, “iperf: Tcp/udp bandwidth measurement tool,” 2005.
- [50] “The DOT Language,” 2011. URL: <https://graphviz.org/doc/info/lang.html>.
- [51] P. GNU, “Free Software Foundation. Bash (3.2. 48)[Unix shell program],” 2007.
- [52] H. Gelabert and G. Sánchez, “Extrae user guide manual for version 2.2. 0,” *Barcelona Supercomputing Center (B. Sc.)*, 2011.
- [53] IBM ILOG, “Cplex optimization studio,” 2014. URL: <https://www.ibm.com/products/ilog-cplex-optimization-studio>.
- [54] B. Brazil, *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. " O'Reilly Media, Inc.", 2018.
- [55] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, *et al.*, “Towards fully autonomous driving: Systems and algorithms,” in *2011 IEEE Intelligent Vehicles Symposium (IV)*, pp. 163–168, IEEE, 2011.
- [56] E. Jones, T. Oliphant, P. Peterson, *et al.*, “SciPy: Open source scientific tools for Python,” 2001.