Departament d'Arquitectura de Computadors
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Programming Models for Mobile Environments

By

FRANCESC-JOSEP LORDAN GOMIS

Advisor: Rosa M. Badia

A dissertation submitted to the Universitat Politècnica de Catalunya in accordance with the requirements of the degree of

DOCTOR OF PHILOSOPHY in Computer Architecture.

Abstract

For the last decade, mobile devices have grown in popularity and became the best-selling computing devices. Despite their high capabilities for user interactions and network connectivity, the computing power of mobile devices is low and the lifetime of the application running on them limited by the battery. Mobile Cloud Computing (MCC) is a technology that tackles the limitations of mobile devices by bringing together their mobility with the vast computing power of the Cloud. Users can run the application on their smartphone or tablet and interact with it through its graphical interface or any of the sensors embedded on the device. Then, the application offloads the execution of the computing-intensive parts of the application onto the Cloud. Thus, the application exploits resource-richer nodes to reduce the execution time of the application. Besides, since the embedded computing elements do not compute, the energy consumption of the mobile device shrinks and its battery lasts longer.

Programming applications for Mobile Cloud Computing (MCC) environments is not as straightforward as coding monolithic applications. On the one hand, developers have to deal with the issues related to parallel programming for distributed infrastructures: application partitioning, data dependencies monitoring, scheduling the computation on the available resources and implementing an offloading mechanism to submit the execution and transfer data values. Besides, the high mobility of the devices adds two new concerns to the cost/benefit analysis: the battery lifetime and the variability of the network. The battery is a limited source of energy; therefore, the amount of energy that an application consumes is important, and developers have to be aware of it when deciding where each execution unit runs. The network variability can rapidly change the costs of transferring the data to and from the remote nodes. Handing over to a mobile data connection from a Wi-Fi network increases significantly the time to transfer a data value and the energy consumed and the price to pay per each byte of data. In extreme cases, the mobile device can become isolated from the rest of the infrastructure. Developers have to control these situations and provide the application with the necessary mechanisms to continue its execution even if the isolation becomes persistent.

As with any other distributed environment, developers turn to programming models to improve their productivity by avoiding the complexity of manually dealing with these issues and delegate on the corresponding model all the management of these concerns. This thesis contributes to the current state of the art with an adaptation of the COMPSs programming model for MCC environments. COMPSs allows MCC application programmers to code their applications in a sequential, infrastructure-agnostic fashion without calls to any COMPSs-specific API. Developers write their applications using the native language for the target platform as if they were to run on the mobile device. When the programming environment builds the distribution package for an application, it bundles a modified version along with the runtime system that supports its execution. At execution time, this runtime system automatically partitions the application written by the developer into tasks and orchestrates their execution on top of the available resources: the CPU of the device, GPUs or other accelerators embedded in it and computing resource in remote nodes. Given that the native language of the device cannot run on the GPU of the mobile, this thesis proposes an extension of the programming model to provide developers with method polymorphism: programmers can implement one method in different ways so that the runtime decides dynamically which of the available version run for each task.

Regarding the runtime system, this thesis contributes with a new architecture redesigned with the characteristics of MCC in mind. For managing the available resources holistically, the runtime runs as a service which all the applications running simultaneously on the mobile device contact for submitting the execution of their tasks. The runtime clusters the computational devices into Computing Platforms according to the mechanisms required to provide the processing elements with the necessary input values, launch the task execution avoiding resource oversubscription and fetching the results back from them. The most simple platform is the CPU Platform which has a static pool of threads to run tasks on the cores of the CPU. The GPU Platform leverages on OpenCL to run tasks as kernels on GPUs or other accelerators embedded in the mobile device. Finally, the Cloud Platform offloads the execution of tasks onto remote resources.

Hosting part of the computation on the local computing devices and offloading part of it onto remote resources forces the runtime to implement a mechanism to share data values among the nodes of the infrastructure. The shared information is potentially privacy-sensitive, and the runtime exposes it to possible attackers when transferring the data values through the network. To protect the application user from data leaks, it is necessary that the runtime authenticates both ends of network connections and encrypts and signs the content of the messages to provide communications with secrecy, integrity and authenticity.

For collaboratively exploiting both, local and remote resources, the runtime has to implement a mechanism that decides whether is worth running a task on embedded or on remote resources. For that purpose, the runtime picks one of the Computing Platforms according to the costs – time, energy and money – of running the computation on each of the platforms. Besides, in the case of the Cloud Platform, the system has to determine also which of all the nodes composing the underlying infrastructure should host the execution and when. In the case of a network breakdown that isolates the mobile device from the remote nodes, the runtime has to ensure that both parts continue with the execution. The mobile device has to respond using only the resources embedded in it, what could incur in the re-execution of computations already ran on the remote resources to re-compute some unaccessible values. Remote workers have to continue with the execution so that in case of reconnection, both parts synchronize its progress to reduce the impact of the disconnection on the application performance.

Acknowledgments

With few words, I would like to show my gratitude to all the people who helped me during this project for their support.

To begin with, I want to thank all the current and former members of the Workflows and Distributed Computing team of the Barcelona Supercomputing Center for their assistance. Rosa M^a Badia, for mentoring my academic formation from the final project of the Computer Engineering degree until the presentation of this thesis. Without her careful direction and her guidance to highlight the major problems and relevant results, this dissertation would have never been possible. Javier Conejero, Jorge Ejarque, Roger Rafanell, Cristián Ramón-Cortés, Raul Sirvent and Enric Tejedor, for their technical perspective on the main problems presented in this document. Adrià Aguilà, Javier Álvarez, Pol Álvarez, Ramon Amela, Víctor Antón, Pedro Benedicte, Arnau Canyadell, Sandra Corella, Carlos Díaz, Marc Domínguez, Fredy Juares, Daniele Lezzi, Jordi Montes, Sergio Rodríguez, Carlos Segarra, Salvi Solà and Albert Serven, for all those long discussions that helped me facing concrete problems from a different point of view or making some design decisions.

I also want to dedicate some words of gratitude to Jens Jensen and Wen-Mei Hwu, who allowed me to visit the Science and Technology Facilities Council (STFC) and the Coordinated Science Laboratory (CSL) of the University of Illinois at Urbana-Champaign (UIUC). Besides the technical education, both stays in the United Kingdom and the United States were once-in-a-lifetime experiences that I will always remember fondly together with all the life lessons learned abroad.

I am also grateful for all those people who showed interest for the thesis, but also, that made me keep a distance from the project and enjoy this period in my life beyond the academical and technical aspect. It would make a too long list to include all their names, but I would like to give a special acknowledgment and express my deepest and sincere gratitude to my family: parents, grandparents, siblings – brothers and sisters-in-law –, uncles, aunts and all the cousins; and closest friends: Lluís, Marc, Ivan and all the members of the volleyball team. Thanks for always standing there, especially on the hard times.

At last but not least, I would like to express my gratitude to the European, Spanish and Catalan societies for funding my research through their institutions. First, to the Ministry of Economy, Industry and Competitiveness (MINECO) from the Spanish Government for the support received through the projects TIN2012-34557 and TIN2015-65316-P, the grant of Severo Ochoa Program (SEV-2011-00067) and the grants of the Research Training Program received for the predoctoral training (BES-2013-067167) and mobility (EEBB-I-15-09808, EEBB-2016-11272). Also, the Generalitat of Catalunya partly supported this work through contracts 2014-SGR-1051 and 2014-SGR-1272; and the European Commission through the ASCETiC (FP7-ICT-2013.1.2 contract 610874) and mF2C (Horizon 2020 contract 730929) projects.

Wholeheartedly, thank you all.

Francesc-Josep Lordan Gomis

Table of Contents

List of Tables

List of Figures

Figure **Page**

Part I

Context

Chapter 1

Introduction

The evolution of IT technologies cannot be understood if it is not related to the computation needs of the society. The first digital electronic computers appeared in the context of World War II and were designed to speed up military processes such as computing artillery firing tables or breaking message encryption [22]. They were sophisticated, user-unfriendly machines that required several engineers to operate them to generate useful results. Governments, research centers, universities and big corporations quickly saw the potential of those systems and invested lots of money to adopt them to process more generic information: statistics on the census, bank accounts, engineering problems, ... Computers quickly evolved to the point that, at the late 1950s, one single computer was giving support to hundreds of trained users that from their terminals, simpler computers, were offloading the heavy-computing or data sensitive processes to a central unit with higher memory and computing capacity (the mainframe).

As the computational complexity of the problems faced by scientist and corporations grew, also did the demand for computational power to the IT industry. Although computer architects enhanced the capabilities of a single computer, the only affordable way to obtain such computing power was to coordinate many computers to give a quicker response. The Cluster [20] and Supercomputer technologies were the result of the research to provide IT users with platforms with a higher computing capacity.

The main issue with these high-performance systems is their economic cost of acquisition and maintenance. For this purpose, at the early 90s, the IT community developed the Grid [43], a set of technologies that allowed IT communities from all over the world to share their data, data storage space, computing power and applications while dividing the costs related to such infrastructure. Mostly adopted by scientific organizations, grids allowed the cooperation to tackle larger, more complex problems and accomplish scientific goals unapproachable without them.

The advent of the Cloud [16, 98] meant one step further in that direction. IT companies with massive computing infrastructures (Amazon, Microsoft, Google, ...) realized that a significant part of their data centers, designed to support peak demand, was unused most of the time, and decided to commercialize computing as a utility. Cloud providers offer access to isolated computing resources running on top of their large IT infrastructures as VM instances. Consequently, the Cloud converted the purchase, maintenance and operation expenses into a pay-as-you-go bill; and emerged as the solution to the computation needs of nowadays' society.

Parallelly to this computing power growth process, there has been a miniaturization of the systems. The discovery of transistors and the development of integrated circuits and microprocessors allowed not only to reduce the size of the devices and increase their capabilities but also fostered the birth of new kinds of devices. During the 1980s, all the power of a mainframe could fit into a single desk-sized device: the personal computer (PC). PCs opened the gates of the universality of IT technologies; slowly, computers put out the head in every house. Due to the high capabilities of these devices and the low requirements of applications targeted to them, developers wrote sequential codes running on their CPUs, which were growing more powerful year after year by increasing their clock frequency.

The continued miniaturization and the technical advancements in batteries made possible the appearance of portable devices, such as laptops, which grew in popularity during the 2000s. Today, their evolution, the mobile devices, dominate the market of computing devices. Smartphones and tablets have a little computing capability compared to laptops and servers. However, they stand out for their high mobility and the wide range of possibilities to interact with the user: multiple microphones, multitouch screens, cameras, positioning, and a large set of sensors such as proximity, light, compass, gyroscope, accelerometers, etc. People always bring a mobile device that connects them to the Internet and provides immediate access to computing services that support them in their work or daily life. For instance, a doctor visiting interned patients in their rooms can read on a tablet the medical history of a patient, look up the results of previous tests, check the patient evolution within the last hours, and then, decide the most suitable treatment.

Mobile Cloud Computing (MCC) [38] deprecates the centralized paradigm used in personal devices and picks up the mainframes model, where people use a simple device to interact with the application, and remote, high-performing resources host the heavy-weighted computations. It brings together the interaction capabilities and immediate network access of mobile devices with the infinite computing capacity of the Cloud. Thus, mobile users can increase the computing capacity of their devices and solve more complex computational problems. Instead of consulting the evolution of patients, doctors could simulate the impact of several treatments on them and pick the most suitable one.

1.1 Motivation

Developing parallel applications targeted to distributed environments is not as straightforward as writing sequential applications. To achieve good performance on complex applications, developers must face the technical concerns related to the parallelization and distribution of the application.

Since the appearance of multicore processors at early 2000s, programmers had to shift their mentality when developing applications: it was no longer sufficient to fit the algorithm in the computer capabilities and make it run as efficiently as possible; the workload of the application was to be distributed among several processors working at the same time. Thus, before coding the application, developers have to study the algorithm to find the parallelism inherent in it, split it into several execution units – known as jobs, tasks or threads – and determine the required data communications among them. The new code has to orchestrate the execution of these units aiming to run the maximum number of them at the same time while guaranteeing the result of the application.

Running parallel applications atop distributed infrastructures adds an extra dimension to the complexity of programming: the job scheduling [47]; i.e., assigning each execution unit to a node where to run at a particular time while trying to minimize the overall makespan. Execution units may use data values computed by other units assigned to a different node; therefore, the involved nodes need to communicate to transfer such data values from the producer node to the consumer before it reads them. Data transfers threaten the performance of the application since they add overhead to the actual computation. The heterogeneity of the system also plays a major role in job scheduling: the difference on the hardware features of each node affects the execution of each block. Job scheduling is an optimization problem in which the programmers try to maximize the number of execution units assigned to the most performing resource and minimize the additional overhead caused by data transfers.

Besides their hardware, infrastructures can also be heterogeneous on their software: different operating systems may manage the nodes composing it and require different protocols and middleware to interact with them. Heterogeneity not only affects the job scheduling but also on how nodes communicate one with each other. Programmers must be aware of which middleware is required to interact with each node and know the programming and running details of each to code the data transfers between nodes and the computation submissions using the proper software for each case.

In addition to the traditional concerns of distributed, parallel computing, mobile computing brings two new concerns to programmers. First, mobile devices are bound to a battery whose lifetime limits the execution time of the applications running on them. Hence, energy consumption becomes one of the heaviest arguments to select one implementation over many others.

The second important aspect is the high-mobility of these devices, which entails a rapid variability of the network conditions: switching mobile data protocols modify the network speed drastically and network breakdowns are likely to happen and isolate the mobile for long periods.

Applications should dynamically adapt their execution according to the current conditions to avoid harming their performance and energy-efficiency. And, in the case of network breakdown, an application should be able to keep its progress by implementing fault-tolerance mechanisms that allow it to run already offloaded computation on the computing devices embedded in the device.

Facing all the issues discussed above and taking into account all the variables requires a high level of expertise. For people coming from areas of knowledge other than parallel and distributed computing, it means to turn to experts to achieve their goals; for experts, it means spending precious time. This difficulty incurs a crucial need for parallel languages and programming models that improve the programming productivity [17, 73] by easing the writing of parallel applications for MCC environments while still achieving a performance comparable to applications written by MCC experts.

Programming model designers decide which of these issues are transparently handled by the model and which ones are exposed to the programmer depending on their objectives. On the one hand, explicit programming models offer a specific language, syntax or API through which programmers specify how to deal with the issue; thus allowing the developer to tune up the application to obtain better performance. On the other hand, implicit programming models hide as many details as possible to their users offering a more comfortable programming experience. Later, the compiler or a toolkit executed along with the application analyzes the application to manage the parallelization/distribution in the best way possible. Ideally, programming models should offer the programmability of the implicit programming models, while applications should get a similar performance as if an experienced developer coded them using an explicit model. In the end, programmers should be aware of the parallel, distributed nature of their code, but agnostic to the details of their management.

1.2 Objectives

Given the difficulties to develop MCC applications and being aware of the approach followed to ease the programming of parallel and distributed applications, the following research question arises:

Could a programming model allow developers to create an application to run on a mobile device and transparently exploit an MCC infrastructure to enhance its performance?

With the purpose of answering this question, this thesis pursues providing developers with an implicit programming model that abstracts away from the programmer the management of the parallelism inherent to the application and the exploitation of the underlying infrastructure as much as possible.

Regarding the programmability of the model, the objective is to smooth its adoption and steepen its learning curve so that developers improve their productivity directly. For that purpose, the model should offer a programming already natural to the user by building on the native language of the target platform and trying to avoid model-specific APIs to construct the application. While coding, developers should focus only on the interface with the user and the logic of their solution as if applications were to run only on the mobile device. Therefore, there is no reason for them to add any reference to the underlying MCC infrastructure on the code of the application. Applications should be infrastructure-agnostic.

Although logical algorithms are inherently parallel, the human mind conceives them more easily as an ordered sequence of operations. However, even when applications are to run on one single device, to get the most out of its computing resources, developers need to exploit this parallelism. To release developers from the additional mental and technical exercise that parallel computing requires, the model should allow programming the applications in a sequential fashion. Thus, they can concentrate on the logic of the solution to the problem of their specific area of knowledge. Generally, mobile applications already separate their logic from the interaction with the user in different threads to improve the responsiveness of the GUI. Therefore, the programming model should also allow developers to code applications using multiple execution threads and exploit the inherent parallelism on each of them.

Regarding the mechanism that converts the code written by the developers into how the application actually runs, the lack of information about the infrastructure on the code and the variability of the network conditions make a compiler unviable. A runtime toolkit has to run along with the application to transform it. This runtime analyzes the code of the application to partition it into several units of execution and detect the dependencies that exist among them. Guaranteeing the sequential consistency of the application, it should orchestrate the execution of these units on the multiple computing devices that compose the infrastructure. The runtime can always count on the computing elements embedded on the mobile: the cores of its CPU, the stream processors of its GPU or any other accelerator integrated on it; to process these execution units. Besides, the runtime can offload the computation onto remote resources reachable through the network: physical or virtual machines connected to the same local network or accessible through the Internet. To decide which resources assigns to each task, the runtime should implement a scheduling policy that considers: the execution time, so that the user gets a better application performance; the energy consumption, so that the battery lasts longer; and the economic cost.

Despite the benefits of using remote resources, using the network incurs new concerns to handle by the runtime. Developers code applications that eventually produce a result using the computing resources embedded on the mobile. Their code does not contemplate any network interactions; the programming model runtime automatically decides to make use of the resources available through the network. Therefore, the runtime has to handle all the details and problems with the network connection transparently and implement fault-tolerance and recovery methods that allow the application execution to progress even in those cases where the mobile becomes isolated from the rest of the infrastructure.

Besides, the information used and generated by mobile applications is likely to contain privacy-sensitive details about the user. The programming model should add no vulnerabilities to the applications that may expose the data on which applications work. For that purpose, all the communication across untrustworthy networks have to be secured by authenticating both ends of the connection and encrypting and signing the content of the transferred messages. If the runtime ensures the authenticity, integrity and secrecy of the network messages, in-transit data is protected from attackers.

1.3 Thesis Contributions

The main contribution of this thesis is answering the research question set out in the objectives section. Yes; a programming model can allow developers to create an application to run on a mobile device and transparently exploit an MCC infrastructure to enhance its performance. For demonstrating so, the following chapters describe a programming model along with its runtime system that achieves the objectives described afore.

Contribution 1:
Extensions to the programming model to support MCC environments

The presented model builds on COMPSs [91]: a task-based programming model with which developers can write sequential, infrastructure-agnostic applications that run in parallel on top of distributed infrastructures. Regarding the programmability of the model, this dissertation contributes in two significant points. First, the extension of the model to support task polymorphism. Thus, a task can have different versions to achieve the same purpose, not only versions implementing different algorithms but also versions targetting different architectures such as CPUs, GPUs or remote web services.

The second contribution to the programmability of the model lies in the integration of the model into the application building and packaging process of Android. To publish and distribute an application, Android bundles the application along with an application description into an Android package (apk) file. The proposal described in this dissertation extends the regular process with an additional step that performs all the necessary modifications to the content of the package so that the runtime can detect the parallelism and exploit the underlying infrastructure.

Contribution 2:
Redesigned architecture for the runtime system

Regarding the runtime of the model, this dissertation proposes a new architecture specially designed with the characteristics of MCC in mind. Despite using the same mechanism to detect the tasks composing one application, the new architecture allows the runtime to holistically orchestrate the parallelism of several applications to achieve a better exploitation of the available resources. Computing platforms group together computing resources and handle the execution of tasks on them; the runtime balances the workload among computing platforms according to the will of the application user. Thus, when the user is in a hurry and needs the result as soon as possible, the runtime fosters those scheduling decisions that pursue reducing the execution time. When the battery is low, the end user can set up the runtime to prioritize those decisions that reduce the energy consumption. In other situations, the runtime could opt for options with a good balance between the temporal, energetic and monetary costs.

Contribution 3:
Design and implementation of computing platforms

This thesis contributes with three different computing platforms. The first, and most simple one, orchestrates the execution of tasks on the cores of the CPU. By just using this platform, the application can already exploit the parallelism of the application transparently by running tasks on several cores at a time. The second platform leverages on OpenCL to offload the execution of tasks onto GPUs or other accelerators embedded on the mobile. Computing platforms have to conduct all the necessary operations to run a task on the managed resources transparently to the runtime. Thus, the OpenCL platform deals with all the memory management to ensure that all the input values are on the device so that the kernel produces the proper results, and collect the results of the kernel execution.

Finally, the third platform allows the runtime to offload task execution onto remote resources. For that purpose, it proposes a mechanisms to submit task executions and sharing data values based on a distributed hash table. The platform completes its basic functionality with a mechanism to tolerate network breakdowns. Mobile devices are likely to experience glitches on the service due to network handovers or long-lasting periods of isolation for entering in out-of-range areas. The described solution allows both, the mobile device and the remote nodes, to keep progressing on the computation autonomously. Thus, in the case of reconnection, both sides synchronize their progress with low impact on the application performance; otherwise, if the device never reconnects to the network, its autonomicity allows the device to provide the application user with the expected result.

Contribution 4:
Security on network communications

Communications due to the offloading and data sharing mechanisms may expose sensitive data by transferring it through untrustworthy networks as discussed afore. To protect the data and avoid that attackers fetch information from nodes impersonating other components of the infrastructure, the platform authenticates the ends of the connection and encrypts and signs the content of the messages. For doing so, the runtime builds on the Generic Security Services API, an interface shared by several security frameworks that allows applications to use interchangeably any of the implementing frameworks, to forward the security management to a security-specific framework. With this, an organization can publicly offer its resources and control that only its members access it and ensure that in-transit data remains protected. Besides, if several organizations federate their identity management, the members of any organization within the federation could benefit from the resources belonging to any federation partner using one single identity (Single Sign-On).

Contribution 5:
Configurable multi-objective scheduling system

All the contributions previously described focus on improving the user experience from an application point of view. The last contribution of this thesis aims to benefit the owner of the infrastructure onto which the mobile device offloads the computation. Unlike when the runtime submits part of the computation to a single laptop, on larger infrastructures, like the ones offered to the members of an organization, the scheduling of tasks may have a significant impact on the operational expences of the infrastructure. To provide infrastructure owners with some mechanism to control the resource usage and its costs, this dissertation describes a multi-objective scheduling system. With it, the platform owner can influence on the scheduling decisions to foster a shorter execution time, a higher energy-efficiency or trying to reduce the monetary cost of hosting the computation.

1.3.1 Publications Related to the Thesis

The following list contains the publications related to the thesis along with a brief summary of their content that highlights the contributions included on them.

Journals

- **Title:** ServiceSs: An Interoperable Programming Framework for the Cloud [67]
  Authors: F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Álvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia
  Journal: Journal of Grid Computing, vol. 12, no. 1, pp. 67-91
  Publisher: Springer Netherlands
  Date of publication: March, 2014
  DOI: 10.1007/s10723-013-9272-5

  This article extends COMPSs to use it in web service environments. Regarding the programming, it allows to define and use web services as the implementation for tasks. Although it introduces the possibility of implementing tasks with mechanisms other than methods written within the application, it only allows one single implementation for each. The second relevant contribution presented in this article refers to the runtime: it adapts the runtime architecture to enable the holistic orchestration of several applications (web service invocations) by splitting it into two parts. The first one detects the tasks on the application and monitors the data dependencies among them. The second part of the runtime orchestrates the execution of tasks on the available resources and dynamically adapts the amount of resources to the current workload.

- **Title:** COMPSs-Mobile: Parallel Programming for Mobile Cloud Computing [66]
  Authors: F. Lordan and R. M. Badia
  Journal: Journal of Grid Computing, vol. 15, no. 3, pp. 357-378
  Publisher: Springer Netherlands
  Date of publication: September, 2017
  DOI: 10.1007/s10723-017-9409-z

  This article presents an initial version of the prototype described in this dissertation. The programming model does not support task polymorphism yet; however, the paper describes all the necessary tools to use the model to develop Android applications. The described runtime already executes on the mobile device, but tasks always run on the CPU or offloaded them onto the Cloud. Although the architecture of the runtime is not the definitive one, it already implements the offloading, data sharing – through a distributed data directory – and network disruption-tolerance mechanisms.

- **Title:** Towards Mobile Cloud Computing with Single Sign-On Access [69]
  Authors: F. Lordan, J. Jensen, R. M. Badia
  Journal: Journal of Grid Computing, pp. 1-20
  Publisher: Springer Netherlands
  Date of publication: September, 2017
  DOI: 10.1007/s10723-017-9413-3

  This article describes the extension of the runtime to secure with authenticity, secrecy and integrity the network communications through which the runtime offloads the tasks and transfers the data. The publication details all the adaptations done to the programming model, basically on the runtime, to integrate GSSAPI on it.

International Conferences

- **Title:** COMPSs-Mobile: Parallel Programming for Mobile-Cloud Computing [65]
  Authors: F. Lordan and R. M. Badia
  Proceedings: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 497-500
  Publisher: IEEE
  Date of publication: May, 2016
  Conference location: Cartagena de Indias, Colombia
  DOI: 10.1109/CCGrid.2016.16

This article is an early version of the homonymous paper published in Journal of Grid Computing. This paper already describes the initial runtime architecture and the offloading, data sharing – with the data directory centralized on the mobile device – and a fault tolerance mechanism that only ensures the autonomy of the mobile.

- **Title:** Energy-Aware Programming Model for Distributed Infrastructures [68]
  Authors: F. Lordan, J. Ejarque, R. Sirvent and R. M. Badia
  Proceedings: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 413-417
  Publisher: IEEE
  Date of publication: February, 2016
  Conference location: Heraklion, Crete, Greece
  DOI: 10.1109/PDP.2016.39

  This article introduces task polymorphism as a mean to allow scheduling policies to control the energy consumed by the infrastructure while running an application. In this case, task polymorphism only considers implementing different algorithms running on the cores of the CPU; however, a task implementation can use more than one core of the CPU.

Workshops

- **Title:** An architecture for programming distributed applications on Fog to Cloud systems [71]
  Authors: F. Lordan, D. Lezzi, J. Ejarque, and R. M. Badia
  Workshop: 1st Workshop on Fog-to-Cloud Distributed Processing (F2C-DP)
  Workshop date: September, 2017
  Workshop location: Santiago de Compostela, Spain
  Proceedings: Euro-Par 2017: Parallel Processing Workshops. Lecture Notes in Computer Science, vol. 10659, pp. 325-337
  Editors: D. Heras and L.Bougé
  Publisher: Springer International Publishing
  Date of publication: February, 2018
  DOI: 10.1007/978-3-319-75178-8_27

  This article introduces the whole solution and discusses its suitability of the programming model for mobile Fog to Cloud environments. The proposal considers one mobile device as the cornerstone of the whole infrastructure. Applications running on it discover the nearby sensors and collect data from them. However, the programming model does not assist programmers with such interactions; they have to handle all the issues related to them programmatically. The mobile processes all the gathered information using the local computing devices, usually the CPU and the GPU, but it can delegate part of the computation and offload it onto the Cloud. The current implementation of the data-sharing and fault-tolerance mechanisms do not support offloading computation onto other peers in the Fog since they do not consider the possibility of losing data values because of the unavailability of one peer.

- **Title:** Enabling GPU support for the COMPSs-Mobile framework [70]
  Authors: F. Lordan, R. M. Badia and W. Hwu
  Workshop: 4th Workshop on Accelerator Programming Using Directives (WACCPD)
  Workshop date: November, 2017
  Workshop location: Denver, Colorado, United States of America
  Proceedings: Accelerator Programming Using Directives. WACCPD 2017. Lecture Notes in Computer Science, vol. 10732, pp. 83-102
  Editors: S. Chandrasekaranand G. Juckeland
  Publisher: Springer International Publishing
  Date of publication: January, 2018
  DOI: 10.1007/978-3-319-74896-2_5

  This article proposes an extension of the programming model that allows applications to benefit from GPUs and other accelerators embedded on the mobile device to improve their performance. For that purpose, developers have to implement the task as an OpenCL kernel and use a new annotation to let the runtime know of the implementation existence. The article describes all the necessary mechanisms so that the runtime manages the content of the device memory and submits the kernel executions transparently to the developer.

Pending for submission

- **Title:** Multi-objective Self-adaptation of Task-based Application Execution in Elastic Distributed Computing Infrastructures
  Authors: F. Lordan, P. Álvarez, J. Ejarque, R. Sirvent and R. M. Badia

  This article will describe the system designed and implemented for scheduling task executions on the available resources and provisioning the necessary resources dynamically. The policies defined in the article take into account application-level information to achieve multiple objectives related to the timespan of the execution, its energy consumption and its economic cost. In practice, the policies try to minimize one of the three parameters while meeting boundaries for the other two parameters. For instance, the policies could try to minimize the energy consumption of the application while guaranteeing that the application ends in less than one hour and spends less than 3 €.

1.4 Thesis Organization

After introducing the topic of the thesis and exposing the motivation behind it and its contributions, this first part of the dissertation continues contextualizing the work and provides some background information about Android in Chapter 2. Chapter 3 analyses the state of the art on programming models for parallel computing and, more specifically, for mobile cloud computing.

The rest of the dissertation is divided into four parts. Part II, entitled General Proposal, explains the solution proposed to ease the development of applications through one single chapter (Chapter 4) describing the programming model and giving an overview of the runtime system that supports it.

In Parts III ("Exploitation of Local Computing Resources", Chapters 5- 6) and IV ("Exploitation of Remote Computing Resources", Chapters 7- 9) the dissertation delves into the exploitation of the available resources. On the one hand, the former details the mechanisms used for running tasks on the computational resources embedded on the mobile device. Chapter 5 describes how the runtime uses the cores of the CPU of the mobile device, while Chapter 6 discusses how the prototype benefits from the computing devices embedded on the mobile other than the CPU, such as GPUs, through OpenCL.

On the other hand, Part IV describes how the runtime exploits remote resources. Chapter 7 details the mechanisms to offload task, share data and tolerate network disruptions. Chapter 8 explains the securing of the network communications with mutual authentication and message integrity and secrecy by means of GSSAPI. Chapter 9 describes the system for scheduling tasks on the remote workers to allow the owner of the infrastructure to control the operational expenses of running tasks there.

Finally, the fifth and last part of the dissertation, entitled Conclusion, wraps up the dissertation with one single chapter that lays out the conclusions extracted from the presented work and introduces possible directions to continue the research.

CHAPTER 2

BACKGROUND: ANDROID

Android is an open-source platform designed primarily for touchscreen mobile devices that offers support at all levels of the software stack, from operating system functionalities to sample applications. Android was initially developed by Android, Inc., a company founded by Andy Rubin, Rich Miner, Nick Sears and Chris White to develop "smarter mobile devices that are more aware of its owner location and preferences". In July 2005, Google acquired Android, Inc. planning to enter the mobile phone market, and in November 2007 founded the Open Handset Alliance, a consortium along with other technology companies, with the goal to create open standards for mobile devices. Since then, the Open Handset Alliance has been the developer of the Android platform.

2.1 Software Architecture

A user of an Android device sees the whole platform as a set of software application (apps) that enacts some functionalities through the mobile hardware device. Despite Android already provides some built-in applications implementing basic features like a phone dialer, an SMS client, a contacts manager, an email reader or a web browser; most of the applications are developed by third-party organizations that publish them on on-line Application stores, such as Google Play.

To ease the development of these applications, the platform offers the Application framework: a set of pre-installed blocks of software that are likely to be reused by various applications that manage the basic functions of the mobile. Thus, it reduces not only the complexity of the application code, but also their size by eliminating repetitive code. Some of the most important blocks contained in this layer are:

- **Package Manager**: a database that keeps track of all the applications currently installed on the device and allows them to interact with each other
- **Window Manager**: manages the many windows that comprise an application: notification bar, main application window or any sub-window defined in it
- **View System**: contains common UI-related elements such as tabs, buttons, icons, text-boxes, labels, ...
- **Resource Manager**: manages all the non-source code entities that compose the application: text strings, images, media content, interface layouts, ...
- **Package Manager**: a database that keeps track of all the applications currently installed on the device and allows them to interact with each other
- **Activity Manager**: coordinates and supports the navigation across different screens of applications
- **Content Providers**: set of databases that allow applications to store and share structured data across applications
- **Location Manager**: provides applications with location and movement information obtained through GPS, sorrounding Wi-Fi networks or cell tower information
- **Notification Manager**: manages the information placed on the notification bar
- **Telephony Manager**: manages all voice phone calls

Developers often use operations that are hardware-specific or performance-sensitive: calls to the operating system, surface management, media reproduction, rendering of web (webkit) and graphics (OpenGL), database operations (SQLite), security (SSL), etc. For that purpose the Application framework leverages on a set of system libraries, usually written in C or C++, optimized for the harware capabilities of the device. These libraries, also known as native libraries, comprise the library layer of the software stack. The implementation/tuning of these high-performing libraries is responsibility of the mobile vendor and may not be included on all devices.

In addition to these system libraries, the layer also includes the Android Runtime (ART) [2], which supports the writting and execution of Android applications. Android applications are written in the Java programming language. This component of the framework contains a set of reusable Java building blocks that include the basic software implementations of data structures, concurrency mechanisms, file and network IO, applications lifecycle management, webservices interaction and testing. Although programmers code in Java, Android applications do not run on a Java Virtual Machine since it is not designed to run on resource-constrained environments. Originally, Android developed the Dalvik VM [24] particularly tailored for that purpose: a register-based VM optimized for low memory requirements that relied on the underlying OS to isolate processes, manage memory and support threads while allowing multiple VM instances to run at once. Since Android 5.0 "Lollipop", the Dalvik VM is discontinued. Android incorporated Ahead-of-time (AOT) compilation which translates the Dalvik bytecode of the application to device-specific instructions at install-time. This allowed to improve applications performance on the phone without sacrificing their portability. Subsection 2.3 elaborates on how the Java code written by the programmer is transformed to run on the device.

To keep Android agnostic about low-level driver implementations, it defines a standard interface for vendors to implement, the hardware abstraction layer (HAL). Thus, vendors can implement funcionalities without affecting software of higher-levels.

The Linux Kernel layer is the bottom of the Architecture and the heart of the whole system by gathering all the core services that any Android device relies on. It provides generic operating system services such as management of memory and processes, storage and network I/O, security settings to grant/deny access to hardware devices or data. It also offers an interface that allows to plugin hardware device drivers so Android can communicate with a wide range of low-level components that are often coupled to the mobile. The layer also includes some Android-specific components that target to mobile-related issues such as the power management, memory sharing or its own interprocess communication mechanism known as binder.

Figure 2.1 illustrates the different layers of the Android Software stack described above.

2.2 Applications

As explained in the previous section, applications are how users access to the device functionalities. A user interacts with an application through the graphical interface developed by its programmer along with all the logic that supports the feature. To help on that job, the Android Framework provides four building blocks: activities, services, content providers and broadcast receivers.

Activities are designed to contain the visual interface through which users give and receive information from and to the application. By convention, an activity should support a single and individually-focused action that user can do and string together to other activities to achieve a common purpose. Tasks gather all the activities related to the same goal, not necessarily from the same application, and organizes them as a stack where the top activity is shown to the user. When the user launches an application, a new task is started along with a new activity which is pushed onto the top of the stack.

From the forefront activity, the user might need to navigate to a new activity. In this case, Android suspends the current activity, captures its current state, creates the new activity and pushes onto the top of the stack. When the activity is no longer useful, it is destroyed and popped

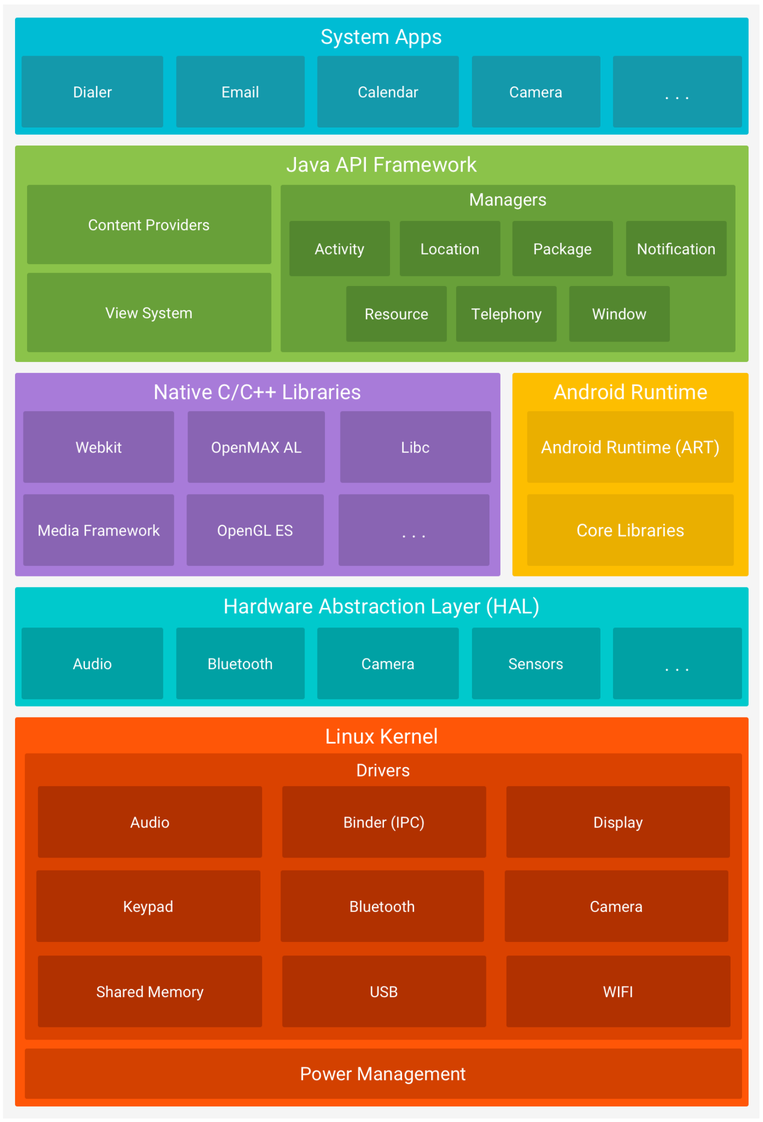


Figure 2.1: Android software stack. (Source: https://developer.android.com)

out of the stack; the new peek activity resumes its execution and is shown to the user. Developers only need to focus on the graphical elements shown to the user in each activity and the actions that need to be taken when interacting with them; the base Activity class encapsulates all the management of the task stack and the lifecycle of that particular activity.

When the application requires two components to interact, e.g. the current activity creates a new activity or opens a web page in the browser, developers turn to Intents. Intents are a class of the Android framework that represents an action to be done: create a new activity or a service, notify an event, etc. The component that the action targets can be explicitly indicated by the developer, for instance, creating a new activity of the same application; or developers may leave it implicit in the code indicating only the action to perform and Android determine which component of which application receives the action through a process known as Intent Resolution. Intent Resolution relies on a manifest (AndroidManifest.xml) attached to each application describing its components and which actions they support. At runtime, when an Intent is created and to be delivered, the system matches the action with some known component (action Id, data type and category). Multiple components can accept the same kind of intent, in that case, Android needs to choose a single one, usually asking the user.

Services are the application component designed to support long running operations without providing any user interface. Mainly they are used for two purposes: performing work in background outlasting the calling application and interaction among different processes. They can take two forms: started and bounded. Started services are started by another aplication component and run in background indefinetely even if the component that started is destroyed. Usually, a service is started to perform a single operation that does not return a result to the caller and it stops by itself when the operation completes. Other application components can bind to a service whether it was created by the same application or by another one. A bound service offers a client-server interface to interact with this service (known as IBinder), send requests and get results. Once the client component gets bound to the service it receives a stub implementation of the IBinder. Multiple components can bind to the service at once; the service is alive as long as other components are bound to it. To allow clients from different applications to access a service it is necessary to define the service interface using the Android Interface definition Language (AIDL).

The purpose of BroadcastReceiver is to enable application to react to system-wide events. Developers specify on the AndroidManifest which events the application receives and which individual BroadcastReceiver handles them. When an event needs to be broadcasted, the source application/hardware creates an event representing it with some additional information about it; Android notifies to any application component registered in the system through the Intent Resolver invoking the onReceive method on the corresponding BroadcastReceiver implementations passing the intent as a parameter.

The last building block of Android applications are ContentProviders which their main purpose is to allow data sharing across different applications. ContentProviders represent centralized repositories of structured data with data access control (specify and enforce permissions). Applications that want to access a particular ContentProvider do so through the ContentResolver class which presents a database-style interface that lets read and write data from and to a ContentProvider and supports methods such as query, insert, update and delete. To use a ContentResolver, applications identify the data they want and the content provider that hosts the data through a URI composed by a scheme (content://), an authority which indicates the specific ContentProvider, a path containing 0 or more segments indicating the specific dataset, and the ID indicating the specific record.

Android comes with a number of standard ContentProviders. For instance, the BROWSER

ContentProvider stores information such as bookmarks and browsing history; CALL LOG keeps track of the telephone calls; CONTACTS manages contact information; MEDIA keeps track of the pictures, songs and videos; and many more.

Sometimes applications need to squeeze extra performance from the device. In these cases, Android allows to build part of the application as a native library and write it in C or C++ and directly interact with the embedded devices of the mobile platform such as network interface, GPUs, accelerators or sensors. Developers dynamically load these libraries at any point of the code and they access to their functions through the Java Native Interface (JNI).

In addition to the code implementing the logic, applications are also include non-source code entities called resources; things like images, sound and video media, the layout of a screen or strings of characters. Managing them separately from the application code and using them properly has a significant impact on the portability of the application since they allow to alter the content shown to the user without need of changing the application or recompiling it. Choosing among different set of strings to be used allows applications to translate to any language. Providing a different layout for each device size or orientation (landscape or portrait) allows applications to adjust to the user interface to the current configuration. For applications to automatically adapt to the current configuration, Android requires the developer to classify the resources in a specific folder hierarchy according to their kind and the device configuration when they sould be used.

2.3 Application Package Building

Android applications are written in Java language and bundled in Android package (.apk) files for distribution that users can obtain from application stores such as Android Store. The Android Software Development Toolkit (SDK) assists the programmer on the building of the apk file from the Java code following a four steps process.

The first step, known as Android Resource Manager, scans the resources of the application and creates a Java class, named R, to ease the access resources. This class classifies all the resource of the applications in other classes according to the file hierarchy and assigns to each resource a unique identifier which is published as a constant of the R class. For instance, the identifier 0x7f020000 could correspond to an image stored in the folder res/drawable/image1.jpg and using the constant R.drawable.image1 they fetch the picture from the ResourceManager framework component. Since this R class is necessary to code the application, IDEs supporting Android programming run this step everytime that a new resource is added to the application.

During the second stage, the Android Pre Compiler, the Android SDK scans all the project looking for AIDL (Android Interface Definition Language) files to generate all the proxy-stub classes required for interprocess communications. At this point, the application has all the Java classes the compose it and they are compiled by the Java compiler to generate Java bytecode on a third stage known as Java Builder.

Finally, the Android Package Builder is the fourth and last stage of the apk building process. Since originally Android applications were running on the Dalvik VM the Java bytecode of all the classes is translated into Dalvik bytecode and stores as a classes.dex file. This file is bundled together along with the AndroidManifest.xml and all the resources into the apk file.

If the application uses native libraries that are not already provided in the device, they are also included in the application package. Libraries to be compiled for the specific device architecture and added into the bundle as dynamic libraries (.so files). To compile the C and C++ codes, Android offers a set of tools named Android Native Development Kit (NDK) that allow to cross-compile the given libraries targeting mobile platforms from any computer. So the application can be portable across multiple architectures, a different versions of the library for each compatible platform needs to be included in the distributable; what increments its size.

Originally, to install the application, the Android Package Manager uncompressed the apk file to copy the .dex file, the resources and any native library that should be included into the file system, and register the application along with all its components, supported intents and required permissions. When the user launched an application from the home screen, a new process was created in the Dalvik VM running the main component of the application. Since Android 5.0 "Lollipop", this procedure changed and incorporated ahead-of-time (AOT) compilation. At installation time, upon package decompression, a tool named dex2aoc compiles the Dalvik code (classes.dex file) optimizing the code for the specific device. Hence, the Dalvik VM is no longer necessary since applications are already composed of instructions supported by the specific processor; thus, removing the overhead of virtualization and improving the performance of the application.

2.4 Process and Thread Management

Processes are self-contained execution environments that have some assigned resources: memory, open files, network connection. Within a process, there can be multiple sequentially executing streams of instructions with its own program counter and call stack, known as threads. Since all of them belong to a single process they share its resources such as the heap and the static memory areas.

When an application is launched, the system creates for the application a new process with a single thread, known as the main thread or UI thread, which creates the main component of the application (usually an activity). By default, this thread hosts the execution of all the components within the application; Android does not create a different thread for each instance of an application component. Consequently, methods that handle user actions, respond to external events and manage the lifecycle of the components run in the UI thread. When an application performs compute-intensive work in response to an event, whether coming from the UI or from an external origin, having a single thread yields low responsiveness and poor performance.

To avoid blocking the UI thread, developers offload the heavy computation to separated threads. On the one hand, they can manually create and manage these new threads as they would do in any regular Java application: instantiating and starting a subclass of the Thread class or creating a new Thread from an implementation of the runnable interface. Since the Android framework is not thread-safe, it only allows the UI thread to directly modify any element visible in the GUI. As a workaround to publish the results of the computations done by worker threads, the Activity class contains methods that enforce the main thread to execute some methods.

On the other hand, the Android framework provides classes that keep developers agnostic to thread management while running operations on background threads and publishing their results on the GUI. One example is the AsyncTask class which allows to define one operation and submit multiple executions of the operation. A background worker thread retrieves these operations from a queue and executes them sequentially. Eventually, the worker publishes the progress of the running operation, and the operation result upon its end. The programmer defines the reaction of the UI thread to both events.

Android also offers to developers the possibility to run different components of an application in isolated processes by specifying that as an attribute in the application manifest. Although, the new process has its own main thread, components no longer share resources and communications among components require IPC mechanisms to interact (through AIDL). This approach is widely adopted to host services that are shared across applications. When the call to a bounded service originates in the same process where the service runs, the method is executed by the calling thread as a regular method. However, if the call is from a remote process the method executes in a thread chosen from a pool that the system maintains in the process of the service. Since multiple calls can be dispatched at the same time, the implementation of an AIDL interface must be completely thread-safe.

At some point of the execution, Android may shut down a process due to a lack of resources. To determine which process to keep and which to kill, the system defines an importance hierarchy based on the components running in the process and their state. It defines five levels of importance:

1. Foreground process: the interacting activity and services bound to it, services in foreground or executing a lyfecycle management callback, and BroadcastReceiver handling an event.

2. Visible process: Activity visible and services bound to it.

3. Service process: Any service-related process not included in any previous categories.

4. Background process: process holding an activity that is not currently visible to the user. They are removed following a least recently used policy.

5. Empty process: process that does not hold any active application components.

Chapter 3

State of the Art

The real world is inherently parallel; multiple events often happen at the same time independently one from each other: planets and asteroids orbit around stars, chemical reactions take place within living organisms, atomic particles move, trading in stock markets, ... Thus, models describing these phenomena are inherently parallel.

Conversely, whether the natural form of human cognitive processing is serial or parallel is still a controvert topic among experts [40, 77, 83]. Speech is a sequence of phonemes one after the other; what forces us to transmit ideas and steps within a process one after the other. Monologue interior is non-vocalized speech; hence, it is also serial. And even the stream of consciousness is serial [19] for events are sequenced in time. Consequently, programmers often implement such models as sequential logico-mathematical processes.

Fundamental physical limits on the technologies used for implementing the computing units cap the performance of sequential computing. The way for current processors to work around these limits and speed up the calculation is to exploit the parallelism of the models and process their operations concurrently. However, the parallelism inherent in a process is finite; often, operations depend on the results computed by another operation enforcing their serial execution. These sequences of operations set a theoretical limit on the speedup that an execution can achieve with parallelization; computer scientists predict the expected execution time of a parallelized application by means of Amdahl's [15] or Gustafson's [50] laws.

Although parallelism is a concept easy to comprehend – much of the human perceptions happen in parallel –, conceiving parallel algorithms is a hard job for the brain. This chapter delves into the sources of parallelism and discusses around the architecture and programming of systems capable of exploiting it.

3.1 Sources of Parallelism

Almasi and Gottlieb define parallel computing in [14] as a type of computation in which many calculations or the execution of processes are carried out simultaneously. According to the grain of these calculations, there are three different levels of parallelism.

The finest possible grain of parallelism (bit-level parallelism) lies on the implementation of a single operation, within the bits that represent the operands. For instance, an 8-bits addition can be represented as a single operation or as a sequence of the addition of the four least significant bits and the addition of the most significant bits plus the carry of the previous addition. Historically, exploiting bit-level parallelism was the technique used to increase the performance of processors in the early days of computer architecture. The most common word-size for nowadays processors is 64 bits.

A coarser grain of parallelism originates in the stringing of several operations; more specifically, the stream of processor instructions that compose a sequential program (execution thread). The potential overlap among these instructions is called instruction-level parallelism (ILP) since the instructions can be executed in parallel [51]. To increase the operation throughput of the processors, the cycle that each instruction had to go through was segmented in several stages (instruction pipelining). That allowed, on the one hand, to shorten the cycle time to the length of its longest stage, and therefore, to increase the clock frequency; and, on the other hand, to reuse the resources dedicated to a previous already performed stage to process a posterior instruction prior the completion of the instruction. The more stages compose the pipeline, the more instructions potentially execute in parallel as depicted in Figure 3.1. Another technique used in processors design to exploit ILP consists in increasing the resources on a stage with the purpose of hosting several instructions at a time (Superscalar Processors). A simple example of a superscalar processor doubling the capacity of every stage in the pipeline achieves a higher performance; doubling the number of instructions across the pipeline could potentially double the instruction throughput of the processor.

	Clock Cycle										
	1	2	3	4	5	6	7	8	9	10	11
Instruction 1	IF	ID	E	M	WB						
Instruction 2		IF	ID	E	M	WB					
Instruction 3			IF	ID	E	M	WB				
Instruction 4				IF	ID	E	M	WB			
Instruction 5					IF	ID	E	M	WB		
Instruction 6						IF	ID	E	M	WB	
Instruction 7							IF	ID	E	M	WB

Figure 3.1: Parallel execution of 7 instructions in a basic five-stage (**I**nstruction **F**etch, **I**nstruction **D**ecode, **E**xecution, **M**emory Access, Register **W**rite **B**ack) pipeline. In the fifth cycle, 5 instructions are running in parallel, the fist has already completed its execution and the last has not entered the pipeline.

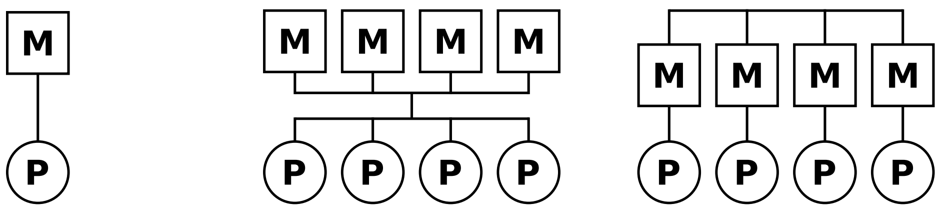
Programs running on pipelined processors cannot achieve its ideal performance because there are situations, called hazards, that prevent the next execution in the stream from executing during its designated clock cycle and stall the execution. The processor is said to be in a structural hazard when resources dedicated to that stage cannot accommodate all the concurrent instructions. For instance, when a load instruction takes several cycles to fetch the value from memory, the whole processor needs to stop since the following instruction will not be able to get into the MEM stage. Instructions in a program are likely to refer to data used in previous instructions of the code; this relation between two instructions is known as data dependency. Bernstein [23] distinguished three situations where two instructions could not be executed in parallel: flow or read after write (RaW) dependencies, an instruction stores a value on a memory location read by a succeeding one; anti-dependence or write after read (WaR) dependencies, an instruction reads from a memory location that a succeeding one writes on; and output or write after write (WaW) dependencies, both instructions write a value to the same memory location. To ensure the sequential consistency of the program, the processor stalls until the proper value is set into the memory location in what is known as data hazard. Finally, when a branch instruction is executed, the processor has no way to ensure which will be the following instruction to run until the new program counter is computed.

Performance penalties due to hazards can be fought both at software and hardware level to improve the exploitation of the ILP. Compilers can emulate the processor behavior and reorder the instruction sequence to delay an instruction until the hazard has been resolved. At hardware level, out-of-order processors allowed to avoid processor stall by delaying the execution of an instruction while executing a succeeding one in its place. However, introduced the possibility of WaW and WaR hazards, that could not happen in the pipeline processors. The most well-known algorithms for dynamic instruction scheduling are Scoreboard [93] and Tomasulo [94].

Finally, the coarsest grain of parallelism, known as thread-level parallelism, arise from the multiplicity of independent streams of processor instructions that can run at a time. Applications are likely to consist of several threads either because of the different instruction sequences composing it (task parallelism) or because of multiple instances of the same sequence each processes a different element of a data set (data parallelism). The exploitation of this kind of parallelism started back to 1970's with vector processors, predecessors of nowadays GPUs, and was the main reason behind the development of multithreaded and multicore processors.

3.2 Parallel Systems

John von Newmann described an *automatic digital computing system* as a device to carry out instructions to perform all-purpose calculations of a considerable order of complexity. In his design, known as von Newmann or Princetown architecture, the system has two main components: the central processing unit (CPU or processor) and the memory. The memory is a device that stores



(a) Von Newmann (b) Shared-memory (c) Distributed-memory

Figure 3.2: Architectures of computing systems.

information. The CPU contains all the necessary organs to perform arithmetic operations (Central Arithmetic) on values read from and stored in the memory and control the proper sequencing of the instructions – also stored in the memory – (Central Control).

Foster defines parallel computers as "a set of processors that are able to work cooperatively to solve a computational problem" [42]; a third component joins the computer architecture: a network to interconnect the processors and the memory modules. The organization and nature of this interconnection network define the major types of parallel computers.

In shared-memory systems, all the processors access to the memory as one single common device composed of multiple memory modules as depicted in the center of Figure 3.2. The organization of the memory modules and the latency for processors to access them creates a whole taxonomy of architectures within shared-memory processors including uniform memory access (UMA), non-uniform memory access (NUMA) and cache-only memory architecture (COMA).

An alternative model, depicted in the right-most part of Figure 3.2, is distributed-memory systems, where each processor has its own (local) memory module which it interacts with to read and write values. For reading and writing values on remote (related to another processor) memories, the processor may send and receive messages through the network. In these systems, accessing to remote values is more expensive than accessing the local memory. The magnitude of this difference lies in the scope of the interconnecting network. It can be few nanoseconds when the network covers few centimeters as happens in heterogeneous systems composed of a CPU and a GPU. In clusters, where the scope of the network is a few meters, latencies grow to milliseconds; and on more loosely-coupled infrastructures where the interconnection network is the Internet, such as environments using Grid or Cloud resources, latencies can grow to tenths of a second.

3.3 Parallel Programming

As noted at the beginning of this chapter, conceiving parallel algorithms for resolving computational problems requires a significant effort. Besides the intellectual part of developing an application, there are the technical concerns related to the actual implementation of the software. Developers need to map the different operations of the application to sequences of instructions understandable for the hardware that computes it.

Programming models are abstract machines that aim to separate the developer's natural model of an application from the concerns of its parallel execution. For that purpose, they provide application developers with an interface with high-level operations, while implementing them on the underlying infrastructure. Abstraction simplifies the structure of the software and releases programmers from learning the intricate details of the architecture and allows the increase their productivity by focusing on specific domains of knowledge.

The second benefit of abstraction is portability. If the software builds on a standard interface to an abstract machine, it can run on any hardware system able to emulate the behavior of the abstract machine described by the model.

Skillicorn and Talia identify four aspects of parallel computing that programming models should conceal: [90]

- Decomposition of the application into pieces (tasks) to run on the processing elements.
- Mapping of each task to the processor that hosts its execution. The capabilities of each specific processor, the size of the data and the speed of the interconnecting networks are factors likely to influence the placement decision. This is a well-known problem widely studied in the bibliography known as Job or Task Scheduling.
- Communication among these tasks to transfer data values.
- Synchronization among tasks so that they all know that they have jointly reached a common state.

If a programming model can successfully abstract these four aspects, that would mean that code could need no references about parallelism; hence, making it implicit in the program.

Another element to consider when implementing the high-level operations on the infrastructure is the interconnecting network. In embedded systems, network controllers are hardwired to achieve high performing networks. To reduce the space of the chip dedicated to the implementation of the protocols, the interacting mechanisms are very rigid: a single well-defined low-level communication protocol is implemented. On the contrary, more loosely-coupled systems, such as grids, the networks are more flexible. The processing elements may be available through different protocols such as SSH or web-services. The interoperability of different processing elements is an additional issue orthogonal to the application parallelism; the translation mechanism from the high-level operations to the infrastructure should also hide away from the programmer the heterogeneity of the system.

The endeavor for programming model developers is to raise the level of abstraction while delivering performance. [17] An abstract model is not of practical interest if the execution of programs written in it is not efficient. Lower levels of abstraction may achieve performance but at the cost of worsening the productivity; industry would dismiss its usage as well.

The following subsections discuss how different programming models (targeting distributed systems) deal with the parallel programming problem and with the heterogeneity of the system.

3.3.1 Handling the Parallelism

When creating new programming models, designers decide for each of the four aspects mentioned above whether it is visible for the programmer to fine control it through explicit API calls or annotations in the code, or it is implicit in the definition of the model which handles it opaquely. Although the individual decision for each aspect is independent of the other three, generally, there is one order in which programming models tackle them as the abstraction level raises.

Historically, the first programming models were developed by hardware manufacturers and aimed to maximize the efficiency of their hardware (hardware-centric). These models gave little importance to programming productivity, and developers have to specify the management of the four issues. Two examples of explicit programming models popular nowadays are OpenCL [81] and CUDA [79], both targeting heterogeneous platforms composed of CPUs, GPUs and other types of accelerators. On both models, the application runs on the CPU and certain parts of the application (kernels) are offloaded to one of the accelerators. To submit a kernel execution, the developer needs to explicitly interact with the target device to copy the necessary input values to the memory from the CPU memory to the device one, command the execution of the code and get back the results to the CPU memory. Sequentializing these operations is also a responsibility of the programmer.

The first aspect that programming model usually abstract away from the user is the mapping of the pieces of the application to the processors of the infrastructure. The developer is making almost all the implementation decisions but where to run each piece of the code. Message Passing Interface (MPI) [36, 92] is a standard that defines a protocol for point-to-point and collective communications. The software developer partitions the applications into several threads, each of which has an exclusive memory space; the runtime library assigns each thread to one of the processors available in the architecture. All the threads run in parallel (concurrently, if the number of threads is greater than the number of processors) and the developer calls the MPI to send to or receive from other processors data value. Threads can synchronize using either communications or calling specific operations like barriers.

The following level of abstraction consists in releasing the developer from making fine-scale timing decisions. Probably the most well-known type of programming models in this abstraction level are the ones that allow the developer to define process networks. A process network consists of a set of entities that react to the arrival of data and potentially sending new data to other entities. In this case, the developer still needs to partition the application into pieces (the entities) and indicate when an entity sends a message to other entities. Synchronization among entities is implicit within the semantics of the communication; the reception of a message is what triggers the computation of an entity.

A successful model to implement process networks is the actors model [12, 52]. Actors are entities with a state that react to the reception of a message. The response given to the message depends on its behavior: a function that determines the actions – changing the current state, sending messages to already existing actors, creating new actors or even destroying other actors that they created – the actor takes according to its current state. An actor processes messages sequentially; messages received during the processing of a preceding message are left in an exclusive mailbox for its later processing. An easy way to understand what actor systems are is to compare them with organizations composed by many people who perform tasks efficiently: actors. When an actor receives a big task, it might divide it into multiple pieces and hire some more actors to compute them. Since the employer is responsible for the execution of the bigger task, it supervises the job of all its employees. While tasks are big enough to be divided, these employees, in turn, can divide their tasks and hire new actors creating a hierarchical structure. When an actor realizes that employed actors are no longer necessary to perform the task assigned to it, the employer can fire its employees destroying the corresponding actors. Two successful implementations of this model are Akka [49] and Erlang [100].

One further step towards full abstraction consists in hiding communications to the developer. A simple approach to achieve this goal is to restrict the communications between tasks. For instance, a model where all the tasks can run in parallel independently of each other. Since tasks cannot communicate, the model does not need to provide any mechanism to command a communication between tasks explicitly. This model, known as bag of tasks, suits well for embarrassingly parallel applications; however, the model does not fit for applications presenting different schemas. Aneka [99] is a framework implementing this programming model.

Another possibility consists in considering tasks as units of work that require and produce data values. For producing their results, tasks often require data values that are results of other tasks. Once developers have identified all the tasks composing the application, they have to describe the flow that task have to follow – known as workflow – so that the system produces the expected result. If developers specify the values that a task requires and produces; the model can automatically infer and manage the required communications. Dryad [55] and Pegasus [33, 34] are two programming models that define a language to construct the workflow; JOLIE [76], Taverna [74] and WS-BPEL [10] allow the developer to construct the workflow via a graphical interface.

A different approach is using shared spaces of memory where tasks can publish values and fetch them from there. One example is the platform offered as a cloud service by Microsoft: Azure [5]. The Azure model spins around two data structures: blobs (binary large objects) and queues; all the instances of these structures are publicly available from all the processing elements. In this case, developers split the application into several functions; each encapsulated within a process (compute) that constantly polls a queue to obtain the parameters to run the function. Computes fetch the input values for that task – identified by a unique name – from the shared memory as blobs; and at the end of the function execution, it stores the results as blobs on the shared space to close the cycle. Although software developers do not need to specify communications among tasks directly, they still need to add all the tasks composing the software into the queue corresponding to the function to execute.

To conclude the classification of programming models according to the level of abstraction, there are those models that hide away from programmers all four problems. Despite this achieving full opacity, some programming models still require the software developer to make explicit the parallelism. For instance, that is the case of programming models, like OpenMP [31], that exploit parallel regions in a fork-join execution model. For instance, parallel loops in OpenMP, developers specify that the iterations of a loop can run in parallel and the number of used threads to compute them. The actual decomposition of the loop in tasks is made by the runtime since the number of iterations executed on each processor may change to balance the computational load.

Conversely, many other programming models achieve full opacity and the software developer does not even need to be aware of the fact that the application runs in parallel.

As for communications, one solution goes through restricting the workflow that developers can define. Algorithmic skeletons implement standard algorithms that can resolve multiple problems. The intellectual difficulty of this approach lies in translating the problem to resolve by the software to the problem resolved by the algorithm. Once this reduction is done, the programmer only defines the behavior of the different methods composing the skeleton algorithm. MapReduce [32] is a well-known programming model following this approach used for processing large sets of key-value pairs. Software developers only need to define two methods Map and Reduce. Map takes a partition of the input set and processes all the contained pairs to produce an intermediate set of key-value pairs containing a partial result of the computations. Reduce is a function that defines how to merge all the partial results for a given key into a single value.

Some computations may be hard or even impossible to fit in an algorithmic skeleton. A more generic approach consists in automatically construct the workflow out of a sequential code. Completely automatic application decomposition is hard, these programming models provide software developers with a mechanism to determine which logical regions of the code create a new task when invoked. As for the models building the workflow, these models can automatically detect the dependencies among tasks by considering the input and output values of each task. In this cases, the usual execution model consists in running the main code of the application on one processor and offload the computation of the tasks to other processors composing the infrastructure. Two examples of programming model within this category are the scripting language Swift [101] and COMPSs [91], described in more detail in Section 3.4.

3.3.2 Handling the Heterogenity of the System

The variety of protocols to interact with the processing elements is an issue that each implementation of the programming model has to tackle if they are not to be exposed to the application developer. On the one hand, there is the solution taken by large IT providers who offer a whole platform (PaaS) to develop services that run on top of their clusters. In this case, the underlying infrastructure is uniform, and the platform only provides an implementation of its API that directly interacts with the storage and computing resources. The developers of the platform only need to focus on how to obtain the most performance of the system without worrying about the interoperability. Some outstanding examples of PaaS are Microsoft Azure [5] Cloud and Google AppEngine [4].

The alternative for not binding the usage of a programming model implementation to a concrete infrastructure without dealing directly with the concerns of interoperability is to leverage on some software solution that abstracts the details of the infrastructure. Within this solution, there are two different approaches.

The first one consists in using software that homogenizes the system and implements the programming model highly-coupled to the features of the software. This solution forces system administrators to bind the infrastructure to a specific software. For example, the Apache Foundation implementation for the MapReduce model, Hadoop [8], builds on the Hadoop Distributed File System (HDFS) taking advantage of its data fragmentation, distribution and replication to obtain a higher performance on the execution of its applications.

The second option, less restrictive for system administrators, is to leverage on middlewares that offer a set of low-level methods and implements their functionalities in several protocols. Thus, the administrator of each component of the infrastructure can manage it with the desired software stack and applications can make use of them regardless the access protocol. This approach was analyzed and solved by the computer scientist who developed the Grid. They defined an abstract API, the Grid Application Toolkit [13], to command remote data transfers and submit jobs (task executions) to remote resources; two implementations of this API are JavaGAT [97] and SAGA [46].

3.4 COMPSs

COMP Superscalar (COMPSs) is a programming model which aims to ease the development of applications for distributed infrastructures, such as Clusters, Grids and Clouds. For the sake of programming productivity, the COMPSs model builds on three pillars:

- Infrastructure unawareness. COMPSs programs do not include any detail that could tie them to a particular platform. Thus, the model releases developers from dealing with the heterogeneity of the system or struggling with the mapping of the tasks to the processing elements of the infrastructure. By keeping applications agnostic to the infrastructure, they achieve portability across different platforms.

- Sequential programming. To hide away parallelism details from developers, COMPSs analyzes the sequential code of the application to build the workflow of the application.

The model automatically detects the tasks composing it and the data dependencies among them. Using this information, the implementation of the model orchestrates the execution of these tasks on the underlying infrastructure taking care of the required communications and synchronizations to guarantee the sequential consistency of the program.

- Standard programming languages and no APIs. To facilitate the learning of the model, COMPSs does not define any specific language nor provides a specific API or construct to build the application. Instead, developers code the sequential application using standard programming languages (Java, Python or C/C++).

The idea behind COMPSs is to apply the mechanisms implemented in out-of-order superscalar processors to exploit the Instruction Level Parallelism but at a coarser grain: method invocations. As the execution progresses, the code invokes methods of the application. Instead of computing the body of these methods in the same processor, the execution is replaced by the creation of an asynchronous task to run the same method code on a node of the underlying infrastructure. The more method invocations the main code does, the more asynchronous tasks coexist and run in parallel on the infrastructure.

As one instruction can use the value stored in a registry by another one, one method can use as an input parameter a value created by another invocation. Therefore, there are data hazards to control by detecting the data dependencies among tasks. Often, the main code of the application needs a value created within the body of one method invocation. These accesses constitute a control hazard since the main code needs to wait until the corresponding task completes and creates the value (synchronization) to go on with the execution.

To allow fine-tuning the grain of these tasks, developers must select the subset of methods whose invocations create new tasks. The selected methods are known as Core Elements (CE) and the main code of the application, Orchestration Element (OE). This selection is done by means of an interface, known as Core Element Interface (CEI), where the developer declares the methods to consider as a CE. Since interfaces allow to define methods but not the class to which they belong, the application developer needs to explicitly point out the class that contains the method implementation. For that purpose, they must annotate the method definition with the *@Method* directive and indicate the class with the attribute *declaringClass*.

An important difference between instructions and user-defined methods is the action performed on the parameters. An ISA has a limited number of instructions, and all of them have clearly defined its parameters and behavior; hence, the processor can detect data dependencies among instructions. Conversely, there are countless user-defined functions; each one has a different set of parameters and operates differently on their values. For determining data dependencies among tasks automatically, developers have to clarify the behavior of the operation by stipulating the action (read, update or create) performed on each parameter. For that purpose, COMPSs provides the *@Parameter* directive to annotate each parameter of the method declaration and describe the action performed on it. There, developers indicate the directionality of the parameter

(IN for value reads, INOUT for value updates or OUT for value creations) and its type (BOOLEAN, CHAR, SHORT, INT, LONG, FLOAT, DOUBLE, STRING, OBJECT or FILE). The model can automatically infer the type looking at the class of the parameter except for file passed as a String.

Besides the arguments, a method can operate on two more values: the callee object of the method and its return. COMPSs considers the former as an additional parameter of type OBJECT with INOUT directionality. Likewise, the return value is a new parameter with type OBJECT, but the direction of the parameter is OUT since the initial value does not exist.

The code snippet in Figure 3.3 contains a simple COMPSs application example. Subfigure 3.3(a) shows the main code of the application which runs one simulation for each argument of the application. The application aggregates the results of all of them in a single report object and prints it at the end of the execution.

Figure 3.4 contains an example of a CEI for the application that selects three methods as CEs. *PrepareParameters* is a static method implemented in the *Simulation* class. It takes one string describing the parameters to run a simulation as input and returns a *SimParameters* object containing the same configuration. *Simulate* is an instance method also implemented in the *Simulation* class. The method takes as the only parameter a *SimParameters* object which it reads to run the simulation. At the end of the execution, *simulate* returns a *Report* object with the result of running the simulation. The third CE corresponds to the static method *aggregate* implemented in the *Report* class. It takes two *Report* objects and updates the content of the first of them to include the values of the second.

When running, the application creates three asynchronous tasks on each iteration, one for each CE. The first detected task – corresponding to the *prepareParameters* CE invocation in line 9 of the *Main* class code – reads a string coming from the arguments of the application. Since they do not depend on any other task, every *prepareParameters* tasks can directly run upon its detection. When the application reaches line 11 of the code, it creates a *simulate* task. In this case, the *simulate* CE reads the *SimParameters* object created by the first task of the iteration; hence, there will always be a data dependency among the *prepareParameters* and *simulate* tasks of the same iteration. The third CE invocation on the iteration, *aggregate* on line 12, creates a task that reads the return value of the *simulate* task of the same iteration to merge it into the result of the *aggregate* task corresponding to the previous execution. Finally, once the execution has gone through all the iteration of the loop, the application reaches the *System.out.println* method invocation on line 14 to print the final result. At this point, the execution needs the actual value of the *globalReport* variable forcing a synchronization with the last *aggregate* task to fetch the proper value.

The directed acyclic graph in Figure 3.5 depicts the described workflow for an execution of the application with four arguments. Each node in the graph represents a task; red tasks correspond to *prepareParameters* tasks; blue tasks are *simulate* tasks; and yellow tasks, *aggregate*. Arches in

```java
package es.bsc.compss.sample;

public class Main {

    public static void main (String[] args) {
        int numSims = args.length;
        Report globalReport = new Report();
        for (int simId = 0; simId < numSims; simId++){
            SimParameters sp = Simulation.prepareParameters(args[simId]);
            Simulation sim = new Simulation();
            Report sreport = sim.simulate(sp);
            Report.aggregate(globalReport, sreport);
        }
        System.out.println(globalReport);
    }
}
```

(a) Content of Main class

```java
package es.bsc.compss.sample;

public class Simulation {

    public static SimParameters prepareParameters (String paramsDescription){
        SimParameters sp = new SimParameters();
        //Update content of sp according to paramsDescription
        ...
        return sp;
    }

    public Report simulate (SimParameters sp){
        Report r;
        //Runs the simulation according to the parameters in sp and generates a report
        ...
        return r;
    }
}
```

(b) Content of Simulation class

```java
package es.bsc.compss.sample;

public class Report {

    public static void aggregate (Report accum, Report diff){
        // Merges the results in report diff into accum
        ...
    }
}
```

(c) Content of Report class

Figure 3.3: Sample application code written in Java.

the graph represent data dependencies among tasks: the task portrayed by the target node of the arch depends on the task corresponding to its source. The red octagon at the bottom of the figure represents the synchronization between the main code and the execution of the last *aggregate* task.

3.5 Mobile Cloud Computing

Mobile or handheld devices are computers small enough to be held in hand and easily carried by users wherever they go. Smartphones and tablets are the most typical examples of this kind of devices. In the recent years, their popularity has increased [63], and applications for them are abundant. Although these devices have high capabilities for user interaction and network

```
package es.bsc.compss.sample;

public interface SampleCEI {

    @Method(declaringClass="es.bsc.compss.sample.Simulation")
    SimParameters prepareParameters (
        @Parameter(direction = IN)
        String paramsDescription
    );

    @Method(declaringClass="es.bsc.compss.sample.Simulation")
    public Report simulate (
        @Parameter(direction = IN)
        SimParameters sp
    );

    @Method(declaringClass="es.bsc.compss.sample.Report")
    public static void aggregate (
        @Parameter(direction = INOUT)
        Report accum,
        @Parameter(direction = IN)
        Report diff
    );
}
```

Figure 3.4: Core Element Interface for the sequential application in Figure 3.3. It defines three CEs: prepareParameters, that creates an object with the simulation parameters out of a string; simulate, which runs the simulation and creates a report of the execution; and aggregate, which merges a report into another.

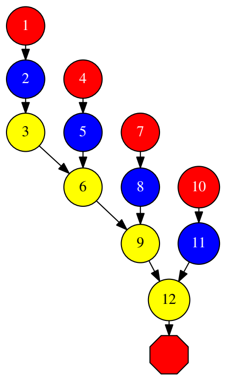


Figure 3.5: Directed acyclic graph representing the task workflow automatically detected by COMPSs for the application presented in Figures 3.3 and Figures 3.4.

connectivity regardless of the movement, their computing power is low and limited by the battery lifetime. Mobile Cloud Computing (MCC) [38] is a technology that tackles this limitation by bringing together the mobility of mobile devices with the vast computing power of the Cloud [16]. In other words, it allows the usage of smartphones and tablets to access/offer computing resources or software as a service. User interfaces of applications, – graphical, microphones and other sensors that might be used to interact with the system – run on the mobile device, and when they reach a compute-intensive point, the execution is offloaded to better-performing resources on the Cloud. An example of applications following this architecture is an app on the phone recording the sound of the heartbeat of a patient and submitting the audio to an external service for detecting cardiovascular diseases such as arrhythmia or tachycardia. Eventually, the service returns the result of the audio processing to the mobile device, and the application displays it.

Applications implemented like this require a fast connection to the Internet to run. Within cities, network protocols that allow these conditions are easily met and applications behave properly. However, they are not likely on rural environments where networks are slower or even unavailable in some areas. Consequently, applications might perform poorly or, in the worst-case scenario, not work since the resources of the mobile are too scarce to host the computation. For these situations, MCC allows offloading parts of the computation onto local cloudlets [88] – nearby resource-rich nodes – or onto peer-to-peer networks made up of several mobile devices [72].

Offloading part of the computation to remote nodes (surrogates) with a higher computing capacity is a technique that appeared with mainframes, where users accessed a central node from simple terminals. Since then, there has been a lot of research on support remote execution; however, due to the short existence of smartphones and tablets, research on Mobile Cloud Computing has only been done in the recent years. A key aspect in MCC is the high mobility of the device. At any point of the execution, the device can undergo changes in the strength and speed of the network, network technology shifts (Wi-Fi to 3G), or experience temporary, or even permanent, network breakdowns. For applications to keep behaving properly, developers must consider these situations and enable intelligent mechanisms on the application to handle them and provide a seamless service.

Revisiting the health application example, the application could keep invoking the same service when using the Wi-Fi interface. When using mobile networks, it could apply a preprocessing of the audio signal to reduce the number of bytes transferred and, consequently, its energetic and monetary costs. And, in the case of being isolated, the application should compute the result using the computing devices embedded in the mobile. Moreover, developers should provide the application with the logic to adapt its behavior to the dynamically changing conditions; and, in the case of a network breakdown while waiting for the result of the service, launch the computation on the local devices.

Programming applications that exploit MCC properly is complex. Fernando et al. perform a conscientious analysis of the issues related to MCC [38] considering not only the operational concerns of this kind of infrastructures but also other important aspects such as data privacy and security, legal restrictions or service-level agreements, among many others. Leaving out all the concerns beyond the implementation of the application, developers have to decide:

- What parts – tasks – compose the application (Application decomposition)
- When is it worth offloading the execution of one task onto surrogates (Cost/benefit analysis)
- Where and when should the offloaded computation run (Task Scheduling)
- How to perform the offloading (Offloading mechanism)

As for any other distributed system, MCC programmers can turn to programming models to ease the development of applications. As explained along Section 3.3, programming models allow developers to code their applications without dealing with these decisions. Automatically decomposing the application into parts is a shared problem for any parallel program regardless the underlying infrastructure. Mapping the tasks to the resources according to a cost/benefit analysis and manage the necessary data transfers and synchronizations to orchestrate the execution according to a Task Scheduling policy is a problem widely studied in the literature. For embedded systems, clusters and grids, these policies pursue maximizing the resource usage to reduce the execution time. Cloud environments evolved them to multi-target policies trying to reduce both, the execution time and the monetary cost of the execution; and for private clouds, they also consider the energy consumption of the whole infrastructure. The required communications between the mobile device and the surrogate to submit job executions is a problem that programming models can abstract away from the developer. Communities built around Cloud Computing have created and implemented several offloading mechanisms and fostered the standardization and homogenization of APIs to achieve interoperability.

The main differences between already existing technologies (Cloud Computing is the more similar one) and MCC lie on two aspects related to the high mobility of the device. First, mobile devices are tied to a battery, a limited source of energy; therefore their lifetime depends on the battery capacity and the usage of the available energy. Reducing the energy consumption of the master node extends the life of the device and applications can last longer. On the other hand, as aforementioned, mobility has an impact on the network: different protocols and interfaces (bandwidth, latency, and energy consumption), instability (performance fluctuation and reliability) and monetary cost (data access fees).

Consequently, the decision of which parts of the code should run on the mobile device and which ones should be offloaded is of great importance. Running a long-lasting, compute-intensive code on the mobile may consume most of the battery, offloading that part could save all that energy. On the other hand, offloading a task may require shipping a big amount of data out from the phone to a cloud node; while sending it through a Wi-Fi connection could speed-up the execution and reduce the energy consumption, transferring it through mobile networks may produce the opposite effect at a higher cost. Therefore, the decision has to be carefully made considering the available resources and their computing features, the locality of the data, the current capabilities of the network and the energy consumption and monetary cost arising from their usage. To tackle the problem, researchers have taken several approaches basing the models of the costs of the execution on the monitoring and profiling of the resources [21, 41], on a parametrical analysis [60], on stochastic methods [53] or even on machine learning algorithms [78].

The differences between MCC and other existing technologies have led to the emergence of frameworks specialized for MCC. The following section gives a glimpse of the ones standing out and compares them.

3.6 Mobile Cloud Computing Frameworks

We have identified a set of three distinguishing features that allow generating a taxonomy of the MCC models. The first factor, the migration granularity, is determined by the size of the application pieces that are offloaded to the remote resources. The coarser the grain is, the more data needs to be transferred to the resource. Transferring the whole state of a VM (or keeping the state of two VMs synchronized) requires more data than transferring only the state of one single thread and the data values it accesses; in turn, offloading a single method execution avoids shipping all the state of the thread.

The second classifying factor lies on how the model decides whether a part of the application runs on the local device or it is offloaded. It could be statically defined in the application code or decided dynamically at runtime depending on the environmental conditions.

Finally, every computation has blocks that can be executed concurrently on different resources to reduce the execution time. Depending on the model, the management of the parallelism is left to the programmer, or the runtime exploits it automatically.

Satyanarayanan et al. define in [88] a coarse-grain model where a whole VM is shipped to a nearby resource-rich computer, the cloudlet, taking advantage of hardware VM technology. They propose two approaches: migrate the whole VM or synthesize a small VM overlay to be applied on a base VM already present in the cloudlet (dynamic VM synthesis). Evidently, offloading a whole VM implies that any parallelism must be explicitly stated in the application. About the offloading decision, they do not specify whether if the programmers specify when to offload or if the runtime toolkit decides it at execution time.

CloneCloud [28, 29] offers the developer a finer level of granularity: threads. The strong point of CloneCloud is its partitioning mechanism that combines a static analysis of the code with a dynamic profiling of the application to pick the optimal migration and re-integration points. When a thread reaches a migration point, it suspends, and its state (including virtual state, program counter, registers, and stack) is shipped to a synchronized clone. When the migrated thread reaches a re-integration point, it is similarly suspended and shipped back to the mobile device. Finally, the returned packaged thread is merged into the state of the original process. Although thread level is finer than VM, it still requires the developer to create new threads and manage the application parallelism.

The partition granularity can still be reduced. Many models operate at method-level granularity. Cuckoo [56] takes benefit of the architecture of Android applications and hides the partitioning problem by exploiting the service component of Android. During the build process, the stubs generated to access service components are replaced by invocations to the Cuckoo framework that decides, at runtime, whether to run the service on the local device or a remote implementation. Since the framework only replaces calls, all the parallelism must be managed by the programmer on the service invocations.

Other models force the programmer to identify the methods to offload (or to consider their offloading). MAUI [30] offloads the execution of .NET methods to a remote clone of the application deployed in the cloud. Developers annotate the remotable methods, and the framework decides whether to offload the method invocation taking into account the application and network characteristics. To submit the method the system computes an incremental delta of the application state (method inputs and some static data) and ships them with the task description. The weakness of this model is the application parallelism. It is completely managed by the programmer, and it only exploits the computing resources of a single clone.

ThinkAir [58, 59] follows the same partitioning method than MAUI, but it works around its parallelism shortcoming by allowing the use of multiple surrogates. ThinkAir already provides a mechanism to automatically parallelize the execution of an offloaded method considering intervals of input variables. The main drawback of ThinkAir is that the offloading mechanism works synchronously: the executing thread is suspended until the method invocation is performed and its result collected. Thus, any subsequent method invocation is not executed until previous ones are executed even when they could run concurrently.

Also the processing network approach has been explored on MCC. There exist tailored applications such as the recommender system introduced by Nawrocki in [78]. The system is built on learning agents and determines the place where to run software operations choosing between several service providers as well as performing them locally on the phone. As a more general solution, AlfredO [44, 85] is a framework that deploys applications built in a modular fashion (OSGI components). Initially, the system extracts a description of the modules composing the application as well as CPU and communication statistics. With that information, the optimizer component –running on the cloud side– identifies an initial partition of the application and offloads on the mobile device the minimum functionality to start the application. At runtime, a profiler component monitors the CPU utilization and network usage and reports them to the optimizer so the latter can adapt the partitioning on the fly.

	Migration Grain	# Surrogates	Execution Model	Automatic Parallelization	Partitioning decision
Cloudlets	VM	1	Synchronous	No	Not detailed
CloneCloud	Thread	Not detailed	Synchronous	No	all methods dynamic profile
MAUI	Method	1	Synchronous	No	candidate methods dynamic profile
Cuckoo	Method	N	Synchronous	No	service calls dynamic profile
ThinkAir	Method	N	Synchronous	Intervals	candidate methods dynamic profile
AlfredO	Component	N	Concurrent	No	all components dynamic profile

Table 3.1: Comparison of MCC frameworks.

Table 3.1 summarizes the features of the MCC frameworks exposed along the section and highlights the characteristic features that distinguish them. It stands out that none of the presented frameworks automatically parallelizes the execution of the application. Using all the frameworks, the application developer deals with all the concerns of the exploiting the application parallelism and manages it manually. ThinkAir is the only framework that automatically deals with it, although it only does so exploiting the parallelism within loops. This dissertation aims to fill this gap by offering a programming framework that allows developers to build applications that exploit their inherent parallelism being totally unaware of it when coding. The key difference between the presented work and all the other frameworks is the execution model followed by the main application. While the other frameworks offload the computation following a synchronous model – once it reaches a migration point, the process waits for the result –, this dissertation describes a solution that leverages on the COMPSs programming model to offer an asynchronous execution of the application. Thus, when the application offloads the computation, it keeps progressing until it reaches a point that requires a value generated by an offloaded part.

3.7 Summary

Although the human mind conceives algorithms as a sequence of instructions performed one after the other, algorithms are likely to have inherent parallelism that allows computing multiple parts of the algorithm simultaneously to reduce the time to run the algorithm. The parallelism can appear due to the implementation of one instruction (bit-level parallelism), to independent operations within a sequence (instruction-level parallelism) or to independent sequences that can run at a time (thread-level parallelism). Amdahl's' and Gustafson's' laws model the performance that an application can achieve by running in parallel and set a theoretical limit to such improvement.

Computers are machines composed of processing devices that operate on a set of data values stored in memory modules. For a computing system to exploit the parallelism of an application, it needs several processing devices and memory modules. Depending on the network interconnecting the modules, there exist different types of parallel system mostly classified into two categories: shared memory, all the processing elements see the memory as a unique data space, and distributed systems, where each processor has its own memory module.

Writing applications for distributed systems is not straightforward. Programmers have to partition the application into computation units and re-code the application so that the execution runs in parallel as many of them as possible. These computation units have dependencies among them; programmers have to analyze the application and add the necessary synchronizations to ensure its sequential consistency. Besides the parallelism inherent in the application, a second limitation on the performance is the infrastructure. The computing capability and the number of the devices composing the system is limited; hence, the programmer has to orchestrate the execution of these computing units avoiding resource oversubscription ordering the consequent data transfers among the nodes to ensure that the computing units operate on the proper values. Besides, distributed infrastructures can be heterogeneous not only on the features of the hardware composing its nodes but also on the software required to interact with it. Thus, the programmer has to struggle also with the details of the interoperability of the nodes.

For easing the development of applications, programmers turn to programming models that provide an interface with high-level operations that abstract them from the technical concerns of distributed computing. Abstraction simplifies the structure of the software leveraging on it and releases the developer from dealing with the technical intricacies; however, as the abstraction level raises, the programmer loses control over the actual execution making harder to apply tweaks to achieve high-performing solutions. A programming model designer endeavors to raise the abstraction level while automatically delivering the same performance as if the developer had tweaked the application.

COMPSs is a programming model targetting distributed systems that provides developers with a sequential, infrastructure-agnostic fashion programming. Developers code their applications using the native language of the target device with no need to call any COMPSs-specific API. The model only demands from the developer to select a set of methods whose invocations become the computation units in which it partitions the application by declaring them on an interface. At runtime, a system running along with the application intercepts these methods invocations, detects the data dependencies that it may have with other invocations and orchestrates its execution top of the underlying infrastructure.

Mobile Cloud Computing is an example of a technology for implementing distributed computing systems. Mobile devices are handheld computers that users can carry wherever they go providing them with access to the Internet. Despite the high capacities for mobility and user interaction that these devices offer, their computing capabilities are very limited because of their scarce computing power and the lifetime of their battery. To overcome this limitation, Mobile Cloud Computing proposes to complement the mobile devices with the infinite computing power of the Cloud. Users could run their applications on their mobile devices and offload the execution of its compute-intensive parts onto resource-richer remote resources that will increase the performance while saving the battery consumed by such computation.

Mobile Cloud Computing adds two new concerns to the difficulties of distributed programming mentioned above. First, the battery consumption is an important parameter to take into consideration while assigning resources to the computation units. When running long computations on the mobile device, the processor can drain the whole battery of the mobile device and achieve a low performance. Data transfers can have the same effect when offloading short computations requiring big amounts of data. The network interface spends more energy and more time to transfer the values than the processor running the computation locally. The second concern is the instability of the network; due to the mobility of the device, it is likely to experience network disruptions due to handovers or the entrance in areas out of signal. Applications have to tolerate these changes and adapt to the new conditions to provide users with the expected response while reducing as much as possible any harm on their performance.

As with other distributed computing infrastructures, developers turn to programming models that struggle with these problems automatically. Despite there exist several programming models targetting MCC, none of them deals with the parallelization of the code. Most of them follow a synchronous execution model that offloads the execution onto the Cloud while the mobile waits for the result. Thus, it is the responsibility of the programmer to parallelize the application to improve the performance of the application. This thesis aims to fill this gap and provide developers with a programming model that abstracts them away from all the concerns. For that purpose, it builds on the COMPSs programming model to offer a clean and easy-to-adopt way of programming while a new runtime system automatically manages the parallelization and distribution of the computation considering the differentiating features of MCC.

Part II

General Proposal

CHAPTER 4

System Overview

This thesis proposes a framework for building mobile applications that leverage on distributed platforms to process their computational load. The cornerstone of such infrastructures is the mobile device. Users launch the application from the mobile and interact with the application via the multiple input devices embedded in it (multitouch screen, microphone, camera, gyroscope, light and proximity sensors, ...). Besides them, applications can also react to information collected from external sources such as wearables or IoT sensors sending the data through the network (usually, personal area networks like Bluetooth).

To overcome the limited computing power of mobile devices, applications offload part of the computation to cloud resources. These resources could be either virtual instances deployed on some datacenter of a commercial cloud provider or a small private cloud owned by some organization; a nearby desktop or laptop; or even other mobile devices or single board computer available on the same network.

The described platform, depicted in Figure 4.1, resembles an infrastructure that could also fit for Fog Computing and Internet of Things; however, both terms consent flexibility on the availability of the remote nodes that Cloud Computing does not. Two of the most characteristic features of the Cloud are the high availability of its services and the reliability of the infrastructure supporting them; conversely, fog nodes can join the infrastructure for a short time and then abandon it. MCC considers disruptions on the network connecting the mobile device to the cloud resources, but it does not contemplate the disconnection of the remote resources.

As explained in Section 3, developing high-performing applications for MCC environments is complex. To ease the work of application developers, the proposed solution raises the possibility of programming the logic of the application without explicitly stating its inherent parallelism

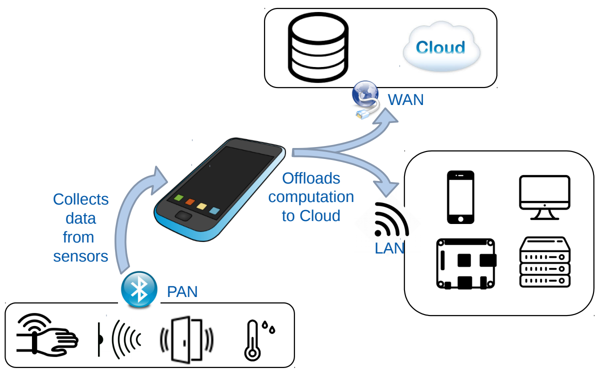


Figure 4.1: Layout of the components of the distributed infrastructure that could be used by applications developed with the proposed framework.

and being agnostic to the computing platform where it will run.

For that purpose, developers have to use a general-purpose programming model able to generate out of the sequential code a set of tasks to distribute across the available nodes while guaranteeing the sequential consistency of the program logic. The proposed solution builds on the COMPSs programming model, already described in Section 3.4, and extends it to allow multiple implementations for a CE. Section 4.1 describes this extension.

To translate from the sequential code written by the developer to a parallel, distributed computation generating the same results, applications run along with a runtime toolkit. This runtime detects the tasks that compose the applications and the data dependencies among these tasks to orchestrate their execution on the underlying infrastructure. It decides which tasks have to run locally on the mobile and which ones run on a remote node. To select the best resource to run a task, the task scheduling policy considers the characteristics of each task, the available resources and its features, the load assigned to each computing device, the conditions of the network and the data values hosted in each node. Although the policy tries to maximize the data locality when assigning each task a resource where to run, data transfers are unavoidable. Remote nodes might need values created by the mobile device; the runtime must provide the workers with the proper value to compute the proper result of the task. Likewise, when the main code of the application accesses a value generated by a remote resource, the runtime has to stall the application execution to synchronize the accessed value. The runtime toolkit manages all the operational issues of MCC transparently to the developer.

Sections 4.2 and 4.3 respectively describe the architecture of this runtime toolkit and how the developer-written code with no calls to any API is linked to it.

4.1 Programming Model Extension: Polymorphism

Often, several existing algorithms achieve the same functionality with different requirements and complexity. Depending on the characteristics of the device running the task, implementations may differ in their behavior; some might show a poorer performance or even not being able to run. For instance, the BubbleSort and RadixSort algorithms sort a set of elements. While RadixSort is faster than BubbleSort, the former requires less memory to run. Computing devices with a high computing power but low memory capacity may rather run BubbleSort to sort a set of elements; while RadixSort would be a better option to run on slower devices with high memory capacity.

The described COMPSs programming model enforces one single implementation for each CE; hence, developers have to select one algorithm to run on all the devices or implement an automatic selection of the algorithm within the CE implementation. The proposed extension to the COMPSs programming model aims to allow the developer to declare several implementations so that the runtime automatically decides which implementation has to run on each processing element. To declare multiple versions for a CE, the programmer adds as many *@Method* directives as different versions and in each one indicates the implementing class as shown in the code snippet in Figure 4.2. Thus, all the versions of the same CE need to be homonymous – *sort* – and share parameters and access patterns. Regardless the specific class invoked, calling any of the methods implementing a CE creates a new asynchronous task of such CE; the runtime determines the implementation that actually runs according to the computing device hosting the computation.

```
@Method (declaringClass = "containing.package.RadixSort")
@Method (declaringClass = "containing.package.BubbleSort")
void sort (
     @Parameter(direction = INOUT)
     int[] values
);
```

Figure 4.2: Sort method CE declaration with two possible versions implemented in RadixSort and BubbleSort classes respectively.

4.2 Runtime Toolkit Architecture

The main purpose of the toolkit is to orchestrate the execution of CE invocations (tasks) to fully exploit the available computing resources (local devices or remote nodes) while guaranteeing sequential consistency. To fulfill it, the runtime offers an API (Figure 4.3) with two different functions: *executeTask* and *accessValue*.

ExecuteTask requests to the runtime the execution of an asynchronous task. The method requires four parameters: the names of the method invoked and the class containing it; a boolean indicating whether invocation of the method is on an instance of the class – true – or static – false –; and the set of values corresponding to the invocation arguments. Besides the regular

```
/**
* Generates a new task whose execution will be managed by the runtime.
*
* @param methodClass name of the class containing the method that has been invoked
* @param methodName name of the invoked method
* @param hasTarget the method has been invoked on a callee object
* @param parameters parameter values
*
*/
 void executeTask(String methodClass, String methodName,boolean hasTarget, Object... parameters);

/**
* Registers an object access from the main code of the application.
*
* Should be invoked when any object is accessed.
*
* @param <T> Type of the object to be registered
* @param o Accessed representative object
* @param isWritter true if the access modifies the content of the object
* @return The current object value
*/
 <T> T accessValue(T o, boolean isWritter);
```

Figure 4.3: Definition of the interface to the Runtime toolkit.

arguments of the method, these parameters may contain two more objects corresponding to the callee object, if it is an instance invocation, and a future object corresponding to the result of the invoked method.

To synchronize the value of the future object with the result of the task execution, the runtime system provides the second method, *accessValue*, that takes two input parameters. The first is an object – a File instance for files – which a preceding task may have accessed and the second a boolean indicating whether that access modifies the content of the object or not. The method checks if any task has previously accessed that object. If no task has accessed that object, it returns the same instance. Otherwise, the runtime fetches the value from the node that computed its generator task and registers the access.

As done for registers in out-of-order processors, the runtime assigns to each datum version a unique ID and applies a renaming technique on each access to the datum with the purpose of preventing false dependencies (WaW and WaR accesses) from reducing the potential parallelism of the application. The first time a task accesses a datum, the runtime designates the ID to the value; for instance, *data1version1*. When one task or the main code of the application accesses the datum to update its content, the runtime assigns a new ID for the new version –for instance, *data1version2* – and preserves the value assigned to the previous ID. Thus, pending tasks reading the old version of the datum can fetch it by using the old ID, and tasks coming after the access will refer to the new ID to obtain the new value.

Since several applications can share computing resources and data values, the runtime library consists of two parts as the layout of the runtime architecture in Figure 4.4 depicts. On the one hand, the application-private part of the runtime controls those aspects of the execution related to the application. In other words, it is the entry point to the runtime; it creates new asynchronous tasks and monitors the private values they access (objects). On the other hand, the *Orchestrator* is in charge of handling all those aspects of the execution that might affect several applications; namely, accesses to shared data (files) and managing the usage of the available computing devices. While each COMPSs-Mobile application instantiates the application-private part of the runtime, there is only one single instance of the former component on the mobile device which runs in a separate process and is publicly available as an Android service.

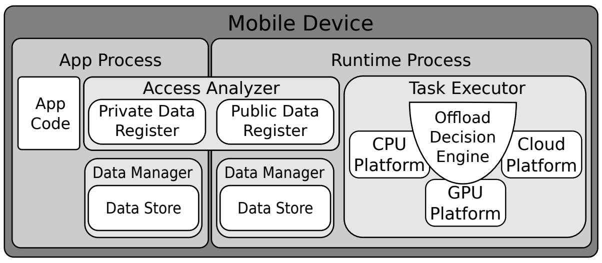


Figure 4.4: Runtime system architecture with three available Computing Platforms: one for the cores of CPU, on to offload tasks to the GPU and one gathering all the remote resources.

For achieving its purpose, the runtime toolkit leverages on two components: the *Access Analyzer* and the *Task Executor*. The *Access Analyzer* is a component partly hosted on the application-private part of the runtime and partly on orchestrator service. As its name suggests, its goal is to monitor all the accesses to the data values to detect data dependencies on task creations and necessary data synchronizations when accessing a concrete datum. The private data register, hosted in the application-private part of the runtime, is in charge of applying the renaming technique to all the private data values such as objects; while the public data registry, hosted in the orchestration service, does the same for the shared data values like files.

The *Access Analyzer* wholly implements the functionality of the *accessValue* method of the runtime API. For *executeTask* invocations, it only pre-processes the task to detect possible data dependencies. At the end of this pre-processing, the *executeTask* implementation creates a *Task* object containing which CE has to be executed and the list of arguments to pass to the method.

Following the example introduced in Section 3.4, when the execution reaches line 12 of the *Main* class on the second iteration of the runtime – calling the aggregation method –, the API of the runtime receives an invocation to the *executeTask* method with parameters "es.bsc.compss.sample.Report","aggregate", false and an object array with the current instance of globalReport and sreport. After the *Access Analyzer* pre-processing, a *Task* object represents the invocation with an attribute *ceID* with the internal ID representing the CE for the aggregate method in the es.bsc.compss.sample.Report class and two parameters representing an updating access to the value known as *d3v2* that at the end of the task will become *d3v3* and a read access to the value known as *d5v2*.

Once the *Access Analyzer* has processed the API invocation, the task object moves forward to the second component of the runtime, the *Task Executor*, for its execution. To decide which resources should host the computation, the runtime relies on the concept of *Computing Platform*: a logical grouping of computing resources capable of running tasks. The resources represented by a platform can vary from one single core from the processor to a set of virtual instances deployed in a multi-cloud platform. The implementation of the platform is responsible for monitoring the data dependencies of the task and scheduling both the execution of the task on its resources – picking one of the available implementations for the corresponding CE – and the obtaining and preparation of any necessary value. To achieve these duties, each platform can turn to different strategies: centralizing the management on the *Orchestrator* process, centralizing it in a remote resource or distributed across multiple resources.

The *Offload Decision Engine (ODE)*, subcomponent of the *Task Executor*, makes the decision of which platform runs the task being unaware of the actual computing devices supporting the platform nor the details of their interaction. The *ODE* polls each of the available platforms – configured by the user beforehand – for a forecast of the expected end time, energy consumption and economic cost of running the task. According to a configurable heuristic, the *ODE* picks the best platform to run the task and requests its execution.

Each part of the runtime has access to the *Data Manager (DM)* component. The *DM* is a distributed key-value store that manages the value assigned to each datum ID. The *DM* is asynchronous; either *Computing Platforms* or the *Access Analyzer*, on behalf of the application code, can subscribe to the existence or value of a specific datum. Upon the computation of a new datum version either on the main code of the application or any resource part of a *Computing Platform*, the generating element publishes its value into the *DM* which notifies all the subscribers. The local instance of the *DM* is responsible for handling the fetching of requested values if they are located in a different process.

4.3 Instrumentation

Section 3.4 defines COMPSs as a programming model with no APIs, while the runtime described in the previous section provides an API with two functions. To close the gap between the application programming and the runtime interface, the proposed framework provides a mechanism that instruments the code written by the developer during the building and packaging of the application.

This mechanism consists in adding a step to the four-step Android building process (described in Section 3.4) after the Java Builder and before the bundling of the application: the COMPSs

Instrumenter. During the first two steps, the building process completes the application code with all the necessary auxiliary classes, and the third step, the Java Builder, compiles it to Java bytecode. Using Javassist, a library for Java bytecode editing, the framework can modify the application classes as done for the regular COMPSs version – leaving aside the differences in the APIs of both runtimes. The framework scans all the classes of the application – not the classes within libraries on which the application depends – mainly looking for four code patterns that require instrumentation:

- Calls to CE methods. The instrumentation has to replace them by executeTask invocations passing as parameters the name of the invoked method, the class to which the method belongs, whether the call is on an instance or not, and the list of parameters. If the method is not statically called, the instrumented code includes the callee object as a parameter. If the method returns any value, the instrumented code creates an empty instance of the same class to use as a future object and also adds it to the list of parameters.

- Calls to non-CE methods on an object instance. Prior the execution of the method, the runtime needs to check if the object is not the result of a task and, if that is the case, it synchronizes its value. Before executing the method, the instrumented code calls the *accessValue* method of the runtime API. It always assumes that the body of the method modifies the content of the object.

- Constructors of Java utility classes to interact with files. Through these classes, the application reads/updates the content of a file that might be accessed by a task; thus, they require the same treatment as calls to non-CE methods on an object instance. Before the execution, the runtime needs to synchronize the content of the file through the *accessValue* method. Depending on the actions that the class allows to do on the file content, the instrumentation indicates whether the access modifies the value or not.

- Calls to non-CE methods whose code has not been instrumented (black-box methods). Besides the callee object, also the values passed as parameters require a synchronization. On instrumented methods, synchronization of parameters values are instrumented in the body of the method; for non-instrumented methods, the runtime cannot delay this verification beyond the invocation method. For each parameter, the code adds an invocation to the *accessValue* method to synchronize the accessed values. Again, the instrumentation assumes that the body of the method modifies the content of the object.

The instrumented code replace the original classes of the application, and the building process continues as usual for a regular Android application: the code is converted into Dalvik bytecode and bundled into the apk file.

Besides the code instrumentation, COMPSs Instrumenter needs to modify the Android-Manifest that is bundled along with the application logic to include the runtime service as an application component and request the permissions that it requires to operate. Namely, it requests the Internet access permission, for interacting and exchanging data with the surrogate nodes; and write access to external storage permission, for using the mobile sdcard to store intermediate values and release the memory from that burden. As discussed in Chapter 8, this thesis considers securing the submission of these intermediate data values over the network; however, it dismisses applying any encryption mechanism automatically to protect data at rest. Figure 4.5 illustrates the differences between the original manifest and the extended manifest.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1"
android:versionName="1.0" package="es.bsc.mobile.apps.bs" >
    <uses-sdk android:minSdkVersion="8" android:targetSdkVersion="21" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <application android:allowBackup="true" android:debuggable="false" android:icon="@drawable/ic_launcher"
    android:label="@string/app_name" >
        <activity android:label="@string/app_name" android:name="es.bsc.mobile.apps.bs.MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name="es.bsc.mobile.runtime.service.RuntimeService" android:process=":newprocess" />
    </application>
</manifest>
```

Figure 4.5: AndroidManifest file extended by the COMPSs Instrumenter. The additional elements are highlighted over the gray code which corresponds the original Android Manifest.

4.4 Summary

This chapter gives a glimpse of the solution proposed in this dissertation to build mobile applications that offload computation onto remote resources to improve their performance. Developers write the application leveraging on an extended version of COMPSs programming model. The extension allows developers to declare multiple implementations for one Core Element; thus, when the application invokes any of the implementations, the programming model creates a new task, and the designated device executes the most suitable implementation given the features of its hardware and the characteristics of the task.

As with the original COMPSs programming model, a runtime system runs along with the application to orchestrate the execution on the underlying infrastructure. This runtime has an API with two methods: one for submitting tasks, *executeTask*; and a second one for register data accesses from the application code, *accessValue*. For holistically managing the available resources for all the applications running on the mobile device, the runtime is twofold. Running on the same process of the application, one part of the runtime handles the invocations to the API, creates asynchronous tasks and detects the dependencies caused by accesses to the private data values of the application like objects. The second part of the runtime, named *Orchestrator*, manages the aspects of the execution shared among applications; for instance, detecting the dependencies among tasks due to access to files or running the tasks on the computing devices. This second part of the runtime runs as a service in an independent process; all the COMPSs applications running on the mobile device contact the same instance of the service. For managing the available resources, the *Orchestrator* groups the computational devices into *Computing Platforms* according to the mechanisms required to provide the processing elements with the necessary input values, launch the task execution avoiding resource oversubscription and fetching the results back from them. The *Offload Decision Engine* evaluates the temporal, energetic and economic costs of running a task on each *Computing Platform*, and decides which of them executes it. Then, each *Computing Platform* manages internally on which resources, on which moment and using which implementation runs to execute that task. For the different processes and *Computing Platforms* to share data values, each process has deployed an instance of a distributed key-value store that asynchronously replies queries about data values: the *Data Manager*.

For the code provided by the developer to invoke the runtime system, the programming environment has to instrument the application monitoring data accesses and replacing the calls to the methods selected as a Core Element by calls to the runtime. This instrumentation happens during the building of the distribution package of the application. The framework extends the default Android process with an additional step that replaces the original code of the application with the instrumented one, attaches the runtime system to the bundle and updates all the necessary configuration files.

Part III

Exploitation of Local Computing Resources

Chapter 5
CPU Exploitation

The first step towards fully exploiting a Mobile Cloud environment is to make a proper use of the computing elements within the mobile. Of all the devices embedded in a mobile, CPU cores are the most natural approach for standard developers to host the computation. Current mobiles are equipped with multi-core CPU; to benefit from their computing capacity, programmers have to deal manually with the management of multiple Java threads or learning the details about the classes provided by the Android framework to offer concurrent execution. In addition to this management, developers need to face the concerns related to the logic of the application already discussed in Chapter 3: partitioning the application and orchestrating the execution of such parts on the cores of the CPU.

COMPSs releases application developers from these concerns and passes the responsibility to the runtime system. Upon a task detection, the *Offloading Decision Engine* picks one of the available platforms to host the computation according to the execution forecasts provided by such platforms and requests the execution of the task to the selected one. At this point, the task may still have some pending data dependencies with previous tasks that need to complete before running it. Computing platforms must hold the execution of tasks with pending dependencies until they are addressed. For that purpose, they monitor the state of the global execution to detect the creation of such data values and notice when a blocked task becomes dependency-free.

The number of processing elements assigned to a *Computing Platform* is limited, and the number of tasks to run in parallel on the platform is likely to be larger than that. To avoid overloading the devices and harming their performance, platforms need to plan the execution of all the tasks on the available resources over time guaranteeing that all the necessary data will be in place before they start running. Arrived the time scheduled for a task to run, the platform triggers its execution on the processing element and monitors it. At the end of the execution, the platform retrieves from the processing element any relevant value generated or updated during the task execution – i.e., the values for INOUT and OUT parameters – and publishes their existence. Thus, other tasks can fetch from the *Data Manager* their value to run on the same processing element, on another processing element within the same platform, or on another platform.

In a few words, a platform is responsible for providing a forecast of the end time, energy consumption and cost of running a task. Once the *Offload Decision Engine* assigns a task to the platform, it monitors the existence and obtains the data values that tasks require to run, schedules task executions on the available processing elements, submits the executions to the actual resources, and collects their results to make them available to other tasks.

5.1 CPU Platform

For the runtime to support task executions on the CPU, it requires a computing platform able to orchestrate the execution of tasks on its cores: the *CPU Platform*. Upon the reception of a new task from the *Offload Decision Engine*, the *CPU Platform* submits the task to an internal *Scheduler* which orchestrates the execution of the tasks assigned to the platform on the available resources. The *Scheduler* plans not only the execution of the tasks but also all the necessary transfers to obtain the proper value of the accessed data values from other processes or nodes.

The first thing that the *Scheduler* does with a just submitted task is checking if there are any pending dependencies. For that purpose, it contacts the local *Data Manager* for querying the existence of every datum used as input for the task. If the value exists, the *Data Manager* directly replies the query; otherwise, it registers the query and delays the existence notification until the value creation.

From the reception of the notification on, the *Scheduler* can decide to trigger the obtention of the actual values to run the task. Arrived that time, the platform contacts again the local *Data Manager* requesting the value associated with the corresponding version of the datum. If the *Data Manager* already contains the value, it instantly notifies the value presence so that the task execution uses it; otherwise, it registers the query and fetches the value from a remote source. Once the *Data Manager* receives the value, it stores it and notifies its presence to the platform.

Often, the body of a task modifies the value of a datum passed as a parameter; however, the *Data Manager* needs to preserve the original value so that tasks running later on the same process can read it. When the platform requests a value for an INOUT parameter, the *Data Manager* clones the value stored for the initial version of the datum, and the task uses the copy as a parameter; thus, the task modifies the clone and leaves the stored value untouched. The *Data Manager* delays the value presence notification of INOUT parameters until the respective copy operation finishes.

Eventually, the *Scheduler* notices that the local *Data Manager* can provide the values to run the task on the cores. From that point on, the *Scheduler* can decide to launch the execution of any of the implementations of the CE at any moment. The platform has a pool of independent Java threads, whose size is configurable, continually polling the *Scheduler* for a task to run. When a thread gets a task from the scheduler, it gathers all the input values of the task and invokes the method corresponding to the selected implementation using reflection. At the end of the method execution, the thread collects the result of the method and the values of all the parameters – possibly modified – and notifies the platform of the end of the execution so that it stores values for the new data values on the *Data Manager*. At this point, the *Data Manager* notifies the existence of the just-stored values so that the *Scheduler* processes the new dependency-free tasks.

Figure 5.1 depicts the 8-step process involving the execution of a task on the CPU Platform. Steps 1 and 2 represent the existence queries and notifications done for each parameter, and steps 3 and 4 are the presence queries/transfer requests for the values and their corresponding notifications. Step 5 illustrates the submission of the task to one of the executing threads; 6, the actual execution of the task on the CPU core; and step 7, the notification of the task completion to the *Scheduler*. Upon the reception of the notification, the platform stores all the updated/new values on the *Data Manager* represented by step 8.

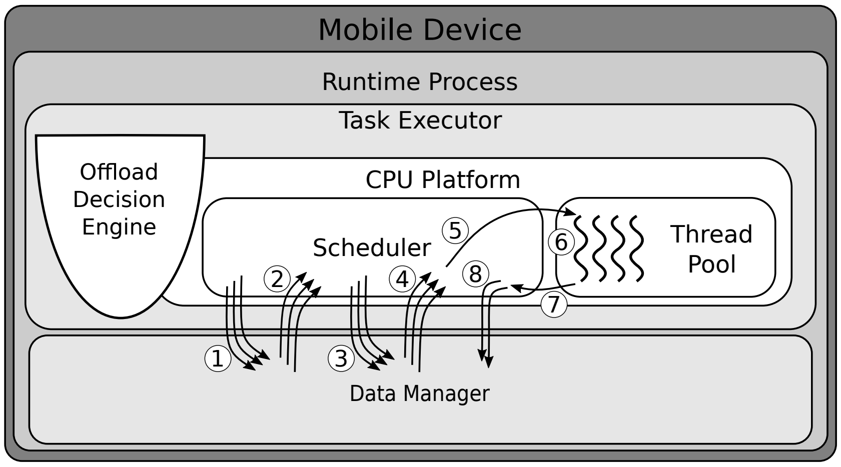


Figure 5.1: Architecture of the CPU platform illustrating the flow involving a task execution.

The current policy for scheduling the transfers consists in requesting all the data values of a task at the same time as soon as the *Scheduler* realizes that the task is dependency-free. For task executions, the *Scheduler* follows a FIFO policy considering the moment when all the parameters of the task are present on the local *Data Manager* as the moment when the task gets in the *Scheduler*.

Last, but not least, the platform needs to provide the forecasts for the execution. To make these predictions the platform takes into account information related to the available resources, like the number of cores and their power consumption; to the underlying infrastructure, such as the speed of the network or data transfer fees; to the task to run, such as the CE, the implementations able to run on the platform or the size of the parameters; and to the pending workload of the platform.

Android directly provides some of these values such as the power consumption of the mobile components (cores, screen, network interfaces, ...). The runtime can infer other values from the current state and standards; for instance, the used network protocol and the strength of the signal can lead to the speed of the network. Values like data fees or the number of available cores require the user to set them up. However, information, like the timespan to run a concrete task implementation on the CPU, requires application-specific knowledge that developers do not provide.

For the runtime to obtain this information, the *CPU Platform* profiles the execution of each task to collect data about the duration of the execution and the sizes of the input and output parameters. At the end of the task, the runtime adds the measured values into a statistical analysis of the historical values obtained throughout all the executions of the application. Currently, this analysis consists in keeping track of the highest, lowest and average value, but it could include other measures such as the standard deviation. Unlike the execution time that depends on the selected implementation and resources, the input and output data sizes are a feature shared by all the implementations of the CE regardless the platform and resources running it. For this reason, the runtime keeps record of the measures common for all the platforms, *Core Profiles*, and each platform owns a data structure to register the execution time of the implementation on its resources, *Implementation Profiles*. The *CPU Platform* assumes that all the cores of the CPU have the same characteristics; an implementation has the same behavior regardless the CPU core running it. Therefore, the data structure groups the profiles only by the implementation executed. To support executions on systems with a heterogeneous computing architecture coupling different types of processor cores, such as ARM big.LITTLE [6], the application user should set up the runtime to use a different platform to manage the cores of each type of processor and use thread affinity mechanism to bind the execution threads of each platform to the cores of the specific processor.

Considering the average execution time observed, the *CPU Platform* can find out which is the fastest implementation for each task. Expecting that all the tasks submitted to the platform waiting for execution require the average execution time to run, the platform can estimate when a core could start a task execution. Another factor important to contemplate when assessing the start time of a task execution is the moment when the input data becomes available. To determine such moment, the platform uses the expected time for the value generation – forecasted end time of the task computing it –, and the expected size of the value – extracted from the *Core Profiles* – and the speed of the network to predict the time to fetch it, if a different process computes the value.

Regarding the energy forecast, the platform takes into account the energy spent for fetching the input values from other nodes and the execution of the task. To predict the energy incurred by transfers, the platform uses the size of the values to transfer them back and the power consumed by the used network interface according to the current network conditions. For the task execution footprint, it uses the expected length of the execution and the power consumption of a CPU core provided by the Android platform. Given that the usage of the cores of the CPU incurs no additional expenses to the execution, the economic cost prediction only considers the costs caused by transferring data into the mobile device. For that purpose, it multiplies the size of data to be transferred back from remote nodes by the data fees – configured by the user – applied to the network interface currently in use.

Figure 5.2 notates the three models described. When the platform receives a new task, it applies the three models to all the possible implementations and selects the best solution according to the same heuristic used to pick the best platform.

End time:

$$RA = \frac{\sum_{c \in CEs} TC_c * XBT_c}{NC}$$

$$DA_d = DC_d + DS_d / BN_R$$

$$DA_t = \max_{d \in ID_t} DA_d$$

$$ST_t = \max\{RA, DA_t\}$$

$$ET_t = ST_t + XT_i$$

Energy consumption:

$$RS_d = DS_d * \mathbb{1}_{MD}(d)$$

$$E_t = PC * XT_i + EN_R * \sum_{d \in ID_t} RS_d$$

Monetary cost:

$$RS_d = DS_d * \mathbb{1}_{MD}(d)$$

$$C_t = CN_R * \sum_{d \in ID_t} RS_d$$

Variable	Description
RA	Time when resources become available
CEs	Application Core Elements
TC_c	Number of pending tasks of core c
XBT_c	Execution time for the best implementation for core c
NC	Number of available CPU cores
DA_d	Time when value d is available on the node
ID_t	Input data values for task t
DC_d	Time when value d is generated
DS_d	Data size for value d
BN_R	Network sensor reception bandwidth
DA_t	Time when all ID values for t are available on the node
ST_t	Start time for task t
ET_t	End time for task t
XT_i	Execution time for implementation i
E_t	Energy consumption for task t
PC	Power consumption of a CPU core
EN_R	Energy consumption of the network sensor when receiveing one byte
RS_d	Reception size for value d
DS_d	Data size for value d
MD	Set of values not on the remote nodes
C_t	Cost for task t
CN_R	Price for receiving one byte

Figure 5.2: Models to forecast the end time, energy consumption and monetary cost of running a task t with implementation i on the local CPU cores.

5.2 Proxied Execution

A superficial analysis of the behavior presented by the described platform reveals a deficiency that unnecessarily harms the performance of the runtime. Just take the example of one application with one single task taking as the only input parameter the content of a text box of the GUI to generate a string to show it to the user. When the application reaches the CE invocation, the runtime registers the content of the text box as *d1v1* in the *Data Manager* instance hosted on the application-private part of the runtime. After that, it submits the asynchronous task to the *Orchestrator* and continues the execution of the main code until it reaches the instruction that sets the result string into the label where the runtime halts the execution due to a value synchronization.

Simultaneously, the *Orchestrator* receives the task, and the *Offload Decision Engine* forwards it to the *CPU Platform*. The *Scheduler* of the platform queries the *Data Manager* of the runtime process for the existence of *d1v1* and receives the corresponding notification. At this point, the *Scheduler* tries to obtain the value for *d1v1*; however, the local *Data Manager* does not contain it and needs to fetch them from the *Data Manager* hosted in the application process. Transfer the value from one *Data Manager* to the other requires interprocess communication (IPC); hence, the source must serialize the value, pass the value, and the destination deserializes it. At the end of the transfer, the *Data Manager* in the *Orchestrator* notifies the presence of the value so that the *Scheduler* forwards the task to the threads. When the task completes, the *Scheduler* stores the result value, *d2v2*, in the local *Data Manager*. At this point, the main code of the application notices the existence of the value and requests the value to the *Data Manager* in the application-private part of the runtime which fetches it from the *Orchestrator* process via interprocess communications. For small objects of a few bytes, the overhead induced by IPC is negligible; however, for large objects, this mechanism may incur a significant overhead of several seconds.

One solution to dodge these value transfers consists in separating the management of the platform from the executing threads. The frontend of the platform, which computes the forecasts and the schedules task executions on the cores, remains on the *Orchestrator*, while the actual execution of the task happens in the backend of the platform hosted in the application process. In the end, the management of the cores is something concerning all the applications, but the CE methods are a private part of the application. By doing so, both, the application and the execution threads, request the same instance of the *Data Manager* for the values of the accessed data values; hence, transfers of data values are no longer necessary. Coordinating both parts of the platform still requires interprocess communications; however, commands follow a clear schema, are quickly serialized/deserialized and take up few bytes.

This division of the *CPU Platform* incurs changes on the flow followed by a task. Although, for the *Scheduler*, the stages of the process are the same – existence check, value obtaining, task execution and storing the values – the location of the components with which it interacts is different. After receiving the task from the *Offloading Decision Engine*, the *Scheduler* queries the *Data Manager* hosted in the *Orchestrator* for the existence of all the input values; steps 1 and 2 remain intact. It is from step 3 on that the process changes since the data values no longer need to be on the local *Data Manager* but on the *Data Manager* in the application process. At this point, the *Scheduler* contacts the backend of the platform which forwards the value request to the corresponding *Data Manager*. The application *Data Manager* acts exactly as the instance in the *Orchestrator* for the original procedure, and it checks whether the value is available in the process or whether it has to fetch the value from another process. Once the *Data Manager* has the value, it notifies the value presence to the *Scheduler* through the backend component alike the original step 4 does. When the *Scheduler* notices the presence of all the input values of the task and decides to launch the task execution, again, it contacts (step 5) the *CPU Backend* so that the execution threads contained in it run the task (step 6). At the end of the execution, the executing thread notifies the task completion so that the backend forwards the notification to the *Scheduler* (step 7) which eventually contacts to the backend to store the output values on the *Data Manager* instance in the application process (step 8).

Figure 5.3 updates the diagram of Figure 5.1 with the architecture and the flow of the task when the platform has proxied executions. Given the higher performance of the Proxied Execution compared to the flow described in Section 5.1, this is the default mechanism included in the final prototype to exploit the CPU of the mobile. However, the user can configure the runtime toolkit to run the tasks within the *Orchestrator* process.

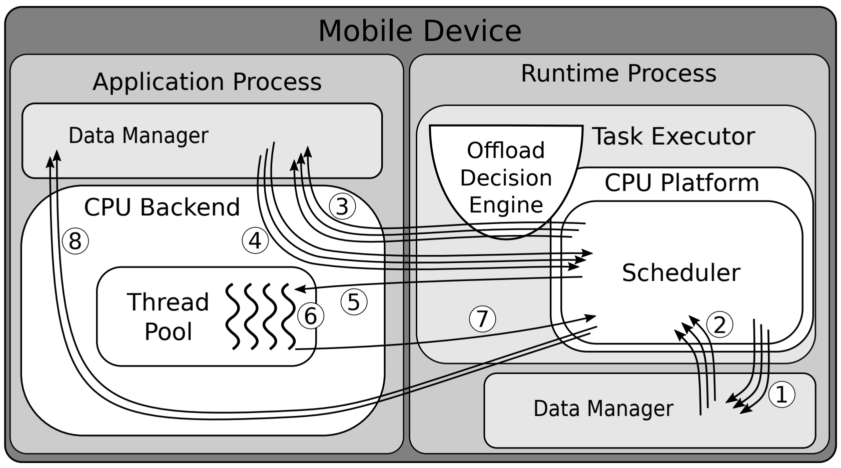


Figure 5.3: Architecture of the CPU platform with proxied execution illustrating the flow involving a task execution.

5.3 Evaluation

This section presents the results of the tests conducted to validate the proper running of the runtime and evaluate the impact of the proposed solution on three applications: Digits Recognition (DR), Bézier Surface (BS) and Canny Edge Detection (CED).

DR is an application based on the well-known method proposed by LeCun et al. [61] to recognize digits out of an image containing handwritten characters using a Convolutional Neural Network. Concretely, the application processes a set of images (a subset of the MNIST database of handwritten digits [62]) and returns an array that contains the recognized values. For doing so, the application goes through eight stages (eight invocations to six different methods) where each processes the whole set of images. To port the application to COMPSs, the six methods become CEs; thus, the application generates a sequence of eight tasks.

BS is a mathematical spline that generates a surface given a set of control points. Unlike interpolation, the resulting surface does not necessarily pass through the control points; they act as attractive forces to the surface. The application splits the output surface, and a method computes the result values within a chunk independently of the others. When porting the applications to COMPSs, this method becomes the only CE, and the application generates a set of parallel tasks.

Finally, CED [27] is an image-processing algorithm for edge detection where each frame goes through a four-stage process (Gaussian filter, Sobel filter, non-maximum suppression and hysteresis) each one encapsulated within a CE. The application runs the algorithm with 30 frames of 354x626 pixels producing a workload composed of 30 parallel chains of four tasks. This selection of applications and implementations is interesting for testing purposes because the diversity of workload patterns as shown in Figure 5.4.

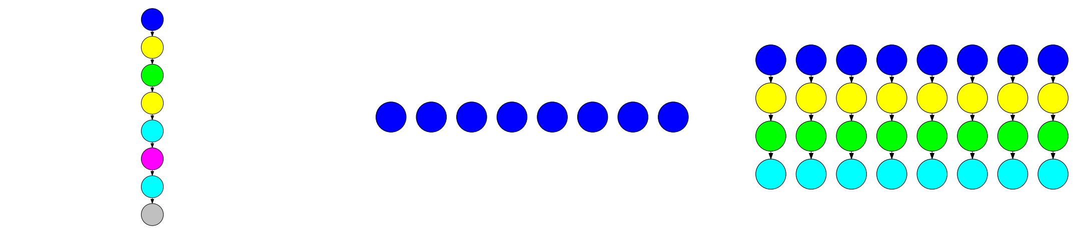

(a) Digits Recognition (DR) (b) Bézier Surface (BS) (c) Canny Edge Detection (CED)

Figure 5.4: Workflow representation for the three applications used during the tests: Digits Recognition, Bézier Surface and Canny Edge Detection (left to right).

The results presented below correspond to the execution of the three applications when running on a OnePlus One smartphone equipped with a Qualcomm SnapDragon 801 processor composed by a Krait 400 quad-core CPU at 2.5 GHz and an Adreno 330 GPU. Up to this point, the dissertation has only described the CPU platform; thus, the conducted experiments only use the Krait 400 quad-core managed by the *interactive* governor included in the Cyanogen OS 13.1.2

implementation of Android 6.0.1. The energy policies implemented in mobile devices, reduce the processor frequency when the screen is off since that fosters the energy savings when the user does not require responsiveness. Table 5.1 shows the normalized performance and power consumption of the OnePlus One. All the measures correspond to executions with the screen on at a 0% brightness and enabling the airplane mode to avoid the frequency reduction of switching off the display while reducing to the minimum the consumption of other devices.

	Idle	**Computing**	
	Power (W)	Normalized IPC	Power (W)
Screen off	0.08	0.042	0.20
Screen on brightness 0%	0.37	1	1.87
Screen on brightness 100%	1.07	0.998	2.55

Table 5.1: Power consumption and computing capacity according to the state of the screen of the mobile device (screen off, the screen on at 0% brightness and 100%) and its activity (idle and computing).

5.3.1 Automatic Parallelization

The first conducted test aims to measure the potential improvement in the performance of the applications, from temporary and energetic points of view, when using the proposed framework. Since all the computations run on the mobile phone, the economic aspect of the execution is not considered because data transfer fees do not apply. The tests consist in running the applications considering two different scenarios: running the sequential version of the application without instrumentation and running the COMPSs version of the application varying the number of CPU cores set up on the CPU Platform.

5.3.1.1 Digit Recognition

DR is an application with no task-level parallelism; thus, the application user cannot get any benefit from COMPSs since it cannot parallelize the execution of the task and, up to this point of the dissertation, the only available computing resources are the CPU cores. Conversely, the application is useful for standing out the overhead induced by the runtime. Charts in Figure 5.5 compare the execution time and energy consumption obtained when recognizing a set of 128, 256 and 512 handwritten digits when running the Android native version of the application (ACPU) and the COMPSs version with one available CPU core (1CPU). Since the application has no parallelism, it makes no sense to include scenarios using more cores. Charts break down the execution time in the number of milliseconds where the runtime is actively computing the result of the application and the number of milliseconds (*Computation*) where the computation is halted because of the overheads of the runtime (*Overhead*) such as IPC and the decision-making process. Regarding the energy, they distinguish the consumption incurred by the processing elements when computing the result of the application code *Application* from the overheads produced by the runtime and the other devices such as the screen *System*.

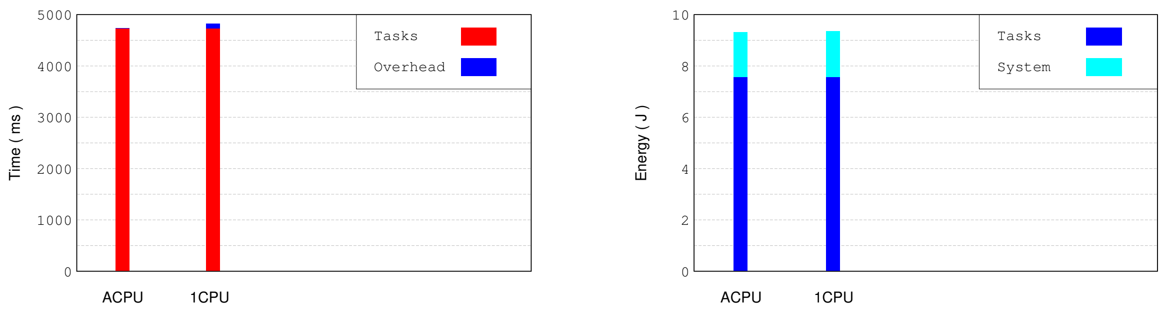
The time spent computing the methods – whether they run as regular methods or tasks – in both scenarios is very similar; the runtime may add a millisecond because of the overhead of calling the methods through reflection. This overhead is negligible and invariable regardless the amount of the processed data. The difference in the total execution time of the application lies only in the time dedicated to interprocess communications required to submit commands and transfer data among processes. As depicted in the charts, the growth of this overhead relates to the size of the data. For the DR application, it represents approximately a 2% of the total execution time: 95 ms, 188 ms and 355 ms respectively when the application processes 128, 256 and 512 images. Regarding the impact on the energy consumption of the runtime, the IPC overhead is negligible since it represents an increase of less than a 0.4% of the energy consumed by the Android native version of the application.

5.3.1.2 Bézier Surface

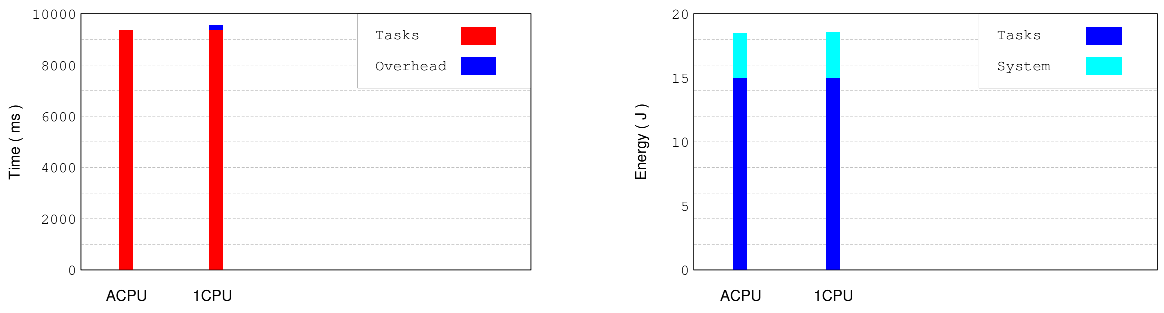
The second analyzed application is BS. The test considers four possible ways to split the surface: one single chunk of 1024x1024, four chunks of 512x512, 16 chunks of 256x256 and 64 chunks of 128x128. Splitting the surface into more than one chunk allows a parallel execution of the computation. While in the *ACPU* scenario, the native application runs on one single CPU core, the other scenarios, running the COMPSs version, vary the number of used cores; *1CPU*, *2CPU*, *3CPU* and *4CPU* respectively use one, two, three and four cores to execute the tasks. Charts in Figure 5.6 depict the execution time and energy consumption broken down as with DR.

When comparing the *ACPU* scenario with *1CPU* in all four partitionings, the time dedicated to processing the application code shows no significant difference. As with DR, the most significant difference in execution time lies solely on the overhead due to the interprocess communication. When the application operates on a single chunk, at the end of its processing the runtime spends 2,639 ms (25% of the whole execution time) on transferring back the surface from the runtime process. The smaller the size of the chunk is, the lower this overhead becomes since the transfer of the results of the earlier-to-run tasks overlap with the execution of other tasks. Besides, the smaller the chunks are, the shorter the time-to-transfer the result of one chunk is because fewer bytes need to be transferred. When the application splits the surface into four chunks, it requires around 730 ms (8%) to transfer a block; 241 ms (3%), for 256x256 blocks; and 56 ms (0.7%), for chunks of 128x128. Regarding the energy consumption, the values for the *Application* part are alike; however, the impact of the IPC overhead raises the consumption but to a lesser extent: from 0.98 J (6%) for the single chunk case to 0.02 J (0.1%) for the case with 128x128 chunks.

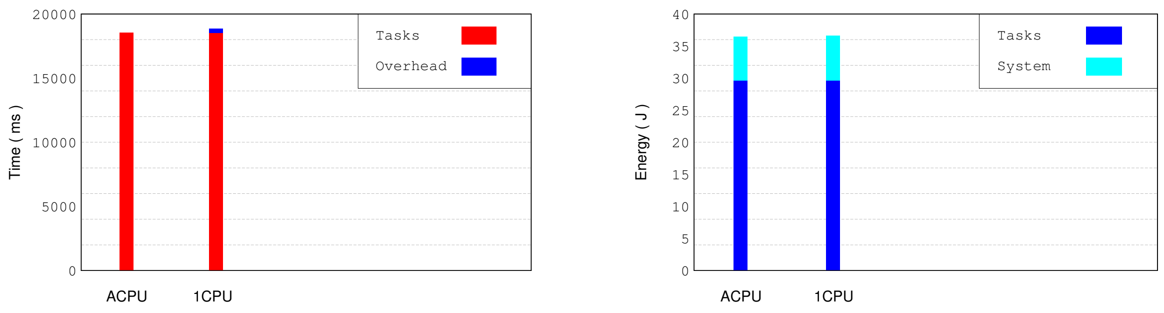
The execution time charts also show that the runtime is automatically exploiting the parallelism. Within each execution time chart, the multiple columns corresponding to the cases running the COMPSs version of the application show how the runtime can help to reduce the time



(a) 128 images.



(b) 256 images.

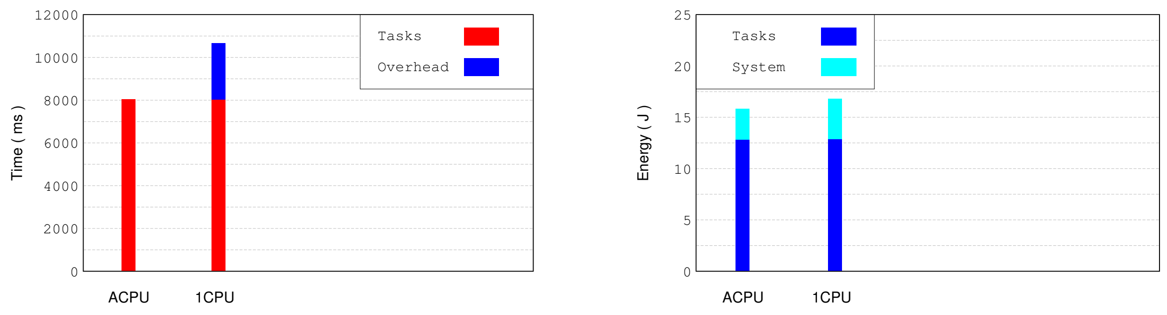


(c) 512 images.

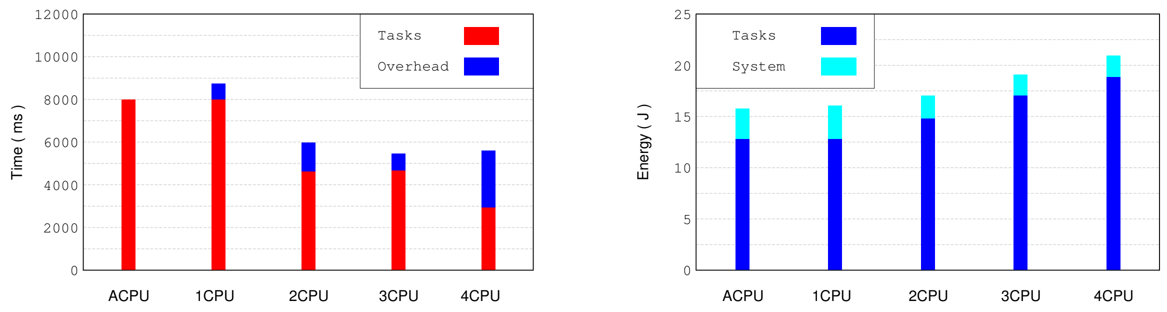
Figure 5.5: Execution time (left) and energy consumption (right) obtained when running DR with the CPU platform.

to compute the result of the application by using multiple cores of the CPU. Approximately, the runtime speeds up the processing of all the chunks by a 1.75x when using two cores. Using three cores allows the runtime to speed up the execution by 2.1x; except for the 512x512 case which for load balancing reasons the runtime requires a similar amount of time to compute the whole surface. When using four cores, the runtime achieves a 2.75x speedup. Although the obtained values show that the runtime system reduces the time to compute the result of the application, the reached speedup is far from the ideal – 2x, 3x and 4x, respectively when using two, three and four cores. The performance loss is caused by the reduction of the processor frequency to avoid overheating. The more cores processing at a time, the lower the frequency needs to be.

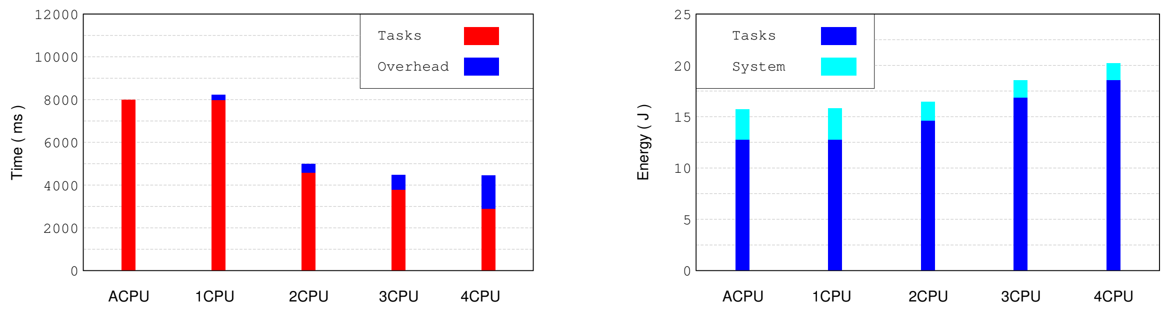
Although the runtime can speed up the processing of the application, its total execution



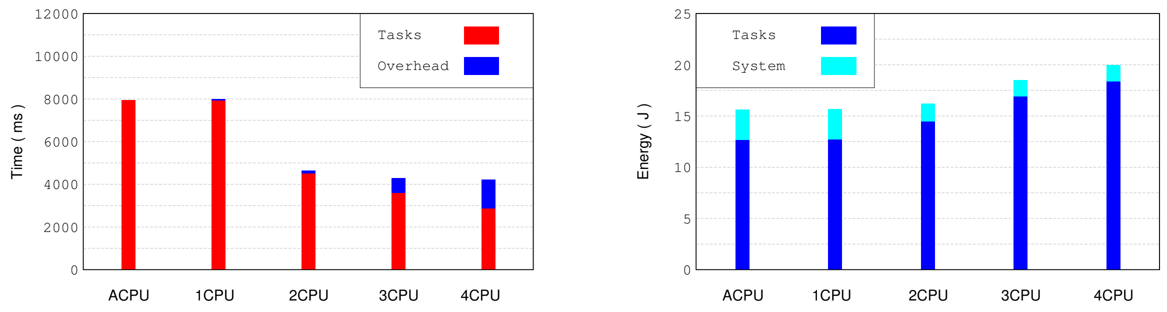
(a) 1024x1024 surface as a single 1024x1024 chunk.



(b) 1024x1024 surface divided into four 512x512 chunks.



(c) 1024x1024 surface divided into 16 256x256 chunks.



(d) 1024x1024 surface divided into 64 128x128 chunks.

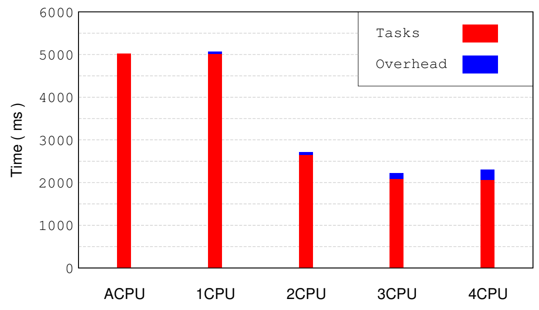
Figure 5.6: Execution time (left) and energy consumption (right) obtained when running BS with the CPU platform.

time cannot be reduced to the same extent since the interprocess communications mechanism is sequential and it transfers the computed values serially. All the executions using four cores show this problem. For the 512x512 case, the CPU processes the whole surface in parallel and the four chunks are ready at the same time; hence, the IPC transfers cannot overlap with any computation and delay the end of the application 2,648 ms (similar to the *1CPU* scenario of the 1024x1024 case). The 256x256 and 128x128 cases have the same problem, but the execution of the first round of tasks is shorter, and transfers begin earlier.

Both, processor frequency reduction and IPC mechanisms, add an important overhead in terms of time; however, the impact on the energy consumption is not that significant. At the end of the execution, the energy consumption only grows a 5% when two cores are available; an 18%, for three cores; and a 30%, for four cores. The energy consumed to execute the tasks is what produces this growth. Despite the frequency reduction lowers the power consumption of each core, it extends the length of the execution of the task and increases the total amount of energy consumed by each task. On the other hand, the earlier the application finishes, the less energy spent by other components within the mobile such as the display.

5.3.1.3 Canny Edge Detection

The third and last application used in this first experiment is CED, an application with similar characteristics to BS. The application generates 30 sequences of four tasks independent from each other; hence, the application has inherent parallelism to exploit by the runtime. Again, the test considers several scenarios: *ACPU*, running the Android native version of the application; and four possible scenarios where the runtime uses a different number of CPU cores – *1CPU*, *2CPU*, *3CPU* and *4CPU* respectively increasing the cores from one to four. Charts in Figure 5.7 show the decomposition of the execution time and the energy consumption of the application as done above for the DR and BS applications.



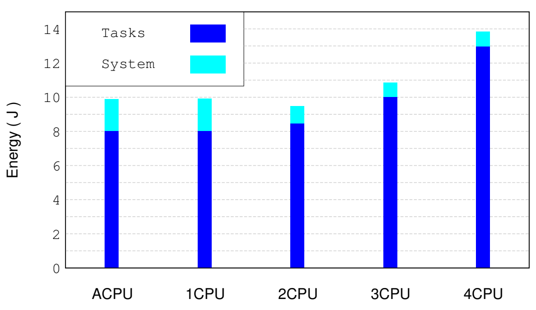


Figure 5.7: Execution time (left) and energy consumption (right) obtained when running CED with the CPU platform.

The obtained results are similar to those of the executions of the BS application with a high degree of parallelism. However, the amount of data to exchange among the application and runtime processes is smaller than in the previous application; hence, the weight of the overhead incurred by the IPC is significantly smaller. Comparing the *ACPU* scenario to *1CPU*, the differences are negligible; IPC increases the execution time by 51 ms (1%), and its impact on the energy is less than 0.02 J (0.2%). Regarding the cases where the runtime exploits the parallelism, the *Application* computation is faster when increasing the number of available cores – 1.9x, 2.41x and 2.43x respectively for *2CPU*, *3CPU*, *4CPU*. CED achieve a performance closer to the ideal than BS on the *2CPU* and *3CPU* scenarios, but this progression worsens on the *4CPU* scenario which roughly improves the execution time of *3CPU*. These behavior differences lie in the characteristics of the tasks and the effects of their concurrent execution; a processor frequency reduction would have a similar impact on the *2CPU* and *3CPU* scenarios. The final execution time is also affected by the IPC bottleneck, the sequential IPC mechanism is overwhelmed when several cores produce a result at a time; the 51 ms overhead on the *1CPU* case becomes 238 ms when using four cores (11.51% of the total exeuction time). Regarding the energy consumption, the conclusions are similar: increasing the number of cores increases the total amount of energy dedicated to the computation of the application. However, the reduction of the execution time leads to a reduction in the energy consumed by the other elements of the system besides the CPU. It is important to notice that for the *2CPU* case, COMPSs achieves an energy consumption slightly smaller than the native version of the application. The high performance of the processors when running the task shortens the processing time of the application by 1,365 ms at the expense of 0.43 additional joules. However, the time saving allows a reduction of the energy consumed by the rest of the system of 0.86 J; hence the application runs 1.87 times faster than the native version consuming only the 96% of the energy.

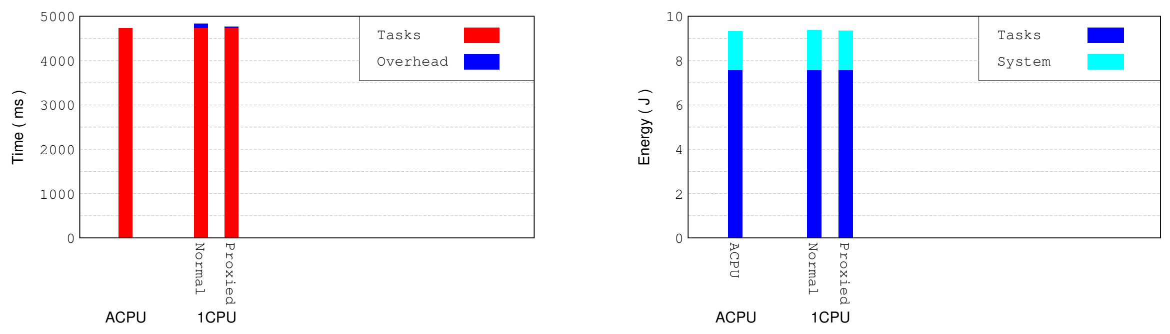
5.3.2 Impact of Proxied Executions

The second test aims to evaluate the impact of splitting the CPU platform to host the execution of the tasks in the same process that runs the applications instead of in the runtime process. For that purpose, the test compares the results presented in the previous section with the execution time and energy consumption obtained when running the same applications but using the proxied execution version of the platform.

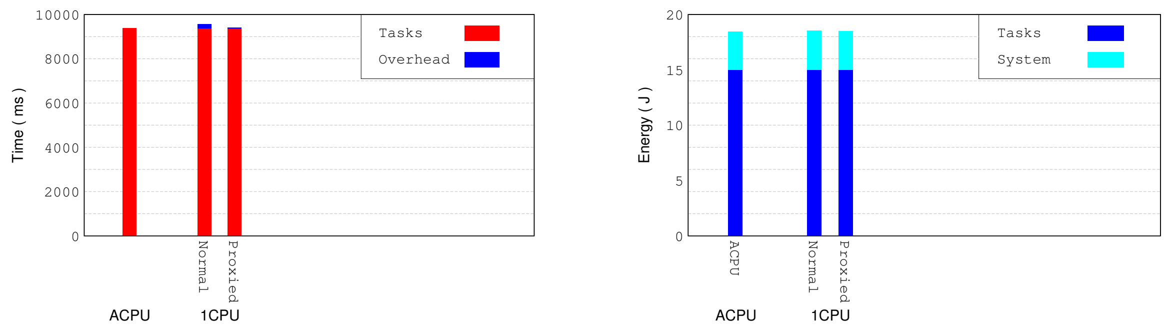
5.3.2.1 Digit Recognition

The DR application running the tasks on the *Orchestrator* part of the runtime did not show any problem related to IPC. The amount of data exchanged among processors is small; the weight on the total execution time around, 2%; and the effect on the energy consumption is almost negligible (0.03% of the total consumption). Hence, the impact of the improvement that proxying the executions may achieve is relatively small as depicted by the charts in Figure 5.8.

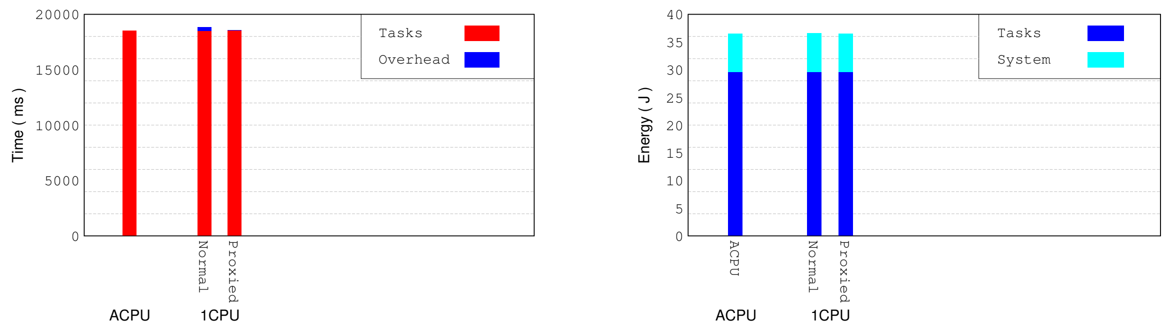
Hosting the execution of the tasks in the application process reduces the time dedicated exclusively to IPC – 95 ms, 188 ms and 341 ms, respectively for processing 128, 256 and 512 images – to a constant 30 ms. In the 512 images case, this reduction reaches a 90% of the IPC time; however, it only represents a 1.6% reduction in the overall execution time. Regarding



(a) 128 images.



(b) 256 images.



(c) 512 images.

Figure 5.8: Execution time (left) and energy consumption (right) obtained when running DR with the CPU platform comparing executions on the *Orchestrator* process (Normal) or on the application process (Proxied).

the energy consumption, proxied execution allows DR to save 0.0246 J, 0.058 J and 0.115 J respectively when processing 128, 256 and 512 images. The reduction is negligible since the overall consumption is 9,368 J for the case processing 128 images, 18,546 J for 256 images, and 36,608 J for 512 images.

5.3.2.2 Bézier Surface

BS is an application where IPC overhead is a major issue. In those cases where the grain of the surface partitioning is coarse, the transfer of the last blocks is expensive in temporal terms

(2,639 ms when a single chunk covers the whole surface). In the fine-grained cases where the runtime has multiple cores of the CPU available, the IPC mechanism is not fast enough to absorb the throughput of computed results and the amount of data to transfer accumulates. In the case where the application uses four cores to process the surface divided into chunks of 128x128, the IPC adds an overhead of 1,350 ms that unnecessarily raises the energy consumption 0.499 J as shown in Figure 5.9.

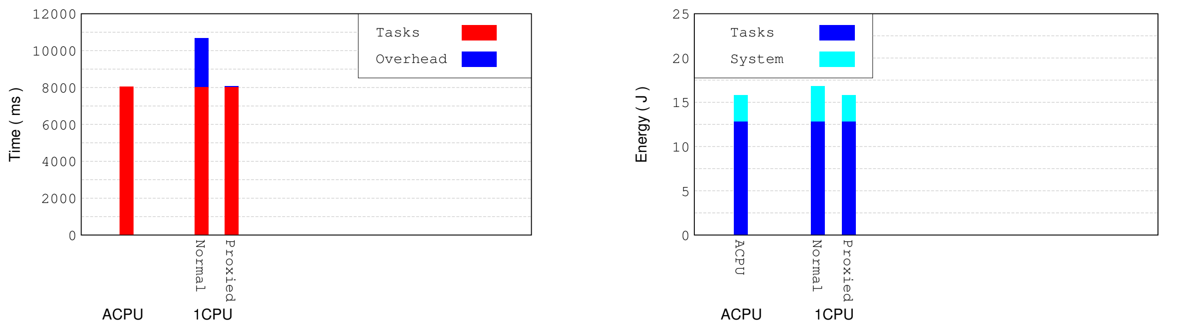
As with DR, splitting the CPU platform to host the execution in the application process reduces the IPC overhead drastically and converts it into a constant. When the application divides the surface into 128x128 chunks, the overhead means adding 30 milliseconds to the computation time. The impact of the measure grows along with the number of cores used: for one core, it only saves 26 ms; for two cores, the saving is already 101 ms; for three cores, 665 ms; and 1,320 ms for four cores. Naturally, shortening the execution time also reduces the energy consumption; however, it does so to a minor extent. Although for four cores the saving in energy reaches 0.488 J, the saving is almost negligible in the one core scenario: 0.010 J.

For the case where the whole surface remains as an only chunk of 1024x1024, the impact of proxying the execution reaches the limit for this application since it converts the 2,639 ms required to transfer the whole surface at a time into 30 ms dedicated to the exchange of the internal commands of the runtime. Although the time needed to compute the surface is the same – 8,035 ms –, reducing the IPC overhead allows a 1.33x speedup for the overall execution time. Shortening 2,603 ms the execution leads to a reduction of the energy consumed by other devices embedded in the mobile, such as the screen. In the 1024x1024 case, the time saving shrinks the consumption from 16.805 J to 15.84 J (6% of the overall energy consumption).

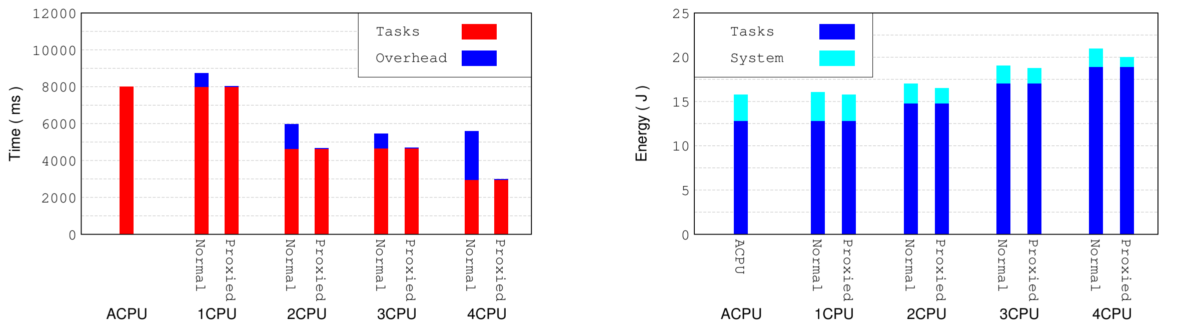
In the scenario using four cores of the CPU processing simultaneously four chunks of 512x512, the application behaves alike. The 2,648 ms of overhead become 30 ms; thus, the application achieves a 1.88x speedup. Although the temporal saving in absolute terms is the same than in the 1024x1024 case, the smaller time needed to process the surface increases the impact in relative terms. Conversely, on the energy aspect, the higher cost of parallel computing reduces the impact of the saving – 0.96 J – from the 6% to a 4%. On scenarios using a smaller number of processors for the same partitioning, proxying the execution reduces 701 ms and 0.26 J for one core, 1,321 ms and 0.49 J for two cores; and for three cores, 753 ms and 0.28 J. The 256x256 case presents the same behavior with a smaller impact.

5.3.2.3 Canny Edge Detection

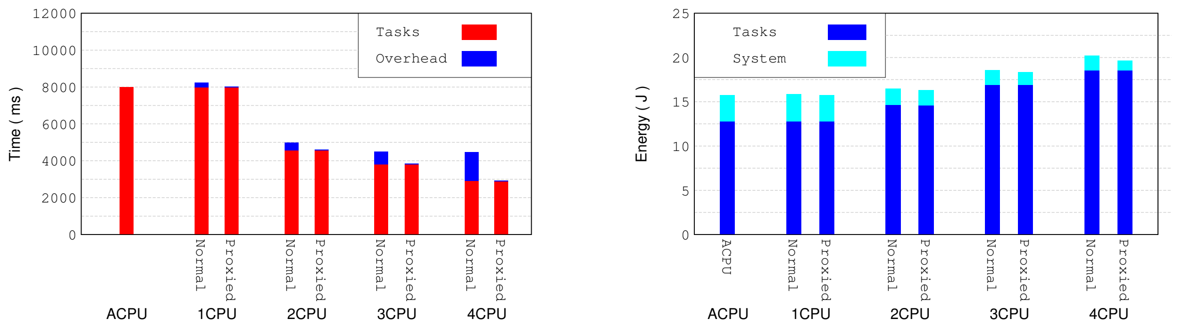
In the last application for this second test, CED, the IPC is also a bottleneck when multiple cores detect in parallel the edges in the frames. As with DR and BS, moving the computation from the *Orchestrator* process to the application process converts the variable IPC overhead – 51 ms when the runtime uses one CPU core, 61 ms for the two cores scenario, 132 ms when having three available cores and 238 ms when using all the cores of the CPU – into a constant overhead of 30



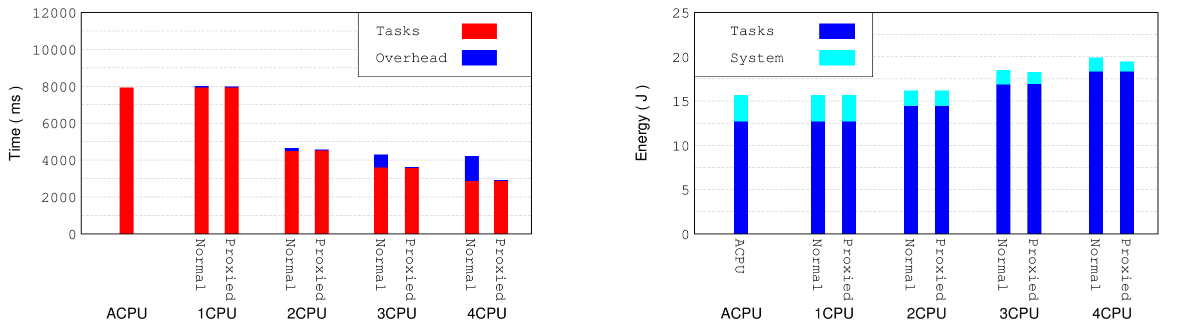
(a) 1024x1024 surface as a single 1024x1024 chunk.



(b) 1024x1024 surface divided into four 512x512 chunks.



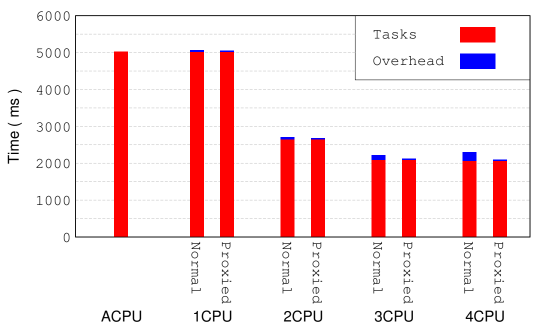
(c) 1024x1024 surface divided into 16 256x256 chunks.



(d) 1024x1024 surface divided into 64 128x128 chunks.

Figure 5.9: Execution time (left) and energy consumption (right) obtained when running BS with the CPU platform comparing executions on the *Orchestrator* process (Normal) or on the application process (Proxied).

ms as shown in Figure 5.10. Thus, the reduction of the overall execution time goes from the 0.4% on the one core scenario to the 9.02% on the four core scenario. The energy impact of the measure is almost negligible in all the scenarios; in *4CPU*, the application saves 0.08 J, only a 0.006% of the overall consumption.



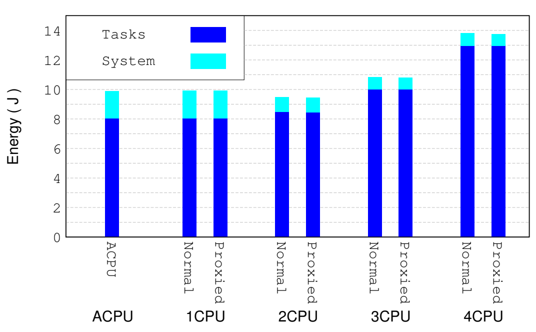


Figure 5.10: Execution time (left) and energy consumption (right) obtained when running CED with the CPU platform comparing executions on the *Orchestrator* process (Normal) or on the application process (Proxied).

5.4 Summary

Up to this point of the dissertation, the presented solution allows mobile developers to code applications in a totally sequential fashion without referring to the parallelism through the COMPSs programming model. During the building and packaging process, the framework instruments the code written by the programmer to call a runtime toolkit that partitions the application into tasks and monitors data dependencies among them and with the main code. The goal of this toolkit is to orchestrate the execution of such tasks on the available resources, grouped in computing platforms.

This chapter introduces one of such platforms, the CPU Platform, which allows the runtime to exploit the CPU cores of the mobile device. Using more than one core allows the runtime to benefit from the inherent parallelism of the application to reduce the time needed to run the application. Figure 5.11 depicts an overview of the runtime architecture including the components corresponding to the CPU platform. The cores of the processor are resources shared among all the applications; hence, the *Orchestrator* process must manage their usage. Nevertheless, hosting the execution of the tasks on the *Orchestrator* process requires transferring values generated by the main code of the application, executed on the application process, to the *Orchestrator*. This interprocess communication imposes an unnecessary overhead to the execution. For small values, it might be negligible; but, if the application works with large objects, it might become a significant part of the execution time. For instance, the Bézier Surface application requires 2,950 ms to process the four 512x512 chunks composing the surface using four cores and 2,648 ms to transfer the whole surface of 1024 x 1024 elements. In this case, the IPC overhead means the 47% of the execution time. Keeping the management of the tasks on the *Orchestrator* process and moving their execution away from there to a *CPU Backend* hosted in the application process works around the problem for tasks can access the same object. Although exchanging commands to launch the execution requires interprocess communication, the smaller size of the data to transfer shortens any delay to 30 ms approximately.

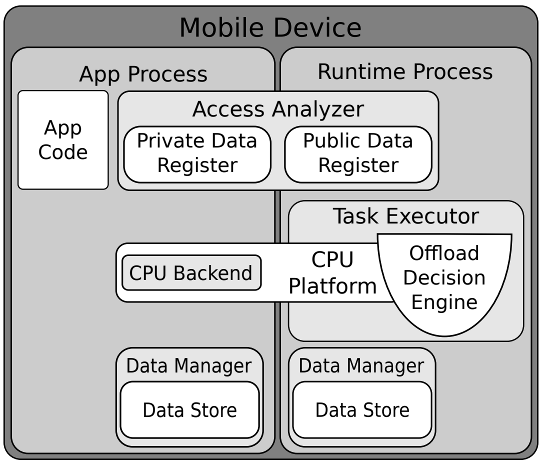


Figure 5.11: Diagram of the runtime architecture with a single CPU platform with proxied executions.

The tests presented show that the solution described up to this point of the dissertation can speed up applications up to 2.74x when using the four cores of the CPU – observed for BS splitting the surface into 64 chunks of 128x128 –, far from the ideal 4x speedup. The more cores compute at a time, the higher the temperature of the processor gets. For controlling the temperature of the processor and avoid malfunctions, mobile systems reduce the clock rate of the processor worsening the performance of the cores. Despite frequency cutbacks lower the power consumption of the processor, longer execution times incur higher energy consumptions for processing the same operation. Generally, the energy consumed by the application to compute the result of the application grows proportionately to the number of cores used. However, the mobile embeds other devices besides the CPU, such as the screen, sensors or the network interface, that also consume energy. If the execution time reduction is significant enough, it is possible that the energy saved on these devices makes up for the additional consumption of the processor as happens in the CED application when using only two cores.

CHAPTER 6

GPU EXPLOITATION

Graphical Processing Units (GPUs) employ SIMD architecture to achieve higher instruction execution rates compared to multi-core CPUs while saving energy through simpler control logic. During the last decade, heterogeneous systems combining multi-core CPU, GPU and other accelerators have become ubiquitous thanks to the general-purpose computing on GPU (GPGPU) frameworks. Even some system-on-chips (SoCs) already have integrated them on the same die; for instance, the Qualcomm Snapdragon and the NIVIDA Tegra. Both target mobile devices where energy efficiency is a major issue and CPU computing power, highly constrained.

To get the most out of the mobile device, applications have to use all the computing devices within it collaboratively. The most widely used programming models for developing applications for GPGPU are OpenCL [81] and CUDA [79]. Both present the hardware as a parallel platform allowing programmers to be agnostic to the actual parallel capabilities of the underlying hardware. On the one hand, these frameworks offer a multi-platform programming language to describe the computation to perform on the computing device; and, on the other hand, they provide an API to handle the parallel platform (launching computations, managing memory, and querying actual hardware details for high-performance purposes). Developers have to decide which parts of the computation run on the CPU and which on an accelerator, code the functionality of the parts not running on the CPU using the multi-platform language accepted by the accelerator and include in the application the code to programmatically manage the parallel platform (memory management and computation submission).

Integrating the use of such programming models natively into the COMPSs programming model and implementing all the necessary mechanisms in its runtime toolkit to support them would allow applications to exploit the internal parallelism within a task and take benefit from all the computing devices embedded on the mobile. Besides, that would release developers from dealing with the concerns of finding the optimal distribution for load balancing and the details of the platform management.

Given that CUDA is a proprietary platform exclusive for devices equipped with the Tegra SoC; the prototype builds on OpenCL: an open standard widely adopted by processor manufacturers, and thus, by a wide range of users. However, the proposed architecture does not lose any generality, and CUDA support could be easily added.

6.1 Related Work

Regarding adaptative heterogeneous computing on mobile devices, Android already provides a natively integrated framework for running computationally intensive tasks at high performance: RenderScript [48]. Programming with a C99-derived language, developers write code portable across the computing devices available on the SoC. At execution time, the RenderScript toolkit parallelizes the work considering the availability of the resources (load balancing) and manages the memory. Although RenderScript achieves performances similar to OpenCL or CUDA, it cannot exploit remote resources.

GPU usage is strongly associated with the implementation of high-performing operation; in this cases, programmers prefer to have an accurate control over the behavior of the platform regardless its cost on their programming productivity. Given the already existence on Renderscript and the low interest of the mobile industry to adopt this kind of frameworks, most of the research on adaptative heterogeneous computing on mobile devices focuses on the implementation of certain algorithms or libraries (especially, game engines) directly on CUDA or OpenCL.

Fortunately, that is not the case beyond mobile computing. There exist other general-purpose programming models/languages aiming to ease the development of task-based applications with GPU support. OmpSs [37] and StarPU [18] are two programming models that leverage on OpenMP pragmas to declare either CPU or GPU task implementations. Conversely, PaRSEC [25] allows programmers to describe the application as a DAG compactly represented in a format called JDF. For each task, JDF indicates the execution space, the parallel partitioning of the data, how the method operates on the parameters and the method to call to execute the task (allowing one CPU implementation and one for the GPU).

Another interesting research field relating GPUs and mobile devices studies how to enable the usage of GPGPU on devices without a GPU by offloading the computation on remote nodes with GPUs. Ratering et al. [84] propose using virtual OpenCL devices as the interface to compute clouds. For CUDA-enabled applications, rCUDA [89] takes a driver-split approach where the driver manages all the necessary details to execute the kernels on the local or remote GPU. A complete framework for computation offloading is the result of the RAPID [75] EU project, which allows CPU and GPU code offloading; however, none of the proposed frameworks automatically deals with load balancing.

6.2 Background: OpenCL

OpenCL (Open Computing Language) is a standard for general purpose parallel programming for heterogeneous devices. In a program using OpenCL, the main code of the application, which controls the execution, runs in a computing device known as host in OpenCL terminology – generally, the CPU of the node running the applications. One host has access to one or more OpenCL platforms; a platform consists of the host and one or more OpenCL devices. OpenCL devices are groups of one or more Compute Units (CUs) which are further divided into one or more processing elements (PEs). PEs process the computations offloaded to that device. Each OpenCL device, including the host, has an exclusive memory space – usually, devices do not share access to the memory modules.

For example, in a mobile device equipped with a Qualcomm Snapdragon 801 – one Quad-core 2.5 GHz Krait 400 and an Adreno 330 GPU – the host running the application is the CPU. This CPU has one OpenCL platform available named *Qualcomm Snapdragon 801* with one OpenCL device, the *Adreno 330*. In turn, the *Adreno 330* has four stream processors (CUs) each equipped with 32 stream processors (PEs).

OpenCL programs consist of a set of kernels, C99-based void functions to execute on the PEs, and the host program which controls the execution of the kernels from the host device. Since kernels are portable across platforms, the host program compiles them at runtime once it has decided which resource will run it. Thus, the host program can generate or modify the code of the kernels as the application execution progresses.

For managing the available platforms from the host program, OpenCL offers an API based on the submission of commands to manage the content of the device memory, execute kernels on its PEs and synchronize with other commands. To coordinate the execution of such commands on each device, the host program creates a command queue. The program pushes the commands into the queue and the device runs them either in in-order or out-of-order mode depending on the preference of the host program. For out-of-order executions to handle a command depending on a preceding command execution, commands produce events that indicate different states of its execution. The OpenCL API allows the host program to specify on which events the command being pushed into the queue depends so that the OpenCL library implementation schedules the execution of the commands preserving the dependency.

The core of the OpenCL model lies in the execution model of the kernels. Unlike with CPU-oriented languages, where one method invocation leads to one sequential execution of the function operating on the parameters, in OpenCL a kernel invocation incurs the execution of several instances of the function operating concurrently on the same parameters. OpenCL terminology calls each of such instances work-items.

For work-items to operate on a specific subset of the input/output data, each work-item has a unique N-tuple ID – usually, up to three dimensions – called *global ID*. To assign them, the library organizes the work-items in an N-dimensional grid and identifies each with the coordinates of its position in the grid plus the offsets in each dimension indicated by an N-dimension array.

To provide a coarser-grain decomposition of the grid, the standard groups work-items into work-groups. All the work-items within a group run concurrently on the same Computing Unit; and thus, they can benefit from hardware memory hierarchies to improve the performance of syncrhonizations and accesses to shared data values. Each work-group has a unique ID, *work-group ID*, and each work-item an ID, *local ID*, corresponding to its relative coordinates in the partition of the N-dimensional grid.

For running a kernel on the PEs of the device, the host program needs to create and set up a *Kernel Object*: an encapsulation of the kernel to invoke and the argument values used during the execution. If any of these values corresponds to a memory object, its content needs to be previously located on the device memory. After that, the host program enqueues a the kernel execution on the queue of the device indicating the *Kernel Object* to execute, the number of work-items running the kernel through the dimensions of the N-dimensional grid (*global_work_size*), the N-dimensional offset array used for computing the global ID of each work-item (*global_work_offset*) and the dimensions of the work-groups partitioning the grid (*local_work_size*). If the kernel depends on other commands and the queue schedules in an out-of-order manner, the host program also needs to specify the predecessor. If the host program does not specify any *local_work_size*, the OpenCL implementation determines how to break the global work-items into appropriate work-groups; by default, OpenCL considers that there is no offset to compute the global id.

6.3 Programming Model Extension

As different algorithms can implement the same functionality, different implementations of an algorithm can target different computing architectures. Programmers can implement one CE to run on a core of the CPU or a GPU thread.

Integrating OpenCL as a possible way to implement CEs arises two issues: the actual implementation of the CE as an OpenCL kernel and the abstraction of the OpenCL platform management. Some state-of-the-art projects, like Aparapi [3] and Sumatra [80], try to hide both issues away from the programmer by automatically generating the code of the kernel out of sequential Java code; mainly they try to automatically parallelize the outmost loop of the code and invoke the kernel with as many work-items as iterations of the loop. The solution proposed in this dissertation does not go as far as these projects, and programmers still provide the code of the kernels. Since OpenCL kernels use a C99-based language which is not compilable by Java, programmers attach their code as resources of the Java application.

To indicate the existence of OpenCL implementations of a CE, programmers annotate the method declaration in the CEI with the *@OpenCL* directive. In this case, instead of pointing out the class implementing the method, programmers indicate the name of the resource containing the OpenCL code of the kernel. To automatically determine the number of work-items running a kernel, the developer has to specify, as an attribute of the *@OpenCL* annotation, the *global_work_size* to use on the submission of the command to execute the kernel. However, the actual value of these variables may depend on the input values or its size. For that purpose, COMPSs allows simple algebraic expressions using the values and length of the parameters as variables. For referring to a parameter the developer uses the reserved word *par* followed by the index of the parameter. For instance, the developer points to the first parameter of the call using *par0*; for the third one, *par2*. If the parameter is a number, it allows to use its value; if the parameter is an array, it can use the value of one of its positions or its length. For multi-dimensional arrays, developers can refer to the length of any of its dimensions. For doing so, the developer uses the reserved names x, y and z to indentify respectively the first, second and third dimensions of the array. For instance, to refer to the length of the first dimension of the first parameter of the call, the developer uses the term *par0.x*; for referring to the second dimension of the third parameter, *par2.y*.

Besides the *global_work_size*, developers can also define values for *global_work_offset* and *local_work_size*. Both attributes are optional; in the case that the programmer does not specify any value for them, COMPSs forwards the decision to OpenCL. For *global_work_offset*, it does not apply any offset and sets the value to (0, 0,... 0); and for *local_work_size*, it delegates the decomposition into work-groups to the library by passing a *NULL* value.

Another important characteristic of OpenCL is that kernels do not return values. To avoid constraining the usage of OpenCL to CEs returning nothing, COMPSs assumes that the return value, if any, is the last parameter of the kernel; therefore, kernels implementing a CE with a return value have an additional parameter compared to its Java method version. As opposed to regular methods, where the return value is created within the method code, the memory space for the return value of OpenCL implementations needs to be allocated prior the invocation of the kernel. The runtime has to manage the allocation of result values automatically when it decides to run an OpenCL kernel. Again, the amount of memory to allocate depends on each CE and, likely, on the input values; therefore, programmers need to specify the number of elements within each dimension of the return value with an algebraic expression as the *resultSize* attribute of the annotation. The actual number of bytes is inferred according to the return type of the declaration.

Figure 6.1 depicts an example of a COMPSs application performing a matrix multiplication. The actual computation of the operation is encapsulated within a CE, *multiply*, implemented as a regular method and as an OpenCL kernel. As aforementioned, kernels have no return value; therefore, the OpenCL implementation of the CE has a third parameter corresponding to the return value of the Java implementation.

```
package es.bsc.compss.matmul;

public class Matmul {
    public static void main(String[] args) {
        int[][] A;
        int[][] B;
        int[][] C;
        ...
        C = multiply(A, B);
        ...
    }

    public static int[][] multiply(int[][] A, int[][] B) {
        // Matrix multiplication code
        // C = AB
        ...
        return C;
    }
}
```

(a) Application Java code

```
__kernel void multiply (
    __global const int *a,
    __global const int *b,
    __global int *c)
{
    //Matrix multiplication code
    // C = AB
    ...
}
```

(b) OpenCL code in matmul.cl

```
public interface CEI {
    @OpenCL(kernel="matmul.cl", globalWorkSize="par0.x,par1.y", resultSize="par0.x,par1.y")
    @Method(declaringClass="es.bsc.compss.matmul.Matmul")
    int[][] multiply (
        @Parameter(direction = IN)
        int[][] a,
        @Parameter(direction = IN)
        int[][] b
    );
}
```

(c) Core Element Interface

Figure 6.1: Example of a matrix multiplication with two implementations: one in OpenCL and one as a regular method. The code of the kernel is in the matmul.cl resource, and it has to be executed by as many threads as the number of rows in matrix a times the number of columns of matrix b. The result of the method is a bi-dimensional matrix with as many rows as matrix a and as many columns as matrix b.

6.4 OpenCL Platform

For the runtime system to run tasks on the GPU, or any other accelerator accessible through OpenCL, it requires a *Computing Platform* that orchestrates the execution of such tasks on the computing device and provides the time, energy and cost forecasts of hosting them. The OpenCL platform represents and manages one single computing device accessible through OpenCL.

As with the CPU Platform, hosting the execution of the tasks within the *Orchestrator* process would incur an overhead caused by the communications to transfer values among the processes that make up the runtime. To tackle this problem, the platform is twofold: the *Orchestrator* keeps managing the execution of the tasks on the device while a platform backend in the application process hosts the execution.

At boot time, the application launches the *Orchestrator* service – if no other application already started it – and waits until it instantiates, sets up and registers the configured platforms so that the *Offload Decision Engine* considers them to run tasks. Once the service is up and running, the application part of the runtime creates all necessary *OpenCL Backends*, setting up an OpenCL context for each with an out-of-order queue to submit commands to the corresponding computing device – identified by the application user through the names of the device and the OpenCL platform containing it. At this point, the platform gets all the CEs with an OpenCL implementation from the CEI of the application and compiles the corresponding kernels stored in the resource indicated in the *kernel* attribute of the *@OpenCL* annotation. After all the OpenCL backends have compiled all the kernels, the instrumented code runs and submits asynchronous tasks for execution.

The three models used to provide the *Offload Decision Engine* with the temporal, energetic and economic forecasts are very similar to those of the CPU Platform. The monetary cost of running the task only considers the data fees applied to bringing the input values from remote locations; thus, it is the same whether it runs on the CPU or in another device embedded in the mobile. Regarding the energy consumption, the only difference lies in the cost of running the task. Although the length of the execution should be shorter and the power consumed by the device higher, neither the theoretical model nor the implementation need changes since the values stored in the *Implementation Profiles* will differ from those obtained from the CPU cores. Finally, the end time prediction takes into account the same parameters: the average execution time on the device, the expected moment when the task becomes dependency-free and the moment when the device can host the execution; the only difference lies in the estimation of the moment when the device is available. As for the *CPU Platform*, the equation assumes that the platform manages several cores to run the pending workload; for the *OpenCL Platform*, it only considers the device as a single computing element. Thus, the resource availability instant corresponds to the summation of the expected – average – execution time for all the tasks pending to run on the device.

If the *Offload Decision Engine* picks an *OpenCL Platform* to host a task execution, the platform forwards the task to an internal *Scheduler* component that, as with the *CPU Platform*, contacts the *Data Manager* in the *Orchestrator* to monitor the existence of the data values on which the task depends. Once the *Scheduler* notices that all the input values exist, it interacts with the *Data Manager* on the application process to ensure that such values are available; if they are not there yet, the *Data Manager* fetches them from any location containing them.

Eventually, the *OpenCL Platform Backend* notifies the presence of the values on the *Data Manager* in the application process, and, from that point on, the *Scheduler* can decide to trigger the execution of the task on the OpenCL device. For doing so, it submits the task description to the *OpenCL Platform Backend* which needs to allocate space on the device memory to host all the values on which the kernel operates, including the output ones, and copy all the input value from the host memory before the kernel runs. Likewise, at the end of the kernel execution, it needs to retrieve the modified values from the device memory to store them into the *Data Manager* so that other tasks can fetch them. To properly manage the lifecycle of a task execution, the *OpenCL Platform Backend* leverages on the out-of-order mode of the OpenCL library.

Upon the reception of a task execution request, the *Backend* creates an OpenCL memory buffer – calling the *clCreateBuffer* method of the OpenCL API – for each parameter. The size in bytes of these buffers is automatically calculated considering the size of the input value corresponding to the parameter. If the buffer corresponds to an output parameter – i.e., the return value of the CE –, the *Backend* evaluates the expression provided in the *resultSize* attribute of the *@OpenCL* annotation. To set in the reserved buffers the corresponding input values, the *Backend* transfers the values, obtained from the *Data Manager*, to the device memory by enqueueing one buffer copy – *clEnqueueWriteBuffer* – for each IN or INOUT parameter.

Immediately after that, the *Backend* enqueues the kernel invocation – *clEnqueueNDRangeKernel* – indicating dependencies with the ordered copies to enforce their completion before the kernel runs. To submit the kernel invocation with the proper arguments, the *Backend* needs to evaluate the *global_work_offset*, *global_work_size* and *local_work_size* expressions provided by the application developer for the implementation to execute. For detecting the end of the kernel execution, the *Backend* registers a listener on the corresponding event. Upon the completion of the kernel, the *Backend* obtains from the library profiling information from the execution.

As with the copies for the input values of the kernel, the *Backend* collects the results of the execution submitting one copy from the device memory to the host memory – *clEnqueueReadBuffer* – for each potentially updated value, i.e., INOUT or OUT parameters. To ensure that these copies obtain the value once the kernel has processed them, the *Backend* submits the commands with a dependency with the kernel execution. Finally, the *Backend* must wait until the copies complete to store the values on the *Data Manager*. To detect the end of each operation, it registers one listener on each event generated for a value collection. Upon the completion of a copy, the *Backend* immediately stores the corresponding value.

For instance, for an invocation to the *multiply* CE introduced in the previous section (Figure 6.1), the *Backend* allocates three memory buffers: two to host matrices A and B, and a third buffer to host the result of the multiplication. Since A and B are IN parameters, the *Backend* enqueues two write commands to transfer the values of A and B into the device memory. After that, it enqueues the kernel execution depending on the two copies and registers the listener to become aware of the end of the task execution. Finally, since there is only one OUT parameter, the *Backend* enqueues a read command to collect the result of the value and registers the corresponding listener to detect when the final C value is in the host memory so that the *Backend* stores it on the *Data Manager*. The directed acyclic graph in Figure 6.2 depicts the dependencies among the commands enqueued on the device queue. Blue nodes illustrate the transfers of A

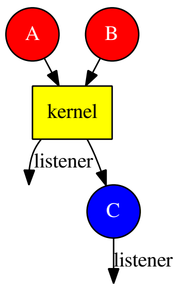


Figure 6.2: Dependency graph of commands submitted to the OpenCL device to run the matmul task from application introduced in Figure 6.1.

and B to the device memory; the yellow rectangle, the kernel execution; and the red circle, the memory transfer to collect C.

The current policy for data transfers is the same as with the *CPU Platform*: as soon as one task becomes dependency-free, the *Scheduler* contacts the *Backend* so that the *Data Manager* on the application process fetches any missing value. For the task executions, the *Scheduler* also applies a FIFO policy considering the moment when all the parameters of the task are in the application process. However, unlike the *CPU Platform*, instead of submitting one task for each core, the *Scheduler* submits up to four simultaneous tasks to the *Backend*. Thus, the out-of-order policy implemented by the OpenCL library manages commands related to multiple tasks at a time; thus, data copies related to a task can overlap with the execution of another one. The limit on the number of tasks simultaneously treated by the *Backend* is arbitrarily set to four; users can change it on the *OpenCL Platform* configuration. However, it is recommended to keep a low number since managing the events related to thousands of tasks could overwhelm the library.

To better exploit locality, the *Backend* monitors the content of the device memory. By keeping track of the buffer containing each data value and the writing event, it can discover an already existing buffer with the value. The *Scheduler* avoids the overhead of creating and filling a new buffer by using the existing buffer as a parameter of the kernel and enforcing its execution to wait upon the corresponding writing event. The renaming mechanism avoids any RaW hazards on data accesses. Before the task operates on the value, the runtime assigns a new ID to the value and creates a copy of the value for the task to modify it. Since the *Backend* looks for the copy instead of the original datum, there is no risk that tasks edit the content of an already existing buffer before another task reads it. Using this registry, the *OpenCL Platform* knows which data values are in the device memory and can predict which values will be created by the tasks assigned to it. Thus, the platform can bypass the query for the existence of those values and use the event produced by the kernel invocation to hand over the proper scheduling of the kernel execution to the OpenCL library.

6.5 Evaluation

This section presents the results of the tests conducted to validate the proper running of the *OpenCL Platform* and evaluate the impact of using the GPU embedded in the mobile on the three applications used in the *CPU Platform* evaluation (Section 5.3): Digits Recognition (DR), Bézier Surface (BS) and Canny Edge Detection (CED). As with the *CPU Platform*, the conducted tests run on a OnePlus One smartphone equipped with a Qualcomm SnapDragon 801 processor composed by a Krait 400 quad-core CPU at 2.5 GHz and an Adreno 330 GPU. The operating system of the mobile device is the Cyanogen 13.1.2 implementation of Android 6.0.1, and it uses the *interactive* governor to control the CPU frequency. Unlike the previous section, this time the tests use both computing devices.

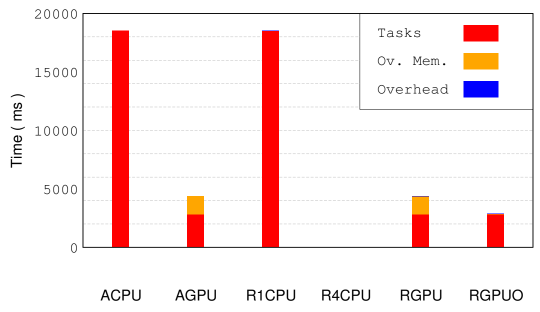
6.5.1 OpenCL Platform Performance

The first test aims to check the proper behavior of the OpenCL platform and evaluate the impact of the implemented optimizations. For that purpose, we executed the three applications considering six possible scenarios: *ACPU*, *AGPU*, *R1CPU*, *R4CPU*, *RGPU*, *RGPUO*. On *ACPU*, the application runs an Android-native, sequential version on the processor of the mobile. The *AGPU* version of the application replaces the CPU code of the functions performing the computation by the necessary OpenCL commands to offload the execution of an equivalent high-performing kernel onto the GPU and transfer all the involved data values in and back from the device memory. To simulate the performance obtained with an application developed by an average programmer, the application uses an in-order queue to submit the commands to the OpenCL platform. On the remaining four scenarios, the developer codes the application following the COMPSs programming model and the final user sets up the runtime to force the runtime to execute on a specific computing platform. On *R1CPU* and *R4CPU*, the runtime uses only the *CPU platform* exploiting one and four cores respectively. On *RGPU* and *RGPUO*, the runtime offloads all the tasks to the GPU through the OpenCL platform. The former disables all the optimizations obtaining a behavior similar to the *GPU* scenario, while the latter enables all the optimizations (reusing memory buffers and overlapping transfers with other kernel executions).

For each scenario, the application measures the execution time and its energy consumption. Within the execution time, it distinguishes the amount of time spent on the execution of tasks (*Computation*) from the overhead surrounding the computation (*Overhead*). This experiment focuses on isolating the part of this overhead corresponding to transfers between main and devices memories (*Ov. Mem.*) to evaluate the benefits of the optimizations implemented on the *GPU Backend*. Regarding the energy consumption, it only separates the energy used for computing the methods (*Application*) from the energy consumed by the whole system including the screen (*System*).

6.5.1.1 Digit Recognition

Charts in Figure 6.3 depict the results obtained from processing 512 images with the Digits Recognition application. It is plain to see that GPU allows a significant improvement on both, time and energy, regardless of using COMPSs. Comparing *ACPU* and *AGPU* scenarios, the execution time shrinks from 18,516 ms to 4,358 ms (23.53%) – 1,531 ms of which correspond to memory transfers –; and the energy consumption, from 36.48 J to 8.68 J (27.8%). *R1CPU* are the results already presented and commented in Section 5.3.2 for the *CPU Platform* with Proxied Execution. The *R4CPU* scenario is dismissed since the application has no task parallelism and it never uses more than a CPU core at a time. Using COMPSs incurs an overhead of 31 ms and 0.02 J due to the interprocess communication to exchange the commands. Obviously, this overhead appears on both scenarios where the runtime uses the GPU since the exchanged commands are the same. Besides this overhead, the application performs as on the *AGPU* scenario when the platform optimizations are disabled and adds an overhead of 1,531 ms due to the transfers of values between the host and device memories. When enabled, the runtime reuses the memory values generated in the device memory by one task as the input of the succeeding one; thus allows to reduce the overhead of data copies from and to the device memory to 5 ms. The optimizations implemented for the management of the device memory speed up the execution of the application on GPUs even when the application has no task level parallelism. Despite the improvement on the execution time, these optimizations have a low impact on the energy consumption (0.56 J) since the cause of the most significant part of it is the actual computation of the kernels.



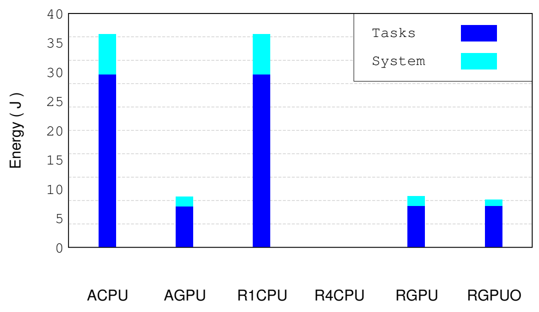
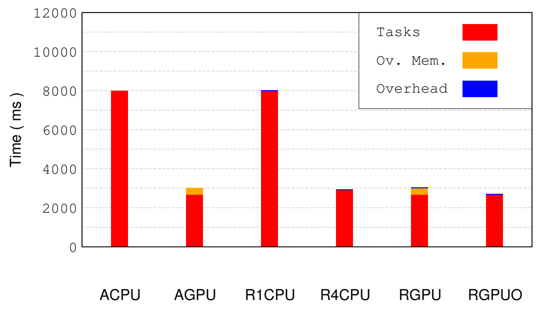


Figure 6.3: Execution time (left) and energy consumption (right) obtained when running DR with 512 images using both devices, the CPU and the GPU.

6.5.1.2 Bézier Surface

Figure 6.4 shows the observed measurements of calculating a surface of 1024 x 1024 points using 256 x 256 blocks with the Bézier Surface application. Tasks in BS have no dependencies; thus, the runtime can exploit the parallelism and use the four cores of the CPU at a time speeding up the execution of the kernels up to 2.72x (2,930 ms) at the cost of increasing the energy consumption up to 19.64 J (124.9%). As with DR, the runtime incurs a little overhead (30 ms and 0.02 J) observed when comparing *ACPU* to *R1CPU* and *AGPU* to *RGPU*.



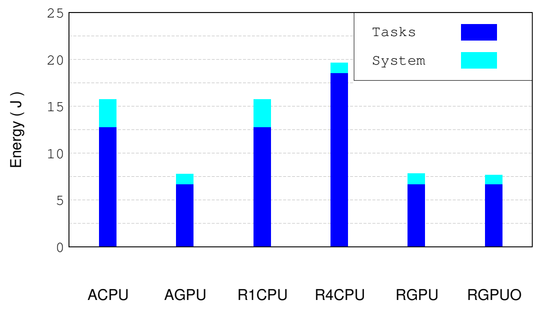


Figure 6.4: Execution time (left) and energy consumption (right) obtained when running BS with blocks of 256x256 using both devices, the CPU and the GPU.

Processing the tasks using the GPU device is 2.99 times faster than using a single core of the CPU as shown by the *Computation* time of the *AGPU* and *ACPU* (2,672 ms vs. 7,984 ms). However, the memory transfers overhead (337 ms) slows down the application; it only achieves a 2.65x lower execution time (3,009 ms): an execution time slightly higher than the one for the *R4CPU* scenario. Since BS tasks have no dependencies, they never read values generated by other tasks; therefore, the runtime cannot reuse values already transferred for preceding tasks. However, the computation of one task can overlap with the transfers of output/input values of the preceding and succeeding ones. This optimization allows the runtime to reduce the time spent on memory transfers from 337 ms to 3 ms on the *RGPUO* scenario. On the *RGPUO* scenario, BS lasts 2,705 ms and consumes 7.68 J.

6.5.1.3 Canny Edge Detection

As seen in Figure 6.5, the GPU device processes the 30 frames in 420 ms, 11.95x faster than a CPU core; and again, the data transfers worsen the application performance adding a 324 ms overhead. In overall, the application takes 5,020 ms to run in the *ACPU* scenario and consumes 9.89 J; while for the *AGPU* case, it needs 744 ms and 1.33 J. The runtime adds an overhead of 30 ms and 0.02 J slightly noticeable when comparing *ACPU* and *AGPU* to *R1CPU* and *RGPU*, respectively.

This application presents task-level parallelism and dependencies among tasks; thus, the GPU can apply both optimizations. The GPU reuses the output of some tasks as the input of its successors; thus, the runtime reduces the number of transfers. Besides, the remaining transfers can overlap with the computation of other dependency-free tasks. Enabling these optimizations allows the runtime to reduce the 324 ms overhead caused by memory transfers to 1 ms. On the *RGPUO* scenario, the application lowers the execution time to 451 ms and its energy consumption to 1.22 J.

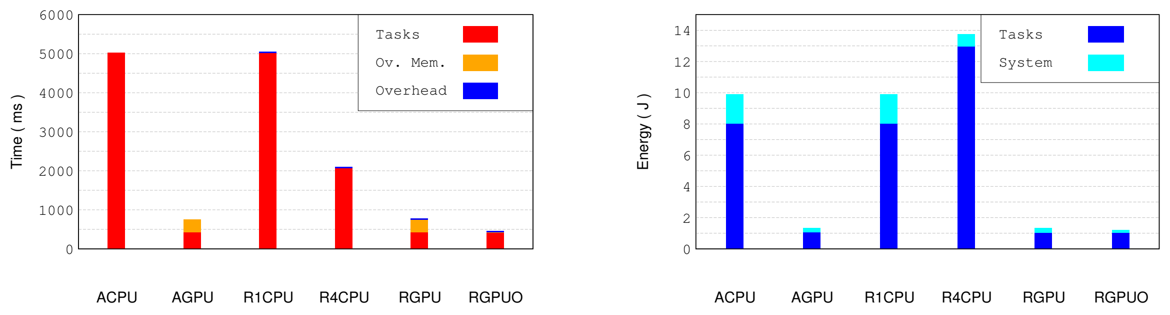


Figure 6.5: Execution time (left) and energy consumption (right) obtained when running CED using both devices, the CPU and the GPU.

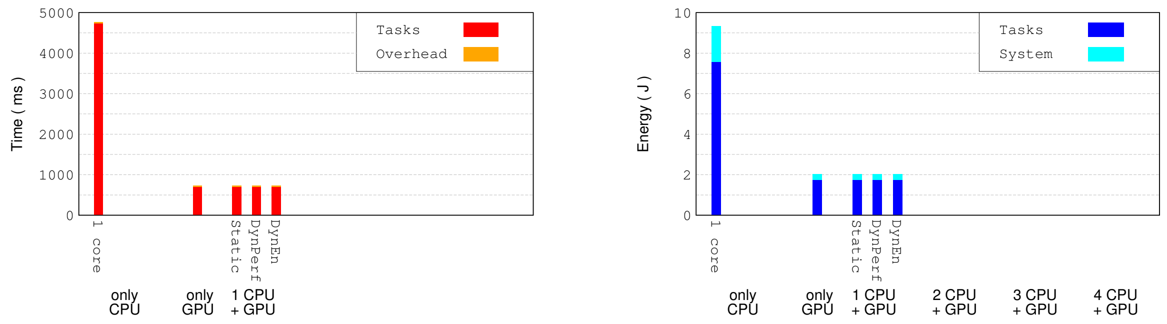
6.5.2 Load Balancing Decisions

The second experiment studies the impact of the platform selection policies on the execution time and energy consumption of the application. For that purpose, the test runs the COMPSs version of each application with different task granularity using every possible combination of resources. For the heterogeneous scenarios – i.e., using both computing platforms –, the test compare the results of three different policies: *Static*, *DynPerf* and *DynEn*. *Static* is a predetermined load distribution that mimics what application developers could easily devise to minimize the execution time. The load arrangement employed on each execution depends on the application workflow, the number of tasks and the time they require to run on each device; the subsection corresponding to each application provides further details on the applied division. With the same purpose, the *DynPerf* policy automatically decides which computing platform executes the task according to the earliest end time forecasted by the platforms. Conversely, *DynEn* aims to find a balance between reducing the execution time and the additional energy that it incurs. For that purpose, *DynEn* takes into account not only the end time of the task but also the energy spent on its processing; it would pick a later task end time if for each sacrificed ms the application can save 5 mJ.

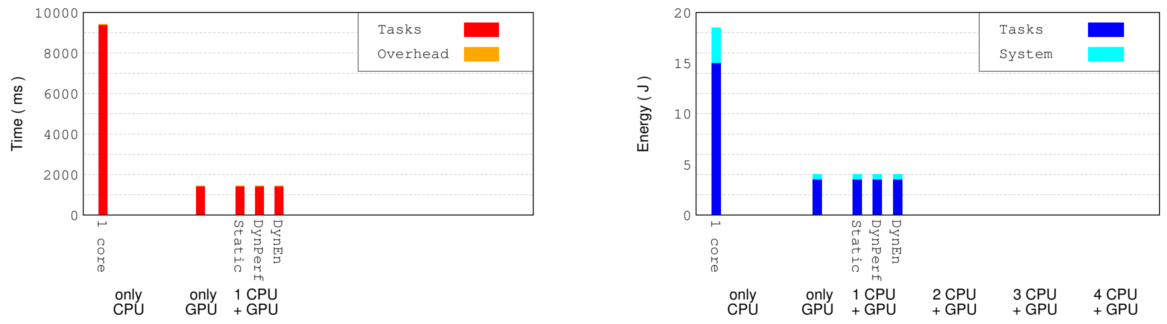
6.5.2.1 Digit Recognition

DR is an application where a set of images go through a 7-stage process. Each stage is encapsulated in a task; thus, their granularity depends on the number of images to process. This experiment uses three different input sets composed of 128, 256 and 512 images. Since DR has no task-level parallelism, it dismisses all those resource configurations using more than one core of the CPU. All the CEs that compose the application take less time and energy to run on the GPU device than on the CPU; therefore, the *Static* policy for DR consists of submitting all the tasks to the GPU.

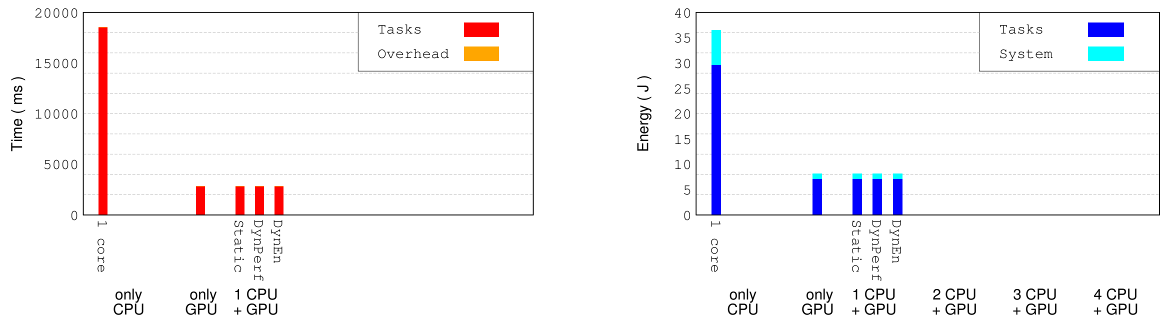
Charts in Figure 6.6 show the execution time (left) and energy consumption (right) obtained when processing 128, 256 and 512 images (from top to bottom). Despite the difference in the



(a) 128 images.



(b) 256 images.



(c) 512 images.

Figure 6.6: Execution time (left) and energy consumption (right) obtained when DR processes 128, 256 and 512 images (top to bottom) using the CPU and OpenCL platforms.

magnitude of the values, the application behaves alike regardless the input size. As the number of images doubles, almost does so the execution time and the energy consumption whether if the application runs on the CPU – 4,768 ms and 9.344 J for 128 images; 9,380 ms and 18.490 J for 256 images, and 18,550 ms and 36.493 J for 512 images – or on the GPU – 730 ms and 2.015 J, 1,445 ms and 4.065 J, and 2,863 ms and 8.127 J respectively for processing 128, 256 and 512 images. Given that the GPU is faster and less energy-consuming than the CPU and that the application presents no task-level parallelism, submitting all the executions to the GPU is the optimal solution either from the performance or the energy point of view. Hence, both dynamic policies schedule all the executions to the GPU as expected. Although all the employed configurations use the runtime which incurs an overhead already measured and analyzed, it is important to notice that dynamically deciding where to run a task adds no significant overhead compared to those cases where the runtime handles a homogeneous system or the decision is statically set beforehand.

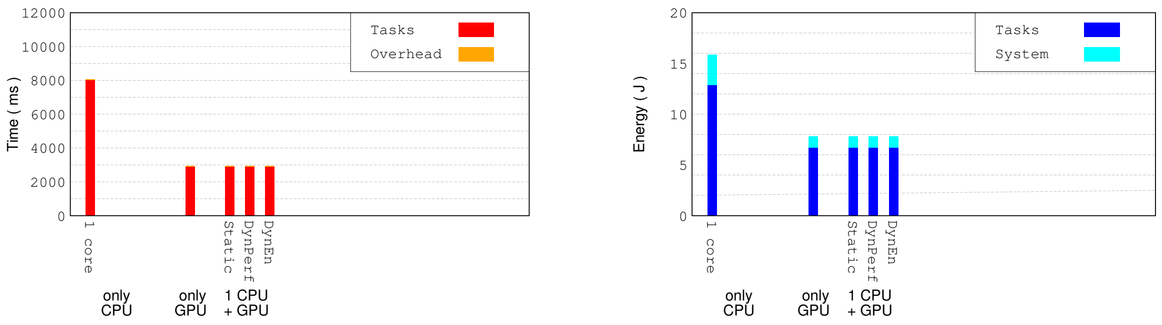
6.5.2.2 Bézier Surface

BS is an application whose task-granularity and parallelism depends on the partitioning of the output. For this experiment, the application computes a fixed-size surface of 1024x1024 points varying the size of the chunk computed within a task from a 1024x1024 block – 1 task –, through 256x256 – 4 tasks – and 512x512 blocks – 16 tasks –, right up to blocks of 128x128 points – 64 tasks. Figure 6.7 depicts the execution time (left) and energy consumption (right) of running the application with the four granularities (top to bottom).

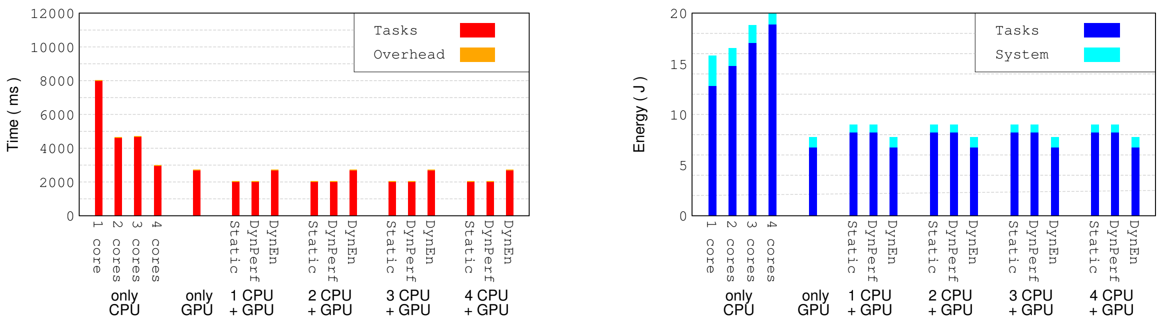
It is easy for the application developers to find the optimal number of tasks to assign to each computing device to minimalize the execution time if they consider the number of tasks, the number of available CPU cores and the speedup obtained when executing a task on the GPU compared to the CPU. The more CPUs are used at a time, the higher this speedup is. When one single core computes tasks, the GPU is about three times faster than the CPU core; when two cores compute tasks simultaneously, this ratio raises up to ~3.4x; ~3.9x, for three cores; and ~4.3x, when the four cores of the CPU compute at a time. For instance, in the case of running BS splitting the surface into 128x128-sized blocks and computing the result using a single core of the CPU, the speedup provided by the GPU is 3.03x. The optimal load balancing from a temporal point of view is to run 48 tasks on the GPU while the CPU core processes 16. The experiment assumes that the application developer is fully aware of all these details when planning the execution of tasks and codes the application; therefore, the *Static* policy emulates that behavior and adapts each execution to this knowledge.

From a temporal point of view, the *Static* policy balances the load in such a way that the execution time is minimal. As with DR, *DynPerf* behaves like *Static* in all executions (as expected) achieving the optimal performance with no significant overhead due to taking the decision dynamically. Regarding energy consumption, running all the tasks on the GPU is the optimal solution in all four cases (7.813 J, 7.736 J, 7.681 J and 7.536 J respectively for 1024, 512, 256 and 128). The cause of this reduction is the better performance of the GPU when processing smaller chunks – 2,893 ms to compute the surface in one single block vs. 2,622 ms to compute 64 blocks, 40.97 ms each –; the CPU behaves alike – 8,035 ms vs. 7, 934 ms.

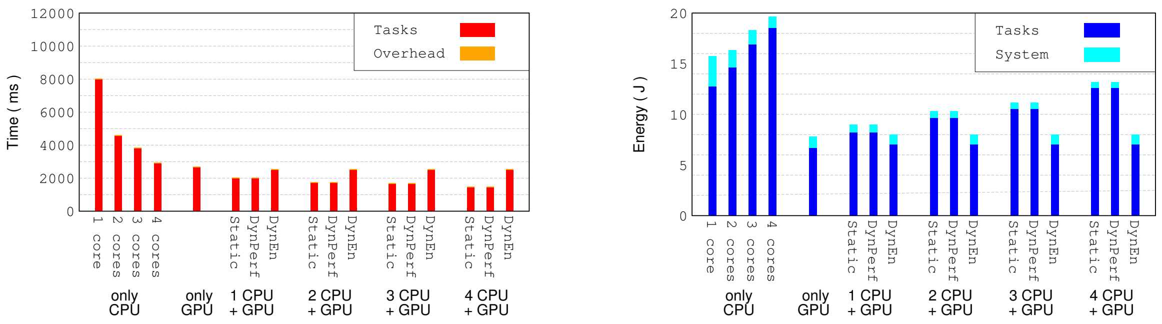
For those cases with a coarse granularity – 1024x1024 and 512x512 –, the low number of tasks and the big difference in the energy consumption of the computing devices lead the *DynEn* policy to schedule the execution of all tasks on the GPU. On finer-grained scenarios, the heterogeneous systems and the GPU present a different behavior. In the case of 256x256, one task is computed on the CPU; thus allows the application to reduce 167 ms despite an increase of 501 mJ when comparing the execution with running all the tasks on the GPU. Using two CPU cores instead of only one increases both the execution time and the energy consumption of each task run on the CPU by 72 ms and 116 mJ; *DynEn* dismisses executing more tasks on the CPU to avoid their growth. Using smaller blocks reduces the difference in time and energy; thus gives more freedom to the *Offload Decision Engine* and allows more diverse schedulings as shown by the four



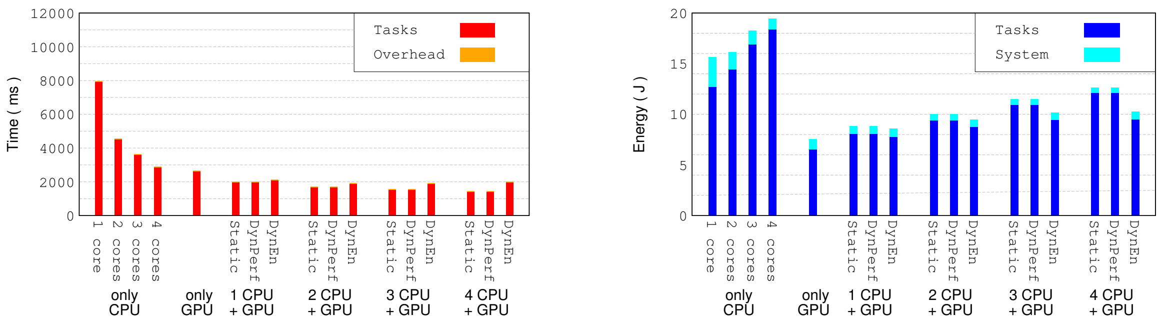
(a) 1024x1024-sized chunks.



(b) 512x512-sized chunks.



(c) 256x256-sized chunks.



(d) 128x128-sized chunks.

Figure 6.7: Execution time (left) and energy consumption (right) obtained when BS computes a 1024x1024 points surface splitted into chunks of 1024x1024, 512x512, 256x256 and 128x128 (top to bottom) using the CPU and OpenCL platforms.

heterogeneous cases using 128x128 blocks. With the GPU and one core of the CPU at its disposal, *DynEn* assigns 12 tasks to the CPU (requiring 2,122 ms and 8.586 mJ to run), while *DynPerf* assigns 16 tasks to the CPU (2,016 ms and 8.835 mJ). For the heterogeneous case using 2 CPU cores, *DynEn* assigns 18 tasks to the CPU vs. the 23 assigned by *DynPerf*. Again the growth on the execution time and energy consumption due to the concurrent exploitation of multiple cores cuts the number of tasks assigned to the CPU; *DynEn* and *DynPerf* assign 18 and 27 tasks to the CPU with three available CPU cores. For the same reason, when using all the computing devices of the phone, *DynEn* reduces the number of tasks assigned to the CPU to 16 while *DynPerf* assigns 30 to it. Thus, *DynEn* shrinks the energy consumption from 12.619 J to 10.247 J while the execution lasts 569 ms with *DynPerf*.

6.5.2.3 Canny Edge Detection

Instead of using different input sizes, the third application always processes a 30-frames video. However, the test considers two different static workload divisions that the developer could easily implement: *Task Partitioning*, where the GPU runs the first two tasks of each frame and the CPU hosts the execution of the last two; and *Data Partitioning*, where one device processes the whole frame. Figure 6.9 and Figure 6.9, which focuses on the heterogeneous cases, show the execution time (left) and energy consumption (right) obtained when running the application and compares them to the ones obtained with *DynPerf* and *DynEn*.

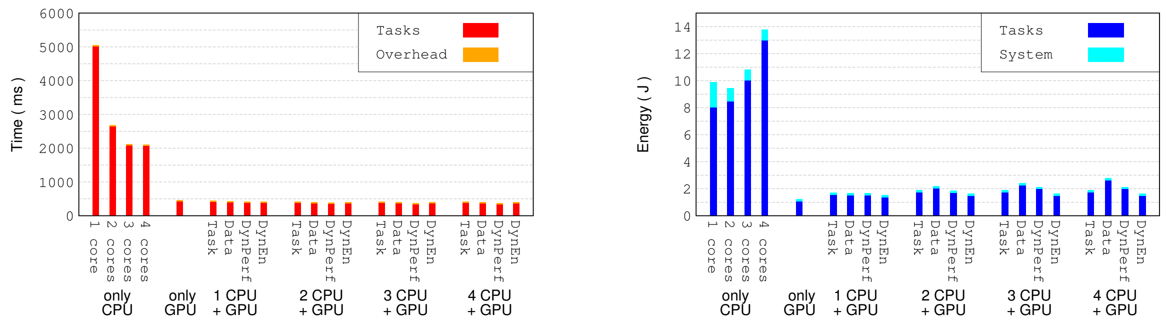


Figure 6.8: Execution time (left) and energy consumption (right) obtained when CED processes runs using the CPU and OpenCL platforms.

Task Partitioning achieves lower energy consumptions while *Data Partitioning* offers better performance. The behavior of *Task Partitioning* remains exactly the same when the runtime has two or more cores at its disposal. The time to process the first two tasks of a frame on the GPU – 12 ms – is higher than what it takes to execute the last two – 9 ms and 5 ms respectively. Thus, the throughput of the GPU is one frame per 12 ms; when only one core is available, the CPU requires 14 ms to process each frame and becomes the bottleneck. When two or more cores are available the execution of the hysteresis of one frame can overlap with the non-maximum suppression of the following frame; thus, despite the CPU still needs 14 ms to process one frame,

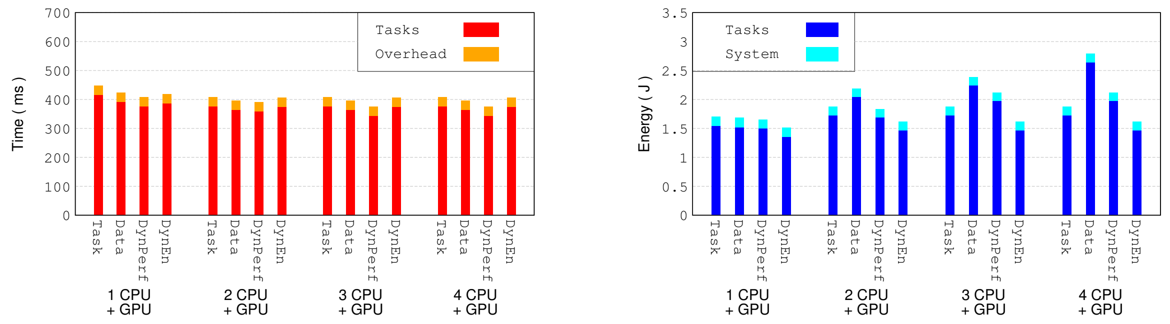


Figure 6.9: Detail of Figure 6.9 comparing the execution time (left) and energy consumption(right) obtained with different schedulers on heterogeneous cases.

its throughput rises to one frame every 9 ms. With one CPU core available, the application takes 447 ms and consumes 1.71 J, vs. 408 ms and 1.87 J when using two or more CPU cores.

Data Partitioning assigns the whole processing of a frame to the same computing unit. The problem with this approach is that the number of frames assigned to the CPU does not progress according to the number of available cores – 2, 4, 4, 4 frames, respectively for one to four cores– due to the performance loss when using multiple cores simultaneously. Using one core, the application takes 423 ms and 1.67 J. When using more than two cores, the execution time shows no improvement – 395 ms with two and four cores available; indeed, using three cores worsens the execution time to 407 ms –; however, the energy consumption reflects the usage of more cores and increases according to the number of used cores – 2.18 J, 2.40 J and 2.79 J.

DynPerf avoids this effect and schedules the executions similarly to *Task Partitioning* but adjusting the load imbalances. When only one core is available, *DynPerf* assigns four non-maximum suppressions and one hysteresis to the GPU to balance the accumulation of load due to the additional 2 ms required by the CPU to process one frame. Thus, the execution time is reduced to 407 ms consuming only 1.65 J. Conversely, when using more cores, the runtime fills their idle time with Gaussian filter tasks. With two cores at its disposal, the *Offloading Decision Engine* decides to run two of them on the CPU reducing the execution time to 395 ms with an energy consumption of 1.84 J. With more cores available, it assigns six Gaussian filter tasks to the CPU achieving a 375 ms execution time (80 FPS) with an energy consumption of 2.11 J.

DynEn tends to schedule more tasks on the GPU to avoid the higher consumption of the CPU. Hence, with one available core, the *Offloading Decision Engine* submits only 14 non-maximum suppressions and 27 hystereses to the GPU; thus obtaining an execution time of 418 ms and an energy consumption of 1.51 J – the GPU alone achieves 451 ms and 1.22 J. From two cores on, the number of non-maximum suppressions assigned to the CPU raises to 24 to shrink the execution time to 405 ms (71 FPS) with an energy consumption of 1.61 J.

6.6 Summary

Up to the beginning of Chapter 6, the presented solution allowed developers to code mobile applications in a sequential fashion that run parallelly on the multiple cores of the CPU. This chapter makes one step further towards the achievement of the goals of this thesis and introduces a new *Computing Platform*, the OpenCL Platform, which allows the runtime to exploit not only the cores of the CPU but also to execute part of the code on other computing devices embedded on the mobile such as the GPU. Figure 6.10 updates the component diagram of Figure 5.11 to depict a typical scenario for a smartphone where the runtime exploits cooperatively both computing devices, the CPU and the GPU, to execute the application.

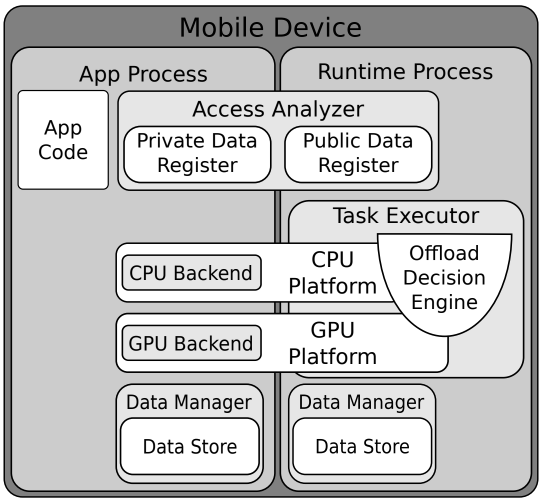


Figure 6.10: Diagram of the runtime architecture with a single CPU platform with proxied executions.

The results of the tests conducted to evaluate the prototype demonstrate the potential benefits of including the usage of accelerators embedded on the mobile device. Offloading a task execution to the GPU instead of running it on a CPU core improves the execution either from the temporal or the energetic point of view. For the CED application, GPU processes a frame ~12x faster and consumes an 87% less energy than the CPU; for BS, GPU computes a surface chunk ~3x faster and spending 54% less energy; and for DR, GPU recognizes the digits ~6.5x faster consuming only a 25% of the energy.

The new computing platform leverages on OpenCL, a standard for general purpose parallel programming for heterogeneous devices, so that the runtime offloads tasks to computing device other than the CPU, such as the GPUs, FPGAs or any other accelerator embedded on the device. Although projects like Aparapi [3] and Sumatra [80] consider automatically generating the kernels – C99-based functions – out of the Java code, the proposed solution does not go that far, and the developer still needs to write them. However, it hides away from the programmer all the details of the interaction between the host code and the OpenCL device and the implementation of a load balancing policy. For the runtime to be aware of the existence of an OpenCL implementation, developers simply need to indicate its existence by adding an *@OpenCL* directive to the CEI describing some parameters required by the OpenCL interface such as the number of work-items running the kernel.

The optimal load division might not be evident as shown in the CED test application. Dynamic load balancing policies can achieve the desired behavior with no strain for the application developer. Besides, they allow the application user to decide whether the application should aim for the lowest execution time, the lowest energy consumption or finding a balanced solution that considers both of them with no additional effort for the developer. Flexibility aside, delegating the load balancing to the runtime system also improves the portability of applications. The time to run a task and the energy consumption of the execution depend on the characteristics of the hardware running the task; therefore, the task scheduling has to be different for different computing infrastructures. Implementing dynamic load balancing policies on the runtime adapts the application to the specific infrastructure with no need of changing its code nor forcing the programmer to write complicated code that manages it.

Part IV

Exploitation of Remote Computing Resources

Chapter 7

Remote Resources Exploitation

Besides the processing elements embedded on the mobile device, which have a very low computing power, applications can turn to the infinity of computing resources available through the network. These resources can be both nearby resources connected to the same wireless network, such as laptops, desktops, servers, single board computers or even other mobiles; and computational services available through the Internet like clusters, grids or virtualized environments deployed on the Cloud. Applications may use remote resources mainly for three reasons. First, to overcome the hardware limitations of the mobile device, applications seek remote, resource-rich nodes able to host processes that require larger amounts of computational resources than the local ones; for instance, memory-bound functions in need of additional memory space.

The second reason to offload computation to remote resources is speeding up the execution. On the one hand, processors on remote nodes are likely to be faster than the ones embedded on the mobile device; hence, running the long-lasting computations on them shortens the overall execution time of the application. On the other hand, the number of task running in parallel – not concurrently – on the local computing devices is small. At its best, within a mobile device, it can run one task on each core (nowadays, high-end devices have octa-core processors) and one on each computing unit of the GPU (typically, up to four). Conversely, the degree of task-level parallelism of an application may be much higher reaching to hundreds of independent tasks. To fully exploit such parallelism, the application may use several remote resources to run of all these tasks at a time; thus, the application shortens its execution time.

Finally, the third reason to use remote resources to host the computation of part of the application is reducing its footprint on the battery of the mobile device. Running tasks on the computational devices embedded on the mobile induces an energy consumption as shown in the previous chapters; offloading the execution of such tasks to remote resources releases the mobile battery from this burden and allows the device to stay up and running for longer.

In contrast, moving the computation away from the mobile device implies bringing all the necessary data to run that part of the application into the remote node. Transferring data values may incur overheads from the temporal, energetic and monetary points of view. The time to place the data on the target node depends on the bandwidth of the network; the network interface and the quality of the network signal set the energetic expense, and fees for mobile data from the network operators and bandwidth from cloud providers influence on the price. Applications need to evaluate the additional costs and benefits of offloading each task versus a monolithic execution to find a proper balance of these three factors. Section 7.1 describes the extensions of the runtime toolkit for handling the task submission and the data management.

Unlike Fog or Edge computing, where remote resources are also devices with high mobility likely to abandon the infrastructure, the Cloud stands out for its stability and the high availability of its resources – usually five nines. Hence, network interruptions disturbing communications among cloud nodes are very exceptional and not considered in this dissertation. However, as a consequence of the high mobility of the mobile devices, network breakdowns disconnecting the main device from the remote nodes are likely to happen. Disruptions can go from glitches caused by network handovers or protocol switches to permanent isolation related to out of range situations. Applications have to adapt dynamically to these changes and recover from them. In the case of having a long-lasting network disruption while running an application whose requirements match the features of the computing resources embedded on the device, the computation should complete and produce the expected result even if the connection never reestablishes. Section 7.4 describes the fault-tolerance mechanisms implemented on the runtime to automatically handle these situations hiding them away from the programmer.

7.1 Cloud Platform

For the runtime to use external resources, it requires a new Computing Platform: the Cloud Platform. As with the CPU and the GPU, this platform has to implement the mechanism that enables the execution of tasks on the resources – other nodes of the infrastructure – and provide the runtime toolkit with the forecasts for the temporal, energetic and monetary costs for the mobile device of running a task remotely.

Unlike the previously described platforms, the Cloud Platform potentially controls a big amount of resources separated across several nodes. If not appropriately addressed, the complexity of managing a large infrastructure may lead to a significant computational overhead. To release the mobile device from this load, the platform consists of two parts. A single, centralized frontend deployed on the mobile device computes the forecasts for the runtime to determine whether offloading the task to remote resources or running it locally. The backend of the platform is a distributed system organized as a hierarchical peer-to-peer network. Each node of the infrastructure hosts an agent that persistently listens to the network waiting for new task submissions.

Once the *Offload Decision Engine* picks the Cloud Platform to host an execution, it forwards the corresponding task description to the frontend of the platform where a *Scheduler* picks one of the nodes to submit the task. For the initial version of this component, the *Scheduler* implements a basic scheduling policy that estimates the end time of the task on each node and picks the one finishing the execution earlier. For that purpose, the *Scheduler* keeps track of the number of tasks waiting to execute on each node and performance-related information of the nodes when running similar tasks, obtained through the profiling of previous tasks. The tiebreaker criteria for those cases where multiple nodes can finish the execution at the same time is the size of input parameters missing on the node. The more bytes of input values are missing on the node, the lower priority it has to host the task execution. Again, two nodes with the same estimated end time can have the same amount of input data missing – this situation is very likely for the first tasks of the application –; for definitely setting a preference on these cases, the *Scheduler* prioritizes those nodes without receiving a task submission in a longer time. Chapter 9 describes an evolved system implementing a more sophisticated policy and releasing the mobile device from the burden of its computation.

Figure 7.1 depicts the architecture of the Cloud Platform and the described flow to execute a task on the remote CPU cores. Upon the selection of one node, the frontend sends through the network an internal command to the backend instance on the corresponding node to offload the execution of the task (step 1). All the communications through the network among components of the runtime toolkit transfer the information using TCP sockets. To enable non-blocking communications that allow sending messages to other nodes without stalling the processing, the asynchronous management of the threads to read from and write to these sockets is enclosed within a *Communication* component that is replicated on every process of the infrastructure. This component leverages on the non-blocking I/O library, a set of APIs offered by the Java language to perform intensive I/O operations. The *Scheduler* in the frontend of platform submits the execution command by asking the local *Communication* component to transfer the command to the remote *Communication* component instance of the corresponding backend so that the latter delivers it to another *Scheduler* component on the backend of the platform.

Unlike the *Scheduler* component on the frontend, which only selects a resource to host the execution, the *Scheduler* on the backend is the responsible for the proper execution of the task on the resource. As with the *Scheduler* components for the CPU and OpenCL Platforms, the *Scheduler* on the backend not only plans the execution of the task but also ensures that all the data values required to produce the proper result are available to the executing resources. To control the data dependencies the *Scheduler* leverages on an instance of the *Data Manager* as

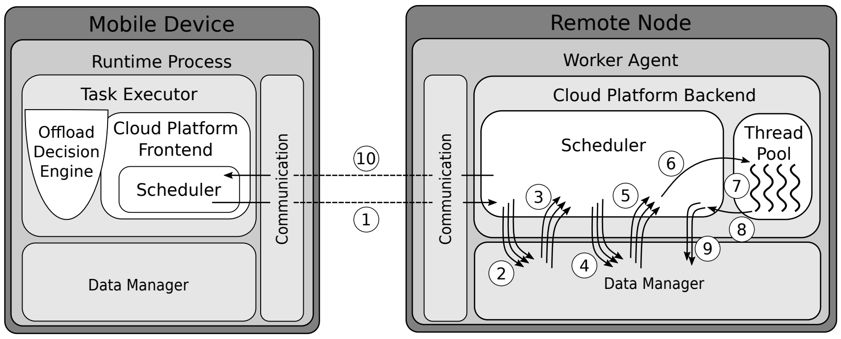


Figure 7.1: Architecture of the Cloud Platform illustrating the flow involving a task execution.

previously described platforms do. First, the *Scheduler* queries the *Data Manager* about the existence of every datum used as an input for the just received task; when the latter notices that the data values are available on some node of the whole infrastructure, it contacts the former back to notify the existence (steps 2 and 3). At this point, the *Scheduler* plans the preparation of all the missing input values and the execution of the task on the remote resources. Currently, the *Scheduler* prepares all the input values as soon as it receives the existence notification for every input datum. Task executions start as soon as there are free resources following a first-come, first-served policy based on the moment when all the input values are ready for being processed.

Given that the only processing devices exploited on the remote node are its CPU cores, ensuring that a data value is on the node is enough to consider it prepared for use. For that purpose, the *Scheduler* contacts again the *Data Manager* to request each of the input values (step 4). If one value is not currently on the remote node, the *Data Manager* fetches it from another location. Once the value is on the node, the *Data Manager* notifies the presence of the value to the *Scheduler* (step 5), which enqueues the task (step 6) so that one of the threads of the *Execution Pool* polls it from there and runs it (step 7). At the end of the execution, the thread informs the *Scheduler* about the task completion (step 8), and the latter publishes the results of the task on the *Data Manager* (step 9). Finally, the backend of the platform reports the execution of the task to the frontend through the *Communication* component (step 10). Along with the task completion command, the backend sends the profiling information of the task execution indicating the execution time and the size of every input and output datum. Thus, the *Offload Decision Engine* and the *Scheduler* of the frontend of the platform can use this information to assess the models for forecasting the costs of future executions.

7.2 Data Manager Implementation

The way how remote workers obtain the data values has a strong impact on the time, energy and money dedicated to any execution. Executing an application with the mobile device shipping to the corresponding remote node all the necessary data values before running the task and collecting all the results at its end leads to very high costs. The limited incoming and outgoing bandwidths of the network interfaces may convert the transfer of data values in a bottleneck for the execution. Besides, the transmission and reception of data through the network incur an additional energy consumption and potentially some monetary costs if the network is subject to fees.

The design of a system that enables sharing data among all the nodes of the infrastructure is of vital importance to avoid raising the costs of the execution unnecessarily passing through the mobile device. For that purpose, the implemented mechanism builds on a data directory that maintains the correspondence between the Id associated to a datum with the locations where to find its value. Upon the obtaining of a specific value – computed by a task or transferred from another node –, the *Data Manager* instance storing it registers on the directory the presence of the datum on that process. To obtain a missing input value, *Data Managers* look up the nodes containing the desired datum on the data directory and select one of the multiple sources to request the transfer of the value directly to the source process. *Data Managers* also turn to this data directory to check the existence of the value. If the datum is registered, the value already exists and the directory notifies the querying *Data Manager*. Otherwise, the directory registers the query and notifies the existence upon the registration of its first location.

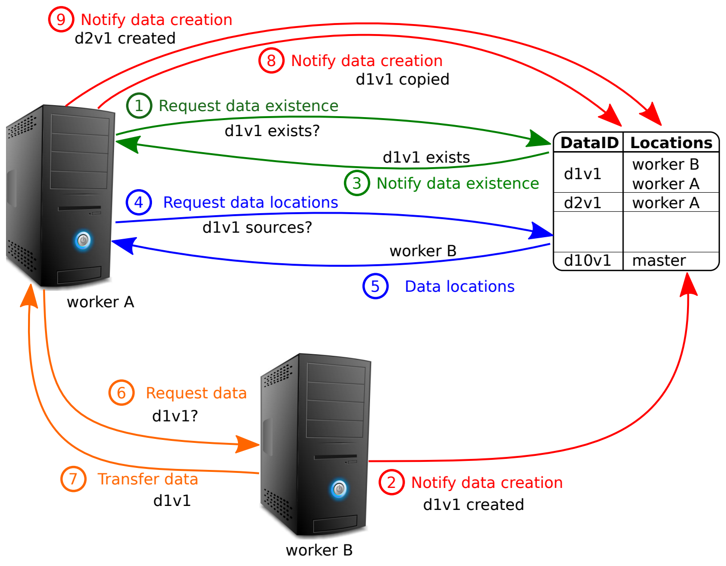


Figure 7.2: Data creation notification and transfer request.

Figure 7.2 illustrates all the interactions that enable *Worker A* to obtain the value *d1v1*, created by another task that runs on *Worker B*. When *Worker A* receives the task, it checks out the existence of all the input values – among them *d1v1* – by querying the data directory (1). If the data value is not available yet, the directory registers the requests and waits for the value creation notification (2); conversely, if the value-existence is already registered, the directory immediately confirms the data availability to *Worker A* (3).

When the worker notices that all the input values already exist, its scheduler processes the task and checks the local availability of the input data. In the case of missing input values – for instance, *d1v1* –, the scheduler decides when to trigger the obtention of that piece of data. To obtain a value, the worker retrieves all the available locations from the data directory (4 and 5). It picks one of the sources trying to avoid the mobile node and opens a connection to the corresponding node – *Worker B* – through which the worker asks for the actual value (6) and the hosting node sends it (7). Once copied, *Worker A* registers a new location for data *d1v1* in the data directory (8)

Once all the input data has been obtained, the worker node can execute the task according to the plan established by the scheduler. Once the task ends, the node publishes the creation of the output values into the directory (9) and the notification is forwarded to every worker that is waiting for that value; thus, enabling the execution of successor tasks, as happened with step 3.

The data directory is a very critical data structure accessed by all the processes of the infrastructure for querying and updating its content. The main application updates it when it creates new values to use during task executions and queries it to retrieve the results of such executions. Worker nodes request the locations of the input values of a task before executing it and update the locations of each execution results at completion. In the same way, the runtime system does so when it decides to run a task on the mobile device itself.

A first simplistic approach consists in centralizing the management of the content of the data directory on the runtime service process since it hosts information about data created by all the applications. This strategy has two strong points. First, implementing such approach is easy and quick; simply, the workers contact the mobile device on every access to the structure. The second advantage is the immediacy of access for the computing platforms using the computing devices embedded on the mobile.

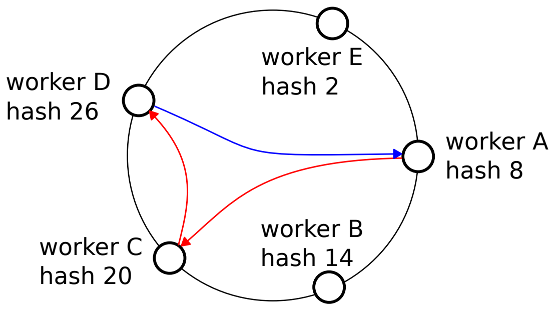
The main drawback of this approach is the overheads caused by hosting this structure. Processing the accesses to the data directory implies an additional computational load that could turn the host into a bottleneck. Besides, the network conditions on the mobile also have a significant impact on the whole system performance. Using a network with a high latency would slow down every query to the directory from the remote nodes. The energy consumption of the mobile also increases with the reception, processing and replying of each access to the data directory, and transferring data in and out from the phone or the cloud may also incur some economic expenses.

Implementing the data directory as a hash table distributed among nodes of the infrastructure mitigates these problems. The computational cost of hosting the data structure scatters across all the nodes containing the information, and the computational bottleneck disappears. Besides, the workers composing the remote infrastructure are usually nodes on the same cluster or VMs deployed on the same Cloud. The interconnection between this nodes is usually a high-speed and low-latency network; thus, also the latency problem disappears when the workers interact with the directory. The mobile network latency only affects to interactions between the mobile device and the directory. However, the mobile device usually only runs tasks whose input data is already on the phone. Transferring the data back from the remote resources and processing the task on the local computing devices, which are likely to be slower, is seldom worthy; the *Offload Decision Engine* tends to opt for offloading the computation onto the remote resources. To mitigate the network latency problem on the queries from the mobile device, *Data Managers* check if the requested values are already in the local process before querying the data directory. Checking the local content before querying the data directory not only lightens the impact of the latency on most of the requests to the *Data Manager*, but also reduces the number of commands incoming and outgoing from the mobile device and, hence, the energetic and monetary costs associated to them.

To implement the distributed hash table, the nodes of the infrastructure build a peer-to-peer network organized as a ring. The results of evaluating the hash function for a random number in every peer determines its respective position on the ring. The neighbors of one peer are those with the closest larger and smaller values. Each peer is responsible for a range of the hash function image corresponding to the hash value of a set of datum identifiers. The first value of the range corresponding to one peer is the hash value that determined its position on the ring; the end of such range meets the beginning of the range controlled by its successor, – i.e., the neighbor with a larger hash value.

On strict ring topologies, one peer only knows its predecessor and successor. Therefore, to query/update the value associated with a hashcode out of its range, the peer contacts one of its neighbors. If the value is out of the range of this immediate neighbor, the latter forwards the message to its other neighbor until it reaches the peer responsible for that hashcode. If the responsible peer needs to reply the message, it submits a notification that retraces the path followed by the query. To reduce the number of hops needed to reach the responsible for a hashcode, each peer knows not only its successor in the ring, but it also has a lookup table that indicates the responsible peer for a set of hashcodes: the first hashcode of its range plus an offset (powers of 2). When the peer needs to interact with the data directory for a given datum, it computes the hashcode of the identifier of the datum, looks for the closest smaller hashcode on this table and sends the message to the corresponding peer. In the case that the receiving peer is not responsible for that hashcode, the latter forwards the message according to the values in its table. Every message registers its original source; therefore, the peers can directly contact that source to reply the message.

Figure 7.3 depicts an example of a five-peer network and 32 possible hashcodes. *Worker A* responds for the interval starting at hashcode *8* and ending at *13*; *Worker B*, *14* to *19*; *C*, *20* to *25*; *D*, *26* to *1*; and *E*, *2* to *7*. The figure also shows the lookup table in *Worker A* and the route followed by an interaction between *Worker A* and the data directory for the datum *d1v1*. In this example, *Worker A* queries the locations of the datum, whose hashcode is *29*. First, it checks if the hashcode is within its responsibilities; it is not, so it looks for the peer responsible for hashcode *29*. Since that entry does not exist, it looks for the closest lower hashcode in the table, $8+2^4$ – i.e., *24* –; and sends the request to the corresponding peer. Upon the query reception, *Worker C* follows the same process: it checks if hashcode is within its range, no; looks for the next possible responsible: *Worker D*; and forwards the request. *Worker D* is the actual responsible for hashcode *29*; therefore, it gets the registered locations for the value and ships them to *Worker A*.



Hash	Resp
8	A
8+1	A
8+2	A
8+4	A
8+8	B
8+16	C

Figure 7.3: Example of a data directory query in a five-node peer-to-peer network sharing a [0-31] hashcode range and the route followed by a query access to hashcode 29 from Worker A.

The described lookup procedure ensures reaching the responsible node in $min(\ N, \mathcal{O}(\log_2 H))$ hops, where N is the number of nodes in the network and H the size of the hash function image. For small networks, this system does not show any inconvenience; for large networks, the request should do several hops before reaching its target. To reduce the number of hops, only a subset of peers are part of the ring and store the data of the distributed hash table. The rest of peers are represented in the ring by one of these selected peers: the one with the closest lower hash to its initial hash value. Represented peers are unaware of the content of the data directory; however, they contain a replica of the lookup table of their representative peer. Thus, they can access the data directory without the need for using their representative as a gateway and avoid an additional hop on their route.

7.3 Cost Forecasting

Besides managing the resources in the platform and ensuring the proper execution of the tasks on them, Computing Platforms have to provide the forecast of the temporal, energetic and monetary cost for the mobile device of running the task on its resources.

The three models for the Cloud Platform are very similar to the models used for the CPU and the OpenCL Platforms; however, two important considerations differentiate them from the ones described in sections 5.1 and 6.5. First, workers can directly fetch data values from other workers; therefore, the mobile device only needs to transfer those values that are not on the remote nodes. The second factor to consider is the heterogeneity of the resources managed by the platform. While the CPU and the OpenCL platforms considered all the processing elements of the platform to have the same features, the remote resources are likely to be different, especially when using nearby resources such as multiple laptops or servers.

The actual computation of the task does not incur any energetic expenditure for the mobile device since the energy dedicated to it is part of the remote resource consumption; hence, it has no impact on the battery lifetime. Regarding the price, computing services usually do not charge for the time that the user is actually computing on the resources, but instead, they charge according to the time that a user reserves them. For instance, Cloud providers charge the same amount for a VM instance whether it is actively computing or idle. Neither the energy nor the economic model considers the costs of executing the task; they only take into account the cost of the data transfers related to the task. Since workers prioritize obtaining the data from another worker, the mobile device has to send only those input values without remote locations. Applications can access any of the values created or updated by the task and transfer it back to the mobile; however, it is impossible to determine, at the moment of computing the forecasts, whether the mobile will fetch the value or not. Assuming a worst-case scenario, both models presume the mobile to bring back all the output of the task. Both models compute the total sizes of the amount of data to emit to and receive from the remote workers and respectively multiply them by the price and energy consumption for emitting and receiving one byte.

To estimate an end time for the task, the platform first needs to determine on which node the task will run. Given that the criteria to pick a resource is the earliest end time, the platform only needs to estimate the end time on each resource to find out on which one runs the task and return the corresponding forecast. As with the CPU and the OpenCL Platform, to compute the end time, the platform considers two aspects: the expected start time and the length of the execution. The heterogeneity of the resources managed by the platform complicates the forecasting of the temporal cost for the task since the platform has to consider a different behavior of each implementation on each node. For that purpose, the platform maintains the performance stats of each node in separated *Implementation Profiles* and uses the shortest average execution time of all the possible implementations of the task as the time required to compute the task on the node.

For estimating the start time of the task, the platform takes into account when the node has free resources to host the execution and when all the data is available on the node to launch it. For the resource availability, the platform just divides an estimation of the timespan to compute the workload assigned to the node sequentially by the number of CPU cores on it. Estimating a time for the obtention of the input data of the task requires knowing the creation moment of each datum and the timespan of the possible transfers from their producing node. To determine a datum availability on a node, the Cloud Platform not only stores the expected end time of the task producing the datum but also for each datum its expected obtention time on each node managed by the platform. Thus, while computing the time forecast, if a datum has a registered obtention time for the node, the platform uses that one; otherwise, it is necessary to estimate the obtention time. If that datum is expected to be on any other node part of the platform, the node should fetch it from there immediately or upon its creation through the high-speed connecting the workers. The platform determines the transfer time using the size of the datum – obtained from the *Core Profiles* – and the network bandwidth – deduced from the profiles of previous executions. Otherwise, if the register has no node of the platform expecting to have the datum, the platform considers that the master node generates it; the node will fetch the value using the mobile network and the transfer time computed by the platform should reflect the lower network bandwidth. When the platform offloads the task, it registers for each output datum the expected end time for the task as the expected obtention time for the value on the node. For those input data values whose obtention time on the node is not registered yet, it registers the computed estimation. Figure 7.4 notates the described models.

7.4 Fault tolerance

As aforementioned, the high mobility of mobile devices leads to temporary or persistent network disruptions; applications have to be prepared to monitor the environmental conditions and react to changes in them.

Typical causes of network disruptions are Wi-Fi network handovers and switches on mobile network protocol which produce a temporary isolation of the mobile and may change the network address of the device. To tolerate these situations and avoid losing the master-workers connection persistently the mobile sends a message to every worker node upon the reconnection describing the new network context – mainly containing the new IP address. Worker nodes update every reference to the master and re-establish any interrupted connections.

For long-lasting disruptions, both, mobile and worker nodes, should keep progressing despite their isolation. Workers autonomy ensures that the network interruption has the smallest impact possible on the performance of the application in the case of reconnection; autonomy on the mobile device allows applications to give the expected result to the user despite the poor performance.

When the network disruption bisects the infrastructure, it is of capital importance that both parts maintain the ability to know which reachable nodes can provide the values required to run a task. Being able to fetch values from other nodes enables the execution of pending dependency-free tasks that produce new values; publishing the existence of such results releases other tasks from pending data dependencies allowing their execution and the whole application progress. Therefore, the data directory plays an essential role on the autonomy of the sections.

On the mobile side, the device becomes totally isolated from the rest of the infrastructure; thus, it can check which values remain available merely by looking at its local data store. Storing the data directory distributed only among the remote nodes protects its content from any problem with the mobile network and guarantees worker nodes access at any time. Therefore, all the offloaded tasks can run except for those involving input values only located in the mobile. In the case of an eventual reconnection, the mobile device behaves as if it were recovering from a glitch,

End time:

$$RA_n = \frac{\sum_{c \in CEs} TC_{cn} * XBT_{cn}}{NC_n}$$

$$BR_{dn} = \mathbb{1}_{MD}(d) * BN_e + (1 - \mathbb{1}_{MD}(d)) * BRN$$

$$TT_{dn} = DS_d / BR_{dn}$$

$$DA_{dn} = DC_d + (1 - \mathbb{1}_{ND_n}(d)) * TT_{dn}$$

$$DA_{tn} = \max_{d \in ID_t} DA_{dn}$$

$$ST_{tn} = \max\{RA_n, DA_{tn}\}$$

$$XT_{tn} = \min_{i \in Impl_t} XT_{in}$$

$$ET_{tn} = ST_{tn} + XT_{tn}$$

$$ET_t = \min_{n \in N} ET_{tn}$$

Energy consumption:

$$ES_d = DS_d * \mathbb{1}_{MD}(d)$$

$$EE_t = EN_E * \sum_{d \in ID_t} ES_d$$

$$ER_t = EN_R * \sum_{d \in OD_t} DS_d$$

$$E_t = EE_t + ER_t$$

Monetary cost:

$$ES_d = DS_d * \mathbb{1}_{MD}(d)$$

$$CE_t = CN_E * \sum_{d \in ID_t} ES_d$$

$$CR_t = CN_R * \sum_{d \in OD_t} DS_d$$

$$C_t = CE_t + CR_t$$

Variable	Description
RA_n	Expected time when resources become available on node n
CEs	Application Core Elements
TC_{cn}	Number of tasks of core c pending to run on node n
XBT_{cn}	Execution time for the best implementation for core c on node n
NC_n	Number of CPU cores available on node n
BR_{dn}	Bandwidth to receive data d on node n
MD	Set of values only contained on the mobile
BN_e	Network sensor emission bandwidth
BRN_e	Bandwidth of the network among remote nodes
TT_{dn}	Time to transfer value d to the node n
DS_d	Data size for value d
DA_{dn}	Time when value d is available on the node n
DC_d	Time when value d is generated
ND_n	Set of values only contained on node n
DA_{tn}	Time when all ID values for t are available on the node n
ID_t	Input data values for task t
ST_{tn}	Expected start time for task t on node n
XT_{tn}	Execution time for task t on node n
$Impl_t$	Set of implementation for task t
XT_{in}	Execution time for implementation i on node n
ET_{tn}	End time for task t on node n
ET_t	End time for task t
N	Set of nodes managed by the platform
ES_d	Emission size for value d
EE_t	Energy to emit input data for task t
EN_E	Energy to emit one byte
ER_t	Energy to receive results of task t
EN_R	Energy to receive one byte
OD_t	Output data values for task t
E_t	Energy consumption for task t
CE_t	Cost to emit input data for task t
CN_E	Price for emitting one byte
CR_t	Cost to receive the results of task t
CN_R	Price for receiving one byte
C_t	Cost for task t

Figure 7.4: Models to forecast the end time, energy consumption and monetary cost of running a task t with implementation i on the local CPU cores.

and submits the message describing the new network context. To update the content of the data directory and include all the values created during the network disruption, it is enough that the mobile device publishes all the values computed during the isolation period since the rest of the nodes already registered the new values as usual. Once the data directory synchronizes its content with the mobile device reality, the execution goes on as if the network never dropped.

Otherwise, if the connection never re-establishes, the mobile device must be able to provide a result by running all the missing tasks on the local computing resources. Initially, the mobile device runs the tasks assigned to the platforms managing the embedded computational resources: the CPU and OpenCL platform. Both platforms prioritize the execution of those tasks whose input values are already on the node over those with pending data dependencies or missing input values; hence, those tasks with all the input values on the mobile at the moment of the disruption can run. Such executions create new values that potentially release from data dependencies other tasks scheduled to run on the local computing devices. These just dependency-free tasks, whose input values are on the mobile, have priority over older tasks with some input value missing, so they can execute and keep on the progress of the execution as if the network never went down.

If the connection is not re-established, the application can reach a point where some computing resources stall because all the scheduled tasks require values created by tasks offloaded onto the Cloud. For being able to go on with the execution, these values need to be computed again by re-running the producing task on the master. When the *Offload Decision Engine* realizes that the mobile is working off-line and that the devices are awaiting for tasks blocked due to external dependencies, it picks one of the blocked tasks and iteratively checks the state of the input parameters. Upon the detection of a missing input parameter, the *Offload Decision Engine* looks for the task producing such datum and checks the viability of its execution. If all its input parameters are on the mobile, it submits the execution of the task to one of the computing platforms handling embedded computing resources. Otherwise, it tries to run the task producing the missing value. This recursive procedure ends up becoming a backtracking mechanism that allows the runtime to generate any missing value to execute a task.

If the disruption persists for longer, the embedded computing resources will execute all the tasks scheduled to run on them, and they will stall not because of the missing input values but for the lack of tasks scheduled to run on them. If the *Offload Decision Engine* notices that some local resources are unemployed and all the tasks assigned to the corresponding platform are already running, it reassigns pending tasks previously offloaded to the *Cloud Platform* and submits their execution to the platform with available resources.

To prevent this backtracking process from running all the offloaded tasks locally to generate the values, the runtime fetches output values of offloaded tasks to establish checkpoints to avoids the re-execution of the whole sequence of tasks preceding the data value generation. Collecting all the results at the end of the producing task guarantees that no task is re-executed on the mobile device; however, it increases the energetic and monetary costs of the execution due to the amount of additional data transferred through the network. To reduce the amount of information brought back to the mobile for checkpointing purposes, the runtime picks some strategic values. For that, it splits the graph – currently, fixed-size partitions according to the chronological order of task generation – and analyzes each partition to determine which of the values generated within the block might be used on other partitions; i.e., it only saves the outcoming version of each datum and dismisses all the intermediate ones. The selected values are transferred back to the mobile device as soon as the producing worker notifies their creation. Once the runtime fetches all the output values from a block, the tasks of the block are removed from the runtime.

7.5 Evaluation

This section presents the results of the test conducted to evaluate the impact of enabling the use of remote resources on applications. In this case, the tests run a compute-intensive application: HeatSweeper, an excerpt of the workflow of several engineering solutions. The goal of the application is to find the optimal placement of 1-to-N heat sources on the surface of a solid body to reduce the time to heat it up. For that purpose, it performs an intensive search algorithm looking for the best combination of 1-to-N location for the heat sources. To simulate the heat diffusion, the application relies on two different solvers based on the Jacobi (used on the tests) and Gauss-Seidel equations.

The COMPSs version of the application encapsulates each simulation within a task – the *simulate* Core Element – that receives the simulation parameters containing the position of the heat sources along with a surface description and some algorithm-constant parameters. At the end of the simulation, the task generates a report describing the results of the simulation. The application compares pairs of these reports using the *getBest* method, also encapsulated as a Core Element, creating a binary tree of comparisons to select the best combination. Figure 7.5 depicts the task dependency graph of a HeatSweeper execution that optimizes the placement of up to three heat sources with four possible locations.

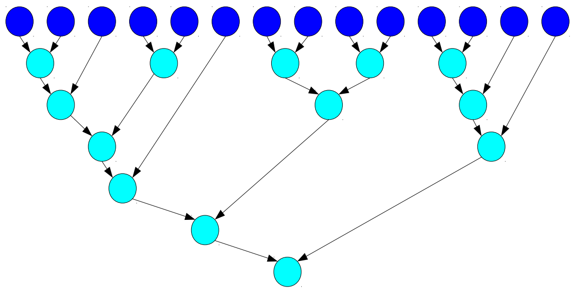


Figure 7.5: HeatSweeper task dependency graph for a three sources optimization on four possible locations resulting in 14 simulations and 13 getBest executions. Dark blue nodes represent *simulate* tasks and cyan nodes depict *getBest* executions.

The tests consider two different configurations that aim to optimize the placement of up to two heat sources. The low-resolution configuration, with only nine possible locations and short simulations of up to 50 time-steps each, which creates only 45 *simulate* tasks and 44 *getBest*. The purpose of this configuration is to verify the behavior of the runtime when dealing with applications with a low number of short-lasting tasks. To emulate large computations, such as the execution of scientific workflows, the tests consider the high-resolution configuration where the sources have 25 possible spots on the surface and simulations take up to 10,000 time-steps. This configuration generates 325 *simulate* tasks and 323 *getBest* task.

As with the tests conducted for the CPU and OpenCL Platforms, the application runs on a OnePlus One smartphone with the the Cyanogen OS 13.1.2 implementation of Android 6.0.1 using the default processor governor (*interactive*). However, neither the CPU nor the OpenCL platforms are enabled to run tasks. The runtime only has one single Computing Platform, the Cloud Platform, which offloads tasks to remote resources. Regarding the resource managed by the platform, the test sets out two different environments: one where the runtime offloads the tasks to a laptop connected to the same local network, and a second one where the resources are in a geographically remote location, and the mobile requires Internet to reach them. For the Local Area Network, the laptop has an Intel i7-2760QM quad-core processor at 2.40GhZ and 8 GB of RAM. In particular, the network connecting both devices is an 802.11g wireless network. On the Wide Area Network scenario, the mobile can use a virtual cluster on a private OpenNebula cloud. The cluster has eight quad-core VMs deployed on nodes with one hexa-core Intel Xeon X5650 processors with hyperthreading at 2.67 GHz and 24 GB of memory interconnected by a Gigabit Ethernet network. In this case, the mobile also connects to the Internet through an 802.11g network; the RTT among the mobile and the remote nodes is 133 ms. Table 7.1 contains the energy and time-related measurements obtained when benchmarking the computing devices that compose the testbed while running the Core Elements of the application.

	50 iters. sim.		**10k iters. sim.**		**Merge**	
	Time (ms)	Energy (J)	Time (ms)	Energy (J)	Time (ms)	Energy (J)
Mobile - screen off	35,549	6.72	6,794,135	1,350	negl.	negl.
Mobile - screen on 0%	1,483	2.85	288,667	561.61	negl.	negl.
Laptop	38	-	6,072	-	negl.	-
Cloud	57	-	27,979	-	negl.	-

Table 7.1: Relation between each computing configuration (Mobile with screen off, mobile with the screen on at 0% brightness, laptop or cloud VM) with the analysis of each core element execution.

Table 7.2 shows the results obtained through a network benchmarking to measure the effective bandwidth of the connection and the energy consumption related to its usage under different environmental conditions. Unlike the computing capacity, where the operating system reduces the processor frequency when the screen is off to save energy, the network performance is not affected by any battery-saving policy since the power difference matches the display consumption. Enabling the Wi-Fi interface of the mobile sets a base consumption of 0.04-0.12 W depending on the strength of the network signal. The power difference due to the signal strength is constant regardless the action performed on the network and the latency of the network. However, it has a significant impact on the effective bandwidth of the network connection what increases the cost per sent/received byte. To evaluate the runtime, the tests consider that the strength of the signal remains above an 80%.

	Sensor Off	**Sensor Idle**	**Phone ↔ Laptop**				**Phone ↔ Cloud**			
			Phone → Laptop		**Laptop → Phone**		**Phone → Cloud**		**Cloud → Phone**	
	Power (W)	Power (W)	Bandwidth (MB/s)	Power (W)	Bandwidth (MB/s)	Power (W)	Bandwidth (MB/s)	Power (W)	Bandwidth (MB/s)	Power (W)
Screen off 100% signal	0.05	0.08	2.31	0.63	2.68	0.42	0.34	0.45	0.50	0.35
Screen 0% 100% signal	0.31	0.37	2.38	0.94	2.61	0.70	0.34	0.73	0.50	0.61
Screen 0% 50% signal	0.31	0.43	0.77	1.00	1.13	0.77	0.32	0.78	0.36	0.69

Table 7.2: Network benchmark results.

7.5.1 Exchanged Message Evaluation

The first conducted test aims to measure the amount of traffic incoming to and outgoing from the mobile phone when using different numbers of worker nodes - one to eight - and different distributions of the data directory - centralized on the mobile, distributed across all the nodes of the infrastructure and distributed only across the worker nodes. The placement of the data directory has a significant impact on the kind and number of messages transferred from and to the mobile. Table 7.3 shows which types of message are sent from and received by the mobile device depending on the deployment of the data directory. Regardless of the locations of the data directory, every time that the mobile device fetches a data value located at a remote node, it sends a data value request to the remote node and receives the value; and vice versa when a remote node needs a value hosted by the mobile. In case of a centralized data directory on the mobile, the device receives the creation notifications of every value remotely computed and receives and replies all the existence or sources queries required by the workers. Conversely, when the directory is located only on the worker nodes, the mobile publishes the creation of data values – locally computed or received from a remote node –, subscribes for the existence and sources of values and receives their corresponding notifications. In the case where the phone has a share of the data directory and the other parts are distributed among the workers, the interactions with the directory depend on the specific hashcode of the value and the responsibilities of the mobile device. If the mobile node is responsible for the hashcode of one data value, it receives any remote data creation notification; otherwise, it sends creation notifications for those values locally computed. Similarly, the subscriptions to existence/sources and their corresponding notifications also depend on specific data hashcode.

Table 7.4 presents the number of messages and the number of bytes transferred in to and out from the mobile device during a low-resolution execution according to the number of workers used and the same data directory placements as in the previous table – only in the phone node (Mobile), shared among the workers (Workers) or across the whole infrastructure (Mobile + Workers). Although the scheduler in the frontend of the Cloud Platform is aware of the data locality, tasks

	Mobile	**Mobile+Workers**	**Workers**
Creation notification	Received	Received/Sent	Sent
Existence request	Received	Received/Sent	Sent
Existence response	Sent	Received/Sent	Received
Sources request	Received	Received/Sent	Sent
Sources response	Sent	Received/Sent	Received
Value request	Received/Sent	Received/Sent	Received/Sent
Value transmission	Received/Sent	Received/Sent	Received/Sent

Table 7.3: Direction of each type of message according to the placement of the data directory: centralized on the mobile device (mobile), hosted by the worker nodes (Workers) or shared among all the nodes composing the infrastructure included the mobile (Mobile+Workers).

running on one node of the infrastructure may have dependencies with values generated on other nodes. The more nodes being part of the infrastructure, the less likely one node is to host all the values that a task requires; thus, nodes fetch values from other peers more often. Besides, as the infrastructure grows, the smaller the local share of the directory gets; and hence, the more queries to the data directory require information stored on other nodes. When the mobile device hosts the data directory – either the whole of it or just a share –, the number of messages processed by it increases as the size of the infrastructure does.

	Mobile			**Mobile + Workers**			**Workers**		
	number of messages	input bytes	output bytes	number of messages	input bytes	output bytes	number of messages	input bytes	output bytes
1 worker	856	124,275	137,387	799	94,753	162,626	765	83,073	168,123
2 workers	961	136,051	148,213	886	97,854	179,228	765	83,984	168,123
4 workers	1,008	140,384	153,054	971	111,861	185,599	765	84,575	168,123
8 workers	1,016	140,525	154,587	1,098	136,211	199,820	765	84,906	168,123

Table 7.4: Number of messages and number of bytes received/transmitted by the mobile during a low-resolution execution according to the size of the underlying infrastructure and the nodes hosting the data directory (the mobile device, the worker nodes or shared across the whole infrastructure.

Distributing the data directory among all the nodes, including the mobile device, may enforce the master to interact with remote nodes to notify every locally created/accessed value and, besides, to reply queries from other nodes fetching values. If the mobile manages the hashes corresponding to all the values locally accessed, it only needs to reply the existence and sources requests from other nodes. Otherwise, if it manages none of the values it accesses, it needs to submit a creation notification for every value creation, request existence/sources requests corresponding to the values it fetches and to reply to queries from remote nodes to the controlled values. Besides, it assumes part of the traffic to forward to other nodes of the ring. Compared with the centralized approach, the option of distributing the data directory among all the nodes can either reduce or increase the number of messages.

Conversely, when only worker nodes host the directory, the mobile interacts with the data directory just to fetch remote data and to notify the local creation of data values. In this case, the number of messages depends on the application itself rather than on the infrastructure. The size of the output data – queries and notifications – remains constant, but the input size may change depending on the number of sources for the accessed values. The more nodes being part of the infrastructure, the more likely they are to grow. When the directory is deployed only atop worker nodes, the number of messages and the size of the input communications is always smaller than in the other deployments; the more nodes the infrastructure has, the more significative this reduction is.

Table 7.5 shows the same information included in Table 7.4 but for a high-resolution execution. Despite the bigger number of messages and the larger number of bytes transferred in to and out from the mobile device, the conclusions extracted from it are the same as with the low-resolution test case. When the mobile device hosts the data directory, either partially or totally, the number of messages and the number of transferred bytes grows along with the infrastructure; while they remain almost constant when the data directory is placed on the workers. Sharing the data directory across the whole infrastructure may increase or decrease the number of exchanged messages depending on the hashcode set associated to each node when compared to the centralized approach.

	Mobile			**Mobile + Workers**			**Workers**		
	number of messages	input bytes	output bytes	number of messages	input bytes	output bytes	number of messages	input bytes	output bytes
1 worker	6,238	912,052	1,007,647	5,712	678,521	1,174,913	5,525	604,123	1,222,494
2 workers	7,817	1,108,741	1,225,531	6,635	737,570	1,330,608	5,525	612,284	1,222,494
4 workers	7,254	1,017,989	1,114,466	8,244	956,214	1,552,528	5,525	614,921	1,222,494
8 workers	7,437	1,037,932	1,135,8892	8,963	1,044,699	1,632,062	5,525	616,648	1,222,494

Table 7.5: Number of messages and number of bytes received/transmitted by the mobile during a high-resolution execution according to the size of the underlying infrastructure and the nodes hosting the data directory (the mobile device, the worker nodes or shared across the whole infrastructure.

7.5.2 Overall Performance Evaluation

The goal of the second test is to evaluate the impact of the Cloud Platform on the overall system performance. For that purpose, the test measures the execution time and the energy consumption of running both, the low and high resolution, using the available resources with different configurations.

7.5.2.1 Data Directory centralized on the mobile device

Running the low-resolution test case as a regular Android application takes 71 s and has an energy consumption of 135.52 J when the screen is on, and, when it is off, 1,631 s and 251.72

J. Given that the former is both, better performing and less consuming, it is the baseline for comparisons.

Figure 7.6 contains two charts that illustrate the relation between the number of surrogate nodes and the application timespan (left) and energy consumption (right) when the data directory is placed wholly on the mobile device. The two isolated points represent the obtained values for the mobile submitting tasks to the laptop, and the continuous lines illustrate the evolution of the runtime while offloading to one, two, four and eight cloud instances (4 to 32 cores). The cross and the dotted line show the ideal values that the runtime is expected to obtain in each platform according to the execution times and energy consumptions displayed in Table 5.1 with a perfect load balancing and without exchanging messages nor data across the nodes of the platform.

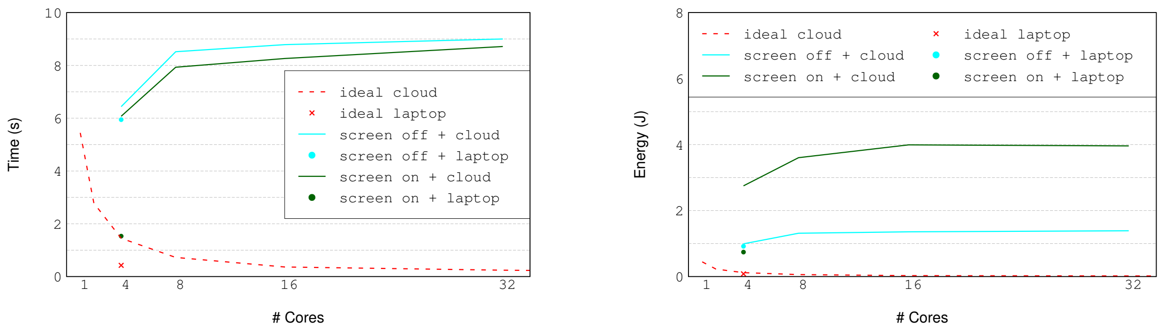


Figure 7.6: Execution time (left) and energy consumption (right) obtained for a low-resolution execution with a centralized data directory.

The best performing testbed for the low-resolution scenario is using the laptop as a surrogate. If the screen is kept on during the execution, the application achieves a speed up 46 times faster than the isolated phone case (1,532 ms) and reduces the energy consumption to a 0.5% of the original (0.74 J). Turning the screen off slows down the execution of the main code of the application – hence, detection of new tasks is also slower – and the runtime processing of the task prior its submission. With the screen off, the application takes 5,945 ms to execute (11.94x) and consumes 0.92 J (0.68%).

Any Cloud scenario behaves better than using only the phone, but the execution time does not shrink when the number of surrogate nodes increases. The high latency on the network slows down the offloading mechanism performance, and the exchange of messages to run a task takes longer than its computation. Indeed, the more nodes the application uses, the longer it takes to execute. When the display is on, the timespan grows from 6,074 ms using one single node to 9,000 ms when the eight VMs are available. The cause of such growth lies on the delay of the propagation of data creation notifications among workers. When a data value is generated, all the tasks within the creating node can already access it while other surrogates need the data directory to notify them the existence and sources of that piece of data. Therefore, the best performing case is the one with a single surrogate since the high network latency only affects the task submission messages and the data sharing protocol messages for the initial data transfers. In those cases with a larger number of nodes, data is less likely to be on the node consuming the value. Since the data directory is only on the master, the producer has to contact the mobile to publish the existence of the value, and the latter forwards the creation notification to the consuming worker. The more workers compose the infrastructure, the more messages the nodes exchange; thus, the latency of the network has a bigger impact.

When using Cloud resources, the impact of turning the display off on the execution time is not as significant as for the laptop case. Since the network latency is high, the overhead caused by it partially overlaps with the slower creation of the tasks; the execution time grows around a 5%. Given that most of the computation runs remotely, the energy consumed by the display is a significant part of the application footprint. By switching off the display, the application reduces its consumption (around 1.3 J) to a third of its consumptions when offloading tasks onto the Cloud with the screen on (around 3.9 J).

Solving the high-resolution problem takes 99,641 seconds (more than 27 hours) on the phone with the screen on, and the phone needs to keep plugged into an energy source. It is an example of the large set of applications whose executions are not viable in current mobile devices; however, offloading computational tasks to remote resources provides the mobile device with the additional computing power necessary to enable the execution of such applications by reducing their execution time and energy consumption.

Figure 7.7 depicts the execution time (left) and the energy consumption measured on a high-resolution execution of the application under the same conditions. Since the execution time of the *simulate* CE and the network latencies are lower when the runtime offloads the computation onto a laptop than when it uses cloud resources, the first behaves much better when the runtime only has four cores available. When offloading tasks onto the laptop, the application lasts 1,368 seconds to solve the problem, achieving a 72.83x speedup compared to running it on the phone. This severe reduction on the timespan has a significant impact on the energy consumption of the application that enables its execution on a mobile device: 621.63 J when the display is on at 0% brightness. Switching the screen off has a little impact on the execution time – 1,401 seconds, 2.4% overhead – and the energy consumption falls to 216 J (34.75%).

On the cloud scenario, when using only four cores, the execution time is significantly higher; and, therefore, the energy consumption too. In the respective best cases, the application lasts 2,318 s (42.99x), and the consumption is 363 J. However, the strong point of the cloud is the amount of resource available for the runtime to offload tasks. When the resource pool has up to 32 cores and the display is on, the application execution time is reduced to 320 seconds, and it consumes 146 J. This is 310 times faster than the isolated phone scenario and 4.26 times faster than offloading tasks to a laptop. Switching off the screen allows the runtime to obtain a lower energy consumption 54.61 J.

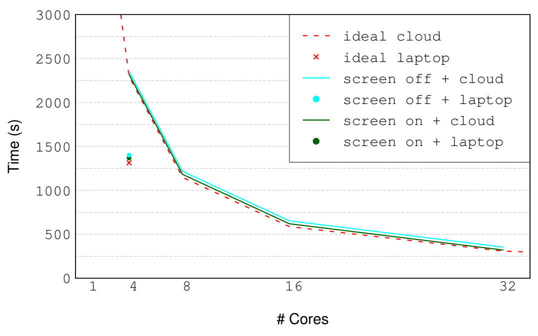


Figure 7.7: Execution time (left) and energy consumption (right) obtained for a high-resolution execution with a centralized data directory.

7.5.2.2 Data directory distribution

Figure 7.8 compares the execution time (left) and energy consumption (right) of a low-resolution execution when the application runs on top of the same infrastructure than in the previous test but changing the nodes containing the data directory. This test only considers the execution of the application when the display of the mobile device is on at 0% brightness and the strength of the Wi-Fi signal is 100%. In the *laptop-mobile* and *cloud-mobile* cases, the data directory is wholly on the mobile device and the runtime offloads tasks onto the laptop and 1-to-8 Cloud VMs, respectively. Their values correspond to the ones depicted in Figure 7.6. The *laptop-1w* points illustrate the measurements obtained when the runtime offloads tasks to the laptop and the latter hosts the whole data directory. The *cloud-1w* lines show the evolution of the measurements when the runtime offloads tasks onto one to eight VMs when only one of the workers contains the data directory. *Cloud-2w* depicts the same evolution when the data directory scatters among two workers; therefore the runtime at least has two nodes – eight cores – at its disposal; *cloud-4w* and *cloud-8w*, respectively distribute the data directory among four and eight workers.

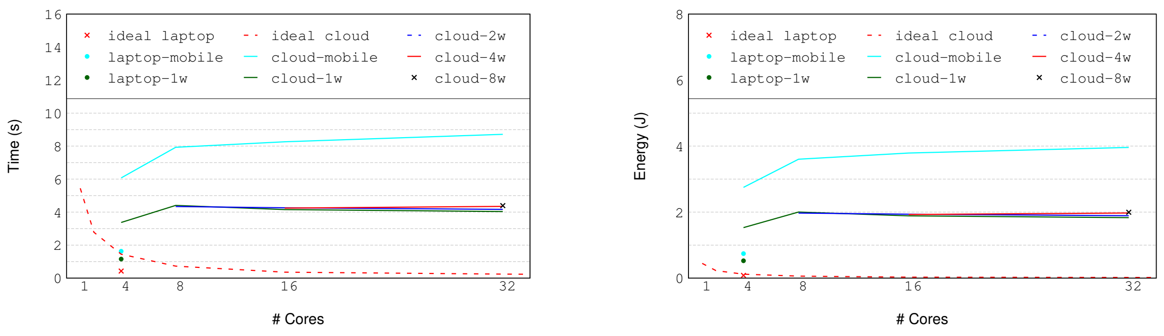


Figure 7.8: Execution time (left) and energy consumption (right) obtained for a low-resolution execution with a data directory distributed among all the nodes.

Both charts demonstrate the performance benefits of moving the directory away from the mobile device. With one single worker (four cores), storing the directory on the surrogates side speeds up the obtention by the workers of the input values created on the phone; thus, tasks start earlier. When using the laptop as the surrogate platform, the execution time shrinks from 1.632 s to 1.156 s. The latency to the cloud nodes is much higher than in a local area network; therefore, the impact when using geographically distributed resources is more significative. For the one cloud worker node (4 cores) scenario, the execution time is reduced up to a 55% (from 6.074 s to 3.372 s). For the 32 cores case, it needs 8.717 s to run when the directory is on the phone and only 4.035 s when it is located in a single worker node (46%). Given the small reduction in the amount of data transferred in and out of the phone, the energy consumption due to the network becomes negligible compared to the energy spent on the display and the processor.

On the other hand, the test also studies the impact of the directory distribution among the workers. All the cases where the worker nodes host the data directory have a similar behavior with slight differences in the execution time. However, the best-performing one when the ring grows is using a single node to host the whole table. The size of the application and the number of exchanged messages are small enough not to saturate the node; conversely, enlarging the data directory ring, increases the number of hops of some queries.

Given the long execution time of the simulations of the high-resolution case, the impact of distributing the data directory on the execution time and energy consumption is negligible since the time required to exchange the commands and transfer the data is several magnitudes smaller.

7.6 Summary

Chapters 4, 5 and 6 describe a solution that allows a programmer to easily write an application that runs automatically in parallel on the computational resources embedded on the mobile device, i.e., the CPU cores and other accelerators such as the GPU. This chapter introduces a new computing platform that enables the offloading of task computations onto remote resources: the Cloud Platform. Through this platform, applications can benefit from the infinity of computing resources accessible through the network and speed up their execution reducing their timespan from days to few seconds. Besides, offloading the computation releases the mobile device from the energy actually consumed by the processing elements; thus, applications that would drain the whole battery of the mobile can run now spending very little amounts of energy. Figure 7.9 updates the architecture diagram in Figure 6.10 to depict the new platform and a single, symbolic remote worker.

Leaving aside its benefits, using remote resources draws attention to the usage of the network. For a remote node to compute a task, its corresponding input values must be on such node; therefore, the runtime must implement a mechanism to share data. An approach where the mobile device transfers all the values to the node when it submits the task and collects the results at the end would unnecessarily increase the costs – temporary, monetary and energetic – of the

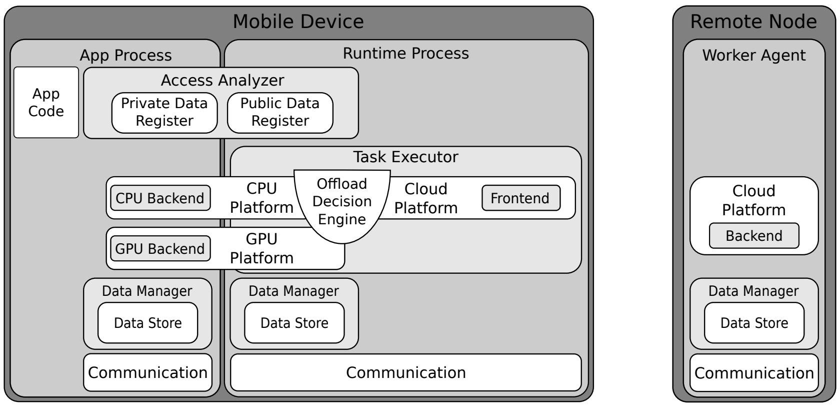


Figure 7.9: Diagram of the runtime architecture with a CPU platform, an OpenCL platform and a Cloud platform.

execution. For that purpose, the runtime implements a mechanism that allows any node of the infrastructure to fetch the necessary values directly from another node without passing through the mobile. This mechanism consists of a data directory that stores the relation between a datum id and all the nodes of the infrastructure that contain it. Thus, when a node produces a new value or obtains a copy of it, it registers into the directory the presence of the value. The consumers of such datum query the data directory which nodes of the infrastructure know the value for the datum and directly contacts one of the sources to fetch it. The current implementation of the data directory is a distributed hash table. The platform organizes the nodes of the infrastructure, including the mobile, as a two-level hierarchical peer-to-peer network. Each of the top-level nodes is directly responsible for a range of hashcodes, and it monitors the sources of all the fata values from which the hash of its identifier is within this range. The lower-level nodes simply contact the corresponding top-level node to get the sources of a datum when needed.

One of the problems associated with devices with high mobility is the stability of the network; devices are likely to go through network handovers or long periods of isolation. It is important that both ends, the mobile device and the remote workers, are autonomous and go on with the execution without contacting the other part. In the case of an eventual re-establishment of the connection, both ends synchronize their progress; workers autonomy ensures, no matter how long the disruption lasts, that the loss of performance is as little as possible since workers nodes execute all the tasks as expected and the mobile device would get the result upon the reconnection. Distributing the data directory only across worker nodes is enough to provide workers with autonomy since workers would keep the ability to monitor the data dependencies of the tasks and fetch data values from other workers nodes.

On the other side, an autonomous mobile device ensures that application users obtain a result even if the mobile never connects back to the network. Assuming that the connection will eventually be re-established, the mobile runs, as usual, all those tasks assigned to the embedded resources and whose input data is already on the phone. If the disruption lasts for long, it is possible that all the tasks assigned to local resources finish except for those depending on values generated by task offloaded to the remote resources. If the runtime detects that there are idle resources and tasks blocked because of missing data dependencies, it looks for the offloaded task generating the missing value and runs it locally. In turn, this task can depend on other offloaded tasks; the runtime tries to run the predecessor also on the mobile what starts a backtracking process that could bring the execution to the very first task of the application. To prevent this mechanism from re-executing the whole application, the master establishes some checkpoints before the network disconnects by fetching some significant values upon their remote generation; thus, the backtracking mechanism never needs to go beyond the task generating these values. Finally, to run all the tasks of the application, the runtime launches the execution on the embedded resources of all the offloaded tasks still pending for execution; eventually, all the tasks run, and the application generates the result by the application user, although slower than expected.

CHAPTER 8

SECURE COMMUNICATIONS

A paradigmatic example of Mobile Cloud Computing is an organization offering its IT resources to its members so that they accelerate applications running on their mobile devices. To avoid unauthorized users and to reliably account for resource usage, the organization needs users to authenticate with the credentials given by the organization through its Identity Provider (IdP) [82]. In addition to its affiliates, this organization could also offer the IT resources to members of other organizations – with their own IdPs. So the service offered by the first organization recognizes the members of others, organizations need to define a set of common policies and protocols to manage and trust the identity of the users and establish a Federated Identity Management (FIM). In this case, using Single Sign-On (SSO) techniques would benefit organizations and users. Resource providers would be released from user account management (managing password strength, keeping account details up to date, resetting passwords,...) since they delegate it to the home organization of the user. It is also more comfortable for users since they no longer have to remember a large number of passwords or reuse a single password for multiple services. Instead, they have a single password – or some other means of authenticating, like a smart card – with which they authenticate to their IdP. Because they typically use this method more frequently, they are also less likely to forget the password. Besides, they do not expose their passwords to remote systems, only to their (trusted) IdP.

On the user end, the data contained on the phone or collected through applications running on it (pictures, videos, lists of contacts, geolocation, movement, etc.) can be privacy-sensitive and should not be accessed without permission of the user. Given the sensitivity of the data, data breaches are a major concern in MCC environments [57]. Clouds often run on resources owned by private companies that offer them as a utility [26]. Other cloud users (multi-tenancy) are a potential threat; however, virtualization should isolate the resources assigned to each user and protect them from the attacks from its neighbors. Malicious insiders are another potential hazard since providers could snoop on the hardware resources and obtain information stored or processed on it. The strong laws enforcing data protection and the strict personnel background checks of commercial providers make malicious insiders not likely to happen, and they can be blissfully ignored. To protect themselves from these attacks, users can apply data-at-rest encryption techniques such as ciphering/deciphering data when interacting with the file system or operate directly on encrypted data using fully homomorphic encryption (FHE) [87].

The biggest concern regarding data breaches is attacks from unaffiliated people to in-transit data. Usually, the network interconnecting the mobile device with the remote nodes is untrustworthy. For instance, when the mobile device connects to the remote workers through Wi-Fi and Internet, attackers could eavesdrop on the interconnecting channel (e.g., the Wi-Fi network). Communication has to provide message secrecy for not exposing user or application information. Another possible attack consists in a disguised attacker impersonating either a remote node, to intercept data transmitted from the mobile, or the mobile device, to fetch data stored in a remote worker. Hence, communications require mutual authentication and message integrity to ensure that both ends are part of the trusted infrastructure and that the content of the messages is the original one and not a malicious command introduced by the attacker.

This chapter describes a solution to secure communications among the components of the runtime with authenticated encryption to protect the integrity, confidentiality and authenticity of the messages in the system. Most organizations already have a deployed authentication infrastructure; adopting a generic approach that avoids a security vendor lock-in is an important design consideration. For this reason, the solution leverages on the Generic Security Services API (GSSAPI) [64], an interface implemented by most of the security services vendors. Thus, applications following the COMPSs model can replace the security framework without modifying their code. For validating the viability of the solution, the runtime builds on Kerberos [1] as the security provider.

8.1 Backgroung: GSSAPI

A secure system consists of a set of interacting participants which authenticate themselves using the credentials issued by an authority. These participants are end users (persons), uniquely identified by their real name, e-mail address or a username; and compute nodes identified either by hostname, IP address or sometimes as individual services or endpoints. Authorities usually are centralized: a single entity manages the credentials for all the participants within the domain (e.g., the members of an organization accessing to its services).

When end users access services offered by different organizations, they often require services to interact with a service provided by a second organization on their behalf. For instance, when the end user runs some program on a computing service provided by *Organization A* that needs to fetch some data from a storage service on *Organization B* that requires the credential of the end user. This kind of situations was very common in Grid systems. The Globus Toolkit [11], a software solution for building Grids, expanded on the original authentication model by introducing delegation [95]. A definition for delegation can be a temporary reassignment of rights; however, in many practical applications, it means forwarding a credential to the server. Through those credentials, servers “impersonate” the user or, at least, perform actions on their behalf. Globus attached X.509 certificates [54] to the communication protocol, whether it was secured HTTP [86], GSS, SOAP, etc.

Federated Identity Management is another approach to allow users to access services from multiple organizations. Multiple domains can share identities and their associated attributes, and organizations can define a common set of policies and protocols to manage the access to their services in a federated way. Thus, members of an organization – *Organization A* – can access services provided by a second one – *Organization B* – using directly the credential obtained from their home organization – *Organization A*.

Several different security technologies implement the described architecture. The Generic Security Services Application Programming Interface (GSSAPI) is an abstraction of the security negotiation that happens when a participant –GSS initiator – authenticates to another one – GSS responder – and both exchange messages securely. The applications at either end call the API and are instructed by the implementation whether authentication is successful, unsuccessful, or needs more calls – some protocols require several back-and-forth communications. A wide range of mechanisms can implement the underlying authentication: username/password, Kerberos, Moonshot, X.509 certificates; clients can be anonymous or named, and they can pass authorization attributes [102]. Initially, the preferences of the initiator determined the authentication protocol; however, GSSAPI was extended to support a common protocol for negotiation [103]. This negotiation protocol builds on mutually accepted trust anchors, and that might not be sufficient; a further proposed extension ("extended negotiation") supports more sophisticated negotiation protocols.

For message-level security, GSSAPI supports not only origin authentication – i.e., sender signs the message – but also message encryption, integrity, replay detection, or detection of receipt out of sequence. Blocks of data, also known as tokens, will have the selected security features applied to it before submission (wrapping), and checked upon reception (unwrapping).

Compared to just implementing one security protocol, using GSSAPI correctly is more complicated for the application programmer; it is also harder to debug because GSSAPI is implemented using ASN.1 as a layer around the actual protocol. However, the generic service, if coded correctly, can then support a range of mechanisms – including future ones – and delegates on GSSAPI many security tasks that may not be obvious to the programmer, such as preventing replay attacks, checking the server identity correctly (preventing man-in-the-middle attacks), and negotiating shared protocols for message security, etc.

8.2 GSSAPI Integration

Extending the runtime to secure its communications brings two challenges to the current architecture. First, the roles comprised in a secure architecture and the roles assumed by the parts of each GSSAPI interaction need to map to the components of the infrastructure running the application. And second, the runtime has to secure all the communications among the nodes of the underlying platform; therefore, the architecture of the system needs to be adapted to include GSSAPI and the security framework.

A secure system consists of several interacting participants authenticated by a trusted third party that acts as an authority. For the computing platform proposed in this dissertation, participants correspond to all the nodes, either the master or the worker nodes. The actual infrastructure acting as the authority and the protocols to interact with it are specific to the choosen security framework. The runtime leverages on GSSAPI to provide an interoperable solution that works with several security frameworks and avoids a security vendor lock-in.

GSSAPI abstracts away from the runtime the authentication infrastructure and the protocols to interact with it and establishes a client-server pattern where the client – GSS initiator – contacts a service – GSS responder – to start a secure connection and exchange messages. This model maps easily with the master-worker approach of the runtime where the application running on the mobile device, the master node, offloads task executions onto remote nodes running a service. Therefore, the master, assuming the role of GSS initiator, authenticates on behalf of the application user to worker services playing the GSS responder role.

However, the model clashes against the peer-to-peer organization used for sharing data. Whenever a node of the infrastructure needs some value located on a remote node, it opens a new TCP connection to a server deployed on the remote node, regardless of whether it is a worker or the master. In TCP terms, any node can act as a TCP client, so every node must listen for incoming connections, including the master. To avoid that anyone fetches a value from a node of the infrastructure, both ends authenticate to each other following the protocol established by the specific security framework. Therefore, the TCP and GSS roles may mismatch. When the mobile opens a connection to a worker, the TCP client acts as the GSS initiator, and the TCP server takes responder role of the GSS. Conversely, when it is the worker the one contacting the mobile, the TCP client is the GSS responder, and the TCP server is the GSS initiator. Regardless which end starts the TCP communication, it is always the node playing the role of GSS initiator the one triggering the GSS negotiation upon the TCP connection establishment. Since establishing a secure context when both ends act as GSS responder is not possible, in worker-worker communications the TCP client assumes the role of GSS initiator.

Traditionally, worker processes are co-located on resources interconnected by trustworthy networks, such as clusters or private clouds. In this case, using secure connections gives no added value but adds unnecessary overhead; for this purpose, the communication component allows to set up a whitelist to indicate which nodes do not require establishing a security context to transfer the data.

The second problem to tackle is the integration of GSSAPI and the security framework within the component architecture of the runtime system. GSSAPI only indicates the format of the messages exchanged among both ends of a connection but does not define their content, which depends on the information to transfer and the selected security framework, nor decides the transport-layer protocols used for delivering such messages through the network. The security framework processes all the applications messages – and is likely to modify their content – before sending them using the same network protocol that the application would use with non-secure communications. Similarly, upon the arrival of new data from the network, the security framework has to process the received bytes to extract the actual application message before forwarding it to the application. The runtime architecture described in the previous chapters already encapsulates all the network interactions within a *Communication* component – concretely, introduced in Section 7.1. Whenever a node of the infrastructure, either master or worker, wants to send a message to another node, it asks the *Communication* component to open a new connection to the target node and delegates on it the transmission of the message. Therefore, the *Communication* component is the only part of the runtime involved in the extension to secure communications among the runtime components; the security framework remains as an internal part of the *Communication* component. Figure 8.1 depicts an overview of the architecture of a deployment of the runtime with one master node (leftmost part of the figure) and two worker nodes (right). As depicted by the red dashed arrow, the security framework processes the messages exchanged between the runtime components before their transmission and upon their reception on the remote node. When another component requests the opening of a connection, the *Communication* component establishes a TCP connection as usual and triggers the GSSAPI negotiation procedure to establish a security context with authenticated ends, an accorded protocol and a keypair to encrypt and sign the messages. Everytime that a component submits a message using a secured connection, the *Communication* component handles the ciphering and the deciphering transparently to the requestor.

As explained in Section 7.1, the *Communication* component leverages on the Non-Blocking IO library provided by Java, which encapsulates point-to-point, ordered network connections in stream-oriented channels. This approach guarantees the reception of all the sent bytes in the same order, but it does not necessarily maintain the groupings; the sender could submit a 128-bytes packet, and the receiver could get two packets of 96 and 32 bytes, respectively. For abstracting this away from the application, the *Communication* component adds 4 bytes to the message header indicating the message size. Upon the reception of the whole message, the *Communication* component delivers the messages to the application/runtime level.

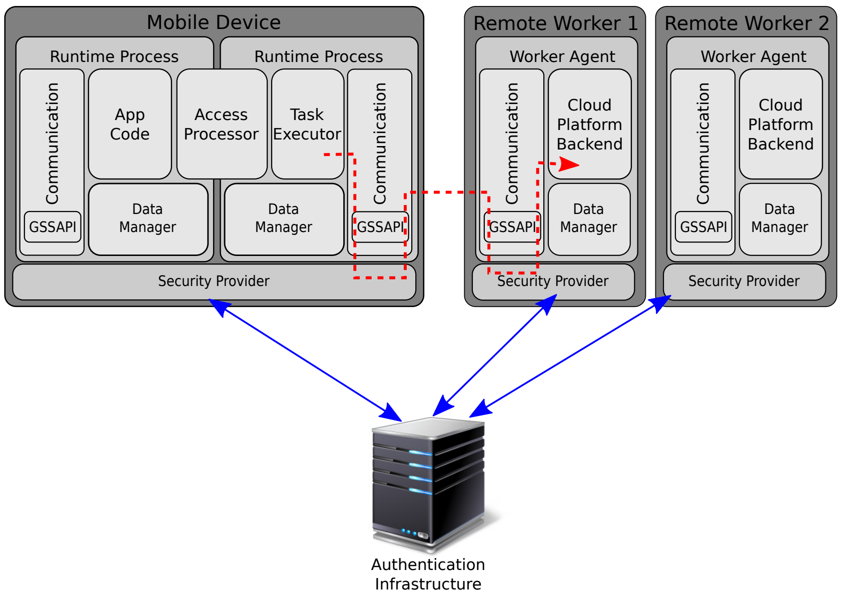


Figure 8.1: Runtime architecture diagram with secured communications. The red dashed arrow shows the flow followed by a task submission command send by the mobile device to one of the workers.

To achieve complete secrecy, both, the message content and its header, need to be encrypted. Hence, the receiver of the message is totally unaware of the length of the message until it decrypts the header. To decrypt ciphertext, some algorithms require the whole text to start processing it; block ciphers require only a complete block; and, on some stream ciphers, the basic unit for deciphering is a single byte. Given that the *Communication* component cannot rely on any specific encryption mechanism, it assumes the most restrictive approach: the whole ciphertext is necessary to decrypt the message. To decrypt the header, GSSAPI needs the entire ciphertext and NIO requires the decrypted length of the message to determine the arrival of the whole ciphertext. Therefore, the *Communication* component reaches a deadlock since both, GSSAPI and NIO, require the other to take the first step to continue the processing. The solution for overcoming this situation consists on fixing the length of the ciphertext; the *Communication* component pads the plaintext of the messages so that their ciphertexts reach a specific size.

Non-secure communications are not to decrypt the tokens received; therefore, the *Communication* component can figure out the length of the message without receiving a whole fixed-size token. To avoid transferring unnecessary bytes while maintaining the same logic for secure and non-secure communications, the *Communication* component adds four bytes as a header of the token indicating its length.

Picking a single size for all messages can lead to issues since the messages transferred by the *Communication* component can have from few bytes, like the commands of the runtime, to several megabytes, if the message ships the content of a file. A token size large enough to fit any message would incur several gigabytes of data dedicated exclusively to the padding of the commands of the runtime; that would kill the performance. A token size too small to fit the content of a file used by the application would impede the execution of the application. To work around the problem, the *Communication* component splits the plaintext of the message to produce several cyphertexts that fit into fixed-size tokens. The first token contains the ciphertext of the header of the message and the first part of the message content. When the receiver gets this first token entirely, it decrypts the header and obtains the length of the message. If the whole message fits in a single token, the *Communication* component forwards the message to the runtime; otherwise, it waits for the following tokens and incorporates them into the message upon their decryption. The chosen token size has a significant impact on the total amount of bytes transferred through the network, and therefore on the time and cost of data transfers. Larger token sizes may add more padding and take more time to transfer; smaller token sizes have more overhead in being processed individually and may be more likely to split important structures.

The security framework used for the validation of the described architecture is Kerberos. The user of the mobile phone obtains the Ticket Granting Ticket – an encrypted identification file valid for a limited period – from the Kerberos key distribution center before running the application. Worker nodes authenticate themselves through a Kerberos keytab, and they are authorized to accept connections either from the master or other worker services. For porting the Kerberos library to Android, it was necessary to cross-compile the official release of the MIT Kerberos for Android to create a native library (libkerberos.so). The runtime dynamically loads this library upon the completion of the TCP handshake corresponding to the first connection that requires securing the messages. Despite being Java the native language to develop Android applications, not all the classes and libraries typical from Java are available on Android. GSSAPI is one example of these libraries; although it is part of the Java SE, there is no GSSAPI implementation within the Android software stack. Besides the security framework, the MIT release for Kerberos also contains a JNI wrapper of the library in fulfillment of the RFC5653 [96], which defines the Java binding for GSSAPI.

8.3 Performance Evaluation

Securing the communications adds some overhead to the application execution which has various causes. On the one hand, the fixed-size token mechanism to transfer messages with an encrypted header through NIO incurs additional costs for each transfer due to the extra bytes attached to pad the messages. On the other hand, GSSAPI processes every token to encrypt, sign, verify and decrypt its content adding a computational overhead. The following tests aim to compare the transmission of data through plain and secured sockets and evaluate the impact of securing the communications on the performance of the application. The tests run on the same infrastructure that hosted the evaluation of the Cloud Platform (see Section 7.5).

8.3.1 Security Overheads

The goal of this first test is to evaluate the overheads introduced by securing the communications on the information transferred through one socket. Thus, it analyzes first the establishment of the secure connection and then it studies how the token size affects the submission of messages. For that purpose, the mobile device opens a new TCP connection to one of the worker nodes deployed on the cloud and, after the secure context negotiation, it uses the connection to send a single message. The test repeats this procedure for different token and message sizes.

After the 3-way handshake to establish the TCP connection, both ends of the connection exchange messages (plain text) to establish the security context (Negotiation). In the timeline depicted in Figure 8.2, the TCP client also acts as the GSS initiator. Upon the connection establishment, it instantiates a new GSSAPI context (average 16 ms) and constructs a message of 612 bytes to authenticate itself and describe the available mechanisms to establish the secure context (average 18 ms). If the Communication component is set up to use very short tokens (256 or 512 bytes), it splits the message into several tokens increasing the total amount of sent bytes (620 bytes in three 256-bytes tokens; and 616 bytes in two 512-bytes tokens). The GSSAPI responder receives the message, verifies the identity of the client and picks the mechanisms and algorithms to establish a secure context (55 ms). After that, it creates a response message of 166 bytes, 142 dedicated to the identification and the agreed terms of the security context. Upon the reception of this response, the GSS initiator verifies the identity of the service (2 ms) and end ups the negotiation. In overall, this process takes around 355 ms (depends on the network conditions). The client emits 632-640 bytes, and the server, 166 bytes.

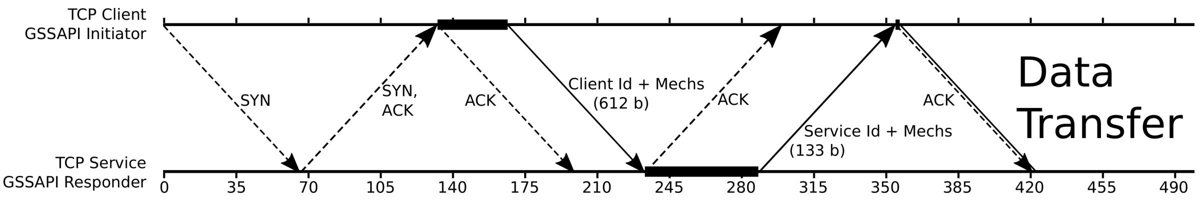


Figure 8.2: Timeline of the TCP Connection Establishment and GSSAPI Negotiation.

Once the negotiation ends, the actual data transfer begins, and secure tokens – “wrapped” in the GSSAPI terminology – are transferred through the network. It is in this second stage where the token size may have a stronger impact depending on the performance. Tables 8.1 and 8.2 compare, respectively, the actual transfer size and the timespan to submit internal commands of the runtime (typically, 250 bytes), small objects (2500 bytes) and files of different sizes (10,000, 100,000, 1,000,000 and 10,000,000 bytes) when using different token sizes for non-secured and secured transmissions.

The *Communication* component adds a header of 16 bytes to every message that indicates its type – raw data or command–, the preferred destination – file or memory – and the length of its content. The larger the message is, the less significant is the overhead of the header (6.4% for

Comm. Type	Token Size	# Bytes					
		250	**2,500**	**10,000**	**100,000**	**1,000,000**	**10,000,000**
non-secure	**256**	274	2,556	10,176	101,604	1,015,892	10,158,748
	512	270	2,536	10,096	100,804	1,007,892	10,078,760
	1,024	270	2,528	10,056	100,412	1,003,940	10,039,232
	2,048	270	2,524	10,036	100,212	1,001,976	10,019,588
	4,096	270	2,520	10,028	100,116	1,000,996	10,009,792
	8,192	270	2,520	10,024	100,068	1,000,508	10,004,904
	16,384	270	2,520	10,020	100,044	1,000,264	10,002,460
secure	**256**	512	3,584	13,568	133,376	1,333,504	13,333,504
	512	512	3,072	11,776	114,688	1,143,296	11,428,864
	1,024	1,024	3,072	11,264	107,520	1,067,008	10,667,008
	2,048	2,048	4,096	12,288	104,448	1,034,240	10,323,968
	4,096	4,096	4,096	12,288	102,400	1,019,904	10,162,176
	8,192	8,192	8,192	16,384	106,496	1,025,808	10,084,352
	16,384	16,384	16,384	16,384	114,688	1,015,808	10,043,392

Table 8.1: Actual size of transferring 250, 2,500, 10,000, 100,000, 1,000,000 and 10,000,000 bytes according to the token size in bytes (256, 512, 1,024, 2,048, 4,096, 8,192, 16,384).

Comm. Type	Token Size	Message size (bytes)					
		250	**2,500**	**10,000**	**100,000**	**1,000,000**	**10,000,000**
non-secure	**256**	74	75	107	193	745	6,734
	512	74	74	85	127	508	3,806
	1,024	73	73	78	127	367	2,146
	2,048	73	73	72	123	327	1,918
	4,096	73	73	75	123	323	1,512
	8,192	73	73	75	121	307	1,446
	16,384	73	73	74	107	278	1,384
secure	**256**	84	90	128	314	1,808	16,265
	512	84	88	96	251	1,068	9,384
	1,024	84	86	93	186	780	6,530
	2,048	83	83	92	179	604	5,001
	4,096	86	86	95	158	574	4,668
	8,192	87	94	99	152	540	4,376
	16,384	96	103	107	162	500	4,074

Table 8.2: Timespan (ms) to transfer 250, 2,500, 10,000, 100,000, 1,000,000 and 10,000,000 bytes according to the token size in bytes (256, 512, 1,024, 2,048, 4,096, 8,192, 16,384).

a 250-bytes message, 0.0000016% for the largest case). In addition to the message header, the component adds a token header of four bytes. The bigger the token size is, the fewer tokens the connection needs and, hence, the fewer bytes the *Communication* component adds.

GSSAPI enables, if wanted, multiple mechanisms to secure the communications using different algorithms. With the used configuration, securing communications implies encrypting and signing the content of each message; thus, the actual payload of each token changes according to the encryption and signing algorithms – in the tests, these algorithms add 60 bytes. This overhead has a significant weight on the message/payload ratio for really small tokens – 25% for 256-bytes tokens – and increases the number of tokens required to send a message. For a 10,000,000-bytes transfer, the overhead reaches up to a 33.35% when using 256-bytes tokens. Conversely, using large tokens reduces the number of tokens to send large messages and, therefore, the additional bytes – 0.43% when transmitting 10,000,000 bytes in 16,384-bytes tokens. On the other hand, the *Communication* component has to pad messages to match the token size. When using very large token sizes to send small messages, the pad has a significant weight on the length of the transmitted message – 6,452.60% increase for a 250-bytes message in a 16,384-bytes token.

Regarding the time to transfer the information, Table 8.2 shows that the latency of the network – 67 ms – is the most important parameter for very short messages since most of the tokens sizes take around 85 ms to transfer the data. It practically shows no difference for tokens shorter than 2,048 bytes; indeed, the lower the number of tokens used is, the lower the time required to transfer the data. Larger token sizes take longer to submit a short message not only because of the time to transfer more data but also because of the time to encrypt and decrypt larger ciphertexts. For this reason, the case using 16KBs-tokens takes 96 ms to transfer a 250-bytes plain message instead of the 83 ms required for the case of using 2KB-tokens.

When transferring larger pieces of data, the delay of the network becomes negligible compared to the time to process the messages – 0.4% of the total transfer time when transferring 10MB using 16KB-tokens. While the non-secure communication using 16KB-tokens transfers the message in 1,384 ms, the secured version needs 4,074 ms, three times more. The additional bytes added by the security algorithms do not explain this difference since they only represent a 0.43% of the total transfer size. Therefore, the time to encrypt, decrypt, sign and verify the content of the message is the most important factor for large data transfers. Although the size of the data to process is the most important factor contributing to the computational overhead, it is important to notice that the number of tokens is also a significant part of it. For instance transferring a 10MB message using 16KB-tokens requires 10,043,392 bytes split among 613 tokens – and 4,074 ms while transferring the same message using 8KB-tokens needs 10,084,352 bytes – 40,960 bytes, 0.4% – divided into 1,231 tokens and 4,376 – 302 ms, 7.4%.

8.3.2 Security Impact on Applications

In the second test, the goal is to measure the impact of using secure communications on applications. For that purpose, the test runs the same application used for evaluating the Cloud Platform, HeatSweeper, and considers the same two scenarios: a low-resolution run, representing applications with a low number of short-lasting tasks, and a high-resolution execution, representing large computations. The test runs the application on the OnePlus One phone equipped with a Snapdragon 801 processor managed by the default governor provided with its operating system, the Cyanogen OS 13.1.2 implementation of Android 6.0.1. The mobile device offloads the tasks onto up to 8 VMs deployed the OpenNebula Cloud described in Section 7.5. The runtime is configured to distribute the data directory to support sharing data only among the worker nodes; and the token size, set to 2,048 bytes since the reports generated by *simulate* tasks are smaller than 10,000 bytes.

In both scenarios, executions using secure and non-secure communication behave alike as shown by the charts in Figure 8.3. However, security adds a delay of 15 seconds – 500-200% overhead depending on the number of cores – for the low-resolution scenario and 50 seconds – 2-10% overhead depending on the number of cores – for the high-resolution. The first cause of this delay is the application-level protocol followed by the runtime to execute one task. On a first stage, the master requests to the network the execution of one task. Then a worker subscribes to the data directory for being notified when other nodes generate the input data values – these commands do not require security since the data directory is deployed across the worker nodes co-located in the same cluster. Upon the creation of all the input data, the task becomes dependency-free, and the worker requests the transmission of the data value to one of the sources – if the source is the master, it secures the connection. For the *simulate* tasks, this protocol enforces the submission of three secured messages: the submission of the task, the request of the datum containing the simulation parameters and the transfer of the value of such datum as an object – request and value transfer happen on the same connection. This message exchange explains 770 ms of this delay according to the results presented before.

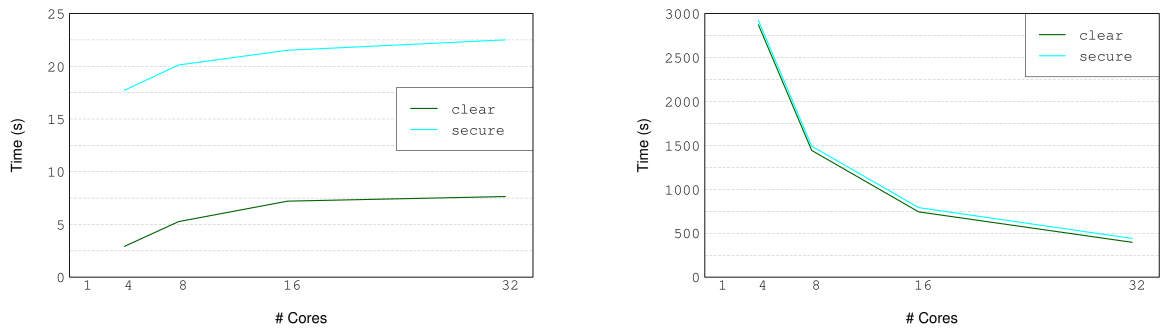


Figure 8.3: Comparison of the execution times for the low-resolution (left) and high-resolution (right) obtained when using non-secure and secure communications.

The second cause for the delay is the number of threads employed by the *Communication* component. All nodes have one single thread dedicated to the reception and submission of TCP packets, a second thread for the application-level management – i.e., token handling and responding to the received commands– and several additional threads to fulfill with its specific duties of its role in the infrastructure. A single thread for handling all the communications allows a concurrent establishment of several connections and message transmission; however, it serializes the computations related to the application-level message, the GSSAPI negotiations and the wrapping/unwrapping of the received tokens. The low-resolution scenario submits up to 89 tasks in parallel; the high-resolution, 649. This increase in the number of parallel incurs a growth on the delay in command submissions.

8.4 Summary

The data contained on the phone or collected through running applications (pictures, videos, lists of contacts, geolocation, movement, etc.) can be privacy-sensitive; therefore, data breaches are a major concern on MCC systems. Assuming that at-rest data is already secure, this chapter explains the mechanisms implemented to secure in-transit data by providing with secrecy and integrity the messages exchanged through a channel where both ends of the communication have mutually authenticated.

Designing and implementing security solutions is complex; it is very easy to add other vulnerabilities to the system because of implementation details like, for instance, a non-uniform response time. To avoid an inadequate implementation of the protocols with security leaks, the runtime leverages on already existing security solutions. Most of the organizations already have a deployed authentication infrastructure; adopting a generic approach that avoids a security vendor lock-in is an important design consideration. For this reason, the solution uses the Generic Security Services API (GSSAPI), an interface implemented by most of the security services vendors. Thus, applications following the COMPSs model can replace the security framework without modifying the code of the application.

The *Communication* component encases the integration of GSSAPI with the runtime; the rest of the runtime transparently benefits from the advantages of security with no modification of their code. The *Communication* component builds on Java Non-Blocking I/O (NIO) to transfer data through the network using TCP. NIO notifies the arrival of information upon the reception of the corresponding bytes; although it respects the order of the bytes, it does not guarantee the groupings. For the *Communication* component to halt the forwarding of the message to the runtime until the reception of its whole content, it adds the length of the message to its header. To achieve complete message secrecy, the component encrypts the header of the message; thus, it needs to decrypt the received information. Some algorithms require the whole ciphertext to decrypt it, and NIO does not allow to know when the whole message has arrived. To overcome the problem, the *Communication* component fixes a token size – configurable parameter. The padding of the messages to match the token size, the negotiation of the security context – strongly influenced by the network latency – and the processing of the messages to encrypt, sign, verify and decrypt them – a single thread processes the messages of all the parallel connections – add a significant overhead to the execution.

To validate the viability of the solution, the prototype uses the Kerberos framework as the security service provider. To authenticate themselves, application users use Kerberos credentials, while worker nodes use Kerberos keytabs as host credentials. Although GSSAPI provides the runtime system with the ability to authenticate and encrypt communication using federated credentials – if supported by the used implementation, and Kerberos does –, it does not provide a generic mechanism for obtaining the credential. The described extension of the runtime requires the device to have the credential already. It can obtain the credential through another application that stores it on the file system, or the same application can provide the required mechanism. While the former option keeps the application agnostic to the authentication mechanism, the latter gathers all the functionality within a single application despite binding it to a particular authentication mechanism. Either way, the content of this chapter is a first, but important, step towards achieving a secure MCC platform with SSO.

CHAPTER 9

OFFLOADED SCHEDULING SYSTEM

The overhead caused by selecting which resource runs each task is negligible when the number of resources is small. However, when the infrastructure and the number of tasks grow, this overhead may become significant. Besides, assigning each task a node upon its detection may end up causing load imbalance if the estimations are not precise. To avoid performance issues, it is necessary to monitor the pending workload assigned to each resource and modify some already-taken decisions to adapt the remaining execution plan to the reality at that time.

Maintaining all this computation on the mobile device not only employs resources that could execute other tasks but also incurs an energy consumption that unnecessarily drains the battery of the device. The solution to avoid this overhead consists in offloading onto the remote resources not only the computation of the tasks but also the scheduling of their execution. Thus, the mobile device considers all the remote resources as a single platform with a global computing capacity; all it needs is an entry point to that platform where to submit task executions.

Delegating the task scheduling not only benefits the mobile device, but it also improves the control that the owner has over its infrastructure. Although the Cloud has allowed reducing the cost of having access to large computing infrastructures from the user point of view, it has rocketed the ownership expenses. On very large datacenters, the energy consumed by the computing nodes and the cooling systems influences on their cost to the extent that energy fees and the average outside temperature condition their placement [45]. For this reason, several research initiatives, like the european projects OPTIMIS [39] and ASCETIC [35], focus on the reduction of the energy consumption of applications across all the stages of their lifecycle: design, construction, deployment and operation.

This chapter describes an extensible scheduling system that runs on one of the remote nodes of the infrastructure. When the *Offloading Decision Engine* submits the task to the *Cloud*

Platform, the latter forwards the task to the node in charge of the scheduling. Upon its reception, the scheduling assigns a resource to the task, that might change during the time, according to the policies chosen by the infrastructure owner. Currently, the set of policies governing the system consider the application-level knowledge – i.e., pending offloaded workflow, profiles of the implementations on each resource, ... – and aim fot minimizing one of the parameters (timespan, energy consumption and monetary cost) while setting global limits for the other two. For instance, minimizing the energy consumed – Wh – by the application without exceeding a threshold of 3,600 s nor 5 €.

Often, achieving the defined boundaries is not possible with the available resources; in these cases, the scheduling system has to adapt automatically the amount of resources exploiting the horizontal elasticity of the Cloud. When the expected execution exceeds the timespan boundary, the system considers scaling-out; conversely, it considers scaling-in to reduce the energy consumption and cost of the application. The system periodically checks whether doing a change in the pool of resource is worth or not.

9.1 General Aspects of the Scheduling System

From an abstract point of view, a scheduling system is a manager that orchestrates the execution of a collection of actions on a set of resources over the time. In the context of this dissertation, these resources are the hardware and software capabilities of the nodes composing the infrastructure, and the actions correspond to activities such as turning on the node (*Poweron Action*),booting the worker process (*Start Worker Action*), running a task (*Execution Action*), stopping the worker process (*Stop Worker Action*) or even shutting down the node (*Shutdown Action*).

An action can carry out its duty in different ways; each implementation of the action requires some resources to host its execution. The scheduling system has to find one node whose capabilities meet the requirements of any of its implementations and reserve the corresponding resources for as long as the execution lasts. Sometimes, actions are bound to a specific node; for instance, an action to start the worker process on the node. In such cases, the scheduling system does not need to look for a compatible node; it only reserves the required resources.

Not only actions per se constrain the scheduling decisions; interactions between actions also can add new restrictions to the system. For instance, data dependencies among actions corresponding to task executions: an action consuming a data value cannot run until the producer action is complete. Furthermore, if some effect of a preceding action used by the succeeding action is pinned to a specific node, the successor has to run on the same node. This kind of dependencies are static: they only change upon the arrival of a new action, when the system analyzes the dependencies with already existing actions, or upon the completion of a preceding action, what releases the dependency.

An action is ready to run when it has no dependencies with other actions. The only reason for delaying the execution of a ready-to-run action is the lack of available resources because all of them are busy running actions with higher priority – in general, actions that entered the scheduling system earlier. The scheduling system holds the action execution until running actions release enough resources to host the action. For controlling the execution flow, it defines a second type of dependency among actions: resource dependencies. A resource dependency among two actions appears when one of them uses the resources that a second action releases at its completion. By properly adding resource dependencies among the actions assigned to the same node, the scheduling system ensures that the requirements of the dependency-free actions assigned to a node never overpass the capabilities of such node.

Static dependencies affect pairs of actions potentially assigned to different nodes, but resource dependencies are only among actions assigned to the same node. For this reason, the scheduling system monitors the static dependencies at a global level in the *Action Scheduler*, and each node has a *Node Scheduler* to manage the resource dependencies within such node.

Unlike static dependencies, originated by the activity performed by both actions, resource dependencies arise from arbitrary decisions made by the *Node Scheduler*; therefore, they can change whenever the system decides when and where the actions run or when it modifies this decision. Despite the flexibility of resource dependencies, it is extremely important that the *Node Scheduler* avoids creating cycles of actions depending on each other; dependency cycles cause a deadlock on the node because all the involved actions are waiting for the completion of another action to start its execution.

Figure 9.1 depicts an example of four dependency graphs for a set of eight actions. Solid edges depict the static dependencies, which are common in all four examples. The dashed edges represent the resource dependencies among the actions. The graph at the top of the figure corresponds to an example with enough resources to host all the executions at a time. The others illustrate three different solutions that the scheduling system could pick to run the same actions on a set of resources able to host up to two actions at a time.

Besides controlling that the execution does not overload the available resources, the scheduling system optimizes the execution according to a user-driven, multi-objective policy. This policy allows the user to minimize one of the following parameters: execution timespan, economic cost and energy consumption; while limiting the overall value for the other two. To perform this optimization, the scheduling system cannot rely on solutions that require complete knowledge of the applications beforehand since the programming model detects the tasks composing the application as it runs. To provide a reasonably good-quality solution in a short time, the scheduling strategy followed takes action in two steps. First, upon the action arrival, the scheduling system applies an initial scheduling policy that greedily assigns the action a node where to run according to the optimization parameter. Later on, the system can revisit the decision taken by this initial policy. A second process of the system, the *Scheduling Optimizer*, checks the whole execution plan and tries to optimize it by re-ordering the actions within a node – changing the resources

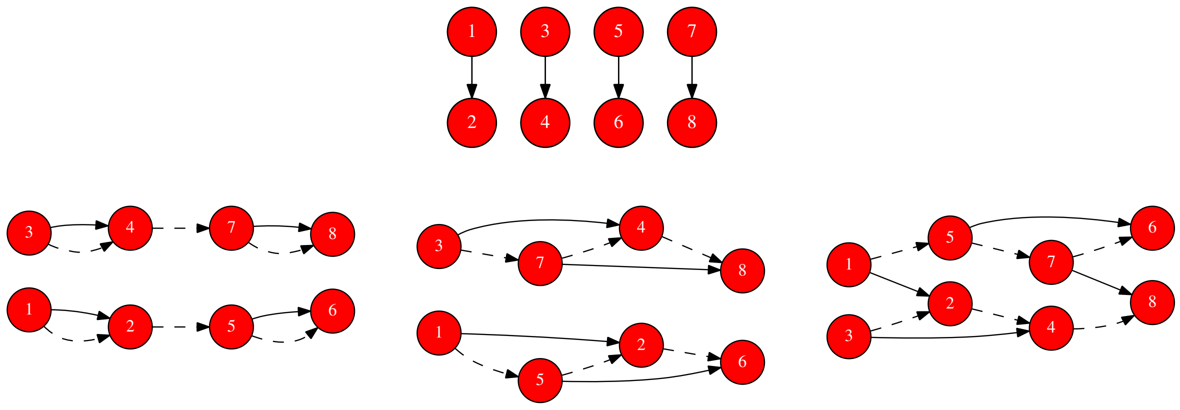


Figure 9.1: Four possible dependency graphs among eight actions. The top graph corresponds to a scenario with resources able to host all the actions at a time, and the others are three different schedulings on resources able to host up to two actions at a time.

dependencies – or re-assigning actions to other nodes. Continually monitoring the state of the execution allows the system to adapt the execution plan to workload variations produced by new incoming actions and correct erroneous decisions caused by workload mispredictions. By hosting the monitoring in an independent process, the system avoids blocking the processing of new incoming actions and the completion of the already scheduled ones throughout the optimization process. While the *Scheduler Optimizer* analyzes the current situation, the main process of the system can provide incoming actions with an initial resource reservation and release the dependencies of the finished ones to keep the execution progress by launching the dependency-free actions.

Finally, a third process, the *Resource Optimizer* complements the scheduling system with dynamic resource provisioning. The *Resource Optimizer* periodically checks the pending workload and the computing capacity at the moment and evaluates whether it is worth expanding or shrinking the infrastructure to better-fit the user restrictions.

9.2 Initial Scheduling

The initial allocation of resources to host one action execution starts on the *Action Scheduler*, where the scheduling system determines the most suitable node-implementation pair to perform the action. To make this initial decision, the *Action Scheduler* lies on *Scores*: a comparable object gathering all the information that any scheduling policy may need to compare two different action-node-implementation options. This information can relate to the action per se, to the fact of hosting of the action on the node, or to the execution of one of specific implementation on the node.

Upon the reception of a new action, the *Action Scheduler* computes the *Score* for every possible node-implementation pair. Initially, it determines the priority of the action and estimates time when the action will become free of dependencies by checking the expected end time of all the static predecessors of the action. After that, the *Action Scheduler* computes for each node the expected delay for transferring those input values missing in the specific node and determines the expected start time for any implementation assuming that the node has enough free resources to host it. Therefore, the last step is to compute the earliest possible start time for each implementation on the node. To contextualize an implementation execution on a node, the *Action Scheduler* requires knowledge about the availability of the resources of such node, information known by the corresponding *Node Scheduler*. For this reason, the former asks the latter to complete the *Score* with an estimation of the end time, energy consumption and economic cost of running such implementation on the node based on historical data from previous executions.

Finally, the *Action Scheduler* only needs to compare all the obtained *Scores* to select the best node-implementation pair. By merely changing a one-to-one comparison function, the owner of the infrastructure can define different policies to select the initial node-implementation selection without any need of looking at the code of the scheduling system. For instance, comparing the expected end time of the options minimizes the execution time of the application; changing the behavior of the comparison function to consider only the energy footprint of each option modifies the system so that it minimizes the energy consumption of the execution. Upon taking the decision, the *Action Scheduler* submits the action to the *Node Scheduler* corresponding to the selected node indicating the selected implementation so that it adds the necessary resource dependencies.

Determining the earliest time when an implementation can start running on a node is not straightforward. It requires keeping track of all the already scheduled executions and finding out when there will be enough resources to host it and check that they will remain available throughout the whole execution. To ease the seeking, the *Node Scheduler* keeps a register of those moments when some resources are idle. For each of these moments, known as *gaps*, it records a description of the available resources, the time when they become available, the action that used them immediately before – the origin of the *gap* – and the earliest time when another action use them again. Initially, the *Node Scheduler* has one single *gap* registered with an unknown origin that contains all the resources of the node – for instance, two CPU cores – from timestamp 0 to the end of the execution.

When the *Action Scheduler* decides to submit an action to the node, the *Node Scheduler* checks if there is any combination of *gaps* that could host its execution. For instance, when the *Node Scheduler* from the previous example receives the first action, which uses one CPU core for 100 ms and has no dependencies with other actions, it decides to reserve the resources from the *gap* with two CPU cores. For doing so, the *Node Scheduler* splits the *gap* into two gaps: one containing the occupied resources from the *gap* start time until the scheduled start time of the action and a second one containing the remaining resources with the same start and end time of the original gap. Both *gaps* maintain the origin of the original gap. In the example, the two-CPU-cores *gap* becomes two gaps: one with one CPU core starting a timestamp 0 until the expected start time of the action execution – timestamp 0 –; and one with one CPU core starting at timestamp 0 until the end of the execution. Since the first *gap* has the same start and end time, the *gap* lasts nothing, and the *node Scheduler* dismisses it.

Besides reserving the resources for the action execution, the *Node Scheduler* also needs to release them at the end of the action. For that purpose, it adds a new *gap* containing the resources released by the action from the end of the action execution until the end of the whole execution. In this case, the action releasing the resources becomes the origin of such gap. Therefore, after scheduling this first action on the example, the *Node Scheduler* would have two gaps: the one with the unused resources and the one containing the resources released by the action starting the end time of the action – timestamp 100 ms – until the end of the execution.

When the *Action Scheduler* assigns a second action to the node, the *Node Scheduler* repeats the process. For instance, it could receive an action exactly as the first one but with a static dependency expectedly released on timestamp 20 ms. In this case, the *Node Scheduler* would check the available *gaps* and find that it can fit the action on the unused resources. Therefore, it takes the *gap* and splits it into two gaps: a first one starting at 0 until the start of the action – timestamp 20 – containing the resources used by the action, and a second one with the remaining resources. However, since the action uses all the resources within the original gap, the *Node Scheduler* dismisses the latter. As with the first action, it registers a new *gap* with the resources released by the second action from timestamp 120 until the end of the execution. The origin of such gap is the second action. After scheduling the second action, the *Node Scheduler* has a list containing three gaps: the unused initial resources – one CPU core from timestamp 0 to timestamp 20 –, the resources released by the first action – one CPU core from timestamp 100 until the end of the execution – and the resources released by the second action – one CPU core from timestamp 120 until the end of the execution.

Actions may not fit in one single *gap* of the register; in such case, the *Node Scheduler* should group several *gaps* for fulfilling the requirements of the action. For instance, in the same example, the *Action Scheduler* could submit a third action to the same *Node Scheduler* requiring two CPU cores for 100 ms. In this case, the action should run on the *gaps* released by the previous actions. Since the action requires both gaps, its execution will not start until the resources of both *gaps* are available; i.e., timestamp 120 ms. Regarding the *gap* coming from the first action, the third action requires all its resources; therefore, it only creates on single *gap* from the *gap* start, timestamp 100 ms, to the start of the action execution 120. For the *gap* with origin the second action, the third action also requires all the resources; however, since the start times of the *gap* and the action execution are the same, the *Node Scheduler* dismisses any possible *gap* between the second and the third actions. Finally, the *Node Scheduler* registers the *gap* corresponding to the resources released by the third action. At the end of this third action scheduling, the register contains three gaps: the unused initial resources – one CPU core from timestamp 0 to timestamp 20 –, the resources from the first action idle until the third action runs – one CPU core from timestamp 100 until timestamp 120 – and the resources released by the third action – two CPU cores from timestamp 220 until the end of the execution.

To ensure that the execution uses the resources as scheduled, the *Node Scheduler* has to add the necessary resource dependencies. When an action employs the resources from a *gap* to run, the *Node Scheduler* adds a resource dependency from the action origin such *gap* to the action being scheduled. In the case of this third action, it employs resources from the *gaps* originated by the other two actions; therefore, the *Node Scheduler* adds two resource dependencies to the third action: one from the first action and one from the second one. Figure 9.2 depicts the evolution of the *gap* register and the dependency graph when the *Node Scheduler* processes these three actions.

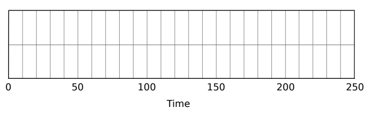
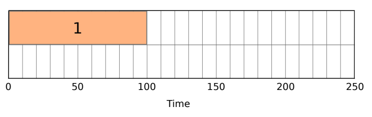
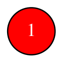
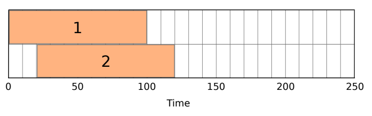
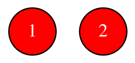
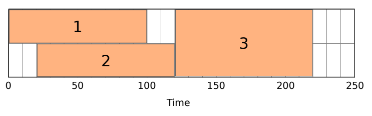
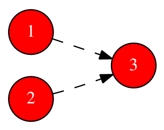
	Execution Plan	**Gap List**	**Dependency Graph**
Initial		< 2 CPU cores, 0, ∞, - >	
First Action		< 1 CPU core, 100, ∞, Action1 >, < 1 CPU core, 0, ∞, - >	
Second Action		< 1 CPU core, 120, ∞, Action2 > < 1 CPU core, 100, ∞, Action1 >, < 1 CPU core, 0, 20, - >	
Third Action		< 2 CPU cores, 220, ∞, Action3 > < 1 CPU core, 100, 120, Action1 >, < 1 CPU core, 0, 20, - >	

Figure 9.2: Evolution of the *gap* list within the *Node Scheduler* and the resource dependencies when scheduling three actions (Action1 and Action2 require one CPU core and Action3 requires two CPU cores) on a node with two CPU cores. Each *gap* is described as a 4-tuple indicating the resources contained, the start time, the end time and the origin action, respectively.

The more actions one *Node Scheduler* processes, the longer its *gap* list may become since *gaps* too short to host an action execution are more likely to appear. To avoid the computational cost of considering these *gaps* when finding the earliest moment when the resource can host the action execution, the initial scheduling policy does not consider backfilling; the *Node Scheduler* only maintains on the register those *gaps* whose end time is not defined.

9.3 Scheduling Optimization

By not considering backfilling, the initial policy leads to inefficient executions. Static dependencies can force a late execution of some actions; if it ignores the *gaps* with resources later-employed on other actions, the resources dedicated to statically dependent actions will remain idle until these run. To improve the performance of the execution, the system runs a secondary process in parallel to the main management of the execution that aims to improve the future execution plan: the *Scheduling Optimizer*.

This process iteratively takes the current execution plan and, acting in a local scope for every node, it tries to reduce the idle time of its resource as much as possible. Once it has shortened the execution time of all the nodes, the *Scheduling Optimizer* tries to reassign one action from a node to another checking if the move improves the solution using the Score comparison function. When the process reaches a convergence state where no action movement improves the current solution, it pauses. After a configurable period of time, the process resumes and performs the optimization taking into consideration any deviation of the execution prediction compared to the real execution and newly-submitted actions.

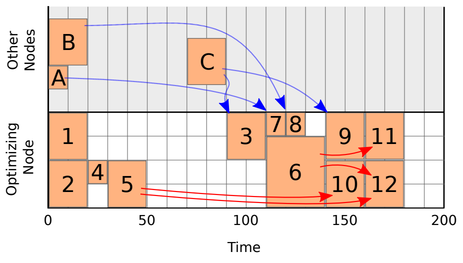
For shortening the execution time on one node, the local scope procedure tries to reorder the execution of the actions assigned to the node while maintaining the implementation selected by the initial scheduling policy. For doing so, the *Scheduling Optimizer* operates on two stages. First, during the Scan stage, it traverses the current plan for the node and classifies all the actions assigned to it into four ordered groups according to the state of its dependencies.

The first group, the *Running* actions, contains the actions that the node is running at the moment – i.e., those actions free of dependencies – and sorts them according to their expected end time. The *Ready* actions group is the second set and contains all those actions with no static dependencies that have not started their execution yet because they have a resource dependency with some other action. This set sorts the actions according to their priority. Currently, except for some higher priority type of actions like *Start Worker Action* or *Stop Worker Action*, the group prioritizes the actions within the same category in strict order of arrival into the scheduling system. However, this policy could change and consider other options such as prioritizing those actions consuming more resources. Finally, actions with some unresolved static dependency split into two groups depending on the node assigned to their predecessors. If all the preceding actions are assigned to other nodes of the infrastructure, the local-scope procedure does not influence on the estimated dependency release timestamp; the *Scheduling Optimizer* groups these actions into the *Pending-Remote* group and sorts them by their release timestamp. Otherwise, if any of the static dependencies relates the action to at least one action assigned to the same node, the *Scheduling Optimizer* cannot estimate the timestamp until it fixes the execution of all the predecessor actions. These actions group into the *Pending-Local* set with an undetermined order.

Once the Scan procedure has classified an action, it unschedules the action execution by blocking any change in its state and removing all its resource dependencies. For stalling the main management process for the shortest time as possible, the Scan process starts the analysis by the end of the execution plan. Using the registered gaps, *Node Scheduler* can quickly determine the actions at the end of the execution plan. By navigating through the resource dependencies of the actions, the Scan process traverses the dependency graph analyzing and unscheduling the actions from the end of the execution plan to the actions at the moment of their analysis, which are the last to get their execution state locked. Thus, the process doing the main management of the execution can deal with the completion of the actions running when the scanning started, and go on with the execution plan while the traversing of the graph does not reach its front.

On the other hand, by starting the graph traversal by its end, the Scan process dismisses any action submitted later than its start. To include these actions on the optimization, the *Node Scheduler* adds them into a fifth group, *New Submissions*, and blocks their execution in spite of applying the initial scheduling policy as usual. At the end of the Scan process, the *Node Scheduler* classifies all the actions in this fifth group into the other groups using the same criteria except for the *Running* group. These actions have no resource dependencies because the *Node Scheduler* has not planned their execution yet; usually, they should have a resource dependency with any of the actions at the end of the execution plan or another action within the *New Submissions* group. Therefore, the *Node Scheduler* moves those actions with no static dependencies into the *Ready* group.

Figure 9.3 depicts an example of the execution plan of twelve actions in a *Node Scheduler*. The Scan process classifies actions 1 and 2 as *Running* actions. Actions 4, 5 and 6 have no static dependencies but have resource dependencies; therefore the *Node Scheduler* classifies them as *Ready* actions. The other seven actions have static dependencies with other actions. Actions 3, 7, 8 and 9 only have dependencies with actions assigned to other nodes of the infrastructure – C, A, B and C respectively –; hence the *Node Scheduler* classifies them into the *Pending-Remote* group. Given the end time of their predecessors the group sorts them following the next order: 7 – A finished at timestamp 10 –, 8 – B finishes at timestamp 20 –, 3 and 9 – C finishes at timestamp 90. Finally, actions 10, 11 and 12 have at least one static dependency with another action assigned to the same node; then, they belong to the *Pending-Local* group.



Running: 1, 2
Ready: 4, 5, 6
Pending-Remote: 7, 8, 3, 9
Pending-Local: 10, 11, 12

Figure 9.3: Execution plan generated only with the initial scheduling policy of twelve actions on one node before performing the local-scope optimization (left) and the group classification at the end of the Scan phase (right).

After deconstructing the execution plan and assigning all the actions into the groups, the local-scope optimization gets into the second phase, the Reschedule, where it builds a new execution plan trying to improve the previous one. For that purpose, the *Scheduler Optimizer* simulates an execution where an online scheduler checks if there are enough available resources to host the highest-priority dependency-free action.

Given that the granularity of the actions can vary from milliseconds to hours, instead of discretizing the time and evaluating the resource availability on each interval, the simulation bases its progress on the events changing the amount of available resources or the runnable actions. The events it considers are the start and end of an action execution on the node and the resolution of all the static dependencies of actions with actions scheduled on other nodes. By keeping a register of all the upcoming events and the timestamp when they are expected to happen, the simulation can directly detect the next change in the available resources by picking the closest future event. Then, it can check whether the highest-priority dependency-free action fits in the currently available resources or it must wait for further upcoming events releasing resources to schedule the action.

Initially, the rescheduling process expects no events and considers all the resources on the node as available; i.e., the simulation has a single *gap* with an undefined origin that contains all the resources of the node from the initial simulation timestamp – corresponding to the moment when the reschedule started until the end of the execution.

The process cannot preempt the actions already running; therefore, for each action within the *Running* group, it registers an action-start event associated to the timestamp when the rescheduling stage started. The rescheduling process neither influences on the moment when actions within the *Pending-Remote* group become dependency-free since it does not control the execution of actions on other nodes. Hence, the simulation knows beforehand when to expect the release of all the static dependencies of those actions; consequently, it registers for each action within the *Pending-Remote* group an event associated to the expected release time. Figure 9.4 illustrates the initial state of the simulation after processing the actions within both groups in the example already scanned in Figure 9.3.

After processing all the actions within both groups, the *Scheduler Optimizer* enters into an iterative procedure that tries to fit the highest-priority action from the *Ready* set as soon as there are enough resources to host its execution. Each iteration of the procedure starts by polling the earliest expected event and compares its associated timestamp to the simulation timestamp. If both timestamp are equal, it means that the time for the event to happen has come; the simulation has to consider from that moment on the effects of such event on the available resources. Otherwise, if the event timestamp is bigger than the current simulation timestep, it means that the simulation has not reached the moment to apply the changes of the event yet; the resources still available at that point of the simulation could host other actions. If there are enough available resources to host the highest-priority action of the *Ready* group, it removes the

Event List	**Gap List**	**Groups Content**
<0, SA, Action1>		
<0, SA, Action2>		
<10, SDR, Action7>	< 4 CPU core, 0, ∞, - >	**Ready:** 4, 5, 6
<20, SDR, Action8>		**Pending-Local:** 10, 11, 12
<90, SDR, Action3>		
<90, SDR, Action9>		

Figure 9.4: Simulation state at the beginning of the reschedule process in the example already scanned in Figure 9.3. The leftmost part of the figure shows the list of events expected to happen (<timestamp, type of event: Start Action (SA) or Static Dependencies Release (SDR), Action>), the center, the list of available resources as gaps, and the rightmost part, the content of the updated groups.

action from the group and registers a new start-action event expected to happen on the current simulation timestamp to start the execution and puts the polled event back into the register. Conversely, if the highest-priority action does not fit in the available resources, the resources have to remain available until the following event; hence, the simulation leaps forward until the time when the polled event takes place and incorporates the effects of such event.

The consequences of each event are different; therefore, so it is how the *Scheduler Optimizer* handles each type. The most simple events to deal with are the ones representing the release of all the static dependencies of an action on actions scheduled in other nodes. In this case, their only consequence is that from that moment on the online scheduler can decide to run the released action; therefore, the *Scheduler Optimizer* only needs to add the task into the *Ready* group.

When handling an event representing the end of an action execution, the simulation can consider the resources employed by the action as available again; therefore, it registers a new *gap* originated by the finished action containing the employed resources from the current simulation timestamp until the end of the execution. Besides the resources, the end of an action also releases the static successors of the action from that dependency. If at that point of the simulated execution, all the static dependencies from the successor are resolved, the action can run from that moment on; hence, the *Scheduler Optimizer* should move the action from the *Pending-Local* group to the *Ready* one. Conversely, if the action still has some pending static dependencies, the *Scheduler Optimizer* has to check if all the pending predecessors of the action run on other nodes. In that case, the rescheduling process no longer influences on the moment when the action is released of all its static dependencies; the process can predict when it becomes dependency-free, register the corresponding static dependencies release event and remove the action from the *Pending-Local* group. Otherwise, if the action still statically depends on at least one not-yet-rescheduled action, it remains in the group until the processing of all the events corresponding to the end of the predecessors.

The last kind of event to deal with is the start of an action execution. At that point, the

Scheduling Optimizer has to reserve enough resources to host the execution. For doing so, it analyzes the list of *gaps* with the available resources at that time and picks those with the closest earliest start timestamps. Given that the only way to process an action-start event is immediately after polling another event associated to a future timestamp and verifying that the highest-priority *Ready* action fits on the available resources, the *Scheduling Optimizer* always finds a set of *gaps* containing enough resources to host the action. The start of these *gaps* might not be the current simulation timestamp; therefore, each *gap* gets divided into two parts, as with the initial scheduling policy: one *gap* with the unused resources maintaining the original time-lapse and a second *gap* with the employed resources finishing at the current simulation timestamp. While the resources of the former remain available for scheduling future executions on them; the regular flow of the iterative procedure would dismiss the ones of the latter. To avoid that these resources remain idle during the real execution, the optimization tries to fill this second *gap* executing actions with a lower priority. For that purpose, the *Scheduling Optimizer* goes through the *Ready* group in strict order of priority looking for any action that could run on the resources within that time.

If no action within the *Ready* group matches the *gap* constraints, the resources remain idle during the lapse of time of the gap. Conversely, if the *Scheduling Optimizer* finds a fittable action, it plans its execution as early as its static dependencies allow within the time frame of the gap. Thus, the original *gap* makes way for an action execution and up to three new gaps. The *remaining gap* represents the availability of the resources unused by the action; therefore, it has all the characteristics of the original *gap* but the amount of resources. The *preceding gap* represents the time that the resources employed by the action before the execution; therefore, its start and origin remain the same of the original *gap*, and its end matches the start of the execution of the selected action. Finally, the succeeding gap is the time lapse that the used resources are available between the end of the selected action and the end of the original gap. The selected action is the last action to use such resources; therefore, it becomes the origin of the gap. Since any of these *gaps*, in turn, can host the execution of other actions, the *Scheduling Optimizer* recursively tries to fill them with lower-priority.

For the real execution to reflect the simulation, the optimization process has to translate its scheduling decisions into resource dependencies. When the simulation decides to run an action on the resources within a gap, the *Scheduling Optimizer* defines a resource dependency of such action on the last resources using them, i.e., the origin of such gap. When handling a start action event, the *Scheduling Optimizer* tries to fit lower priority actions on each of those *gaps* before running the action. If it cannot find any action to run, it assigns the action a dependency on the origin of such gap. Otherwise, if the *Scheduling Optimizer* finds an action to run within the gap, the action depends on all the last actions using the resources within the *succeeding* and *remaining* gaps. In turn, the selected action depends on the last action using the resources within the *preceding gap*. If no action fits on it, it would depend on the origin of the *preceding gap*;

otherwise, it would depend on the last actions using the *succeeding* and *remaining gaps* of the recursively selected action.

During the Scan stage, the *Scheduling Optimizer* locks the status of every action to avoid any unexpected modification during the optimization process. Once the optimization has already scheduled an action execution – handled the corresponding end event – and has determined all its immediate successors – the *gap* created at its end and all the possible divisions have been fully assigned as resource dependencies –, no change on the action status can affect the result of the optimization. Hence, it is no longer necessary to hold the lock on the action status; the *Scheduling Optimizer* releases it so that the real execution progresses while it optimizes the remaining part of the execution.

Finally, every started action finishes at some point; to end up the handling of an action-start event, the *Scheduling Optimizer* registers the corresponding action-end event associated the expected timestamp of its completion: the current simulation timestamp plus its expected length.

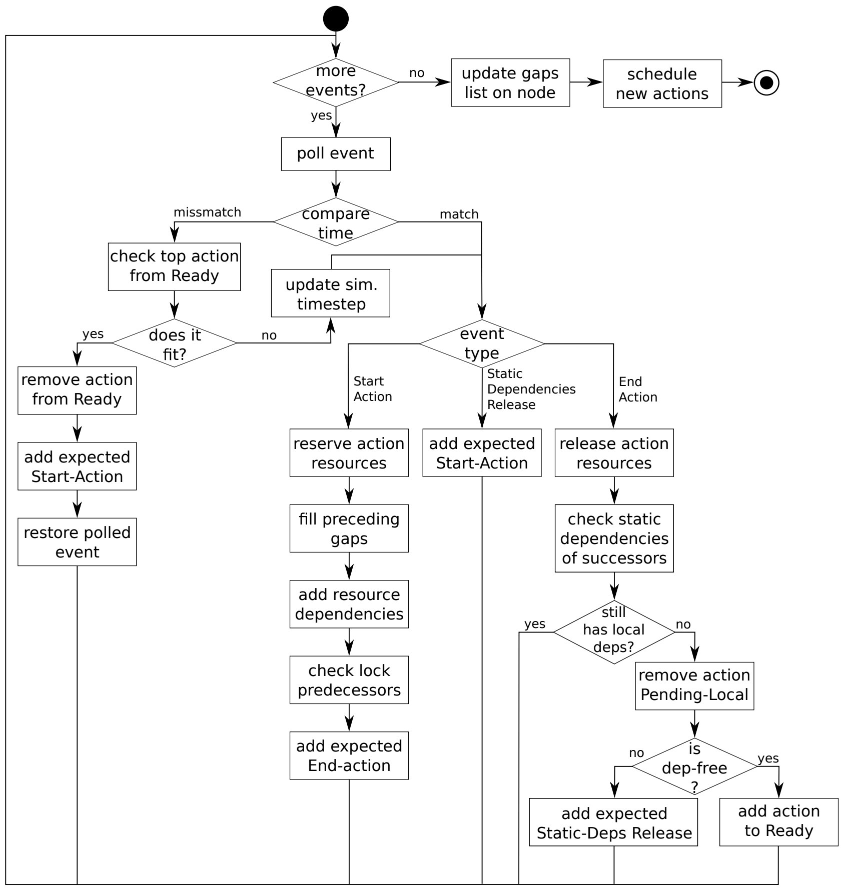


Figure 9.5: Flowchart of the iterative process leading the Reschedule stage of the local optimization.

The whole iterative process, depicted as a flowchart in Figure 9.5, finishes when there are no more registered events. At that point, the *Scheduling Optimizer* updates the list of *gaps* registered in the corresponding *Node Scheduler* according to the new execution plan and schedules any action submitted to the node during the optimization process using the initial scheduling policy as if the *Action Scheduler* had submitted the action upon the optimization completion.

The described procedure allows the initial execution plan in the left part of Figure 9.3 to reduce its execution time from 180 ms to 130 ms as depicted in Figure 9.6. By reducing the execution time while maintaining the implementation chosen by the initial scheduling policy, the optimization also targets a lower energy consumption and price to pay for the optimized node. With a more time-efficient solution, the application avoids spending the power dedicated to idle resources and its impact on the overall execution bill.

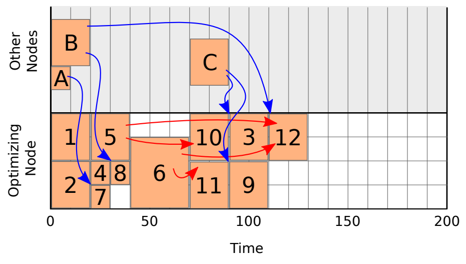


Figure 9.6: Execution plan after the local-scope optimization processes the execution plan on the left part of Figure 9.3.

After shortening the execution plan for every node, the *Scheduling Optimizer* tries to balance the workload among the nodes by changing the selected node and implementation for one of the actions. There exist several different ways to estimate the workload assigned to a node: the number of pending actions, the expected resource usage percentage, the overall energy consumed by the node, ... Since the goal of the optimization is to reduce the execution timespan, the currently implementated optimization uses as workload metric the time when the resources of the node have run all the assigned actions; i.e., the biggest end time of all its actions, which matches the start time of the latest *gap* registered in its *Node Scheduler*.

For the *Scheduling Optimizer* to decide which action movement to apply, it looks for an action assigned to the node with the highest workload level that is worth moving to another node. Given that the *Scheduling Optimizer* only moves one action of the node per iteration, it needs to establish a priority among the actions assigned to each node. The current policy implementing this intra-node priority considers only the end time of the actions: the later an action finishes, the higher priority it has; thus, actions scheduled at the end of the execution are more likely to change the node where they run. To decide the new host for the action execution, the *Scheduling Optimizer* checks the workload levels of the nodes again. First, it checks if it is worth to move the action to the node with the lowest workload level. For doing so, it computes the *Score* for the current action-node-implementation tuple and for every implementation able to run on the candidate node. If none of the implementation on the candidate node improves the current solution, the process tries to move the same action to the next node according to the ascending order of workload levels. If no node can improve the current scheduling for the highest priority action on the node, the *Scheduling Optimizer* looks for another action assigned to the same node worth moving by iteratively applying the same procedure to the remaining actions in strict order of intra-node priority. If the *Scheduling Optimizer* processes all the actions assigned to the node without finding anyone worth moving, it tries to do the same for the next node according to the descending order of workload level. If after processing all the nodes of the infrastructure, no action movement improves the current scheduling, the optimization process reaches a convergence state and finishes.

Conversely, if while considering any action movement during the process, the *Score* computed for one implementation on a different node improves the *Score* for the current action-node-implementation combination, the *Scheduling Optimizer* has to apply the change. For that purpose, first, it needs to unschedule the action execution; therefore, all the resource dependencies of the action have to disappear, and the current successors of the action replace any resource dependency on the unscheduled action by resource dependencies on the predecessors of the unscheduled action. Once the *Scheduling Optimizer* completes the unscheduling of the action, it needs to reschedule its execution on the node. For that purpose, it submits the action to the *Node Scheduler* corresponding to the selected node so that this schedules the execution of the selected implementation on the node and adds the necessary resource as if it was scheduling the action for the first time.

Taking an action out from the execution plan for a node leaves the resources expected to host the action idle during the time-lapse when the action was to run. Therefore, if the removed action was not at the very end of the execution plan, those idle resources could host the execution another action during that period. Likewise, as discussed at the beginning of this section, the initial scheduling policy can also lead to an inefficient execution plan with idle resources at some point of the application. Hence, the execution plan for both nodes, the donor and the receiver, should be optimized again. Rescheduling the execution of these nodes may change the release moment of some static dependencies on actions assigned to these nodes; the optimization of these nodes may affect actions assigned to others. Therefore, if the *Scheduling Optimizer* decides to move one action, it triggers the re-execution of the whole optimization procedure. First, it performs a local-scope optimization on every node of the infrastructure, and then, it looks for another worthy action movement.

9.4 Dynamic Resource Provisioning

A good planning of the execution on top of the available resources can achieve shorter execution times and lower costs from the energetic and economic point of view. Complementing the scheduling system with a mechanism to exploit elasticity of the Cloud providing the resource pool with dynamicity boosts the effects of the system. By adding new resources into the pool, the system can reduce the execution time of the application when they allow a better-exploitation of its inherent parallelism. Conversely, when the parallelism degree of the execution falls and some resources remain idle, shutting down some of them allows the system to reduce the energetic and economic costs of the application. On the one hand, powering them off avoids consuming the power to feed idle resources; and thus, their energy footprint and corresponding expenses disappear. On the other hand, if the infrastructure is composed of virtual machines deployed on a third party, shutting down a VM avoids the charges for the additional time that the VM remains on.

The *Resource Optimizer* is a third process part of the scheduling system that runs in parallel to the *Scheduler Optimizer* and the main management processes. Its goal is to monitor the pending workload and provide the scheduler system with the amount of available resources that fits better with the characteristics of the application and the requirements of the infrastructure owner. For that purpose, it monitors the amount of time, energy and money already spent by the application and computes the respective budgets to run the remaining part of the application without exceeding the boundaries defined by the user. Being aware of the current limits for the execution, the *Resource Optimizer* computes the expected costs following the current execution plan and triggers the best actions to adapt the resource pool in accordance with the necessities.

To estimate the costs of the current execution plan, the *Resource Optimizer* uses the information stored in the *Action Scheduler* and its *Node Schedulers*. For forecasting the timespan of the execution, it only needs to check the end time of the very last action running on the platform. Practically, it can get that information from the list of *gaps* registered in every *Node Scheduler*, the latest start time of all the gaps of the system is the estimated end time of the execution. Regarding the energy consumption, the system relies on a model that considers the base consumption of the platform and the additional consumption caused by application workload. For computing the base consumption of a node, the model multiplies its power consumption when all the resources are idle by the time they remain on. Then, to compute the energy consumed by the node, it has to incorporate to this value the consumption of all the actions assigned to the node, which depends on the characteristics of the selected implementation for each task. Finally, the overall energy consumption of the application is the accumulation of the energy spent by all the nodes of the infrastructure. In terms of total monetary cost, as with the energy, the cost of an execution is the accumulation of the cost of all the nodes composing the infrastructure. The pricing model used to estimate the cost of each node combines a fixed expenditure, motivated by the reservation of the resources throughout the execution, with a variable bill induced by the utilization of such resources. Usually, the fixed part of the price is a proportional share of the amortization and maintenance costs of the cluster, and the variable part of the cost corresponds to the energy bill. When the node is a virtual machine deployed in a commercial cloud provider like Amazon, the cost is constant; the price of the energy is 0. The total cost of the node corresponds to the fixed part of the model which depends on the fees charged by the provider for that VM.

Figure 9.7 contains the equations notating the described models.

End time:

$$ET_{exe} = \max_{a \in A} ET_a$$

Energy Consumption:

$$EC_{exe} = \sum_{n \in N} EC_n$$

$$EC_n = Pi_n * T_n + \sum_{a \in A_n} EC_{an}$$

Monetary Cost:

$$MC_{exe} = \sum_{n \in N} MC_n$$

$$MC_n = P_n * T_n + EC_n * PE_n$$

Variable	Description
ET_{exe}	Expected execution end time
A	Set of all actions to perform
ET_a	Expected execution end time of action a
EC_{exe}	Expected execution energy consumption
N	Set of nodes composing the infrastructure
EC_n	Expected energy consumption for node n
Pi_n	Power consumption when node n is idle
T_n	Time that node n is on
A_n	Set of actions to perform on node n
EC_{an}	Energy consumed by the computation of action a on node n
MC_{exe}	Expected execution monetary cost
PE	Price for the energy
MC_n	Monetary cost of having node n
P_n	Price of having node n

Figure 9.7: Models to estimate the end time, energy consumption and monetary cost of running a set of actions A on a set of nodes N.

Given the current resource configuration, the *Resource Optimizer* considers two possible changes: obtaining one additional VM instance to act as a new node of the infrastructure incorporating new resources into the pool (scale-out) or releasing the resources of one node by destroying the corresponding VM (scale-in). Although it is possible to create VMs tailored to the necessities of the client, cloud managers usually offer a set of VM templates with predefined characteristics. The infrastructure administrator has to configure the scheduling system to indicate to which cloud providers the *Resource Optimizer* can request new VMs and the characteristics of the templates available on each provider.

To decide whether to change the resource pool or not, the *Resource Optimizer* compares the costs of running the application on every resource pool reachable through one single change. For every template defined by the user, the system considers one scenario where it scales-out and one instance of such template joins the resource pool. Likewise, it makes no distinction among nodes and considers one scenario for each already available VM where it scales-in and destroys such VM. To estimate the time, energy and money that it takes to run the application on the resource pool available on each scenario, the *Resource Optimizer* runs a coarse-grain simulation. For carrying out this simulation, the *Resource Optimizer* groups the pending-to-run actions performing a similar activity and counts the number of actions in each group. Then, it checks how many actions of each group can run in parallel on each VM and how long do these simultaneous actions take to run. Finally, it balances the load among the nodes. For that purpose, the *Resource Optimizer* sorts the available resources by the moment when they end the execution of the actions currently running on the nodes and gets into an iterative procedure that distributes the actions of every group among the nodes able run them. For doing this distribution, it picks a group, gets the compatible node becoming idle earlier, assigns to it as many actions of such group as it can host at a time, and delays the release timestamp according to the execution time of those actions on the node. The *Resource Optimizer* repeats this procedure until it has assigned all the actions; then, it can determine the end time of the execution of each node and the number of actions of each group assigned on it.

These simulations are agnostic to the static dependencies among the actions; hence, the actual execution of the workflow may completely diverge from the simulated plan. In an attempt to improve the forecasts, the *Resource Optimizer* applies a correction factor to the end time of the execution. The value of this factor is the relation between the estimated end time computed with the current execution plan and the estimated end time obtained from the simulation of the execution on the current resource pool. Thus, for one application whose execution plan on the current resources takes 1000 s, and the simulation on the same resources forecasts an execution of 500 ms; the correction factor is 2. If the simulation for a scenario removing one of the nodes forecasts an execution of 700 ms, the corrected end time of the execution using the new resource pool is 1,400 s. By providing the corrected end time and the simulation workload distribution to the afore-described models, the system forecasts the time, energy and money necessaries to run the application on the modified pool of resources.

Once it has the cost forecast for all the possible scenarios, the *Resource Optimizer* compares them, picks the one achieves the best result according to the preferences of the infrastructure owner, and performs the corresponding change. The priority of the system when choosing the solution has to avoid exceeding the consumption boundaries; it dismisses any candidate solution surpassing them unless that the current resource configuration is already beyond the limits and the candidate gets the costs closer to them. After the filtering, the *Resource Optimizer* looks for the resource pool achieving a lower cost for the parameter to optimize – time, energy or money –. If multiple solutions achieve the lowest cost for the optimized parameter, the tiebreaker criterion is trying to reduce of the other two. Often, the cost differences for the other two parameters are contrary. For instance, in a system minimizing the monetary cost of the execution, two scenarios can require the same amount of money; however, one scenario can achieve a lower energy consumption while the other can achieve a shorter execution time. In this cases, the implemented policy establishes that reducing the execution time has the highest priority and saving money has the lowest.

If the system has to scale-out, the *Resource Optimizer* creates a new *Node Scheduler* managing no resources and submits to the *Action Scheduler* one actions bound to this new *Node Scheduler*.

This action, the *Poweron Action*, has the highest priority and no resource requirements. Its purpose is to request the creation of a new instance of the selected template to the corresponding cloud provider, monitor the booting process of such VM and contextualize it. Usually, cloud providers give access to VMs with the requested features; however, the amount of computational resources may mismatch if they are scarce. At the end of the action execution, it has to update the amount of resources managed by the *Node Scheduler* with the granted resources.

So that other actions can use these resources to compute tasks on them, it is necessary to start the worker process of the runtime. For that purpose, the *Poweron action* submits a second action to the *Action Scheduler* bound to the node with the highest priority: the *Start Worker Action*. This second action requires all the resources of the node; given its higher priority, other actions cannot start their execution until the worker process is up and running.

Conversely, if the system has to scale-in, the *Resource Optimizer* has to retrace the steps done when scaling-out. For doing so, it submits a high priority action, *Stop Worker Action*, bound to the node. This action requires all the resources of the node with the purpose of stopping the worker process on it. Since the action requires all the resources of the node, its execution does not start until all the actions running on the node finish; given its higher priority, the *Node Scheduler* plans its execution as early as possible, coming ahead of the execution of other actions previously assigned to the node. From the moment when the worker process stops, the node cannot receive any more application-level commands; therefore, the other nodes of the infrastructure cannot retrieve data from it. Before turning down the worker process, it is important to replicate on any other node of the system every data value only located on the node being switched off. Once all the data in the node is available through another node of the infrastructure, the system submits the command to terminate the worker process on the node and notifies the *Node Scheduler* that all its resources no longer exist.

To end up its activity, the *Stop Worker Action* requests the *Action Scheduler* to run one last action bound to the node: the *Shutdown Action*. The goal of this last action is to contact the cloud provider hosting the VM to destroy it. As with the *Poweron Action*, the *Shutdown Actions* does not require any resources to run; thus, it is the only action that can run on the node after the *Shutdown Action* has removed all the resources from the *Node Scheduler*. At the end of this action, the VM does no longer exist; therefore, the *Action Scheduler* dismisses the corresponding *Node Scheduler*.

For the application to go on with the execution, the system has to reschedule all the action assigned to the node before dismissing it. Once the *Stop Worker Action* has reduced the resources managed by the *Node Scheduler*, the *Scheduler Optimizer* should evict all the actions from the *Node Scheduler* moving them to other nodes. The lack of resources makes the start time of such actions undeterminable; and once the load balancing procedure detects that, it moves the actions to other nodes. If the *Shutdown Action* finishes before the *Scheduler Optimizer* re-assigns all the actions, the *Action Scheduler* plans the execution of the ones remaining as if they just arrived into the system, using the initial scheduling policy; and then, it deletes the *Node Scheduler*.

9.5 Evaluation

This section presents the results of the test conducted to validate the self-adaptation capability of the described system. For that purpose, it uses the jEPlus application [9] developed by the GREEN PREFAB company, a simulation manager that runs a set of parametrized EnergyPlus [7] executions. Each of these runs simulations provides architects with an evaluation of the thermal quality and indoor comfort of a building given certain climate conditions and its architectural design to assess them on the election of materials or designs.

To port the application to the COMPSs programming model, the application has one single Core Element, *simulate*, that contains the execution of one EnergyPlus simulation. Since all the simulations are independent of each other, all the tasks composing the application can run at a time if the infrastructure has enough resources to host them. The application has two different implementations of the *simulate* Core Element using two different versions of the EnergyPlus software generated with different compiler flags.

GREEN PREFAB identified three different profiles of jEPlus users whose requirements become the preferences defined by the user to lead the scheduling system. Urgent computations require getting the result as soon as possible regardless the monetary cost and the energy consumption of the execution. The second profile corresponds to academic executions where users do not care about the performance of the execution while the result of the execution is ready when they go back to work next morning (execution should end in 16 hours). However, users classified in the Academic profile do care about the final bill; the monetary cost of the execution is the parameter to optimize. Finally, the third profile corresponds to users that run the application trying to achieve an energy-efficiency certificate. These certificates take into account the energy efficiency at all the stages of the building lifecycle; thus, they also measure the energy spent during its design. In this case, users within this profile, named green profile, aim to optimize the energy consumption of the application, but they limit the total bill for the execution to 20 € and its timespan to one week (604,800 s). Table 9.1 summarizes the description of these three profiles. The test compares the behavior and costs of executing a project using a medium-sized dataset that generates up to 4,608 simulations when users configure the scheduling system according to meet the requirements of each profile.

Profile Name	**Optimization Parameter**	**Boundaries**
Urgent	Execution time	-
Academic	Monetary cost	57,600 s (16 h)
Green	Energy consumption	604,800 s (7 d) and 20 €

Table 9.1: Scheduling system preferences according to the user profiles detected by Green Prefab.

The jEPlus executions run in a private cloud running on a cluster of 32 nodes, each equipped with one Intel Xeon E3-1230 V2 (quad-core) processors at 3.30GHz, 16 GB of RAM and 1.8 TB of local disk, interconnected by a Gigabit Ethernet network. The cluster leverages on the KVM hypervisor to virtualize the nodes, and OpenStack manages the virtual machines. The system is configured to use one single VM template where each instance has one virtual CPU core at 1.5 GHz, 3 GB of RAM and 4 GB of instance storage. The operating system installed in the VM is a Debian GNU/Linux 7.8 (wheezy).

The nodes are connected to a PDU (power distribution unit) that physically measures their power and energy consumptions. When the VMs are idle, each of them consumes 6.1 W. Since the cluster is located in Berlin, the test determines the economic cost due to the energy consumption using the price fees provided by the European Commission on Eurostat for Germany: 0.149 €/kWh. The fixed part of the monetary cost of the VMs is 0.03€/h, a value similar to the prices charged by Amazon for its EC2 instances. The total price of a VM idle for an hour is 0.0309 € as shown by the breakdown in Table 9.2.

Fixed Price per VM	0.03 €/h
Power Idle VM	6.1 W
Energy Price	0.149 €/kWh
Total Price per Idle VM	0.0309 €/h

Table 9.2: Idle VM price breakdown.

Running different versions of the EnergyPlus binary on the VM has different costs. Regarding the basic version of the binary, the average execution time is 331 s and its execution incurs an average power consumption increase of 5 W. Thus, the average energy consumption of a task using the basic implementation is 1,655 J and the corresponding bill amounts to 0.00007 €. Conversely, for the binary generated with the optimization flags on, an execution lasts 175 s with an average power consumption increase of 15.1 W. Thus, the total amount of additional energy spent on the execution is 2,642.5 J incurring an additional cost of 0.00011 €. Besides the costs directly related to its computation, tasks reserve some resources to run with a cost; therefore, the task assumes the corresponding share of the costs of these resources (0.0000083 € per second of reserved CPU core plus the costs of the 6.1 W consumed by the resources) as part of its own cost. Table 9.3 summarizes the differences among both implementations of the *simulate* Core Element distinguishing the costs observed due to the differential in the measures and the costs including the infrastrucutre.

The Urgent execution has no limits for the money and energy dedicated to the execution; therefore, the system requests the necessary resources to exploit all the parallelism of the application. The execution scaled-out up to 31 VMs, the top capacity of the infrastructure, and maintained that number of VMs until the end of the execution. The execution lasted for 26,100 s (7 hours and 15 minutes) spending 5.22 kWh and 8.21 €. All the tasks executed the optimized

	Basic	Optimized
Average Execution Time (s)	331	175
Average Δ Power (W)	5	17.1
Estimated Δ Energy Consumed (J)	1,655	2,992.5
Estimated Δ Energy Bill (€)	0.00007	0.00012
Average Total Power (W)	11.1	23.2
Estimated Total Energy Consumed (J)	3,674.1	4,060
Estimated Total Energy Bill (€)	0.000152	0.000168
Estimated Total Bill (€)	0.00276	0.00163

Table 9.3: Average measures for each version of the *simulate* Core Element running an EnergyPlus execution.

version of the Core Element since its faster than the basic one.

Both, the Green profile and the Academic profile try to use the lower number of VMs to reduce the additional energy and money spend by idle resources that appear at the end of the execution with load imbalances. For the Academic profile, the selected implementation is also the optimized one because its overall cost is lower than the basic one (0.00163 €vs. 0.00278 €). Using this implementation, the lowest number of VMs that can achieve an execution time lower than 57,600 s is 15. However, adding one more VM allows to shortens number of hours charged per each VM from 15 hours to 14, reducing the resources bill from 6.75 €to 6.72 €. When the scheduling system is configured to fit the requirements of the Academic profile, the application lasts 50,339 s, consumes 5.19 kWh and spends 7.49 €.

For the Green profile, the system picks the basic implementation of the Core Element due to its lower energy consumption (1,655 J vs. 2,992.5 J). In this case, running the application with one single VM would last more than 17 days, more than the one-week boundary. To reduce that difference, the application scales-out up to three VMs that the application uses for the first four days of the execution and then, it scales-in to two VMs. The overall execution time is 592,102 s (almost 6 days and 21 hours). The overall energy consumption shrinks to 4.72 kWh, but the money spent raises to 13.47 €.

Table 9.4 summarizes the results obtained for the executions with the three profiles. These results highlight, on the one hand, the importance of selecting the proper implementation for achieving the goals of the system. Using the basic implementation of the method slows down the execution since it requires 1.9x more time than an execution running the optimized implementation on the same resources. Although the execution time is longer, the power consumption of the overall system is 2.1x lower; thus, the total amount of energy shrinks. However, the price to pay for reserving the resources (0.03 €/h per VM) surpasses by far the savings on the energy bill.

On the other hand, the results also highlight the importance of the *Resource Optimizer* to fulfill the boundaries. The results obtained for the three profiles demonstrate that the system scales-out to provide the necessary amount of resources to achieve the desired execution time.

The Green profile allows to see the necessity of dynamic resource provisioning; the application initially detects that it needs to scale-out and use the resources of at least three VMs to meet the one-week deadline. After almost five days running the application on those resources, the *Resource Optimizer* notices that it can run the remaining workload using only two nodes and still meet the deadline. Thus, at that point, it decides to scale-in and reduces the number of VMs to two for reducing the power consumption of the system.

Profile Name	**Max. VMs used**	**Selected Implementation**	**Exec. Time (s)**	**Energy (kWh)**	**Price (€)**
Urgent	31	Optimized	26,400	5.22	8.21
Academic	16	Optimized	50,339	5.19	7.49
Green	3	Basic	592,102	4.72	13.47

Table 9.4: Results obtained after executing the application configuring the system according to the three profiles: Urgent, Academic and Green.

9.6 Summary

This chapter describes a system that allows the Cloud Platform to offload not only the execution of the tasks onto the remote resources but also the scheduling of such executions. Once the *Offload Decision Engine* decides to execute a task on the resources managed by an instance of Cloud Platform, this forwards the task execution request directly to the endpoint of such system so that it manages the execution of the task on the underlying infrastructure. On the one hand, offloading the task scheduling releases the mobile device from the overhead of the computation associated with it and allows it to use these resources for the execution of tasks. On the other hand, the higher computing capacity of the remote node running this system allows the development of policies with a higher computational load than the ones hosted on the mobile; policies lead either by the interests of the application user or by the owner of the infrastructure hosting the computation.

The purpose of this system is to orchestrate the execution of the received tasks on the computational resources of the nodes managed by the Cloud Platform. From a more general point of view, this system has to schedule the execution of a set of actions, computing tasks among them, on the set of resources composing the underlying platform. Therefore, the purpose of the system is to determine an execution plan that pursues an optimal execution according to the configured goals. This plan has to be consistent with the dependencies among actions and guarantee that the resources employed by actions running simultaneously on a node never outnumber its computing capabilities. To ensure that the execution of the actions follows the decided plan, the system expresses the order of the actions using the same resources defining a new set of dependencies, resource dependencies. Considering both, the static and the resource dependencies, the system can build a dependency graph of the actions to analyze the behavior of the execution plan. It is extremely important that the graph of the decided plan has no cycles for a cycle of dependencies would lead the execution to a deadlock.

For carrying out its duty, the system is organized in two levels. At a high level, an *Action Scheduler* is in charge of determining the where and how the actions run; i.e., which node executes which implementation of the action. At a low level, every node is represented by a *Node Scheduler* that manages the resources that belong to the node and decides when the action runs without oversubscribing the resources of the node. To take these decisions, the system runs a two-stage procedure. Every time that a new action reaches the system, the *Action Scheduler* asks each *Node Scheduler* a forecast of the end time, energy consumption and monetary expenses if the node hosted the action execution. After comparing the costs, the *Action Scheduler* picks a combination of implementation and node to run the action according to the optimized parameter and requests the corresponding *Node Scheduler* to schedule the execution of the selected implementation on its resources. At this point, the *Node Scheduler* greedily plans the execution of the action at the end of the execution without considering backfilling. Infrastructure manager – the owner or the user – can create new policies to select the node and implementation to run by merely overriding a function that compares two different options.

By not considering backfilling, the initial policy leads to inefficient executions. To improve the performance of the execution, the system runs an iterative process in parallel to the main management of the execution that aims to improve the future execution plan: the *Scheduling Optimizer*. To optimize the execution plan, it acts first on a local scale of each node; every *Node Scheduler* reorders the execution of all the remaining actions according to their priority. To determine the priority of each action, the currently implemented policy is aging; however, the infrastructure manager can easily modify how the system determines the priority of each action by overriding the corresponding method. Finally, once the optimization has acted on the local scope and reordered the actions of all the nodes, it tries to improve the execution at a global level by re-assigning one action from one node to another. To decide which action to move, the *Scheduler Optimizer* sorts the nodes according to their workload level and seeks an action worth moving. To select which of the actions should be assigned to another node, the *Scheduler Optimizer* subsequently checks every action in strict order of donation priority. Using the end time of an action to determine the donation priority of an action fosters reassigning those actions scheduled at the end of the execution while consolidating the most immediate decisions. Besides, considering the end of the last action to run on the node as a good indicator of the workload assigned to such node allows the runtime to balance the workload among nodes. As with the initial scheduling and the local-scope optimization, the infrastructure administrator can easily override both methods to create new policies for the global optimization.

Providing the system with dynamic resource provisioning boosts its effects by adapting the amount of available resources to the computation needs of the moment. Adding a new node to the current infrastructure allows the system to exploit higher levels of parallelism, and hence, reduce the timespan of the application, at expenses of increasing the power consumption and the price paid for the infrastructure. Conversely, releasing one node can save this costs when having this resources available makes no difference in the execution. To take the decision of scaling-in or scaling-out, the system periodically runs a coarse-grain simulation of the execution with all the possible scenarios after adding or releasing one of the nodes to forecast its end time, its energy consumption and the money spent. Comparing the results obtained for each possible scenario, the system can determine which is the best option and contact the corresponding resource manager to apply it. Currently, the elasticity policy pursues optimizing one parameter – execution time, energy consumption or money paid – while limiting the cost for the other two. As with the scheduling policies, the infrastructure manager can change the elasticity policy by merely overriding the function that selects an option comparing the costs.

Part V

Conclusions

Chapter 10

Conclusion

To wrap up this dissertation, this last chapter presents the conclusions extracted from this thesis and suggests possible research lines to continue the work described in this document. As explained in the Introduction chapter, the frame of the thesis is Mobile Cloud Computing technologies; specifically, it delves into the complexity of the development of mobile applications that offload part of the computation onto the Cloud.

As detailed in the State of the Art chapter, writing applications for MCC environments is not straightforward. To begin with, developers have to deal with the issues related to parallel programming for distributed infrastructures. First, they need to analyze the code of the application to partition it into several units of execution. Then, they need to modify the application to orchestrate the execution of such units on top of the underlying infrastructure guaranteeing the dependencies among them to produce the expected result. At that point, developers need to evaluate which units are worth running on the device and which ones to offload and decide on which node and when each unit should run. Finally, once they have decided the best resources and moment to execute each unit, developers have to implement a mechanism that allows them to submit the execution to the corresponding resources and provide them with all the necessary values to produce the expected result. Besides, the high mobility of the devices adds two new concerns to the cost/benefit analysis: the battery lifetime and the variability of the network. The battery is a limited source of energy; therefore, the amount of energy that an application consumes is important, and developers have to be aware of it when deciding where each execution unit runs. The network variability can rapidly change the costs of transferring the data to and from the remote nodes. Handing over to a mobile data connection from a Wi-Fi network increases significantly the time to transfer a data value and the energy consumed and the price to pay per each byte of data. In extreme cases, the mobile device can even get into an area out of signal and become isolated from the rest of the infrastructure. Developers have to control these situations and provide the application with the necessary mechanisms to continue its execution even if the isolation becomes persistent. All these issues bring into the spotlight the strong necessity of programming models that ease the creation of MCC and increase the productivity of the developers by releasing programmers of the management of these issues.

The efforts made throughout the thesis focus on answering the research question set out on Section 1.2: *"Could a programming model allow developers to create an application to run on a mobile device and transparently exploit an MCC infrastructure to enhance its performance?"*; and developing a prototype of the model that substantiates the answer. This dissertation proposes a programming model that successfully tackles the issues of MCC and hides them away from the application developer; thus, the described prototype already satisfies the requirements for the solution. Therefore, this thesis concludes that it is possible to create such a programming model.

Building on the COMPSs programming model, the described solution achieves the objectives of the thesis regarding the programmability of the model. Developers following the model code their applications being agnostic to the details of managing the parallelism and the underlying infrastructure. Instead of handling them manually, programmers implement the computational logic of their software in a sequential fashion with no references to the infrastructure. Besides, they write the code using the native language of the target platform; therefore, the adoption of the model is smooth, and developers can improve their productivity with no additional effort. For detecting the parallelism at runtime and executing the application on the infrastructure, the programming environment extends the process that builds the application distribution package (Contribution 1). This extension instruments the application code provided by the developer to invoke a runtime system that orchestrates the execution and replaces the original code with the modified version when bundling it into the application package on which also adds the runtime system.

Besides extending the application package building process, this thesis also proposes an extension of the COMPSs programming model to support method polymorphism. With it, developers can provide several implementations for one method; thus, the system can decide at runtime which of all the available versions runs according to the circumstances of the moment. Everything developers have to do is to declare the multiple versions of the method on the Core Element Interface.

The runtime system has to analyze the code of the application and partition it into execution units, detect the dependencies among them, and to orchestrate their execution on the available resources guaranteeing the sequential consistency of the logic programmed by the developer. For achieving this purpose in MCC environments, it is necessary a new runtime system implementing an architecture specially designed with the needs of MCC in mind (Contribution 2). Mobile devices usually host several applications running at the same time; therefore, for orchestrating their execution on the available resources holistically, all the applications have to contact a shared runtime system deployed as a service. Each application detects the tasks composing it and the dependencies among them using the mechanism already implemented in the original COMPSs runtime and then requests its execution to the shared runtime service. The system groups together the computational resources into several Computing Platforms according to the mechanisms required to provide the processing elements with the input values of a task, submit such task execution and collect its results. The runtime system picks which Computing Platform hosts the execution of each task according to a forecast of the costs – end time, energy and money – of the execution on each platform. The selected Computing Platform, via an internal task scheduler, decides the resources and the moment to run the task avoiding resource oversubscription.

Chapter 4 describes in more detail the overall solution proposed. Sections 4.1 and 4.3 explain respectively the extensions to the programming model and the building process, and Section 4.2 gives an overview of the runtime architecture. From Chapter 5 until this last chapter, the dissertation describes the design of different Computing Platforms (Contribution 3) and presents the results obtained from the tests conducted to evaluate their implementation.

The first platform described by the dissertation is the CPU Platform, which allows the runtime to run tasks using the cores of the main processor of the mobile device. For doing so, the platform has a static pool of threads that execute actions sequentially when the scheduler within the CPU Platform decides; however, once a task execution has started, the scheduler cannot preempt the thread to run another task. By using this platform, the runtime can already automatically exploit the inherent parallelism of the application to reduce the execution time. The results presented in Section 5.3.2 show that, on a mobile equipped with a quad-core processor, applications can achieve up to a 2.74x speedup. The more cores are running tasks at a time, the higher the temperature of the device gets and to control it the mobile reduces the frequency of the processor. Despite lowering the frequency reduces the power consumed by the processor, the energy consumed by each task grows since they take longer to run.

Chapter 6 describes the second implementation of a Computing Platform: the OpenCL Platform. This platform enables the execution of tasks on other computational devices embedded on the mobile such as GPUs, FPGAs or other accelerators. For doing so, the platform leverages on OpenCL, a standard for general purpose parallel programming for heterogeneous devices; developers have to code the tasks to run on the platform using a C99-based language and indicate the existence of the OpenCL implementation with the *@OpenCL* annotation (Contribution 1). The programming model hides away from the developers all the details of the interactions between the host code and the OpenCL devices; the runtime system performs all the necessary data transfers among the host and the device memories and handles the execution of the tasks as OpenCL kernels. The runtime system also deals automatically with the balancing of the workload giving applications flexibility to adapt to the necessities of the user. The conducted tests achieve up to 13.39x faster executions when the runtime applies policies pursuing the lowest execution time, and energy reductions eight times lower than running the application sequentially on one core of the CPU. Another benefit of delegating the load balancing to the runtime system is the portability of applications. The time to run a task and the energy consumption of the execution depend on the characteristics of the hardware; the runtime system can adapt the application to the specific infrastructure automatically.

Using both, the CPU and the OpenCL platforms, the runtime system can exploit collaboratively all the computing devices embedded on the mobile. The third and last implemented Computing Platform is the Cloud Platform, which allows the runtime to widen the available resources for the application with remote computing devices. These remote resources can be from nearby desktops, laptops or single-board computers to virtual machine instances deployed in a cloud passing through remotes servers offered an organization to its members. For a remote device to run tasks, the corresponding node has to host a process persistently listening to the network for task submissions. When it receives a new task, the worker fetches the necessary input data values and launches the task execution when there are enough available resources. For sharing data among nodes, it is necessary a mechanism that allows every node to know from which locations on other nodes it can fetch such value. This mechanism consists in a distributed hash table storing all the locations for every data value. When a node needs to fetch a data value, it looks up the identifier of the value on the table to get its locations of such value. Nodes can query the locations for yet-to-compute data values; for avoiding that the querier node stalls, the distributed hash table follows a publish-subscribe execution model. If no other node has published a location for the data value, the hash table registers the query and, upon the existence registration of the value, it forwards the notification to the querier node. Thus, this mechanism is not only useful for sharing data values but also allows the nodes of the infrastructure to become aware of the release of the data dependencies of the tasks to run.

The results presented in Chapter 7 show the importance of enabling the use of remote resources. Offloading computation onto one single resource-richer remote node allows applications to speed-up their execution according to the performance difference of both devices; offloading it onto multiple remote resources allows the runtime to exploit higher degrees of parallelism and shorten even more the execution. Besides, since the local resources no longer compute, the energy consumption of the application falls drastically. In the presented example, the programming model reduces a one-day execution that drains the whole battery of the mobile device to a five-minute execution consuming less than 55 J.

Despite the benefits of exploiting remote resources, using the network incurs new concerns to handle by the runtime system since developers code applications without specifying any network interaction. Mobile devices are likely to experiment glitches on the network service due to handovers or long-lasting periods of isolation due to entering in out of signal areas. Hosting the distributed hash table only on the remote nodes allows them to keep executing tasks even if the master node is down. In the case of an eventual reconnection, both ends synchronize their progress; workers autonomy ensures, no matter how long the disruption lasts, that the loss of performance is as little as possible since worker nodes execute all the tasks as expected and the mobile device would get the result upon the reconnection.

On the other side, the autonomy of the mobile device is also important for returning a result even if the connection never re-establishes. In this case, the mobile has to run all the tasks assigned to its computing devices (higher priority) and the offloaded ones still pending to run. Often, a pending task needs a data value produced by a task offloaded onto the remote resources; and due to the network breakdown, the mobile device cannot fetch it. The only solution in these cases is to re-execute the offloaded task on the computing devices embedded on the mobile to re-generate the data value. In turn, this preceding task can require unaccessible values generated by other offloaded tasks; thus, the runtime gets into a backtracking procedure that may end up re-executing the whole application. By automatically ordering the transfer of strategic values, the Cloud Platform avoids the backtracking from going beyond the tasks generating them, and thus, it prevents the system from re-executing the whole application workflow.

The second problem that appears due to using the network is data leaks. The information contained on the mobile and used by the applications is likely to be privacy-sensitive. Transferring these data through insecure networks exposes it to external attackers eavesdropping the communication channel or trying to disguise as another node of the computing infrastructure. Given that the application developer is not in control of the data transfers, the runtime system is responsible for securing the communication channel. Chapter 8 describes the adaptation of the architecture of the runtime system to adopt external security solutions that provide communications with secrecy, integrity and authenticity (Contribution 4). Secrecy avoids that eavesdroppers can read the information sent through the network. Integrity denies the possibility that attackers modify messages to harm the execution. Mutual authentication allows both ends to verify the identity of the counterparty. For instance, in the case where the mobile device offloads the computation onto remote servers owned by an organization, the application user can make sure that it ships data to a node belonging to the organization, and worker processes can check that the source of a task submission or data transfer request is an authorized user of the system. For validating the viability of the solution, the prototype uses the Kerberos framework as the security service provider. To authenticate themselves, application users use Kerberos credentials, while worker nodes use Kerberos keytabs as host credentials. Since Kerberos supports federated identity management, several federated organizations could offer access to their computing devices to their members regardless their origin and provide them with a unique identity that grants them access to the whole infrastructure (Single Sign-on).

Generally, organizations offering resources where to offload the computation already have a deployed authentication infrastructure to control the access to their services. For avoiding a security vendor lock-in, the runtime system builds on the Generic Security Services API (GSSAPI). This interface abstracts the security framework user away from its actual API by providing a model where a client – GSS initiator – contacts a secure service – GSS service. At this point, both ends negotiate the security techniques – message encryption, signing and mutual authentication – necessary to secure the channel and the algorithms used to apply them. At the end of the negotiation, both parts can exchange messages securely applying the agreed algorithms to create the ciphertext to transfer before shipping it – wrapping – and unwrapping it upon its reception to recover the original message. Since some mechanisms require the whole plain/ciphertext to encrypt and decrypt it, the system splits the messages into fixed size tokens. The padding of the messages to match this size, the negotiation – strongly influenced by the network latency – and the processing of the messages add a significant overhead to the execution.

Although GSSAPI provides the runtime system with the ability to exchange the identities and messages using federated credentials – if supported by the used implementation, and Kerberos does –, it does not provide a generic mechanism for obtaining the credential. The described extension of the runtime requires the device to have the credential already. It can obtain the credential through another application that stores it on the file system, or the same application can provide the required mechanism. While the former option keeps the application agnostic to the authentication mechanism, the latter gathers all the functionality within a single application but binds it to a particular authentication mechanism.

As the infrastructure and the number of tasks grows, selecting the best resources becomes more complicated incurring significant overhead. Besides, forecast mispredictions when assigning the resource to run each task may lead the runtime system to decisions causing load imbalances. Hence, it is necessary to monitor constantly the workload assigned to each node for detecting and correcting these situations. Hosting all this processing is a workload not assumable by the mobile device. For this reason, Chapter 9 proposes offloading not only the computation onto the remote nodes but also the scheduling of the tasks (Contribution 5). With this solution, the overhead on the mobile device becomes negligible since it only needs to contact the remote endpoint of the scheduling system.

The scheduling system works with a two-level hierarchy. On the lower level, each node is represented by a *Node Scheduler* that manages the resources of the node and plans the execution of actions on them; on the higher level, an *Action Scheduler* coordinates the all the *Node Schedulers* to orchestrate the execution. When the mobile device submits a new task to the system, the *Action Scheduler* picks one of the nodes to execute the task, and the corresponding *Node Scheduler* decides on which resources and moment the task runs. Periodically, each *Node Scheduler* tries to optimize the execution plan of the tasks assigned to the corresponding node, and the *Action Scheduler* monitors the workload assigned to each resource and tries to balance it.

This solution not only benefits the mobile device but also to the owner of the remote resources since it can improve the control over the usage of the infrastructure. By configuring the scheduling system to follow different policies, the system can take scheduling decisions fostering a shorter execution time, a lower energy consumption or a reduction of the amount of money spent on the execution. Besides, a third component, the *Resource Optimizer* monitors the amount of resources available for the runtime and adapts the resource pool requesting new virtual instances or destroying the available ones so the application meets the defined temporal, energetic and monetary boundaries.

10.1 Future Work

Although the solution described in this dissertation tackles all the issues related to the development and execution of MCC application, it only establishes the foundations for handling these problems and further research can delve more deeply into all of the subjects to improve the results obtained on this thesis. This section suggests possible lines to continue and complement the already presented work.

Regarding the detection of dependencies, the programming model only detects those dependencies among data values passed as parameters on the invocation of the Core Element. Thus, if one of the arguments of the method, which no previous CE invocation generates or modifies, contains a reference to an object previously accessed by another task, the runtime system will not detect the dependency among both tasks. Improving the detection of nested dependencies would simplify even more the programmability of the model. Besides, supporting the detection of dependencies on collections of data values would allow the runtime system to apply techniques that improve the performance of the applications by reorganizing the execution of several tasks to follow map or reduce patterns. However, that would require additional annotations to allow the developer to hint which transformations the runtime should apply.

A second problem regarding the management of the dependencies is that the runtime considers that data values are created at the completion of the task that computes them. However, a task can compute the data value long before its completion; hence, runtime could release the dependencies on that value upon its generation so the consumer tasks can start earlier their execution. Furthermore, the dependency might not be on one single data value but on a collection of values constantly generated that consumer tasks can individually use as the producer task computes them. Enabling a stream-like dependency would allow the runtime system to overlap the execution of both tasks, producer and consumer, instead of running them sequentially.

Focusing on the runtime system, one aspect with room for improvement is the selection of the computing platform hosting a task execution. Currently, the system assigns the task upon its detection onto a platform according to the forecasts of three models and only changes this decision when the mobile device gets isolated from the remote workers. One the one hand, enabling a mechanism that monitors the pending workload assigned to each platform and reconsiders that decision would allow the runtime to amend those decisions made taking into account predictions differing from the real execution. On the other hand, more accurate models would reduce the decision to correct. The models currently implemented only consider the average of the measured time and energy required to run each Core Element on a computing device or remote node. Although the average gives an idea of the behavior of that task, the real execution can differ a lot from it. Including in the models other parameters that influence on the execution, such as the relation between the input size and the execution time or the number of CPU cores running other tasks when the system expects to run the analyzed task, would produce more realistic predictions.

Within the computing platforms, the scheduling policies are another research niche. Both platforms exploiting computing devices embedded in the mobile device requests the transfer of all the necessary input data as soon as the task becomes dependency-free. Then the CPU platform executes the tasks in a first come, first served basis according to the moment when all the input values are on the mobile phone and there are enough resources to host the execution, whereas the OpenCL platform leverages on the internal kernel scheduler implemented in the OpenCL library. Applications could better exploit the local computing devices if the runtime implemented more complex policies. For instance, these policies could order the data transfers giving priority to those transfers corresponding to values on which the earlier-to-run tasks operate; thus, the runtime would use the network bandwidth more efficiently and avoid that it becomes a bottleneck. Another important element to consider is the overheating prevention; these more complex policies could also decide the number of computing devices executing in parallel according to the impact of such mechanisms pursuing to maximize the throughput of the processor or reducing its energy consumption.

The computing platform with a bigger scope for improvement is the Cloud Platform. Regarding the task scheduling, the platform already has a more sophisticated system that allows not only to correct decisions previously taken but also dynamically adapt the amount of available resources. However, the policies leading these decisions – score comparison to select the node, initial scheduling on the *Node Scheduler*, priority of an action execution during the local optimization, order of donors-receivers and priority of the action to leave the node during the global optimization) – could also consider more parameters or include new objectives on their boundaries like limiting the power consumption of each node or the total price per hour of execution.

The policy determining the significant data values to transfer back to the mobile with the purpose of establishing checkpoints currently consists in partitioning the graph in fixed-size chunks according to the order of task detection. Other strategies to pick these values could be bringing back the results of those tasks that would require to re-run N tasks to re-compute it from the already selected checkpoints, or that require the re-execution of a sequence of N predecessors. Other approaches could even consider the time or the energy necessary to re-compute the preceding tasks instead of the number of predecessors.

Concerning the security of the network communications, the described solution splits the encrypted messages into fixed-size tokens. This size has a significant impact on the amount of bytes transferred and the timespan of the communication; and, currently, the application user defines this value statically as a configuration parameter. Enabling a mechanism that dynamically adapts the size of such tokens would improve the efficiency of the network communication. This mechanism could directly check the type of data to transfer to determine a token size; for instance, tokens would be short for internal commands, mid-size for transferring objects and larger for files. A complementary technique would be using a scaling token-size, start communications transferring short-size tokens and incrementally extend them. Regarding the security provider, Kerberos allows federations only through cross-realm authentication while other frameworks, like Moonshot, support full federation. Porting other security frameworks to Android and including them to the prototype would enhance the result of the project and foster its adoption by organizations with already deployed authentication infrastructures other than Kerberos.

Another limitation of the presented Cloud Platform is that tasks can run only on the cores of the CPU of the remote nodes. As demonstrated in Chapter 6, GPUs achieve shorter execution times and lower energy consumptions; thus, enabling the offloading of tasks onto the GPUs or other accelerators available on the remote nodes would improve the performance of the application.

To end up, this thesis considers the use of remote resources with a very high availability. Therefore, it does not acknowledge the possibility of a failure within one of the remote nodes that isolates the remote node from the infrastructure; if the mobile device remains connected to the network, it can always contact all the remote resources. For this reason, the name of the platform leveraging on remote resources is Cloud Platform. However, there exist other devices that occasionally are nearby the mobile device that could host part of the computation.

Exploiting resources that dynamically appear and disappear through the implementation of a new computing platform, the Fog Platform, opens a wide range of research possibilities. This platform should deal with the discovery of new nearby devices onto which the runtime can offload the computation to include it to the resource pool and detect when one of them is no longer reachable for not considering offloading task onto it. Another problem to tackle is the data distribution: when one node becomes unreachable, also do all the data values only contained in it. Transferring back all the outputs of every task executed on a fog device ensures the availability of these values; however, it would increase the network traffic of the mobile incurring a significant energy consumption and probably causing a new bottleneck for the runtime system. Replicating data values on other fog nodes distributes the overhead among all the nodes of the infrastructure. The policies deciding the locations where to replicate the data values should consider the trustfulness of each node. Likewise, the policies leading the scheduling of the tasks should also take into account the risk of losing the node while computing.

Bibliography

[1] MIT Kerberos Consortium.
URL `http://www.kerberos.org/software/index.html`.

[2] Art and dalvik - android open source project.
URL `https://source.android.com/devices/tech/dalvik/`.

[3] Aparapi: Api for data parallel java.
URL `http://code.google.com/p/aparapi`.

[4] Google App Engine.
URL `http://code.google.com/p/appengine-mapreduce/`.

[5] Microsoft Azure.
URL `http://microsoft.com/azure/`.

[6] arm bigLITTLE.
URL `https://developer.arm.com/technologies/big-little`.

[7] EnergyPlus Web Site.
URL `https://energyplus.net/,Dec.2015`.

[8] Apache Hadoop.
URL `http://hadoop.apache.org/`.

[9] jEPlus Web Site.
URL `http://www.jeplus.org/,Dec.2015`.

[10] OASIS Web Services Business Process Execution Language.
URL `http://www.oasis-open.org/committees/wsbpel/`.

[11] Globus Toolkit, 2017.
URL `http://toolkit.globus.org/toolkit/`.

[12] Gul A Agha.
Actors: A model of concurrent computation in distributed systems.
Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[13] Gabrielle Allen, Kelly Davis, Tom Goodale, Andrei Hutanu, Hartmut Kaiser, Thilo Kielmann, André Merzky, Rob Van Nieuwpoort, Alexander Reinefeld, Florian Schintke, Thorsten Schütt, E. D. Seidel, and And Brygg Ullmer.
The Grid application toolkit: Toward generic and easy application programming interfaces for the Grid.
In *Proceedings of the IEEE*, volume 93, pages 534–549, 2005.
doi: 10.1109/JPROC.2004.842755.

[14] George S. Almasi and Allan Gottlieb.
Highly Parallel Computing.
Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.
ISBN 0-8053-0177-1.

[15] Gene M Amdahl.
Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.
In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.
doi: 10.1145/1465482.1465560.
URL `http://doi.acm.org/10.1145/1465482.1465560`.

[16] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, and Others.
Above the clouds: A berkeley view of cloud computing.
2009.

[17] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A Yelick.
The Landscape of Parallel Computing Research: A View from Berkeley.
Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.
URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

[18] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier.
StarPU: a unified platform for task scheduling on heterogeneous multicore architectures.
Concurrency and Computation: Practice and Experience, 23(2):187–198, 2011.
ISSN 15320626.
doi: 10.1002/cpe.1631.
URL `http://onlinelibrary.wiley.com/doi/10.1002/cpe.1631/full`.

[19] Bernard J Baars.

A Cognitive Theory of Consciousness.
1988.
ISBN 0521301335.
URL `http://www.loc.gov/catdir/description/cam032/87020923.html`.

[20] Mark Bakery and Rajkumar Buyya.
Cluster computing at a glance.
High Performance Cluster Computing: Architectures and Systems, 1:3–47, 1999.

[21] Rajesh Krishna Balan, Mahadev Satyanarayanan, So Young Park, and Tadashi Okoshi.
Tactics-based remote execution for mobile computing.
Proceedings of the 1st international conference on Mobile systems applications and services (MobiSys '03), pages 273–286, 2003.
doi: 10.1145/1066116.1066125.
URL `http://portal.acm.org/citation.cfm?doid=1066116.1066125`.

[22] Miquel Barceló.
Una historia de la informática, volume 125.
Editorial UOC, 2010.

[23] Arthur J. Bernstein.
Analysis of Programs for Parallel Processing.
IEEE Transactions on Electronic Computers, EC-15(5):306–307, 1966.
ISSN 0367-7508.
doi: 10.1109/PGEC.1966.264565.

[24] Dan Bornstein.
Dalvik vm internals.
In *Google I/O developer conference*, volume 23, pages 17–30, 2008.

[25] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J. Dongarra.
PaRSEC: Exploiting heterogeneity to enhance scalability.
Computing in Science and Engineering, 15(6):36–45, 2013.
ISSN 15219615.
doi: 10.1109/MCSE.2013.98.

[26] Rajkumar Buyya, Rajkumar Buyya, Chee Shin Yeo, Chee Shin Yeo, Srikumar Venugopal, Srikumar Venugopal, James Broberg, James Broberg, Ivona Brandic, and Ivona Brandic.
Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility.

Future Generation Computer Systems, 25(June 2009):17, 2009.
ISSN 0167-739.
doi: 10.1016/j.future.2008.12.001.
URL `http://portal.acm.org/citation.cfm?id=1528937.1529211`.

[27] John Canny.
A Computational Approach to Edge Detection.
IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-8(6):679–698, 1986.
ISSN 01628828.
doi: 10.1109/TPAMI.1986.4767851.

[28] Byung-Gon Chun and Petros Maniatis.
Augmented Smartphone Applications Through Clone Cloud Execution.
In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, page 8, Berkeley, CA, USA, 2009. USENIX Association.
URL `http://dl.acm.org/citation.cfm?id=1855568.1855576`.

[29] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti.
CloneCloud: Elastic Execution Between Mobile Device and Cloud.
In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.
ISBN 978-1-4503-0634-8.
doi: 10.1145/1966445.1966473.
URL `http://doi.acm.org/10.1145/1966445.1966473`.

[30] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl.
MAUI: Making Smartphones Last Longer with Code Offload.
In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.
ISBN 978-1-60558-985-5.
doi: 10.1145/1814433.1814441.
URL `http://doi.acm.org/10.1145/1814433.1814441`.

[31] Leonardo Dagum and Ramesh Menon.
OpenMP: An Industry-Standard API for Shared-Memory Programming.
IEEE Computation Science & Engineering, 5(1):46–55, 1998.
ISSN 1070-9924.
doi: 10.1109/99.660313.
URL `http://ieeexplore.ieee.org/document/660313/`.

[32] Jeffrey Dean and Sanjay Ghemawat.
MapReduce: Simplified Data Processing on Large Clusters.
In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, page 10, Berkeley, CA, USA, 2004. USENIX Association.
URL `http://dl.acm.org/citation.cfm?id=1251254.1251264`.

[33] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, Anastasia Laity, Joseph C Jacob, and Daniel S Katz.
Pegasus: A Framework for Mapping Complex Scientific Workflows Onto Distributed Systems.
Sci. Program., 13(3):219–237, 2005.
ISSN 1058-9244.
URL `http://dl.acm.org/citation.cfm?id=1239649.1239653`.

[34] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira Da Silva, Miron Livny, and Kent Wenger.
Pegasus, a workflow management system for science automation.
Future Generation Computer Systems, 46:17–35, 2015.
ISSN 0167739X.
doi: 10.1016/j.future.2014.10.008.

[35] Karim Djemame, Richard Kavanagh, Django Armstrong, Francesc Lordan, Jorge Ejarque, Mario Macias, Raül Sirvent, Jordi Guitart, and Rosa M. Badia.
Energy efficiency support through intra-layer cloud stack adaptation, volume 10382 LNCS. 2017.
ISBN 9783319619194.
doi: 10.1007/978-3-319-61920-0_10.

[36] Jack J Dongarra, Steve W Otto, Marc Snir, and David Walker.
An introduction to the mpi standard.
Communications of the ACM, page 18, 1995.

[37] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas.
OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES.
Parallel Processing Letters, 21(2):173–193, 2011.
ISSN 0129-6264.
doi: 10.1142/S0129626411000151.

[38] Niroshinie Fernando, Seng W. Loke, and Wenny Rahayu.
Mobile cloud computing: A survey, 2013.
ISSN 0167739X.

[39] Ana Juan Ferrer, Francisco Hernández, Johan Tordsson, Erik Elmroth, Ahmed Ali-Eldin, Csilla Zsigri, Raül Sirvent, Jordi Guitart, Rosa M. Badia, Karim Djemame, Wolfgang Ziegler, Theo Dimitrakos, Srijith K. Nair, George Kousiouris, Kleopatra Konstanteli, Theodora Varvarigou, Benoit Hudzia, Alexander Kipp, Stefan Wesner, Marcelo Corrales, Nikolaus Forgó, Tabassum Sharif, and Craig Sheridan.
OPTIMIS: A holistic approach to cloud service provisioning.
In *Future Generation Computer Systems*, volume 28, pages 66–77, 2012.
ISBN 0167-739X.
doi: 10.1016/j.future.2011.05.022.

[40] Rico Fischer and Franziska Plessow.
Efficient multitasking: Parallel versus serial processing of multiple tasks, 2015.
ISSN 16641078.

[41] Jason. Flinn, SoYoung Park, and Mahadev Satyanarayanan.
Balancing performance, energy, and quality in pervasive computing.
Proceedings 22nd International Conference on Distributed Computing Systems, pages 217–226, 2002.
ISSN 1063-6927.
doi: 10.1109/ICDCS.2002.1022259.

[42] Ian Foster.
Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering, volume 5.
1995.
ISBN 0201575949.
doi: 10.1109/MCC.1997.588301.
URL `http://portal.acm.org/citation.cfm?id=527029`.

[43] Ian Foster and Others.
The anatomy of the Grid.
Berman et al.[2], pages 171–197, 2003.

[44] Ioana Giurgiu, Oriana Riva, Dejan Juric, Ivan Krivulev, and Gustavo Alonso.
Calling the Cloud: Enabling Mobile Phones As Interfaces to Cloud Applications.
In *Proceedings of the ACM/IFIP/USENIX 10th International Conference on Middleware*, Middleware'09, pages 83–102, Berlin, Heidelberg, 2009. Springer-Verlag.

ISBN 3-642-10444-4, 978-3-642-10444-2.
URL `http://dl.acm.org/citation.cfm?id=1813355.1813362`.

[45] Iñigo Goiri, Kien Le, Jordi Guitart, Jordi Torres, and Ricardo Bianchini.
Intelligent placement of datacenters for internet services.
In *2011 31st International Conference on Distributed Computing Systems*, pages 131–142, June 2011.
doi: 10.1109/ICDCS.2011.19.

[46] Tom Goodale, Shantenu Jha, Harmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John Shalf.
SAGA: A Simple API for Grid applications, High-Level Application Programming on the Grid.
Computational Methods in Science and Technology, 12(1):7–20, 2006.

[47] Ronald L Graham.
Bounds for certain multiprocessing anomalies.
Bell System Technical Journal, 45(9):1563–1581, 1966.

[48] Hervé Guihot.
RenderScript.
Pro Android Apps Performance Optimization, pages 231–263, 2012.

[49] Munish K Gupta.
Akka Essentials.
Packt Publishing, 2012.
ISBN 978-1849518284.
URL `http://books.google.com/books?hl=en&lr=&id=TkycGHohXmEC&oi=fnd&pg=PT10&dq=Akka+Essentials&ots=7q1R6bcaMT&sig=ueaS98k2pkMrapIKCILQW-IkLYE`.

[50] John L. Gustafson.
Reevaluating Amdahl's law.
Communications of the ACM, 31(5):532–533, 1988.
ISSN 00010782.
doi: 10.1145/42411.42415.
URL `http://portal.acm.org/citation.cfm?doid=42411.42415`.

[51] J L Hennessy, D A Patterson, and D Goldberg.
Computer Architecture: A Quantitative Approach.
Morgan Kaufmann, 2002.

[52] Carl Hewitt, Peter Bishop, and Richard Steiger.

A Universal Modular ACTOR Formalism for Artificial Intelligence.
In *Proceeding IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, 1973.
ISBN 9781450336697.
doi: 10.1145/359545.359563.

[53] Dijiang Huang, Xinwen Zhang, Myong Kang, and Jim Luo.
MobiCloud: Building Secure Cloud Framework for Mobile Computing and Communication.
In *2010 Fifth IEEE International Symposium on Service Oriented System Engineering*, pages 27–34, 2010.
ISBN 978-1-4244-7327-4.
doi: 10.1109/SOSE.2010.20.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5569935.

[54] Marty Humphrey and Mary Thompson.
Security Implications of Typical Grid Computing Usage Scenarios, 2000.
URL https://www.ogf.org/documents/GFD.12.pdf.

[55] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, Dennis Fetterly, and Others.
Dryad: Distributed Data-parallel Programs from Sequential Building Blocks.
In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, volume 41 of *EuroSys '07*, pages 59–72, New York, NY, USA, 2007. ACM, ACM.
ISBN 978-1-59593-636-3.
doi: 10.1145/1272996.1273005.
URL http://doi.acm.org/10.1145/1272996.1273005.

[56] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri E Bal.
Cuckoo: A Computation Offloading Framework for Smartphones.
In Martin L Gris and Guang Yang 0001, editors, *MobiCASE*, volume 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 59–79. Springer, 2010.
ISBN 978-3-642-29335-1.
URL http://dblp.uni-trier.de/db/conf/mobicase/mobicase2010.html#KempPKB10.

[57] Abdul Nasir Khan, M L Mat Kiah, Samee U Khan, and Sajjad a Madani.
Towards secure mobile cloud computing: A survey.
Future Gener. Comput. Syst., 29(5):1278–1299, 2013.
ISSN 0167739X.
doi: 10.1016/j.future.2012.08.003.
URL http://dx.doi.org/10.1016/j.future.2012.08.003.

[58] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang.
Unleashing the Power of Mobile Cloud Computing using ThinkAir.
CoRR, abs/1105.3, 2011.
URL `http://arxiv.org/abs/1105.3232`.

[59] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang.
ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading.
In Albert G Greenberg and Kazem Sohraby, editors, *INFOCOM*, pages 945–953. IEEE, 2012.
ISBN 978-1-4673-0773-4.
URL `http://dblp.uni-trier.de/db/conf/infocom/infocom2012.html#KostaAHMZ12`.

[60] Karthik Kumar and Yung Hsiang Lu.
Cloud computing for mobile users: Can offloading computation save energy?
Computer, 43(4):51–56, 2010.
ISSN 00189162.
doi: 10.1109/MC.2010.98.

[61] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner.
Gradient-based learning applied to document recognition.
Proceedings of the IEEE, 86(11):2278–2323, 1998.
ISSN 00189219.
doi: 10.1109/5.726791.

[62] Yann LeCun, Corinna Cortes, and Christopher Burges.
THE MNIST DATABASE of handwritten digits.
The Courant Institute of Mathematical Sciences, pages 1–10, 1998.
URL `http://yann.lecun.com/exdb/mnist/`.

[63] Rob Lineback.
Cellphone IC Sales Will Top Total Personal Computing in 2017.
URL `http://www.icinsights.com/data/articles/documents/987.pdf`.

[64] John Linn.
Generic Security Service Application Program Interface Version 2, Update 1.
RFC 2743, RFC Editor, 2000.
URL `https://tools.ietf.org/html/rfc2743`.

[65] Francesc Lordan and Rosa M. Badia.
COMPSs-Mobile: Parallel Programming for Mobile-Cloud Computing.

In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 497–500, may 2016.
doi: 10.1109/CCGrid.2016.16.

[66] Francesc Lordan and Rosa M Badia.
COMPSs-Mobile: Parallel Programming for Mobile Cloud Computing.
Journal of Grid Computing, 15(3):357–378, sep 2017.
ISSN 1572-9184.
doi: 10.1007/s10723-017-9409-z.
URL `https://doi.org/10.1007/s10723-017-9409-z`.

[67] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Álvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M Badia.
ServiceSs: An Interoperable Programming Framework for the Cloud.
Journal of Grid Computing, 12(1):67–91, 2014.
ISSN 1570-7873.
doi: 10.1007/s10723-013-9272-5.
URL `http://dx.doi.org/10.1007/s10723-013-9272-5`.

[68] Francesc Lordan, Jorge Ejarque, Raül Sirvent, and Rosa M Badia.
Energy-Aware Programming Model for Distributed Infrastructures.
In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 413–417, feb 2016.
doi: 10.1109/PDP.2016.39.

[69] Francesc Lordan, Jens Jensen, and Rosa M. Badia.
Towards Mobile Cloud Computing with Single Sign-on Access.
Journal of Grid Computing, 2017.
ISSN 15729184.
doi: 10.1007/s10723-017-9413-3.

[70] Francesc Lordan, Rosa M Badia, and Wen-Mei Hwu.
Enabling GPU Support for the COMPSs-Mobile Framework.
In Sunita Chandrasekaran and Guido Juckeland, editors, *Accelerator Programming Using Directives*, pages 83–102, Cham, 2018. Springer International Publishing.
ISBN 978-3-319-74896-2.

[71] Francesc Lordan, Daniele Lezzi, Jorge Ejarque, and Rosa M Badia.
An Architecture for Programming Distributed Applications on Fog to Cloud Systems.
In Dora B Heras and Luc Bougé, editors, *Euro-Par 2017: Parallel Processing Workshops*, pages 325–337, Cham, 2018. Springer International Publishing.
ISBN 978-3-319-75178-8.

[72] Eugene E Marinelli.
Hyrax : Cloud Computing on Mobile Devices using MapReduce.
Science, 0389(September):1–123, 2009.
URL `http://www.contrib.andrew.cmu.edu/~emarinel/masters_thesis/emarinel_ms_thesis.pdf`.

[73] Paul E McKenney.
Is parallel programming hard, and, if so, what can you do about it?
Linux Technology Center, IBM Beaverton, 2011.

[74] Paolo Missier, Stian Soiland-Reyes, Stuart Owen, Wei Tan, Aleksandra Nenadic, Ian Dunlop, Alan Williams, Thomas Oinn, and Carole Goble.
Taverna, reloaded.
In M Gertz, T Hey, and B Ludaescher, editors, *SSDBM 2010*, Heidelberg, Germany, 2010.
URL `http://www.taverna.org.uk/pages/wp-content/uploads/2010/04/T2Architecture.pdf`.

[75] Raffaele Montella, Sokol Kosta, David Oro, Javier Vera, Carles Fernández, Carlo Palmieri, Diana Di Luccio, Giulio Giunta, Marco Lapegna, and Giuliano Laccetti.
Accelerating Linux and Android applications on low-power devices through remote GPGPU offloading.
In *Concurrency Computation*, volume 29, 2017.
doi: 10.1002/cpe.4286.

[76] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro.
Jolie: a Java orchestration language interpreter engine.
Electronic Notes in Theoretical Computer Science, 181:19–33, 2007.

[77] Jonathan J. Nassi and Edward M. Callaway.
Parallel processing strategies of the primate visual system, 2009.
ISSN 1471003X.

[78] Piotr Nawrocki, Bartłomiej Śnieżyński, and Jakub Czyżewski.
Learning Agent for a Service-Oriented Context-Aware Recommender System in Heterogeneous Environment.
COMPUTING AND INFORMATICS, 35(5):1005–1026, 2017.

[79] Nvidia.
Compute unified device architecture programming guide.
2007.

[80] JDK Open.

Project sumatra.
URL `http://openjdk.java.net/projects/sumatra/`.

[81] Khronos Opencl.
OpenCL Specification.
ReVision, pages 1–385, 2009.
doi: 10.1016/j.actamat.2006.08.044.
URL `http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:OpenCL+Specification#0`.

[82] Andreas Pashalidis and Chris J. Mitchell.
A taxonomy of single sign-on systems.
In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2727 LNCS, pages 249–264, 2003.
ISBN 9783540405153.
doi: 10.1007/3-540-45067-X_22.

[83] Hillel Pratt, Naomi Bleich, and Nomi Mittelman.
Spatio-temporal distribution of brain activity associated with audio-visually congruent and incongruent speech and the McGurk Effect.
Brain and Behavior, 5(11), 2015.
ISSN 21623279.
doi: 10.1002/brb3.407.

[84] Ralf Ratering and Hans-Christian Hoppe.
Accelerating opencl applications by utilizing a virtual opencl device as interface to compute clouds, 2011.
URL `https://www.google.ch/patents/US20110161495`.

[85] Jan S Rellermeyer, Oriana Riva, and Gustavo Alonso.
AlfredO: An Architecture for Flexible Interaction with Electronic Devices.
In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 22–41, New York, NY, USA, 2008. Springer-Verlag New York, Inc.
ISBN 3-540-89855-7.
URL `http://dl.acm.org/citation.cfm?id=1496950.1496953`.

[86] Eric Rescorla.
HTTP Over TLS.
Technical Report 2818, Internet Engineering Task Force, may 2000.
URL `http://www.ietf.org/rfc/rfc2818.txt`.

[87] Ronald L Rivest, Len Adleman, and Michael L Dertouzos.
On data banks and privacy homomorphisms.
Foundations of secure computation, 4(11):169–180, 1978.

[88] Mahadev Satyanarayanan, P Bahl, R Caceres, and N Davies.
The Case for VM-Based Cloudlets in Mobile Computing.
Pervasive Computing, IEEE, 8(4):14–23, oct 2009.
ISSN 1536-1268.
doi: 10.1109/MPRV.2009.82.

[89] Federico Silla, Javier Prades, Sergio Iserte, and Carlos Reano.
Remote GPU Virtualization: Is It Useful?
In *High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), 2016 2nd IEEE International Workshop on*, pages 41–48. IEEE, 2016.

[90] David B. Skillicorn and Domenico Talia.
Models and languages for parallel computation.
ACM Computing Surveys, 30(2):123–169, 1998.
ISSN 03600300.
doi: 10.1145/280277.280278.
URL `http://doi.acm.org/10.1145/280277.280278`.

[91] Enric Tejedor and Rosa M Badia.
COMP Superscalar: Bringing grid superscalar and gcm together.
In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 185–193. IEEE, 2008.

[92] The MPI Forum.
MPI: A Message Passing Interface.
In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing - Supercomputing '93*, pages 878–883, 1993.
ISBN 0818643404.
doi: 10.1145/169627.169855.
URL `http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf%5Cnhttp://portal.acm.org/citation.cfm?doid=169627.169855`.

[93] James Thornton.
Parallel operation in the control data 6600.
AFIPS '64 (Fall, part II): Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems, 1964.
URL `http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=pubmed&cmd=Retrieve&dopt=AbstractPlus&list_uids=1464039.1464045`.

[94] Robert M. Tomasulo.
An Efficient Algorithm for Exploiting Multiple Arithmetic Units.
IBM Journal of Research and Development, 11(1):25–33, 1967.
ISSN 0018-8646.
doi: 10.1147/rd.111.0025.

[95] Steven Tuecke, Von Welch, Doug Engert, Laura Pearlman, and Mary Thompson.
Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile.
Technical Report 3820, Internet Engineering Task Force, 2004.
URL `http://www.ietf.org/rfc/rfc3820.txt`.

[96] Mayank D Upadhyay and Seema Malkani.
Generic security service api version 2: Java bindings update.
RFC 5653, RFC Editor, August 2009.

[97] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal.
User-friendly and reliable grid computing based on imperfect middleware.
In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC '07*, page 1, 2007.
ISBN 9781595937643.
doi: 10.1145/1362622.1362668.
URL `http://portal.acm.org/citation.cfm?doid=1362622.1362668`.

[98] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner.
A break in the clouds: towards a cloud definition.
ACM SIGCOMM Computer Communication Review, 39(1):50–55, 2008.

[99] Christian Vecchiola, Xingchen Chu, and Rajkumar Buyya.
Aneka: A Software Platform for .NET-based Cloud Computing.
CoRR, abs/0907.4, 2009.

[100] Robert Virding, Claes Wikström, and Mike Williams.
Concurrent Programming in ERLANG (2Nd Ed.).
Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.
ISBN 0-13-508301-X.

[101] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster.
Swift: A Language for Distributed Parallel Scripting.
Parallel Comput., 37(9):633–652, 2011.
ISSN 0167-8191.
doi: 10.1016/j.parco.2011.05.005.

URL http://dx.doi.org/10.1016/j.parco.2011.05.005.

[102] Nico Williams, Leif Johansson, Sam Hartman, and Simon Josefsson.
Generic Security Service Application Programming Interface Naming Extensions.
Technical report, 2012.

[103] Larry Zhu, Paul Leach, Karthik Jaganathan, and Wyllys Ingersoll.
The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism.
RFC 4178, RFC Editor, 2005.
URL https://tools.ietf.org/html/rfc4178.