Self-Adaptive Solutions for Managing Performance and Power Consumption of Parallel Applications

Daniele De Sensi

Supervisor
Prof. Marco Danelutto

Supervisor
Dr. Massimo Torquati
To my family
Power consumption management in computing systems is gaining an increasing attention due to its environmental and economic impact. On one side, it has been estimated that \(CO_2\) emissions of U.S. data centers during 2010 were on par with those of an entire country like Argentina. On the other side, the cost of the energy required to keep the system running is quickly going to overcome the cost of the physical system itself. Despite the improvements in the hardware technology, a proper management of the applications running on top of that hardware is still of paramount importance, since estimations show that the power demand of datacenter will rapidly grow 10% - 12% per year. Power consumption is an important issue even on smaller devices. For example, on mobile devices, an efficient power management leads to a longer battery life and to a better user experience. Moreover, a better battery management would enable the diffusion of more complex and richer applications.

In this thesis, we focus on scenarios where the user wants to express specific Quality of Service (QoS) requirements on his application, regarding performance and power consumption. Requirements can be enforced by using self-adaptive algorithms, to dynamically and automatically adapt the number of resources allocated to the application (i.e., its configuration). To select a proper configuration, most existing approaches rely either on heuristics or on machine learning models. While heuristics may work fine in general, they are usually less accurate than machine learning solutions. On the other hand, despite being more accurate than heuristics, machine learning techniques typically require extensive training phases to be performed offline, before the application starts its execution. Moreover, machine learning techniques may not be able to find a suitable configuration for applications with characteristics different from those of the applications used to train the models.

In this thesis we try to close the gap between these two approaches, by proposing an online learning algorithm which, like the heuristics, will not rely on any training phase to be performed offline and, at the same time, can achieve the same accuracy of complex offline machine learning approaches. Since a
ABSTRACT

simulation would not capture all the runtime related overheads introduced by such algorithms, we designed and implemented a full software stack (released as open source), ranging from the management of hardware control mechanisms to parallel applications to be used as benchmarks. This led to the design of Nornir, a framework to be used by application users to enforce explicit performance and power consumption requirements on their applications. Nornir can be customized by algorithms designers to implement new self-adaptive strategies easily and to compare them with the state of the art algorithms already provided by the framework. Eventually, we will show that if the application has some clear and well-defined structure, it would be possible for the algorithms to select better configurations regarding power/performance tradeoffs.
# Contents

Abstract

Contents

## 1 Introduction

### 1.1 The Power Efficiency Challenge

### 1.2 Facing the Challenge

### 1.3 Thesis Main Contributions

#### 1.3.1 Self-Adaptive and Power-Aware Algorithms

#### 1.3.2 Parallel Patterns and Concurrency Throttling

#### 1.3.3 A Customizable Framework for Self-Adaptive Applications

### 1.4 Thesis Structure

### 1.5 List of Publications

#### 1.5.1 Articles in Peer-Reviewed Journals

#### 1.5.2 Articles in Peer-Reviewed Conferences, Workshops and Symposiums

## 2 Background and State of the Art

### 2.1 Background

#### 2.1.1 Power Management

#### 2.1.2 Self-Adaptive Algorithms

#### 2.1.3 Parallel Patterns

### 2.2 State of the Art

#### 2.2.1 Power Management

#### 2.2.2 Self-Adaptive Algorithms

#### 2.2.3 Self-Adaptive and Power-Aware Frameworks
II Self-Adaptive Algorithms and Evaluation

3 Self-Adaptive Algorithms

3.1 Heuristic
  3.1.1 Performance Prediction
  3.1.2 Power Consumption Estimation
  3.1.3 Algorithm

3.2 Online Learning Algorithm
  3.2.1 Stationary behaviour
  3.2.2 The prediction algorithm
  3.2.3 Optimal Configuration Re-evaluation
  3.2.4 Streaming Applications

3.3 Summary

4 Evaluation

4.1 Test Environment

4.2 Evaluation Methodology

4.3 Online Learning Algorithm Alternatives
  4.3.1 Performance Prediction Models
  4.3.2 Threads Placement Impact

4.4 Comparison with State of the Art Strategies
  4.4.1 Requirements Violations
  4.4.2 Training Steps and Training Time
  4.4.3 Suboptimality

4.5 Summary

5 A Concurrency Control Algorithm

5.1 Problem and Background

5.2 Algorithm Design
  5.2.1 Base Mechanisms
  5.2.2 Power-Aware Data Pipelining
  5.2.3 Generic Streaming Graphs

5.3 Experimental Results
  5.3.1 Protocol identification application
  5.3.2 Malware detection application

5.4 Summary

III Parallel Patterns Based Application

6 On the Relevance of Parallel Design Patterns
## CONTENTS

6.1 On Concurrency Throttling ........................................... 109  
  6.1.1 Programming effort ........................................... 111  
  6.1.2 Overhead ....................................................... 112  
  6.1.3 Achievable Performance ...................................... 112  
  6.1.4 Concurrency Throttling Requirements ....................... 113  
  6.1.5 Task-Based Models .......................................... 117  
6.2 Common Patterns .................................................. 118  
6.3 Examples of Parallel Patterns-Based Code ...................... 122  
6.4 Summary ........................................................... 125  

7 A Benchmark for Parallel Patterns-Based Applications 127  
  7.1 Design of the P³ARSEC Benchmark Suite ....................... 127  
  7.2 Evaluation ....................................................... 134  
    7.2.1 Programming Effort ...................................... 135  
    7.2.2 Performance Evaluation ................................ 138  
  7.3 Summary ........................................................ 144  

IV Framework Design .................................................. 145  

8 Using the Framework .................................................. 147  
  8.1 Black-Box Interaction .......................................... 150  
  8.2 Instrumentation ................................................. 151  
  8.3 Interaction with Supported Runtimes ......................... 156  
  8.4 Nornir Programming Framework ................................ 160  
    8.4.1 Threads-Based, Parallel Patterns Interface .......... 160  
    8.4.2 Tasks-Based Interface ................................ 164  
  8.5 Summary ........................................................ 176  

9 Customizing and Extending the Framework ......................... 179  
  9.1 Monitor .......................................................... 179  
  9.2 Execute .......................................................... 181  
    9.2.1 Mammut Design ........................................... 182  
  9.3 Analyze and Plan ................................................ 188  
  9.4 Summary ........................................................ 190  

V Conclusions and Future Research Directions .................... 193  

10 Conclusions .......................................................... 195  

VI Appendices ........................................................... 199  

A Heuristic Implementation in Nornir ................................ 201  

B P³ARSEC Speedups ................................................... 205
Part I

Introduction
Chapter 1

AIM AND SCOPE

Power consumption management is becoming one of the main factors to be considered when designing a new computing system. On large scale systems, power consumption has a direct impact on the datacenter density (i.e. how many servers to allocate per area unit) and, in turn, on the maximum performance that can be delivered. At the other end of the spectrum, power consumption is a major issue also for mobile devices, since battery life is often the main element to be considered when designing an application for a mobile device. For these reasons, different solutions have been designed to reduce or to control the power consumption of computing systems. We will describe these issues more in detail in Section 1.1.

In this thesis we focus on solutions which can be used to explicitly control the performance and power consumption of applications, as we will discuss in Section 1.2. In general, higher performance implies a higher power consumption and the best trade-off usually depends on user preferences. Finding an optimal operating point (a configuration) so that both performance and power consumption are compliant with the user requirements is a relevant problem, and many solutions have been proposed and are currently adopted to address this scenario. However, predicting how performance and power consumption change in different configurations is a hard problem and existing approaches have still a limited applicability. Indeed, most solutions rely on machine learning techniques, where a model is trained by using a large dataset of performance and power consumption data collected over a wide set of applications and for different configurations. However, these solutions share the same issues that affect machine learning approaches in general, i.e. they could not manage applications which were not represented in the training set. Moreover, from a practical perspective, building such algorithms is still a cumbersome task, due to low-level details related to interactions with the underlying system and with the applications. As a consequence, the programmer may spend more time trying to solve these low-level issues rather than focusing on
the algorithm logic. As we will discuss more in detail in Section 1.3, one of the main contributions of this dissertation is addressing these limitations.

Eventually, in Section 1.4 we outline the structure of this document and in Section 1.5 we list the papers we published which cover different parts of this thesis.

1.1 The Power Efficiency Challenge

Power consumption management is becoming a critical factor in designing applications and computing systems.

On mobile devices, having an efficient power management translates directly into how long the battery lasts while using the device. This problem is exacerbated on mobile phones due to the convergence of multiple functionality on a single device. A modern smartphone/tablet is often used as a phone, a music player, a camera, a portable gaming console, etc... Indeed, the highest ranked metric in mobile devices evaluation is usually the battery life and improving it will lead to even increased functionality and richer applications. Beside smartphones, battery life is also crucial on sensors networks, since the sensors may often be locate in remote or inaccessible locations and an extended battery life would decrease the maintenance cost for such devices.

Power management is also an issue for tethered devices (connected to a power supply). For example, in current datacenters, the energy cost is quickly going to overcome the cost of the physical system itself [1, 2]. Increased power consumption also leads to more complex power supplies that in turns add costs. A strictly related problem is the waste heat generated by power dissipation; consequently, when talking about power management we also include the heat management system. Indeed, studies concerning hardware components failures have shown that operating electronics and a higher temperature than their operational range can lead to significant degradation of reliability, with a 50% increased chance of server failure for each 10°C increase over 20°C. To prevent the heat from damaging the system’s components, increasingly complex thermal management solutions are adopted, adding one dollar of cost for each dollar spent on electricity [3]. To mitigate this issue, some companies are exploring “creative” solutions, like building datacenters in Iceland and other Nordic countries due to the cooler climate, that helps in lowering the temperature inside the datacenter\(^1\). In some cases, excess heat is captured from equipment inside the facility and conducted it into the district heating system to help warm homes in the neighboring community\(^2\). More recently, data centers’ computing nodes are being used both as computational nodes and as radiators\(^3\). By locating the nodes inside private houses, the heat produced by the computation is used for home heating purposes.

\(^1\)http://www.icelanddatacenter.com/

\(^2\)https://www.apple.com/pr/library/2015/02/23Apple-to-Invest-1-7-Billion-in-New-European-Data-Centres.html

\(^3\)http://www.nerdalize.com/
1.2. FACING THE CHALLENGE

More importantly, beside economic considerations, power consumption has a considerable impact on the environment, since during 2010 the CO$_2$ emissions of U.S. data centers were on par with those of an entire country like Argentina or Netherlands [4] (and close to the CO$_2$ emissions produced by the worldwide airline industry).

The importance of power management techniques is going to increase in the future rapidly. On mobile devices, the increase in performance requirements due to new functionality will overcome the advances in battery capacity. The problem may be partially addressed through new battery technologies. However, a proper design of power-efficient systems and applications will still be of primary importance. Concerning tethered devices, in 2010, the energy consumed in U.S. by data centers reached the 3% of the overall energy production [5], and this power demand is estimated rapidly growing 10% – 12% a year [4, 6]. If we also add the steadily increasing costs for electricity reported by U.S. Department of Energy$^4$, we can clearly see the scale of the problem. Moreover, increased server density in data centers will quickly hit the physical limits of current air-cooled solutions and despite new cooling technologies are being investigated by the research community, it will still be important to decrease the generated heat in the first place.

1.2 Facing the Challenge

To address these problems, different research communities proposed in recent years many solutions at various levels of the computing stack, ranging from hardware level solutions to virtualization and cloud computing. An in-depth description of these techniques will be done in Chapter 2. However, to better understand the target and the contributions of this thesis, we report here the main concepts.

In this thesis, we would like to address scenarios where the user wants to express some Quality of Service (QoS) requirements on his application, regarding performance and power consumption. There are different reasons why an application user would need QoS guarantees on his application. We here highlight some of them:

**Power Capping** To avoid possible electric surges, data center operators have traditionally over-provisioned data center power, considering a worst-case power consumption [7]. Albeit this ensures reliability with high confidence, it is wasteful in terms of power infrastructure utilization. This leads in turn to an increase in the capital cost, since every megawatt of power capacity can cost around 10 to 20 million US Dollars per year [8, 9]. Moreover, power over-provisioning is particularly inefficient since power is frequently the bottleneck resource limiting the number of servers that a data center can house, thus limiting the maximum performance that the data center can provide.

$^4$http://www.eia.doe.gov/
CHAPTER 1. AIM AND SCOPE

To improve efficiency, over-subscription of data center power is recently becoming more common [7, 10]. Accordingly, the data center power demand is intentionally allowed to exceed the power supply, under the assumption that correlated spikes in servers’ power consumption are infrequent. However, this exposes data centers to the risk of power outages, caused by unpredictable power spikes (e.g. due to an increase in the power consumption of more servers at the same time). Such an event would have catastrophic effects, since it would lead to degradation in the final user experience or to service outages. Moreover, an outage in a data center could cause a redistribution of the workload to other data centers, increasing their power consumption and possibly leading to a cascading power failure event.

For these reasons, to achieve power safety and to avoid having under-utilized power provisioning, power capping techniques have been recently proposed [11, 12, 13]. These techniques monitor the data center power consumption and, when it gets close to the available capacity, they request server to reduce their power consumption.

Performance Requirements In other cases, the user may not need his application to run at maximum performance. Nevertheless, he would still like to have some minimum performance guarantees [14, 15, 16, 17]. For example, suppose to have a data backup application that, each night, collects and compresses the data produced by the employees of a company during the day and sends the compressed data to remote storage. The user could require the application to complete this process in a 12-hour time frame. Even if the application could probably finish the process in a shorter time frame, this would not necessarily have an advantage from the user perspective. On the other hand, by slowing it down it is possible to reduce its power consumption and to enforce user requirements.

Of course, this scenario also includes the case when the user just wants his application to run at maximum performance.

Of course, in many cases these scenarios can be mixed, and we could have situations where the user needs to express both power consumption and performance requirements for the application. Moreover, other requirements are possible as well, like economic cost, energy-delay product (EDP) and others. We do not consider them explicitly since they can often be derived from the previous ones. Despite we are assuming that these requirements are specified by application users, this may not always be the case. For example, if the user is executing his application on a public datacenter, the requirements may be decided by a negotiation between the user and the datacenter operator, as we will describe in Section 2.1.2.

A possible way to enforce these requirements is to rely on self-adaptive, software-level solutions. We use the definition of self-adaptive software reported in [18]:
1.3. THESIS MAIN CONTRIBUTIONS

“Self-adaptive software is capable of evaluating and changing its own behavior, whenever the evaluation shows that the software is not accomplishing what it was intended to do, or when better functionality or performance may be possible.”

In this thesis, we focus on self-adaptive software which is also power-aware, i.e. it would change its behavior when it detects that its performance and power consumption do not comply with the user requirements. Self-adaptive and power-aware approaches often operate on control knobs, to find the correct amount of resources to allocate to the application to comply with the requirements specified by the user. For example, this may imply dynamically changing the number of cores used by the application, scaling down their frequency or acting on other software or hardware mechanisms. A specific combination of the knobs values is often referred as a “configuration”.

Self-adaptive computing is strongly related to other existing notions like autonomic computing, self-aware computing, elastic computing and autotuning. Autonomic [19] and self-adaptive computing are often used as synonyms. On the other hand, autotuning expresses the capacity of automatically find the best values for some application parameters, but does not necessarily consider the case where such parameters are changed and adapted throughout the application execution. In self-aware computing, the focus is on the capacity of the system to learn and reason about itself [20]. However, self-awareness does not imply self-adaptation since, in principle, the system could learn a model about its behavior but without directly taking decisions on how to alter its behavior to comply with user’s requirements. Concerning elastic computing [21], it is a narrower concept, since it is related to one specific user requirement, i.e. matching the number of resources used by the system with the current workload demand.

For these reasons, in this thesis we will always refer to self-adaptive computing systems.

1.3 Thesis Main Contributions

The main target of this thesis is to design solutions capable of reconfiguring parallel applications to enforce specific requirements regarding performance and power consumption. We try to address both the scientific and the engineering sides of the problem, by improving current existing algorithms and by building a full software stack which can be used in a real environment. We now briefly introduce the main contributions of this thesis. A more detailed picture will be provided in Chapter 2, while discussing in detail the available state of the art solutions.

1.3.1 Self-Adaptive and Power-Aware Algorithms

Most existing self-adaptive and power-aware reconfiguration algorithms are either based on heuristic solutions or on machine learning models. While heuris-
tics may work fine in general, they are usually less accurate than machine learning solutions. On the other hand, despite being more accurate than heuristics, machine learning techniques usually require extensive training phases to be performed offline, before the application starts its execution. Moreover, machine learning techniques may not be able to find a good configuration for applications with characteristics different from those of the applications used to train the models. For example, this may be a sensitive issue in cloud environments, when there is no control on the type of applications submitted by the user, and where it would be common to experience behaviors never seen for any other application previously executed [22]. Moreover, the collection of a proper amount of data to be used to train the algorithm and the subsequent computation of the prediction models could take a significant amount of time (up to a week for the applications and the architecture we used in our experiments). Albeit this is done only once for each architecture, it is still a cumbersome and tedious process, which needs to be executed from scratch when the applications are executed on a different machine.

In this thesis, we try to close the gap between these two approaches, by proposing an online learning algorithm which, like the heuristics, will not rely on any training phase to be performed offline and, at the same time, can achieve the same accuracy of complex offline machine learning approaches.

We will validate our algorithms over a comprehensive set of real applications, showing that they are often able to find better configurations than those found by existing solutions, in a shorter time frame. For our validation, we will use the applications from the PARSEC [23] benchmark. PARSEC is a well-known benchmark suite containing applications from many different domains and with varying characteristics concerning performance and power consumption, thus allowing the assessment of our algorithms over a wide range of real-world scenarios.

### 1.3.2 Parallel Patterns and Concurrency Throttling

The algorithms we propose works on general parallel applications. However, for applications with a well-defined structure, it would be possible to exploit some additional knobs, which would allow the algorithms to improve the quality of the selected configurations significantly.

This means that, for example, if the user requires some specific performance, we could be able to provide that performance with a much lower power consumption than the case where the algorithm is applied to a generic application. If we consider that current data centers run many thousands of servers, any improvement on the single server would be magnified by several orders of magnitude. In particular, we consider the possibility of exploiting concurrency throttling, i.e., dynamically remove and add threads to the application. This is something which cannot be done on any generic parallel application, since using this mechanism requires the applications to have a well defined
1.3. THESIS MAIN CONTRIBUTIONS

and regular behavior. In particular, we consider the possibility of applying such technique to applications which were designed and implemented by using parallel design patterns. However, there is often skepticism around parallel design patterns, which are usually considered not flexible enough to be used for real applications. For this reason, we created P³ARSEC, a benchmark for parallel design patterns-based applications. We were able to model and implement all but one PARSEC applications by using parallel patterns, proving that they are flexible enough, that implementing a parallel application by using patterns requires a lower programming effort and that the performance of the patterns-based version are comparable (and in some cases even better) than those achieved by using Pthreads, OpenMP, TBB or OmpSs.

1.3.3 A Customizable Framework for Self-Adaptive Applications

Since power-aware algorithms need to interact with both the application and the underlying hardware architecture, and since this is usually a cumbersome and error-prone task, we decided to design and implement a new framework, called Nornir, to help reconfiguration strategy designers in building their power-aware algorithms.

The underlying idea is that the algorithm designer should only focus on how to find the optimal configuration for the application, while the interaction with the system and with the application itself should be performed through our framework. This would allow the designer to rapidly prototype new algorithms, thus reducing the time to solution. Moreover, since the framework already provides some ready-to-use reconfiguration strategies, the designer can easily compare his algorithm with state-of-the-art ones, to analyze strengths and weaknesses of his approach. To reach our target we built an entire software stack, achieving three main goals:

- We can evaluate the self-adaptive algorithms we designed in this thesis by actually executing them to enforce performance and power consumption requirements on real applications. This is an important step since usually self-adaptive algorithms are validated through simulations.

- We created a software which could be used in real scenarios to enforce power consumption and performance requirements on parallel applications.

- We provide monitoring and actuation tools which could also be used different contexts from the one we are considering in this thesis.

The common thread in the contributions of this thesis is the simplification of the process of building self-adaptive and power-aware parallel applications: i) by simplifying the structure of the algorithms, via avoiding using big training set; ii) by simplifying the structure of the application, thus enabling the exploitation of more efficient reconfiguration mechanisms; iii) by simplifying the process of creation and validation of new reconfiguration strategies.
Contributions Balance  We tried to balance the effort dedicated to each of the three contributions, as shown in Table 1.1, where we report both the peer-reviewed papers we published for each contribution as well as the software we developed to validate the contributions. The full list of publications will be provided in Section 1.5.

<table>
<thead>
<tr>
<th>Contribution</th>
<th>Journal Papers</th>
<th>Conference/Workshop Papers</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Patterns and Concurrency Throttling</td>
<td>[30]</td>
<td>[31, 32, 33]</td>
<td><a href="https://github.com/ParaGroup/p3arsec">https://github.com/ParaGroup/p3arsec</a></td>
</tr>
</tbody>
</table>

Table 1.1: Balance between the three main contributions of this thesis, in terms of publications and software developed.

1.4 Thesis Structure

This thesis is organized in six parts, one for each main contribution, plus introduction, conclusion and appendices parts.

Part I introduces the thesis and, in Chapter 2 provides some background concerning power-aware computing and self-adaptive systems. Then, existing state of the art solutions, both from academic and industrial world are introduced.

Part II describes the Self-Adaptive and Power-Aware Algorithms contribution. In Chapter 3 we present some algorithms operating on the number of cores allocated to the application, on their clock frequency and on the displacement of the threads over the physical cores. We then evaluate these algorithms in Chapter 4, over a wide set of real applications and by comparing them with state of the art solutions. Then, in Chapter 5 we describe an algorithm which dynamically optimizes the way in which some data structure are concurrently accessed by threads in a parallel application, to provide specific performance and power consumption guarantees.

Part III introduces the Parallel Patterns and Concurrency Throttling contribution. In particular, in Chapter 6 we show how our algorithms could potentially find better solutions if the application has a clear and well-defined structure and to this purpose, we introduce parallel design patterns. In Chapter 7 we prove, for the first time on this scale, that parallel patterns are a suitable alternative to more common programming methodologies, since they reduce the programming effort while providing comparable performance.
Part IV presents the **Customizable Framework for Self-Adaptive Applications** contribution, i.e. the framework we designed and implemented during this thesis to quickly prototype and implement new power-aware algorithms. Chapter 8 describes how to use the framework and how to connect it to an application. Then, in Chapter 9 we describe how the framework can be used by algorithm designers to implement new reconfiguration strategies.

Part V concludes this thesis, outlining some possible future directions for this research, trying to extend the scope and applicability of the proposed algorithms.

Part VI is an appendix providing some additional information and results.

The diagram in Figure 1.1 different possible ways to read this document. Although we suggest reading it in its entirety, in principle it should be possible to read the Parts corresponding to every single contribution individually. However, we suggest to do not skip Part II since it contains the most important contribution of this thesis. Part VI contains some additional results which we report for completeness.

![Diagram of Parts](image)

Figure 1.1: Different possible ways to read this document.

### 1.5 List of Publications

Some parts of this thesis have been published in the following peer-reviewed papers:

#### 1.5.1 Articles in Peer-Reviewed Journals

1. De Sensi D., Torquati M., Danelutto M.
   
   *A Reconfiguration Algorithm for Power-Aware Parallel Applications*
DOI: 10.1145/3004054

2. De Sensi D., De Matteis T., Torquati M., Mencagli G., Danelutto M.
Bringing Parallel Patterns out of the Corner: the P3ARSEC Benchmark Suite
DOI: 10.1145/3132710

3. De Sensi D., Torquati M., Danelutto M.
Mammut: High-level management of system knobs and sensors
SoftwareX, Vol. 6, pp. 150 – 154, 2017
DOI: 10.1016/j.softx.2017.06.005

4. Danelutto M., De Sensi D., Torquati, M.
A Power-Aware, Self-Adaptive Macro Data Flow Framework
Parallel Processing Letters, 2016, pp. 1740004
DOI: 10.1142/S0129626417400047

1.5.2 Articles in Peer-Reviewed Conferences, Workshops and Symposia

1. Danelutto M., De Matteis T., De Sensi D., Mencagli G., Torquati M.
P3ARSEC: Towards Parallel Patterns Benchmarking
Proceedings of the 32nd Annual ACM Symposium on Applied Computing (SAC), 2017, pp. 1582 – 1589
DOI: 10.1145/3019612.3019745

2. De Sensi D., De Matteis T.
Nornir: A Customisable Framework for Autonomic and Power-Aware Applications
Proceedings of Euro-Par Workshops, pp. 42 – 54
DOI: 10.1007/978-3-319-75178-8_4

3. Danelutto M., De Sensi D., Torquati, M.
Energy Driven Adaptivity in Stream Parallel Computations
Proceedings of 23rd Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP), 2015, pp. 103 – 110
DOI: 10.1109/PDP.2015.92
4. Danelutto M., De Sensi D., Torquati, M.
   *A Power-Aware, Self-Adaptive Macro Data Flow Framework*

5. De Sensi D.
   *Predicting Performance and Power Consumption of Parallel Applications*
   Proceedings of 24th Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP), 2016, pp. 200 – 207
   DOI: 10.1109/PDP.2016.41

6. Aldinucci M., Danelutto M., De Sensi D., Mencagli G., Torquati, M.
   *Towards Power-Aware Data Pipelining on Multicores*
   Proceedings of HLPP: International Workshop on High-Level Parallel Programming, 2017

7. Danelutto M., De Matteis T., De Sensi D., Torquati, M.
   *Evaluating Concurrency Throttling and Thread Packing on SMT Multicores*
   Proceedings of 25th Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP), 2017, pp. 219 – 223
   DOI: 10.1109/PDP.2017.39

8. De Sensi D., Kilpatrick P., Torquati M.
   *State-aware Concurrency Throttling*
   Proceedings of Parallel Computing (ParCo) Conference, 2017
   DOI: 10.3233/978-1-61499-843-3-201
BACKGROUND AND STATE OF THE ART

In this chapter we first provide some background in Section 2.1. Then, in Section 2.2, we describe state of the art solutions for self-adaptive and power-aware computing. The contributions of the thesis will be highlighted in different parts of this Chapter (not in order of importance) since they will now be clearer after a discussion of the background and the related work. Some contributions are more related to the scientific aspects of this thesis while other contributions involve the development of tools and software solutions that may be helpful for those working with self-adaptive algorithms. They will be summarized again in Section 2.3. Eventually, we summarize this chapter in Section 2.4.

2.1 Background

In this section we provide underpinning to this work, by introducing some concepts related to power management (Section 2.1.1) self-adaptive software (Section 2.1.2), and parallel design patterns (Section 2.1.3).

2.1.1 Power Management

First of all, we need to make a distinction between energy and power. While power is an instantaneous metric which may be measured at any point in time, energy has to be measured over a specified period, e.g. a second, an hour, or a year. These two metrics are correlated by the simple equation $E = P \times T$, where $P$ is the power (expressed in Watts), $E$ is the energy (in Joules), and $T$ is the time period expressed in seconds (e.g. the execution time of the application). In general, minimizing energy is different than minimizing power. For example, if an application is running on 10 computing nodes, to minimize power we may want to turn off 9 nodes and let the application running just on one of those. However, this significantly increases the execution time and,
CHAPTER 2. BACKGROUND AND STATE OF THE ART

in turn, may increase the energy consumption. According to the specific scenario, it could be convenient for the user to express his requirements on either energy or power consumption.

We will mainly consider power consumption in the rest of this thesis. Considerations about energy will be done when appropriate.

Since the self-adaptive algorithms we propose in this thesis target the power consumption of CPU (which according to [36] is still the most power consuming component), we will now describe the main factors contributing to its power consumption. However, many concepts are general enough and are still valid for other system’s components.

According to [37, 38, 39], the power consumption of a CPU can be modelled as:

\[ P_{CPU} = vI_{leak} + \frac{Av^2fn}{Dynamic\ Power} \] (2.1)

Static power dissipation (also known as leakage power), occurs regardless of the processor activity, and is denoted as \( vI_{leak} \), where \( v \) is the voltage supplied to the CPU and \( I_{leak} \) is the leakage current.

The component \( Av^2fn \) denotes the so-called dynamic power. \( A \) is the activity factor, a fraction representing how often the circuit’s internal wires do a transition from 0 to 1 or vice-versa. \( c \) is the capacitance, and it depends on the specific architectural design of the CPU. \( v \) is again the supply voltage, while \( f \) is the clock frequency of the CPU. Eventually, \( n \) represents the number of active cores (i.e., cores executing instructions) on the CPU. It’s worth noting that even if there are no active cores (i.e., \( n = 0 \)), the CPU is still dissipating its static power.

2.1.1.1 Static Power

Static power usually represents around 20% of the total dissipation [40] but, in general, depends on the specific architecture [41] and, as shown in [42] it may range from 13% to 70%. Static power consumption can be either reduced by scaling down the voltage \( v \) or by reducing the leakage current \( I_{leak} \). Since voltage affects both static and dynamic power, we will analyze it later in Section 2.1.1.2, when discussing about dynamic power. To reduce the leakage current, when no applications are scheduled to run on a specific core, the operating system can decide to switch off some of the core components, by changing its C-state. C-states are power saving states, and they are denoted with \( C_0 \) to \( C_n \), with a higher index corresponding to a lower power consumption. Going from \( C_0 \) to \( C_n \), different parts of the core are turned off (e.g. caches), thus incrementally reducing its power consumption. However, the latency to switch from an idle state to the active one is higher for higher C-states. For example, on an Intel Sandy Bridge CPU, this wake-up latency may range from 1.5 mi-
2.1. BACKGROUND

cross seconds from moving from C1 to C0, to 90 microseconds to move from C6 to C0 [43]. For this reason, the operating system will move a core into higher C-states only if it predicts that there will be a long inactivity period on that core. The most common C-states of the cores are described in Table 2.1.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>Core executing code.</td>
</tr>
<tr>
<td>C1</td>
<td>Core is halted.</td>
</tr>
<tr>
<td>C1E</td>
<td>Core is halted and set to the lowest clock frequency.</td>
</tr>
<tr>
<td>C3</td>
<td>Core flushes L1 instruction cache, L1 data cache, and L2 cache to the L3 shared cache. Clocks are stopped.</td>
</tr>
<tr>
<td>C6</td>
<td>Architectural state is saved to a dedicated SRAM. Voltage reduced to 0V.</td>
</tr>
</tbody>
</table>

Table 2.1: Core C-States description

These states can slightly change on different architectures, i.e., there can be some additional C-states or some of the C-states described before could be missing. More details can be found in the ACPI interface documentation\(^1\). Resolution of C-states occur at the virtual core (i.e. a context when Simultaneous Multithreading (SMT) is present), physical core, and CPU level.

A physical core C-state is determined by the lowest virtual core state, i.e., a specific state can be entered only if all its virtual cores are in that state or in a deeper state.

CPU C-states reduce the consumption of uncore components (i.e., components shared between multiple cores, like L3 caches) and are determined by the lowest numerical C-state amongst its physical cores.

For example, by analyzing Figure 2.1, we can see that when Virtual Core 2 enters in C6 state and Virtual Core 3 enters the C1 state, Physical Core 1 will enter the lower of such state (i.e., C1). However, since Physical Core 0 is still in C0, CPU 0 is forced to remain in C0 state since it is the lower state among those of its physical cores.

Decisions taken while designing the application can drive the operating system in changing the C-State of the cores. To this purpose, we consider two commonly used techniques: threads placement and concurrency control.

**Threads Placement** The way in which the threads (or processes) of active applications are mapped to physical resources (also known as threads pinning or thread mapping) may impact the overall power consumption of the system. Indeed, let us suppose to have an application composed by 2 threads, running on a node with 2 CPUs, each one with 2 cores (for a total of 4 cores). This application could run one thread on each CPU, thus minimizing the contention on last level caches to improve performance, as shown in Figure 2.2a. Alternatively, it could run by placing both threads on the first CPU, as shown in Figure 2.2b.

\(^1\)http://www.acpi.info/
CHAPTER 2. BACKGROUND AND STATE OF THE ART

This solution would increase contention on L3 cache and may decrease application performance. However, it would reduce the power consumption of the system, since now the operating system can put CPU 1 in deeper C-States. Accordingly, by acting on the thread placement it is possible to trade performance for power (and vice versa), and the best choice usually depends on the specific application [44] and on the user requirements.

Concurrency Control  As recently demonstrated in [45], a simple and effective way to reduce power consumption is to optimize the use of synchronization and coordination algorithms when threads access shared data structures (also known as concurrency control). Consider the producer-consumer problem, i.e. two threads sharing a fixed-size buffer (a queue), where one thread (producer) inserts data in the queue and the other thread (consumer) reads and removes data from the queue. Despite being a basic example, this mechanism is fundamental in many applications [46] and runtime systems implementations [47], since it allows building complex parallel applications where their components interact by exchanging message through these buffers. When the consumer needs to read a message from the queue and there are no available messages, different solutions could be adopted\(^2\).

A common approach is to suspend the consumer thread on a condition variable. We refer to this approach as “blocking”. In this case, the suspended thread is moved in a waiting state and the hardware context is released to the operating system, thus allowing the core to enter the higher C-states, reducing its power consumption. When a new message is inserted into the queue, the producer will notify the consumer, which will be resumed. However, sus-

\(^2\)The same approaches are used when the producer needs to insert a message in the queue but there is no room for new messages
2.1. BACKGROUND

(a) Performance Optimal Solution

(b) Power Optimal Solution

Figure 2.2: Thread Placement Strategies

Performance and restart mechanisms impair application performance due to many factors such as context switch, compulsory cache miss, or core migration [45].

To mitigate such problems, the consumer could perform active-waiting on the queue, waiting for a new data element to be inserted by the producer. We call this approach "nonblocking". These solutions are usually characterized by higher throughput and lower latency [48, 49]. In some cases, like in fine-grained parallel computations, using nonblocking access to shared data structures is necessary to achieve the required performance. Unfortunately, the nonblocking approach is not power-efficient due to the busy-waiting loop executed while waiting for new data, which keeps the CPU core active in C0 state doing nothing useful.

Even in this case, the choice between blocking and nonblocking depends on the application and on the user requirements, since involves choosing the proper tradeoff between performance and power consumption. For example, if the shared queue is almost never empty nor full, using a nonblocking algorithm could be a good choice. However, if the computation is coarse-grained, a nonblocking approach may not provide any significant performance improvement, and a blocking algorithm could be suited for this scenario.
2.1.1.2 Dynamic Power

Dynamic power can be reduced by acting on the frequency $f$ of the cores, on the number of cores $n$ or on the activity factor $A$.

**Dynamic Voltage and Frequency Scaling** Since the voltage $v$ supplied to a CPU depends from its clock frequency $f$ (i.e. by increasing the frequency we also increase the voltage), due to equation 2.1, when increasing the frequency we have an almost cubic increase in the dynamic power consumption. For this reason, power consumption of a CPU may be significantly reduced by reducing its clock frequency $f$ (and consequently, its voltage). This technique is also known as Dynamic Voltage and Frequency Scaling (DVFS). Cores in modern CPUs are characterized by different $P$-states, each one corresponding to a specific voltage and clock frequency level. $P$-states are denoted with $P_0$ to $P_n$, with a higher index corresponding to a lower voltage and thus a lower power consumption. Due to hardware limitations, in many cases groups of cores will share the same voltage, and DVFS will be applied to a group of cores instead of individual cores. We will denote this situation by saying that these cores belong to the same voltage domain or voltage island [50]. More info may be found in the ACPI interface documentation³.

**Thread Packing and Dynamic Concurrency Throttling** To reduce the number of active cores $n$, two different techniques can be used: Thread Packing and Concurrency Throttling. First of all, it is worth noting that reducing the number of active cores may also have an impact on static power consumption, since the operating system could put the inactive CPUs into deeper C-states, thus reducing the leakage current. Let’s assume an application is composed by $t$ threads, but we want to run it on $n$ cores, with $n < t$. Thread packing would reach this goal by pinning the threads on $n$ cores, so that more threads share the same core. However, this may lead to a performance decrease caused by an increased contention on cores’ resources. On the other hand, Dynamic Concurrency Throttling (DCT) would dynamically remove $t - n$ threads from the application, so to have $n$ active threads (one thread for each core). Despite being a powerful mechanism, DCT cannot be applied to any application, since it may require some support and interaction with the application or with its runtime. Indeed, changing the number of threads on-the-fly may involve a redistribution of the internal state of the application, to reflect the new number of threads active in the application. This is, in general, a complex problem and in some cases this mechanism cannot be applied [51]. In the most general case, as we will show in Section 8.4.1, the programmer may need to specify the actions to be performed each time that a thread is added or removed. This operation can be in some cases extraordinarily costly and impair the effectiveness of the reconfiguration itself. Despite being a complex technique, there are scenarios where it can be safely and easily applied. For example, in parallel pattern based

³http://www.acpi.info/


2.1. BACKGROUND

applications (Section 2.1.3), the structure of the computation is well defined, and the programmer may easily reason about the effect of concurrency throttling on the internal state of the application and on how to properly manage it. Moreover, in some cases, information about the application structure could be exploited to reorganize the internal state automatically, without an explicit programmer intervention.

Concurrency throttling is a powerful mechanism which, as we will show in Section 6.1, is characterized by better power/performance tradeoff than thread packing. Indeed, if instead of placing more threads on a smaller set of cores we actually reduce their number, we could reduce the contention on shared resources (e.g., memory or disks) and improve the overall performance [52, 53].

To summarize, in Table 2.2 we report the different alternatives which can be used to control CPU power consumption.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Static Power</th>
<th>Dynamic Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads Placement</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Concurrency Control</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>DVFS</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Thread Packing</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Concurrency Throttling</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Switching Activity</td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Possible alternatives to control CPU power consumption.

Activity Factor  Eventually, power consumption can also be reduced by lowering the number of times the internal circuit’s wires switch between 0 and 1, thus reducing the activity factor $A$. For example, some compile-time solutions [54, 55, 56, 57] optimize the order of machine instructions execution, to minimize the number of bits flipped between two subsequent machine instructions. By doing so, they can reduce the power required to switch between two following instructions, since it is proportional to the number of bits flipped.

2.1.2 Self-Adaptive Algorithms

As anticipated in Chapter 1, the central point of this thesis is to design novel techniques to enforce performance and power consumption requirements on parallel applications, to overcome some serious limitations of current state of the art solutions.

Despite some compile-time [58, 54], link-time or binary rewriting [59] power-aware solutions exist, most approaches need to be applied at runtime, since applications are often characterized by workload fluctuations [60, 27], different phases [61, 62] and can also be altered by external disturbances [63, 64]. Even if some decisions can be taken statically, they need to be continuously verified against the user’s requirements and modified if required.
For these reasons, in this thesis we focus on self-adaptive techniques, i.e., solutions which can manage the system without any human intervention, to reach goals specified by the user. Such approaches, by definition, are applied at runtime, are usually time-driven and, at each sampling interval (also known as control step), act by executing an iteration of the so-called Monitor-Analyze-Plan-Execute (MAPE) loop [19], depicted in Figure 2.3.

Figure 2.3: MAPE (Monitor-Analyze-Plan-Execute) loop.

In the monitor phase, data is collected from “sensors” on the computing node and on the application (e.g., power consumption of the system, performance of the application, etc...). Such data is aggregated, for example by averaging the monitored samples over a time interval, and then passed to the analyze phase, which verifies that the monitored data is compliant with the user’s requirement. If this is not the case, the plan phase is triggered, which decides the actions to be executed to achieve the goals specified by the user. These actions can be elementary (e.g., scale down the clock frequency of the CPU) or could also be expressed as a complex sequence of operations. Eventually, the execute phase applies the reconfiguration plan by using proper “actuators” (also known as “knobs”). Such actuators may be present in both the computing node and the application. In general [19, 65], the loop is executed by an entity external to the application, also called manager, which is in charge of driving application execution to enforce the requirements expressed by the user.

A common underlying assumption in these approaches is the iterative nature of the managed application [66, 67, 68]. With iterative we refer to an application that performs roughly the same (or very similar) computation in each iteration or over a set of contiguous iterations, thus exhibiting a certain degree of repetitive behavior. Many real applications fall into this category [68, 69, 66, 70, 30] and this is a crucial property for the execution of the MAPE loop, since the manager can assume that the decisions taken at the current control step will still be valid at the next control step, because in the meanwhile the
characteristics of the computation have not significantly changed. It is worth noting that the application may still be characterized by different phases, unless there are extreme behaviors like one different phase at each control step. In such cases, decisions which were taken at the current control step of the MAPE loop would not be valid anymore at the next control step since the application would most likely be in a different phase, thus executing a computation with different characteristics. This is however not the case in many real applications [71], which are in general characterized by an iterative behavior and by a small number of different phases.

We classify applications in two types:

**Streaming Applications** These applications receive a “stream” of data, i.e., a continuous flow of elements. The elements to be processed are not already available but will be received at a variable rate, usually characterized by “ebbs” (decreases in the arrival rate) and “flows” (increases in the arrival rate), as shown in Figure 2.4. The same function is applied to each received element, or on a window (i.e., a subset) of recently received elements. The number of elements to be received is, in some cases, “infinite” (i.e., the application doesn’t have to process a predefined number of elements and could run “forever”). For example, consider a network monitoring application. In this case, the same function (e.g., identify the network protocol) is applied to every packet. In some cases, since more than one packet may be needed to detect the protocol, the function is applied to a subset of recent and contiguous packets. Moreover, the data is not already available but is received at a variable rate while the application is running, according to the traffic generated by the users of the network. Since the application is active 24hours/7days and there is not a predefined number of packets to process.

**Batch Applications** This class includes all the applications that need to process data which is already available and can be accessed by the application at any time. This is the case, for example, of a video processing
application for 3D movie rendering. In this case, the scenes models are already available on the disk, and the application needs to translate them into an actual video. In this case, one iteration could correspond, for example, to the generation of a video frame. Many batch applications have an iterative behaviour, as we will see in Chapter 4.

In the following, we will make an explicit distinction between these two classes of applications only when necessary.

Eventually, it is worth remarking that in this thesis we focus on self-adaptive solutions on a single computing node. Research in self-adaptive management on multiple computing nodes have been carried on in the last years by different communities, like algorithmic skeleton community [72], software engineering community [73] and autonomic computing community [74]. Solutions adopted on the single node and multiple nodes usually complement each other, since solutions used on a bigger scale (e.g., on the cloud or in a distributed data center) mainly involve the coordination of different computing nodes and, eventually, they rely on the self-adaptive capabilities of each individual node [74].

Eventually, we would like to spend few words on how these concepts can actually be applied in a scenario where the user may not have the complete control on the underlying hardware. For example, in public datacenters, the user most likely would have limited control on the virtualized hardware, and probably would not even able to monitor power consumption. To further complicate the scenario, users and datacenter operators usually have different goals. In general, users need to control performance according to their needs. However, there are situations where they may also be interested in reducing power consumption, for example to have economical incentives from the datacenter operator [75, 76]. On the other hand, the operator is mainly interested in controlling power consumption, since unexpected power peaks could lead to power outages. Since the user may not have any direct access to the hardware actuators required to control the performance and power consumption, we can envision having a global manager, managed by the datacenter operator, in charge of controlling the knobs available on the underlying hardware. Applications users would issue to the manager their request to enforce specific performance and power consumption objectives. The manager will enforce the requested objectives only if they are compliant with the operator requirements (e.g. on power consumption).

2.1.3 Parallel Patterns

We will now introduce the concept of parallel patterns, which will be useful in Chapter 6 to show how, by having some information about the structure of application, it could be possible to apply concurrency throttling instead of thread packing, improving the quality of the solutions found by any self-adaptive algorithm. Parallel design patterns have been envisioned as a viable solution to improve the quality and the efficiency of parallel software development while re-
2.2. STATE OF THE ART

Reducing the complexity of program parallelization and enhancing performance portability [77]. Parallel patterns are schemes of parallel computations that recur in many real-life algorithms and applications. Each of them usually has one or more well-known implementations of communication, synchronization and computation models. Algorithmic skeletons [78] were developed independently of parallel patterns to support programmers with the provisioning of standard programming language constructs that model and implement common, parametric, and reusable parallel schemes. The two concepts are closely related [79] and algorithmic skeletons may be considered as a practice of implementation of parallel design patterns.

We will describe some of the most commonly used patterns in Section 6.2.

2.2 State of the Art

In this section we describe some state of the art solutions, which will be helpful in outlining the contributions of this thesis. In Section 2.2.1 we discuss the main solutions adopted to manage power consumption of applications. Then, in Section 2.2.2 we describe different self-adaptive algorithms. Since most of the times these solutions are only simulated or executed on post-mortem data, in Section 2.2.3 we will explicitly focus on some frameworks which can be used to apply build self-adaptive software. Eventually, in Section 2.2.4 we briefly discuss the state of the art in parallel patterns-based applications.

2.2.1 Power Management

Power management techniques can be applied at each level of the computing stack, and different algorithms have been proposed to optimize or to control power consumption [80, 3]. We mainly focus on software-level solutions, and we will describe the main existing approaches in the following.

2.2.1.1 Performance and Power Consumption Requirements

Many solutions explicitly control performance and power consumption in parallel applications. For example, Facebook recently proposed an approach to hierarchically enforce power consumption requirements on the server in its datacenter [11]. Similarly, Google provides to the users of its cloud platform the possibility to specify performance requirements by using low-level metrics like CPU utilization, or high-level metrics like number of HTTP requests to be served per time unit4. Google's algorithms will then autonomously decide how many instances to use to satisfy user’s requirements and to minimize the economic cost due to the number of active instances (virtual machines).

Many solutions have also been proposed in academia. Since, as described in Chapter 1 this thesis focuses on these techniques, we will describe them

4https://cloud.google.com/compute/docs/autoscaler/
more in detail in Section 2.2.2, when discussing self-adaptive state of the art approaches.

2.2.1.2 Power Proportional Computing

Computing systems are usually designed to manage the highest expected load peak. For example, an on demand video service (e.g. Netflix) would probably have a very high load demand in the evening but not so much traffic during office hours. The peak load usually happens in a limited time frame, leaving the system underutilized for the remaining part of the day. Similar patterns occur in most computing systems. Indeed, according to [1, 3], the average servers’ utilization is usually between 10% and 50%. This is an important source of inefficiency and, for this reason, many solutions work by dynamically changing the number of resources allocated to the application, trying to have a high utilization during the entire working period of the system. This includes finding the proper number of cores [46] or number of servers [81] to use to serve all the requests with minimum power consumption or minimum number of resources. This is often referred as elastic computing, particularly in the context of streaming applications.

For example, Amazon Autoscaling\(^5\) provides the user possibility to dynamically scale the number of Virtual Machine (VM) instances used by his application. By doing so, applications can automatically follow the demand curve, reducing the need to manually provision the number of instances to be used in advance. Since the billing plan is based on the number of allocated instances, by using autoscaling the user can reduce the costs required to provide its service while not reducing its performance. The scheduler monitors some metrics like CPU utilization, disk read/writes or number of bytes received/sent from/to the network and the user can specify how many instances to add/remove when these metrics exceed some specified threshold.

In other cases, elasticity is adopted to reduce the power consumption of data centers. For example, Facebook\(^6\) uses a control theoretic approach to reduce energy consumption by consolidating the workload on a smaller set of servers with a medium-high utilization instead of running the workload on all the server at a low utilization. The amount of active server changes during the day according to the actual amount of received requests, allowing to reduce the energy consumption of 10-15% over a 24 hours period. Netflix\(^7\) moved forward with these concepts by proposing Scryer. In this case, instead of reacting to changes in the workload, they predict how the workload will change in the future based on historical data. By using these predictions, they can perform earlier the needed reconfiguration actions thus improving performance and service availability.

\(^5\)https://aws.amazon.com/autoscaling/getting-started/
\(^6\)https://code.facebook.com/posts/816473015039157/making-facebook-s-software-infrastructure-more-energy-efficient-with-autoscale/
\(^7\)http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html
2.2. STATE OF THE ART

It is worth noting that this is a subset of the scenario considered in this thesis and in Section 2.2.1.1. Indeed, elasticity can be achieved by using a performance requirement, where the requirement corresponds in having a performance which matches, time by time, the input rate of data to the application. We will show in Chapter 3 that this case can be managed as well by the algorithm we are proposing.

2.2.1.3 Approximate Computing

In some situations, it is possible to trade off the accuracy of the produced output for a lower power consumption. For example, if a user is watching a video on a mobile device, he may be willing to reduce the quality of the video to have a longer battery life. Such solutions [82] usually allow the user to control the balance between power consumption and the quality of the end-result, without sacrificing performance.

For example, [83] focuses on a task-based programming model like OpenMP, by adding the possibility to assign a significance to each task. This is a value between 0 and 1, with higher values denoting that the task is more important for the overall quality of the final result. For each task with a significance lower than 1, it is possible to specify an approximate version, that consumes less power, takes less time to be computed but provides a lower quality result. Moreover, for each task, the user must specify the maximum ratio of tasks that can be executed on the lower quality version. Eventually, it is possible to specify the final desired quality of the results (between 0 and 1). Different policies are then applied by the runtime system to decide how many less-significant tasks should be executed in their approximate version. The advantage of this solution is that it allows a graceful quality degradation of the output. However, users need to specify for each task its significance and the maximum ratio of acceptable lower-quality task. This may be non trivial for the user to determine, and wrong decisions may lead to results with unacceptable quality.

Similar solutions are adopted by [84, 85] and some programming languages like PetaBricks [86] provide such features by design.

These works are orthogonal to this dissertation, since our algorithms do not alter the accuracy of the output.

2.2.1.4 Heterogeneous Architectures

The wide usage of accelerators (e.g. GPUs (Graphics Processing Units), FPGAs (Field-Programmable Gate Arrays) and DSPs (Digital Signal Processors)) and of asymmetric multicore systems (e.g. ARM’s big.LITTLE) opened a wide spectrum of possibilities concerning power consumption optimization. Indeed, such architectures are characterized by different performance and power consumption for different applications [87, 88, 89, 90, 91]. By exploiting this variety, it is possible to execute the application on the most suitable architecture according to the user requirements. In some cases, it is possible to improve
both performance and power consumption compared to a CPU only execution.

These techniques address a different use case with respect to the one consider by this thesis, and are orthogonal to the algorithms we will present.

2.2.2 Self-Adaptive Algorithms

We now describe the state of the art on self-adaptive and power-aware algorithms. We organize this description by following the different steps of the MAPE loop. We first discuss in Section 2.2.2.1 the main existing approaches used in the Monitor and Analyze phase, while in Section 2.2.2.2 we show different techniques used in the Execute phase. Eventually, in Section 2.2.2.3 we describe solutions adopted for the Plan phase.

2.2.2.1 Monitor and Analyze

Existing self-adaptive approaches adopt a wide variety of solutions to monitor and analyze data collected from the application and the computing system.

Power consumption can be monitored by using external power meters to be physically connected to the hardware, can be estimated by analyzing other metrics (e.g. CPU utilization) or can be measured by reading some special CPU registers available on some architectures. This broad spectrum of possibilities led to the creation of several power consumption monitoring tools. Different meters and tools monitor different system’s components, ranging from the power consumption of the entire node to that of the CPU only.

Concerning performance, different monitoring techniques can be used. They differ in intrusiveness, applicability and on how they impact the optimality of the decisions taken in the plan phase. We describe them starting from the less intrusive (and less optimal) ones to the more invasive but more optimal techniques.

Hardware Counters Most performance monitoring techniques rely on hardware counters, for example by monitoring the number of assembler instructions executed per time unit (i.e., instructions per second (IPS)). Hardware counters are usually chosen since they do not require any modification to existing applications.

Such solution has however different drawbacks. Firstly, correlating the IPS (or other hardware counters) to the actual application performance is not an easy task and not so intuitive from the user perspective. For example, it is simpler for the user to express a performance requirement on a web server regarding number of requests served per time unit rather than expressing the same requirement in IPS. Additionally, as shown in [67, 84, 101] hardware counters may not always be a good performance proxy since they are not always strictly...
correlated to the actual application-level performance. Eventually, there may be some portability issues, since not all the performance counters are available on all the architecture.

**Instrumentation** A better solution would be to monitor actual, high-level, application performance. For example, by monitoring the number of requests served per time unit in a web server, it would be possible for the user to understand and express better his performance requirements and for the manager to clearly monitor and understand the effect of the taken decisions. Such monitoring data can be communicated by the application to the manager by instrumenting the application. This solution has been adopted in many state-of-the-art approaches [102, 103, 104] and different libraries can be used to instrument applications [66, 105]. Despite being a more intrusive approach than using the hardware counters (since it requires to modify the existing application code), the required effort is still reasonable, since the programmer should only insert few instrumentation calls inside the code.

**Limitation 1**

The main limitation of the existing instrumentation tools is the impossibility to monitor metrics which are particularly useful for streaming applications. For example, they do not provide any information which could be used by the **plan** phase to distinguish whether a performance drop is caused by some intrinsic characteristic of the application or by a drop in the arrival rate of the data to the application.

In the former case, the manager could take some decision in the **plan** phase, trying to improve application performance (e.g. by increasing the number of resources allocated to the application).

In the latter case, it would be useless to execute such actions since, even if the application could process data faster, there would be not enough data to process. This would only increase the power consumption without any beneficial effect on the performance.

**Contribution 1**

For this reason, we designed and implemented **Riff**, a C++ library to instrument parallel applications which, besides monitoring the metrics usually monitored by the existing tools [66, 105], also monitors some metrics which are specific to streaming applications, improving the decisions taken during the **plan** phase, as we will show in Section 3.2.4. Differently from existing monitoring tools, **Riff** allows monitoring applications running on both local and remote computing nodes.
Framework- and Application-Specific Solutions  Other monitoring solutions are tailored towards specific runtimes [106]. By accessing their internals, they can gather more detailed data, to take better decisions in the Plan phase and to exploit more fine-grained knobs in the Execute phase. For example, as we will show in Chapter 5, it would be possible to optimize some framework-specific aspects like concurrency control. In some cases, self-adaptive solutions explicitly address applications expressed by using a specific programming model, e.g. by using parallel patterns [27, 72, 107, 108, 109] or dataflow model [25]. Eventually, some solutions are targeted towards a specific application [110, 111]. By monitoring application-specific metrics, they achieve, in general, better results compared to more generic solutions.

2.2.2.2 Execute

We described in Section 2.1.1 the main factors contributing to the CPU power consumption and the actions which could be taken to control it. We now describe how those control knobs are actually used by some state of the art solutions. Moreover, for completeness, we will also describe of the techniques used to optimize and control power consumption of other system’s components.

CPU  As we saw in Section 2.1.1.1, static power can be reduced by the operating system when the CPU is idle, by changing its C-State, thus reducing the leakage current. Instead of relying on the operating system, C-states can be explicitly forced to limit system’s power dissipation. For example, in [112] the authors provide a technique to switch between active and idle states dynamically. By predicting the impact of idleness on both performance and power consumption, they can provide the application user with guarantees on performance and power consumption.

Moreover, we recall that also the choice of the concurrency control algorithm in multi-threaded applications can have an impact on the static power consumption. A common approach is to have a nonblocking algorithm and to delay the busy-waiting loop by performing very short duration suspensions (e.g. by using the usleep or equivalent calls) at each iteration, according to a fixed or variable duration. For example, this is often done by using exponential backoff [113], i.e. by having variable and exponentially increasing the duration of the suspensions in successive iterations.
2.2. STATE OF THE ART

**Limitation 2**

When using exponential backoff, the threads may be not reactive enough since a new data element may arrive while the consumer is sleeping, and it will not start to process the data before the sleeping period is over. Finding good values for the minimum and the maximum sleeping time is not straightforward. Since it may depend on the target application, this solution could require some non-trivial tuning of these parameters.

**Contribution 2**

Differently from existing approaches, we designed an algorithm which automatically switches from a nonblocking to a blocking concurrency control (and vice-versa) according to the power consumption and performance requirements specified by the user. Differently from exponential backoff, this approach does not require any application-specific tuning.

**Insights 2**

Chapter 5.
Publications: [29]

Concerning the dynamic power, DVFS is one of the most commonly used techniques. Historically, DVFS has been used in memory intensive applications, to lower the CPU frequency while the processor is stalled, waiting for memory load request to be served [114], without any significant impact on application’s performance. However, as shown in [115], due to the emergence of more efficient memory technologies, this technique is not efficient anymore in such a context. Moreover, operating systems already provide the so called governors, i.e. algorithms which can be used by the operating system to automatically scale the frequency according to the CPU load. Nevertheless, frequency scaling is still profitably used to explicitly trade off power consumption for performance (and vice-versa) [14, 116, 117].

*Thread Packing* and *Dynamic Concurrency Throttling* are both used to explicitly control performance and power consumption of parallel applications [116]. In some cases, due to contention on shared resources, an application may increase its performance by using less cores than those available, and many solutions have been proposed for finding the optimal number of cores to be used [53, 118, 119].
CHAPTER 2. BACKGROUND AND STATE OF THE ART

**Limitation 3**

In general, these low-level architectural mechanisms are managed via command line tools or by using appropriate Application Programming Interfaces (API). However, when the API is present, it usually does not provide a sufficiently high abstraction level. Moreover, existing tools are often targeted towards one specific knob, without any possibility to interact and extract information obtained with other tools, hampering their effectiveness. Indeed, integrating APIs from different tools may result in a time-consuming task, since the programmer needs to convert data representation between all these tools and force them to work together.

**Contribution 3**

For these reasons, in this thesis we designed and developed Mammut, a C++ framework which provides an object-oriented abstraction of the knobs available on the system. Mammut allows to easily express interactions among different knobs, thanks to a simple and easy to use interface. Differently from most existing tools, Mammut can also be used to control knobs over remote computing nodes, which is an emerging requirement [120, 121] and is of paramount importance in some decentralized MAPE loop designs [73]. Moreover, Mammut can also be used to monitor some architecture-specific metrics such as power consumption.

**Insights 3**

Section 9.2.1  
Publications: [34]  
Software: http://danieledesensi.github.io/mammut/

**Memory and Caches**  
Some works report main memory power consumption to range between 20% [122] and 40% [123] of the overall system power consumption. Some solutions [122, 124] scale down the voltage of the main memory to find good tradeoffs between power consumption and system performance. In some cases [125, 126] the memory consumption is reduced by scaling down the memory voltage and refresh rate below the minimum allowed by design. Since this may introduce memory errors (e.g., bit flips), such approaches require the programmer to specify which data is non-critical for the application, i.e., which data can be possibly corrupted without a significant impact on the applications or on the meaningfulness of the produced result.
This is, however, an intrusive approach which requires a deep understanding of the application logic and is prone to errors. Moreover, there are critical applications which cannot tolerate even a single bit flip.

Other solutions try to determine the maximum working set size of an application. By doing so, it is possible to turn off some cache parts which are not needed for the computation, thus reducing the power consumption of the system. This can be done either at compile time [58, 127, 128] or at runtime [70].

**Interconnection Network**  Interconnection networks are another important contributor of the overall system consumption. At exascale level, it contributes for the 10%-20% of the total dissipation [129] and it is estimated to grow in the future up to the 30% [130]. For such reasons, some techniques to automatically shut down unused links at each computing node have been proposed [131, 132].

**Hard Drive**  Eventually, other works optimize the power dissipation of hard drives. In [133] this is possible by consolidating the workload on the smallest set of drives possible, turning down the unused ones. More recently in [134] a solution have been proposed to limit the maximum power consumption of hard drives while they are active, by dynamically resizing the disks’ queue.

### 2.2.2.3 Plan

To describe the existing self-adaptive algorithms, we divide them between those who need to collect and exploit data collected before the application is started (e.g., profiling of previous runs) and those that do not need such kind of data.

**Profiling-Based Solutions**  A first set of solutions is characterized by the use of complex analytical models, obtained by an offline training phase [117, 118, 116, 15]. These approaches are based on the collection of profiling data for an extensive set of different applications. Such data will be used to train machine learning algorithms, which correlate some metrics to the observed performance/power consumption. When an application is executed, metrics collected at runtime during the *monitor* phase will be used as input to the model, to predict performance and power consumption of the application in different configurations and to take the best decision during the *plan* phase. For example, in [135, 136] the authors propose a cluster management system to allocate the right amount of resources to different applications given a performance requirement. By using classification techniques, they can perform this decision by exploiting information on previously executed applications.

Albeit offline approaches are usually characterized by a low runtime overhead, their accuracy profoundly depends on the choice of the data used to train the model. For example, if a behavior that has never been experienced
in the training phase occurs, these methods could produce sub-optimal solutions [137, 138, 14, 17]. This is typical for example in cloud computing [22], where applications may have characteristics never experienced on other applications. Moreover, due to its complexity, offline training could require several hours to be completed [117]. Considering that the models need to be retrained if the application is executed on a different platform, this could be an annoying overhead.

Recently, Bard [103] and Poet [102] have been proposed. They use a combination of control theory and linear programming to provide both performance and power consumption guarantees. Despite being a robust approach, since the algorithms work with a generic combination of resources, they need to know a priori the performance and power consumption of the application in each possible configuration. This significantly limits the usability of the approach since these values need to be collected by the user, for example by running the application in all the possible configurations.

To mitigate these limitations, some models integrate the offline collected data with additional data obtained while the application is running [14, 139]. However such solutions still share some of the limitations of the full offline approaches.

**Online-Only Solutions** A different approach is to not rely on any previously collected information and to only make decisions on the base of the data collected while the application is running.

The most intuitive way to do that is to analyze all the possible configurations, as done by [140] to select the most performing configuration under a power budget. However, this is a costly process, which can be prohibitive due to a large number of possible knobs configurations. Moreover, the process should be repeated if the application enters into a different phase or if an external interference occurs, since performance and power consumption behaviors would be different. In practice, this analysis phase could not terminate before the application end.

More sophisticated solutions are based on heuristics [16, 141, 142, 143]. For example, the authors of [106] proposed an extension to OpenMP to specify power and performance constraints, which will be enforced during the plan phase by using hill-climbing heuristics. However, as we will show in Chapter 4, heuristics are usually less accurate than machine learning approaches.

To address such issues, more complex and accurate techniques are used, based on control theory [66]. Albeit they avoid some typical problems of the offline approaches (like the strong correlation between prediction accuracy and the choice of the training set), the collection of runtime data and the computation of prediction models while the application is running may introduce a significant overhead.
2.2. STATE OF THE ART

Limitation 4

Existing online algorithms have some limitations:

1. Some of them are either simulated or the analysis is done on post-mortem data [144, 16, 52, 119, 145]. Although a simulation may provide a first approximation about the precision of the model, it is difficult to estimate the run-time overhead of these methods accurately.

2. Some solutions do not explicitly consider power consumption, thus only providing the possibility to specify performance constraints [135, 16]. Similarly, other algorithms only optimize performance under a given power budget [146, 124, 147, 148, 149], not providing any means to express explicit performance requirements.

3. Other approaches [143, 135, 150, 141, 151] only rely on one knob. For example, the well known Intel’s RAPL power capping tool only uses DVFS to enforce user requirements. However, as shown in [117, 27], when more knobs are exploited at the same time, the range of available configurations is extended and is possible to find better performance/power tradeoffs.

4. Many existing algorithms do not consider stream processing applications and characterize every performance fluctuation as a phase change [14, 140]. This is not efficient for stream processing applications, for the reason discussed in Section 2.2.2.1.

Contribution 4

We try to address such existing limitations by providing different algorithms which can be used to express explicit performance and power consumption requirements [24, 25, 26], by exploiting more control knobs at the same time. Such algorithms can significantly cut down the number of reconfiguration on stream processing applications, thus having a lower impact over the application execution. In Chapter 4 we validate these algorithms over a comprehensive set of real applications, comparing them with some state of the art solutions. We will show that they are more accurate than state of the art approaches and they provide different tradeoffs between accuracy and runtime overhead.

In Table 2.3, we report a summary of the main characteristics of the existing algorithms for performance and/or power capping (our algorithms are indicated as [25, 26, 24]).
## Constraints vs. Knobs

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Knobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>IDLE</td>
</tr>
<tr>
<td></td>
<td>Cores</td>
</tr>
<tr>
<td></td>
<td>DVFS</td>
</tr>
<tr>
<td></td>
<td>DCT</td>
</tr>
<tr>
<td></td>
<td>THREADS</td>
</tr>
<tr>
<td></td>
<td>PLACEMENT</td>
</tr>
<tr>
<td></td>
<td>ONLINE</td>
</tr>
<tr>
<td></td>
<td>LEARNING</td>
</tr>
</tbody>
</table>

Table 2.3: Comparison between some existing techniques with the online learning algorithm developed for this thesis ([25, 26, 24]).

### 2.2.3 Self-Adaptive and Power-Aware Frameworks

Despite many self-adaptive strategies for power-aware computing exist, as we described in Section 2.2.2.3, most of them are usually only simulated. Indeed, implementing such algorithms is a cumbersome and error-prone duty for application programmers. When implementing a self-adaptive technique, the programmer has to deal with many architectural low-level issues related to hardware mechanisms management like voltage, frequency, cores topology, etc. Even interfacing with applications to collect monitoring data may not be an easy task. When such strategies are implemented, they are usually embedded inside the application code, and it is challenging to port them and to validate them on different applications.

Some self-adaptive frameworks have been proposed in the literature [152, 153, 154] to address this problem, allowing to customize different parts of the MAPE loop.

A reference work is SEEC [67]. In this work, the authors describe the design of a framework for self-aware computing. In this framework, the plan phase uses a combination of control-theory and machine learning. Despite it is not possible for the user to change the algorithms used the plan phase (e.g., by
replacing them with an ad-hoc heuristic), custom execution mechanisms can be specified.

On the other hand, Adam [104] allows the customization of the plan phase but the execute and monitor phases are fixed, allowing the interaction with only two knobs (DVFS and Thread Packing).

Similarly to SEEC, Bard and Poet provide an algorithm which is general enough to work with different types of resources. By default, DVFS and Thread Packing are supported and, when other knobs need to be added, appropriate actuators need to be implemented.
**Limitation 5**

Existing frameworks have some limitations. Some of them [152, 153, 154] provide insufficient customization opportunities, are quite outdated and the source code is not publicly available.

*IBM StreamS* [155] is a commercial and proprietary framework for the development and deployment of (distributed) stream processing applications. Complex applications are expressed by means of computational graphs of *Processing Elements* (PEs) that cooperate by exchanging data. Each PE can be internally parallelized exploiting a certain number of threads. StreamS may change at runtime the number of used threads to maximize throughput (using its own strategy [156]), but it does not allow users to enforce a specific throughput or power consumption for the application or to define custom decision strategies.

*SEEC* [67], *Bard* [103] and *Poet* [102] allow the customization of the Monitor and Execute phases of the MAPE loop, but provide their own Plan algorithm. Albeit being a flexible strategy, it is not possible to replace it with a different one. This is an important limitation, since in some cases the designer may exploit some knowledge about the application or the architecture to design more efficient plan algorithms with respect to those already provided by the framework.

Moreover, *Bard* and *Poet* require the user to specify changes in power consumption and performance for each possible application. This is a cumbersome task for the user, since it needs to execute the application in all its possible configuration and to collect its performance and power consumption beforehand. Furthermore, such frameworks cannot be used if the applications to be executed are not known a priori.

In addition to this, a common limitation of such frameworks is the lack of explicit support for stream processing applications. As shown in [24] this can lead to unnecessary reconfigurations since it would not be possible to know whether workload fluctuations are caused by intrinsic changes in the application or by changes in the arrival rate of data to the application. Being able to distinguish these two situations would improve the stability of the system and reduce the overhead caused by application reconfigurations.

For these reasons, we believe that providing a customizable framework would allow the designer of the plan strategy to just focus on the algorithm, exploiting the infrastructure provided by the framework to collect the data and to apply the decisions. This is a fundamental step in building efficient self-adaptive solutions and for their wide adoption.
2.2. STATE OF THE ART

Contribution 5

One of the main contributions of this thesis is the design and implementation of Nornir, a customizable C++ framework to implement self-adaptive and power-aware algorithms to control parallel applications’ execution on shared memory multicore machines. Differently from existing frameworks, Nornir allows the Plan strategy designer to quickly prototype and validate his own decision strategies. The proposed design is flexible enough, and different state of the art self-adaptive algorithms have been implemented by using the framework. This also provides the algorithm designer the possibility to compare his new algorithm with other existing ones. Moreover, differently from other frameworks, Nornir is designed to manage stream processing applications efficiently.

Insights 5

Part IV.
Publications: [24, 35, 25]
Software: http://danieledesensi.github.io/nornir/

2.2.4 Parallel Patterns

Combinations of parallel design patterns and algorithmic skeletons are used in different parallel programming frameworks such as Microsoft PPL [157] and Intel TBB [158] as well as in niche pattern-based research frameworks such as SkePU [159], Muesli [160], FastFlow [47], SkeTO [161], SkeCl [162], Skandium [163] and OSL [164] just to mention a few of them. Other frameworks such as Google MapReduce [165] are instead built around a single powerful pattern.

To raise the level of abstraction in parallel software development for specific application domains, some research works proposed DSLs (Domain Specific Languages) built on top of pattern-based frameworks [166, 167, 168]. Their main aim is to help the domain experts to prototype different parallel variants of their code easily and to introduce parallel runtime optimizations more selectively. A similar approach, which leverages the new C++1x features, consists in annotating the sequential code with C++ attributes to introduce parallel patterns in specific regions of code (usually compute-intensive kernels). Then, a source-to-source compiler is responsible for translating the annotated C++ code into a parallel code linked to the proper pattern-based runtime library [87, 169].

The use of parallel patterns in the development of applications provides several advantages both concerning time-to-solution as well as concerning the
automatic or semi-automatic applicability of different optimization strategies (e.g., like the ones proposed in [170, 171, 172]). This last aspect is usually manually enforced in non-pattern-based parallel programming models such as MPI and Pthreads.

**Limitation 6**

However, despite the diffusion of parallel design patterns and although many applications have been implemented by using such programming abstraction [27, 46, 173, 174], there is still some skepticism about the flexibility and the advantages of using this approach.

**Contribution 6**

We addressed this issue by modeling applications in the PARSEC benchmark [23] suite by using parallel patterns. We implemented the applications with two different parallel pattern based frameworks, comparing them with alternative implementations with OpenMP [175], TBB [158], Pthreads and OmpSs [176]. On one side, we tried to remove skepticism around parallel patterns, by proving that parallel patterns are a suitable candidate for designing parallel applications, reducing the programming effort required to implement them while also improving their performance. On the other hand, we created a benchmark (which we called P3ARSEC) which can be used to validate techniques targeted towards parallel-pattern based applications (included concurrency throttling) on realistic applications, with different behaviors regarding performance and power consumption.

**Insights 6**

Part III
Publications: [30, 31, 32, 177]
Software: https://github.com/ParaGroup/p3arsec

2.3 Contributions

Now that we provided the background and we described the state of the art solutions, we briefly summarize the main contributions of this thesis.

One of the main contributions of this work is the proposal of an algorithm to enforce performance and power consumption requirements on parallel applications. To reach this goal, our algorithm builds prediction models for performance and power consumption by using online learning techniques. While
most existing solutions either rely on extensive offline training phases or on simple heuristics to be applied at runtime, our solution takes the best from both worlds. Indeed, as we will show in Chapter 4, it is characterized by an accuracy comparable (or even higher) than that of an offline learning approach, while being characterized by an overhead similar to that of an heuristic.

In addition to that we prove, for the first time at this scale, that parallel patterns are a suitable candidate for designing and implementing parallel applications. Indeed, in Chapter 7, we will show that they are characterized by a performance similar to that of an handwritten implementation, while requiring a lower programming effort. Beyond being a contribution per se, this led to the creation of the P3ARSEC (Parallel Patterns PARSEC) benchmark suite, which would enable further research on techniques which target pattern based applications. Due to the relevance of concurrency throttling and to its

Even for pattern based applications, this could provide useful insights on this power-aware technique in the future. ally, we designed and implemented a framework to enforce performance and power consumption requirements on parallel applications, by using self-adaptive algorithms. As we will describe in Chapter 8, it can either support legacy applications, as well as applications implemented by using specific programming frameworks. The latter case, despite being more restrictive, can lead to better solutions, since the self-adaptive algorithm could control some framework-specific knobs which would not be available otherwise. Moreover, differently from most existing solutions, our framework can be easily and fully customized to implement new self-adaptive algorithms and to support additional knobs and monitoring mechanisms.

This thesis also introduced several minor contributions, which were described in this chapter and which are not reported here for brevity’s sake.

2.4 Summary

In this Chapter, we provided some background and we presented state of the art solutions for self-adaptive and power-aware applications, proposed by academia or currently adopted by Information Technology (IT) companies. We also introduced the main limitations of existing approaches, at different levels, and briefly outlined how such limitations are overcome in this thesis or how the techniques that will be presented in the next Chapters could be used jointly with existing ones.

In this Chapter we mainly focused on subjects close to the original target of this thesis, i.e., providing explicit performance and power consumption guarantees to the application user, for applications running on multicore machines. We did this by analyzing existing solutions for each step of the MAPE control loop.

Power-aware computing is also addressed at other layers of the computing stack, for example at the virtualization level, where many techniques exist to consolidate virtual machines on the minimum set of computing nodes, or at
the hardware level, where solutions are proposed to reduce the energy foot-
print of chips. The discussion of such techniques is however outside the scope
of this thesis. Although this Chapter is far from being a complete and exhaust-
tive presentation of all the existing power-aware techniques, we provided the
necessary background to understand the content of the following chapters and
to assess the original contributions of this thesis.
Part II

Self-Adaptive Algorithms and Evaluation

In this Part of the thesis we describe our first and most important contribution, the design of self-adaptive and power-aware reconfiguration algorithms for enforcing user requirements concerning performance and power consumption on parallel applications. In Chapter 3 we introduce two algorithms for selecting the most appropriate configuration according to the user requirements, regarding number of cores, clock frequency and placement of threads on the physical cores. These algorithms will be evaluated in Chapter 4, by executing them over different applications and for a wide set of different performance and power consumption requirements. To assess their effectiveness, we will also compare them with some existing state-of-the-art algorithms. Then, in Chapter 5 we adopt a slightly different approach. Instead of acting on hardware resources, we will dynamically modify the algorithm used by the threads composing a parallel application when accessing some shared data structure, showing that it is possible to optimize both performance and power consumption of streaming applications.
In this Chapter, we propose two self-adaptive algorithms to reconfigure the number of cores, clock frequency and mapping of the threads on the physical cores, to enforce specific user requirements regarding performance and power consumption. As we will see in the evaluation part (Chapter 4), the algorithms we designed complement each other. We first describe in Section 3.1 a heuristic that can be used to enforce performance requirements only. Then in Section 3.2 we propose a self-aware algorithm for enforcing both performance and power consumption requirements, exploiting online machine learning to accurately predict performance and power consumption of a parallel application in different configurations.

Parts of this chapter have been published in [24, 26, 28, 25, 27].

3.1 Heuristic

The first algorithm we are going to present is a heuristic that can predict, with a reasonable accuracy, the throughput of the application in different $<\text{Cores, Frequency}>$ configurations. Since, as anticipated in Chapter 2, self-adaptive algorithms work mainly on iterative computations, we define the throughput as the number of iterations performed by the application in 1 second. The algorithm does not explicitly predict power consumption. Instead, it will perform a rough estimation, to decide whether a configuration is more or less power consuming than another.

Let us define $n$ as the number of cores\(^1\) and $f$ as the frequency. The basic idea behind this algorithm is that, when the user requirements are violated, the algorithm will search for another $<n, f>$ configuration, with acceptable performance according to the requirements. For doing that, the algorithm needs first to predict the throughput of the application for all the possible $<n, f>$

\(^1\)The number of cores $n$ can be changed by using either thread packing or concurrency throttling.
pairs. Then, if required by the user, among all the configurations with acceptable performance, the algorithm could pick the one with the lowest estimated power consumption. We first show how to predict performance. Then, we show how to estimate power consumption.

### 3.1.1 Performance Prediction

Let us denote $R$ as the throughput of the application and $\pi$ and $\bar{f}$ as the current number of cores and the current clock frequency, respectively. Without loss of generality, we can assume $R$ to be proportional to the clock frequency [41]. Accordingly, for a given $\pi$, we can estimate the throughput the application will have when frequency $f$ is changed, by computing:

$$R(\pi, f) = R(\pi, \bar{f}) \cdot \frac{f}{\bar{f}}$$

However, performance is not always proportional to frequency since clock frequency mostly impacts on the parts of the application executed on the CPU. Frequency scaling does not affect the time spent in I/O and memory accesses. However this is a heuristic and, as we will show in Chapter 4 it works well in practice.

We define with $S(n)$ the scalability of the application: $S(n) = \frac{R(n, \bar{f})}{R(1, \bar{f})}$. For a fixed frequency $\bar{f}$, if the number of cores $n$ changes, we can predict the throughput as:

$$R(n, \bar{f}) = R(\pi, \bar{f}) \cdot \frac{S(n)}{\pi}$$

We consider the application to have an ideal scalability, i.e. $S(n) = n$. Again, this is usually not true and, in general, we have that $S(n) < n$. However, as we will show in a while, the algorithm can automatically adjust such mispredictions.

It follows that, in general, if both the number of threads and the frequency change:

$$R(n, f) = R(\pi, \bar{f}) \cdot \frac{n \cdot f}{\pi \cdot \bar{f}}$$

(3.1)

### 3.1.2 Power Consumption Estimation

Now that we can estimate the throughput for all the available <Cores, Frequency> configurations, we want to pick, among all the configurations with a proper throughput, the one with the lowest power consumption. As motivated in Section 2.2.2.2 we are interested in controlling the power consumption of CPUs. We also described why the power consumption of an active CPU (i.e. a CPU where at least one core is executing instructions) can be modelled as:

$$P(n, f, v) = vI_{\text{leak}} + Acv^2 fn$$

(3.2)
where $v$ is the voltage, $I_{\text{leak}}$ is the leakage current, $A$ is the activity factor and $c$ is the capacitance. For our purposes, we can consider $A$, $c$ and $I_{\text{leak}}$ as constants. These constants, however, may change according to the architecture, to the application and even to the application input. In Section 3.2 we will show another algorithm which can derive these constants at runtime by using online learning techniques. However, since for this algorithm we just want to find the configuration with the lowest power consumption, we are not interested in knowing the exact power consumption but only a proportional estimation that we can use to order configurations according to their estimated power consumption. For this reason, we drop $vI_{\text{leak}}$ (i.e. the static power) from the equation, and we remove the constants $c$ and $A$, obtaining the following proportionality [38]:

$$P(n, f, v) \propto v^2 fn$$

(3.3)

To remove the dependency from $v$, we can observe that the voltage is strictly correlated to the frequency, since by increasing the frequency we raise the operating voltage and vice versa. Consequently, we can use a tabular function $V(f)$ to get the voltage value associated to a specific frequency level $f$. This function is architecture specific and may be obtained programmatically or by using the values provided by the CPU vendor. Our equation then becomes:

$$P(n, f) \propto V(f)^2 fn$$

(3.4)

### 3.1.3 Algorithm

Putting all the pieces together, we obtain the algorithm shown in Algorithm 1.

Function `Main()` resembles the structure of the MAPE loop. The implementation of the `Monitor()` and `Execute()` functions depend on the way in which the manager interacts with the application. This is implementation dependent and we will show the alternatives we provide inside our framework in Chapter 8. The `AnalyzeAndPlan()` function, if the requirement is violated predicts the throughput and power consumption of all the possible `<Cores, Frequency>` configurations$^2$, selecting the one with a throughput higher or equal than the required one and with the lowest power consumption. Function `Predict()` computes Equation 3.1 and Equation 3.4 over a specific configuration. Moreover, for clarity’s sake, we only show how to enforce the throughput requirement. However, if the user knows the number of iterations $s$ that the application will perform, instead of setting a throughput requirement, he can express the performance requirement as a maximum execution time $T$. Since $T = R \cdot s$, the algorithm will automatically translate the execution time requirement into a throughput requirement. If the average throughput changes during application progress, the throughput requirement will change too. Similarly, the

$^2$The range of possible frequencies is discretized according to the capabilities of the underlying hardware. In general, only some specific frequencies can be selected.
ALGORITHM 1: Heuristic algorithm for the selection of the best configuration.

```plaintext
Function Predict(n, f)
1. \( R_{\text{pred}} = R_{\text{real}} \cdot \frac{n}{f} \)
2. \( P_{\text{pred}} = V(f)^2 f n \)
3. return \( R_{\text{pred}}, P_{\text{pred}} \)

Function AnalyzeAndPlan()
4. if \( R_{\text{real}} < R_{\text{required}} \) then
5.     forall \( n, f \) do
6.         \( R_{\text{pred}}, P_{\text{pred}} \leftarrow \text{Predict}(n, f) \)
7.         if \( R_{\text{pred}} \geq R_{\text{required}} \) and \( P_{\text{pred}} < P_{\text{min}} \) then
8.             \( P_{\text{min}} = P_{\text{pred}} \)
9.             nextConfiguration \( \leftarrow < n, f > \)
10.         end
11.     end
12. end
13. return nextConfiguration;
14. return currentConfiguration;

Function Main()
15. while true do
16.     Sleep(controlStep);
17.     \( R_{\text{real}} \leftarrow \text{Monitor()} \)
18.     \( c \leftarrow \text{AnalyzeAndPlan()} \)
19.     \( \text{Execute}(c) \)
20. end
```

The algorithm will estimate the energy consumption by multiplying the estimated execution time by the predicted power consumption.

There is one last point to address before moving on in describing the next algorithm. In Equation 3.1 we assumed the application to be characterized by an ideal scalability, i.e., if we double the number of threads, the throughput will be doubled as well, which usually is not true. For this reason, the prediction will, in general, overestimate application’s performance. However, since the prediction is performed again until the requirement is satisfied, after few steps the algorithm will find the correct configuration. To better describe this process, we depict in Figure 3.1 how performance prediction changes in subsequent control steps in the bodytrack application.

We show one subplot for each control step of the algorithm. In each subplot, we show the throughput prediction for the next step and the actual throughput for different core numbers. We also show the minimum throughput we required in this specific test. The dot represents the configuration of the application in the specific step.

At step 0, the algorithm starts by using only one core. Since it predicts the throughput to scale linearly with the number of cores, it decides to use six cores at the next control step.
However, when it moves to step 1, it realizes the throughput was overestimated and that this configuration does not satisfy the user’s throughput requirement. The algorithm performs a new prediction, always assuming the throughput to be proportional to the number of threads, but this time using as a starting point the throughput measured with six cores. As we can see from the step 1 subplot, the prediction is still proportional to \( n \) but the slope changes, underestimating performance for lower number of cores and still overestimating it for a higher number of cores.

This process keeps going on until, at step 3, we reach a configuration with the proper throughput.

To summarize this process, we can notice that step by step the algorithm gets closer to the optimal configuration. This is because the throughput proportionality to \( n \) property holds locally to the current configuration but not for configurations distant from the current one.

There are still two considerations that we need to make. The first is that the number of steps does not significantly increase if the target configuration is distant from the starting one. As shown in Figure 3.2, even if the throughput requirement is higher (and thus the optimal configuration uses a higher number of cores), the algorithm is still able to reach it in 3-4 steps.

The second consideration is that if the throughput is not monotonically increasing (e.g., with 24 cores we have a throughput lower or equal than that with 20 cores), the algorithm could not work correctly. The algorithm we will present in the next section will be able, however, to manage this situation.
3.2 Online Learning Algorithm

In this section, we will describe an algorithm which, by adopting machine learning techniques, overcomes some limitations of the heuristic algorithm, allowing the user to express also explicit power consumption requirements. Moreover, it can also predict how performance and power consumption of the application will change if different mappings of the threads over the physical cores are applied. As we will show in Section 4.4, this algorithm may be slower than the heuristic in finding a proper configuration, and there are situations where the two algorithms complement each other.

As extensively described in Section 2.2.2.3, one of the main issues with current machine learning-based reconfiguration algorithms is that they require a long training phase to be performed offline, before the application starts its execution. Such training phase consists of collecting profile data for a broad set of different applications. These data will be used to train a machine learning model which, eventually, will be able to correlate changes in the number of cores and frequency to changes in performance and power consumption. The main issue with this approach is that, if an application has a performance/power characteristics different from those of the applications used to train the prediction models, these methods could produce sub-optimal solutions [137, 138, 14, 17]. For this reason, we will consider a worst-case scenario where no previous data is available or where previously collected data cannot be exploited to infer the models for new applications, due to significant differences in performance/power of the application. This is typical in cloud computing [22], where applications may have characteristics never experienced on other applications.
The target of this algorithm is to achieve an accuracy comparable to that obtained by offline algorithms but without using any previously collected training data. To take decisions, the algorithm will make predictions by only exploiting monitoring data about the current application and reconfiguring it on-the-fly. In principle, this can be a costly process, due to the data collection and to the cost to compute the learning model from scratch. However, as we will show in Section 4.3, we obtain an overhead comparable to offline technique and a better accuracy, while not requiring any previously collected profiling data.

First, in Section 3.2.1 we will describe the algorithm starting from the simplest scenario, i.e., there are no internal or external interferences that alter the application behavior and once a proper configuration has been found it will never be changed. Then, in Section 3.2.3 we describe how to improve the algorithm so that it can adapt the configuration of the application when internal and external conditions change during the application execution. Eventually, in Section 3.2.4 we explicitly consider streaming applications, showing why most existing algorithms are inappropriate and describing how we modified our online learning algorithm to manage streaming scenarios.

### 3.2.1 Stationary behaviour

To simplify exposition, in this section we consider the situation where once the algorithm finds a configuration, it will never change again during the execution. Then, we will extend the algorithm to consider a more general case where the best configuration can be re-evaluated due to changes in the application itself or to changes in the external environment.

![Figure 3.3: Different stages of our algorithm.](image-url)

We consider two distinct stages throughout the application execution: *training* phase and *steady* phase (see Figure 3.3). When the application starts, the algorithm starts the training phase, by executing the following steps:

1. The \( \langle n, f \rangle \) configuration of the application is changed, by selecting a configuration not yet visited.
2. The throughput and power consumption of the application are monitored for a short predefined period.

3. The monitored data is used to refine power consumption and throughput prediction models.

4. Throughput and power consumption of the current configuration are predicted by using such models. Predicted values are compared with real monitored values and, if the prediction error is lower than a specified threshold, the training phase finishes. Otherwise, the process is iterated.

Briefly, the algorithm is training machine learning models during the application execution, by using data about the application itself. Step 4 is based on the assumption that the error is uniformly distributed over all possible configurations so that, by looking at the prediction error of the current configuration, we may have an estimation of the error in all other possible configurations. We observed that this property holds in practice for the prediction models we will propose. However, if the error is not uniformly distributed, the effect is that the algorithm will potentially mispredict some configurations and will select a sub-optimal configuration.

When this process finishes, the models are used to predict throughput \( R \) and power consumption \( P \) for all the configurations and to select the best one according to the user requirements. This process is more formally described in Algorithm 2.

The \texttt{Main()} function first executes the training. The \texttt{Train()} function executes the 4 steps we described before\(^3\). Functions \texttt{Refine()} and \texttt{Predict()} will be described later in Section 3.2.2.

Concerning the \texttt{Plan()} function, it is executed after the training phase finishes, once we have accurate models for predicting throughput and power consumption of the application in all possible configurations. By using these models, the algorithm selects the best configuration according to the requirements given by the user and applies that configuration to the application, thus entering in the steady phase. For simplicity, in lines 7-10 we only show how to find the configuration with the lowest power consumption among those with a throughput higher than the required one. However, similarly to the heuristic case, the algorithm can be easily modified to consider the other possible requirements on metrics derived from throughput and power consumption (e.g. execution time, energy, energy-delay product (EDP), etc...). If according to the models there are no configurations that satisfy the requirements, the algorithm selects the closest one to the requirements (not shown in the algorithm for clarity’s sake).

It is worth noting that, during the training phase the application is running, and, when switching from training to steady phase, the application is neither

\(^3\)We inserted comments in the algorithm to associate the different parts to the 4 high-level steps described before.
Algorithm 2: Online learning algorithm - Stationary case

Function Plan()

1. \( P_{\text{min}} \leftarrow \infty \);
2. \( \lambda \leftarrow \text{GetArrivalRate}() \);
3. for all \( n, f, p \) do
4. \hspace{1em} \( R_{\text{pred}}, P_{\text{pred}} \leftarrow \text{Predict}(n, f, p) \);
5. \hspace{2em} if \( R_{\text{pred}} \geq R_{\text{req}} \) and \( P_{\text{pred}} < P_{\text{min}} \) then
6. \hspace{3em} \( P_{\text{min}} \leftarrow P_{\text{pred}} \);
7. \hspace{3em} best \( \leftarrow \langle n, f, p \rangle \);
8. end
9. end
10. return best

Function Train()

11. \( \epsilon_R \leftarrow \infty \);
12. \( \epsilon_P \leftarrow \infty \);
13. repeat
14. \hspace{1em} \( c \leftarrow \text{PickUnvisitedConfiguration}() \); \hspace{1em} // Step 1
15. \hspace{1em} Execute(c) ; \hspace{1em} // Step 1
16. \hspace{1em} Sleep(controlStep);
17. \hspace{1em} \( R_{\text{real}}, P_{\text{real}} \leftarrow \text{Monitor}() \); \hspace{1em} // Step 2
18. \hspace{1em} \( \text{Refine}(c, R_{\text{real}}, P_{\text{real}}) \); \hspace{1em} // Step 3
19. \hspace{1em} \( R_{\text{pred}}, P_{\text{pred}} \leftarrow \text{Predict}(c) \); \hspace{1em} // Step 4
20. \hspace{1em} \( \epsilon_R \leftarrow \left| \frac{R_{\text{pred}} - R_{\text{real}}}{R_{\text{real}}} \right| ; \hspace{1em} // Step 4
21. \hspace{1em} \( \epsilon_P \leftarrow \left| \frac{P_{\text{pred}} - P_{\text{real}}}{P_{\text{real}}} \right| ; \hspace{1em} // Step 4
22. until \( \epsilon_R < \text{threshold} \) and \( \epsilon_P < \text{threshold} \);

Function Main()

23. Train();
24. \( c \leftarrow \text{Plan}() \);
25. Execute(c);

stopped nor the previously computed results are discarded. However, during the training phase, the algorithm still does not have the throughput and power models, and thus it could force the application to run in configurations which might violate the requirements. For this reason, training phases should last as short as possible. Alternatively, it is possible to set a maximum duration for the training phase (number of training steps or absolute time) and to trade lower prediction accuracy with shorter training time. In Section 4.4 we will analyze the impact of the training phase. The algorithm could explore multiple configurations in parallel to shrink the duration of the training phase. We do not consider this possibility in our experiments due to the limited frequency scaling capabilities of our target architecture (we can partition the cores at most in two groups running at two different frequency). Another way to reduce training time is to exploit information about previously executed applications
However, as anticipated earlier, we are considering a worst-case scenario, where such information is not present or cannot be exploited, for example, because the application is too different from those used to train the prediction models.

3.2.2 The prediction algorithm

After showing how the training process works, we can focus on the Refine and Predict functions, which are used to build the prediction models and to perform predictions. We first show how to do that when the algorithm changes \( n \) and \( f \). Then, we extend the algorithm to consider different threads placement \( p \) as well.

To predict power consumption and throughput of the application, we decided to start with some general analytic models and to derive parameters specific to the application and architecture at runtime by using linear regression analysis. By using linear regression [178], we model the relationship between two or more independent variables (called predictors) and a dependent variable (called response) by fitting a linear equation to observed data. In our case, the predictors are the number of cores used by the application and their frequency, while the response is the throughput or the power consumption. Suppose we made a set of \( n \) observations to obtain the values of the responses \( y_1, y_2, \ldots, y_n \). Let \( x_i = x_{i,1}, x_{i,2}, \ldots, x_{i,s} \) denote the \( s \) predictors for the observation \( i \). Then we have:

\[
y_i = a_0 + a_1 x_{i,1} + \cdots + a_s x_{i,s} + \epsilon_i
\]  

(3.5)

where \( a_i \) is a regression coefficient and \( \epsilon_i \) is a term representing a random error due to measurement error or fluctuations in the results.

By fitting a regression model to observations, we determine the \( a_i \) coefficients, thus enabling the prediction of the responses for the unobserved predictors. To fit the model, we use the least squares method, minimizing the sum of the squares of the residuals, where a residual is the difference between the real value of the dependent variable and the value predicted by the model.

In our algorithm, the Refine routine will add a new observation and the Prediction routine will use the observations we made to derive the \( a_i \) coefficients. To let the process work, we need to express throughput and power consumption in a form similar to the one in Equation 3.5, where \( a_i \) coefficients are architecture and application-specific values. Moreover, we need to express them such they are only dependent on the number of used cores \( n \) and from the frequency \( f \).

Concerning the throughput, we propose two different prediction models. The first one (Section 3.2.2.1) is based on Amdahl’s Law [179], while the second one (Section 3.2.2.2) is based on the Universal Scalability Law [180]. Eventually, in Section 3.2.2.3 we will describe the power consumption prediction model. Then, in Section 3.2.2.4 we extend the models to consider different threads placements \( p \).
3.2. ONLINE LEARNING ALGORITHM

3.2.2.1 Amdahl’s Law

According to the *Amdahl’s Law* [179], we can model the throughput $R$ of a parallel application as:

$$R(n) = R(1) \cdot \frac{n}{(E \cdot n) + (1 - E)}$$

(3.6)

where $E$ is the percentage of the application that is strictly sequential.

We can generalize this equation also to consider the effects of frequency scaling on the execution time. Similarly to what we did for the heuristic (Section 3.1), we consider the throughput to be proportional to the clock frequency, i.e., if the frequency is doubled, throughput is doubled as well.

$$R(n, f) = R(1, f_{\text{min}}) \cdot \frac{n}{(E \cdot n) + (1 - E)} \cdot \frac{f}{f_{\text{min}}}$$

(3.7)

To bring back this equation to a form similar to Equation 3.5, we consider the reciprocal on both sides of the equation, i.e.

$$\frac{1}{R(n, f)} = \frac{1}{R(1, f_{\text{min}})} \cdot \frac{f_{\text{min}}}{f} \left( E + \frac{(1 - E)}{n} \right) =$$

$$= \frac{1}{R(1, f_{\text{min}})} \frac{f_{\text{min}}}{f} (E) +$$

$$+ \frac{1}{R(1, f_{\text{min}})} \frac{f_{\text{min}}}{f \cdot n} (1 - E) =$$

$$= a_1 \frac{f_{\text{min}}}{f} + a_2 \frac{f_{\text{min}}}{f \cdot n}$$

(3.8)

We use this form of the equation to compute the linear regression. Then, we get the throughput prediction by calculating its reciprocal.

This model has two issues. The first one is that, like the heuristic, it assumes the performance to be proportional to the frequency. The second problem is that it expects the performance to be monotonically increasing with the number of cores. This is not always true since in some cases, due to contention on shared resources, using fewer cores than the maximum could lead to better performances. We will try to solve both these problems with the next performance prediction model.

3.2.2.2 Universal Scalability Law

According to the *Universal Scalability Law* [180], we can model the throughput as:

$$R(n) = R(1) \cdot \frac{n}{1 + \alpha \cdot (n - 1) + \beta \cdot n \cdot (n - 1)}$$

(3.9)
where $\alpha$ and $\beta$ are application dependent constants, representing two possible issues that limit the achievable throughput. $\alpha$ represents the weight of the contention on shared data, while $\beta$ represents the cost needed to keep the writable data coherent. Differently from the Amdahl’s Law, considering these two factors allows the model to fit situations where the throughput is not monotonically increasing with the number of cores $n$.

Even in this case, to simplify the calculation we consider the reciprocal $\frac{1}{R(n)}$:

$$\frac{1}{R(n)} = \frac{1}{R(1) \cdot n} + \frac{\alpha \cdot (n-1)}{R(1) \cdot n} + \frac{\beta \cdot (n-1)}{R(1)} = a_1 \frac{1}{n} + a_2 \frac{n-1}{n} + a_3 (n-1)$$

Then, after we use this equation to get a prediction, we compute its reciprocal to get the throughput $R(n)$. Note that in this equation we assumed not to know $R(1)$, since we inserted it into the $a_i$ elements that will be predicted by the regression model. Alternatively, since we need to perform the training phase anyway, we could obtain the value of $R(1)$ during the training by forcing the `PickUnvisitedConfiguration()` function to return the configuration $<1, f_{\text{min}}>$ when it is called for the first time. In this way, we can exclude it from the $a_i$ so that the regression model will only predict the actual unknown values (i.e., $\alpha$ and $\beta$). We will consider the effect of this optimization in Section 4.3.

Moreover, Equation 3.10 does not depend on the frequency $f$. We now describe how to predict the effect of frequency scaling. First, we use Equation 3.10 to predict throughput only for configurations running at the minimum frequency $f_{\text{min}}$. Then, we monitor the throughput $R(n, f_{\text{max}})$ and we compute $\gamma = \frac{R(n, f_{\text{max}})}{R(n, f_{\text{min}})}$. Now we can compute the throughput for any configuration where $f = f_{\text{max}}$ as

$$R(n, f_{\text{max}}) = R(n, f_{\text{min}}) \cdot \gamma$$

To compute throughput for frequencies different from $f_{\text{max}}$ and $f_{\text{min}}$, we observe that the throughput has a linear relationship with the frequency [181, 182]. Therefore, for a given $n$, we can express $R(n, f) = a + b \cdot f$. Note that in this case, by using the factor $a$, we are explicitly considering situations where some parts of the computation are not directly affected by the frequency scaling\(^4\). Since we already know $R(n, f_{\text{min}})$ and $R(n, f_{\text{max}})$, we can obtain the $R(n, f)$ by using the equation for a line passing through two given points. In our case, where the two points $[x_1, y_1]$ and $[x_2, y_2]$ are $[f_{\text{min}}, R(n, f_{\text{min}})]$ and $[f_{\text{max}}, R(n, f_{\text{max}})]$, we get:

$$R(n, f) = \frac{R(n, f_{\text{max}}) - R(n, f_{\text{min}})}{f_{\text{max}} - f_{\text{min}}} \cdot (f - f_{\text{min}}) + R(n, f_{\text{min}})$$

\(^4\)On the contrary, for the Amdahl’s Law we had $R(n, f) = b \cdot f$ (Equation 3.7).
3.2. ONLINE LEARNING ALGORITHM

3.2.2.3 Power Consumption Model

Concerning the power consumption model, we start from the same model we used for the heuristic algorithm (Section 3.1). We briefly recall it here for clarity. According to [37, 38, 39], the power consumption of an active CPU can be modelled as:

\[ P(n, f, v) = vI_{\text{leak}} + Acv^2fn \]  

(3.12)

where \( v \) is the voltage, \( I_{\text{leak}} \) is the leakage current, \( A \) is the activity factor and \( c \) is the capacitance. For our purposes, we can consider \( A, c \) and \( I_{\text{leak}} \) as constants. Differently from the heuristic case, now we will ignore neither the static power \((vI_{\text{leak}})\) nor the \( A \) and \( c \) constants, and we will try to derive all of them dynamically with our regression algorithm.

First of all, we need to consider the case where more CPUs are present on the system since static power \( vI_{\text{leak}} \) must be accounted for each of them. To further reduce power consumption, we can run the unused CPUs at the minimum available frequency \( f_{\text{min}} \), with a corresponding operating voltage \( v_{\text{min}} \). Since for unused CPUs we have \( n = 0 \), we do not account dynamic power, and their power consumption is just the static one, i.e., \( v_{\text{min}}I_{\text{leak}} \). If \( k \) and \( \bar{k} \) are the number of active and inactive CPUs respectively, \( f \) is the frequency of active CPUs and \( v \) is the voltages of the active CPUs, then we have:

\[ P(n, f, k, \bar{k}) = kV(f)I_{\text{leak}} + (K - k)v_{\text{min}}I_{\text{leak}} + AcV(f)^2fn \]  

(3.13)

To remove the dependency from \( v \), we can observe again that the voltage is strictly correlated to the frequency, since by increasing the frequency we raise the operating voltage and vice versa. Consequently, we can use a tabular function \( V(f) \) to get the voltage value associated to a specific frequency level \( f \). This function is architecture specific and may be obtained programmatically or by using the values provided by the CPU vendor. We can then rewrite equation 3.13 as:

\[ P(n, f, \bar{k}, \bar{k}) = \bar{k}V(f)I_{\text{leak}} + kV_{\text{min}}I_{\text{leak}} + AcV(f)^2fn \]  

(3.14)

Let \( K \) be the number of CPUs available on the target architecture, then we have \( k = K - \bar{k} \).

\[ P(n, f, \bar{k}) = \bar{k}V(f)I_{\text{leak}} + (K - \bar{k})v_{\text{min}}I_{\text{leak}} + AcV(f)^2fn = \]
\[ = I_{\text{leak}}(\bar{k}V(f) - v_{\text{min}}) + Kv_{\text{min}}) + AcV(f)^2fn \]  

(3.15)

Since the number of active CPUs depends on the specific thread placement, we will discuss how to remove the dependency from \( \bar{k} \) in Section 3.2.2.4.
3.2.2.4 Different threads placements

Some applications may be sensitive to thread placement (also known as thread-to-core affinity).

We will consider in this work two different types of threads placements: linear and interleaved. Linear placement minimizes the number of used CPUs, by placing a thread on a new CPU only if all the cores on the used CPUs have been already allocated to other threads. On the other hand, interleaved placement minimizes the number of resources shared by threads, by allocating one thread per CPU in a round-robin way. For example, suppose to have a machine with 2 CPUs, each one with 4 cores on top and that the algorithm must place 3 threads over them. In the linear case (Figure 3.4a) it will place all 3 threads on the first CPU (one per core) while with an interleaved placement (Figure 3.4b), the algorithm will allocate 2 threads on the first CPU and 1 thread on the second one.

The difference between these two cases is that, for example, due to contention on shared caches, some memory intensive applications may benefit from a the interleaved placement. However, as we described in Section 2.1.1.1,
3.2. ONLINE LEARNING ALGORITHM

this arrangement is more power consuming since a CPU may enter the deepest C-state and shut down its last level cache only if none of its cores is active.

The algorithm will simultaneously derive one performance model for the linear case and one for the interleaved case, to predict the performance of these two different alternatives. During the training phase, configurations characterized by a linear placement will be used to refine the model for the linear case while configuration using an interleaved threads placement will be used to improve the other performance model. When the optimal configuration must be selected, both models will be evaluated.

A similar process is performed for the power consumption model. However, in this case, the number of used/unused CPUs changes according to the specific thread placement. Let $k$ be the number of cores available for each CPU. For the linear placement we have:

$$\bar{k} = \left\lceil \frac{n}{k} \right\rceil$$

(3.16)

While for the interleaved placement we have:

$$\bar{k} = \min(n, K)$$

(3.17)

These two values correspond to minimizing and maximizing the number of used CPUs for a given $n$.

Eventually, in case of linear placement, we may rewrite Equation 3.15 as:

$$P(n, f) = I_{leak} \left[ \frac{n}{k} \left( V(f) - v_{min} \right) + Kv_{min} \right] + AcV(f)^2 fn =$$

$$= a_0 \left[ \frac{n}{k} \left( V(f) - v_{min} \right) + Kv_{min} \right] + a_1 V(f)^2 fn$$

(3.18)

For interleaved placement, Equation 3.15 can be rewritten as:

$$P(n, f) = I_{leak} \left[ \min(n, K) \left( V(f) - v_{min} \right) + Kv_{min} \right] + AcV(f)^2 fn =$$

$$= a_0 \left[ \min(n, K) \left( V(f) - v_{min} \right) + Kv_{min} \right] + a_1 V(f)^2 fn$$

(3.19)

We would like to point out that this approach to application placement is general and can be extended to consider other threads placements as well. However, since it requires building a different model for each placement, it could be not the best solutions if many different placements are considered. In such cases, it would probably be better using other approaches which are not based on an analytical model and which could capture the effect of different placements without the need to model them explicitly [14]. However, as we will show in Section 4.3.2, the effect of considering different threads placement is often negligible, at least for the applications we considered in our evaluation.
### 3.2.3 Optimal Configuration Re-evaluation

Up to now we only considered a situation where, once the best solution is found, it never changes during the application execution. However, this is not realistic since, in general, internal or external disturbances may let the once optimal configuration not optimal anymore. One typical case concerns applications characterized by different phases. For example, an application may have a first I/O bound phase when it loads some big file from the disk, and then a compute-intensive phase when the real calculation is done. Of course, the power and performance characteristics of these two phases may be completely different, and the prediction models we computed for the I/O bound phase may be inappropriate for CPU bound phase. In principle, this means that, when the algorithm detects that the application entered a different phase, a new training must be performed to obtain new prediction models. Fortunately, as shown in [71] many real applications are characterized by a small number of phases (4-8). Thus the number of different training stages to perform is usually limited and would not significantly impact application performance. Algorithm 3 shows how the algorithm is modified to consider the possibility of re-training the prediction models.

**Algorithm 3**: Online learning algorithm - General case

```
1. firstRun ← true;

2. Function TrainingNeeded()
   3. if firstRun then
      4. firstRun ← false;
      5. return true;
   6. end
   7. if PhaseChanged() then
      8. return true;
   9. end
  10. return false;

11. Function AnalyzeAndPlan()
   12. if TrainingNeeded() then
      13. Train();
      14. return Plan();
   15. end
   16. return currentConfiguration;

17. Function Main()
   18. while true do
      19. Sleep(controlStep);
      20. \( R_{\text{real}}, P_{\text{real}} \) ← Monitor;
      21. c ← AnalyzeAndPlan();
      22. Execute(c);
   23. end
```
3.2. ONLINE LEARNING ALGORITHM

We introduce a new function, $\text{PhaseChanged}$, which can tell us if the application entered a different phase. Several techniques have been recently proposed [61, 62] to detect such situation. These methods can be applied online while the application is running, thus are particularly suitable for our purposes. Moreover, it is possible to detect if the current phase was already experienced in the past. In that case, it is possible to use the same models previously computed for the same phase, instead of training again the models from scratch.

Figure 3.5 shows how the online learning algorithm behaves when an application has different phases. The application is composed of three different phases: one between 0 and 40 seconds, one between 40 and 100 seconds and the last one between 100 and 140 seconds. We can see from the figure the trainings occurring around 0, 40 and 100 seconds from the beginning, causing the throughput to fluctuate since different configurations are explored. Moreover, as anticipated earlier, we can also see that while training the system cannot guarantee that the requirement is satisfied.

![Figure 3.5: Reaction of the online learning algorithm to phase changes.](image)

External interferences may lead as well to the re-evaluation of the optimal configuration. For example, consider the case where the models were computed when the application was the only application running in the system. If, while the application is running, another application is started, the computing resources would now be shared between the two applications, and the power and performance models will most likely be not valid anymore. External interferences lead to a behavior similar to the one we showed in Figure 3.5.

3.2.4 Streaming Applications

In streaming applications, there is another situation which may lead to performance fluctuations. Indeed, the performance of streaming application depends not only on the intrinsic characteristic of the application and of the computing node they are running on but also from the rate at which the data to be processed is generated. In a nutshell, if no data are sent to the streaming application, we will experience a throughput of 0 elements per second, even if the application would be able to process much more data than that. Before moving forward, let us describe such concepts more formally.

Without loss of generality, we assume that the considered parallel application can be modeled as a queueing network [183]. Briefly, the application is seen as a graph whose nodes are threads (service centers), and edges among
threads are communication channels implemented as queues. We characterize such queueing system with few metrics described in Table 3.1. In general, both the power consumption and the service rate depend on the amount of resources used by the application. Since in this work we are considering the number of physical cores $n$ used by the application, their frequency $f$ and the threads placement $p$, when needed we will denote the service rate as $\mu(n, f, p)$ and the power consumption as $P(n, f, p)$.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Time $T_S$</td>
<td>Average time between the beginning of the executions on two consecutive elements to be processed.</td>
</tr>
<tr>
<td>Service Rate $\mu$</td>
<td>Inverse of the service time, i.e. the average number of elements that can be processed by the application per time unit $\mu = \frac{1}{T_S}$.</td>
</tr>
<tr>
<td>Interarrival Time $T_A$</td>
<td>Average time between the reception of two consecutive elements to be processed by the application.</td>
</tr>
<tr>
<td>Arrival Rate $\lambda$</td>
<td>Inverse of the interarrival time, i.e. the average number of elements received per time unit $\lambda = \frac{1}{T_A}$.</td>
</tr>
<tr>
<td>Utilization $\rho$</td>
<td>Ratio between the arrival rate and the service rate $\rho = \frac{\lambda}{\mu}$.</td>
</tr>
<tr>
<td>Throughput $R$</td>
<td>Average number of elements produced by the application per time unit. When $\rho &lt; 1$ (i.e. $\lambda &lt; \mu$), the system can manage all the arriving requests. In such case, we have $R = \lambda$. If this is not the case, the system is saturated and we have $R = \mu$. In general, we have $R = \min(\lambda, \mu)$.</td>
</tr>
<tr>
<td>Power Consumption $P$</td>
<td>Power consumed by the application.</td>
</tr>
</tbody>
</table>

Table 3.1: Basic metrics that characterize a parallel application.

Figure 3.6: Left: Modelling of the application with respect to the metrics of interest. Right: A possible behaviour of interarrival rate, service rate and throughput with respect to the amount of used number of cores $n$.

On the left part of Figure 3.6, we characterize the application (the square box) through its service rate $\mu$. The application receives data to be processed at a rate $\lambda$, and has a throughput of $R$ which, of course, will be the minimum between $\lambda$ and $\mu$. Indeed, if $\lambda > \mu$, that the application is not able to process all the incoming data, and the experienced throughput will be equal to $\mu$. On the other hand, if $\lambda < \mu$, the application is underutilized and could process (per time unit) more data than what it is receiving. However, the experienced throughput will be equal to $\lambda$ since the application cannot process more data than what it is receiving. On the right part of the figure we show how these metrics change for different numbers of cores $n$. We consider $\lambda$ to be independent from $n$ since it only depends on how fast data is generated by the “exter-
nal world”, while the throughput is always the minimum between λ and μ. Is quite common, in streaming applications, to have a different interarrival rate λ during application execution. If we consider for example a network monitoring application, the interarrival rate will be higher during working hours, since there will be more people using the network, thus producing more data, while it will be lower during the night. Every time λ changes, the throughput will change as well. Since fluctuations in throughput will most likely be detected as phase changes [61, 62], this could have a dramatic effect on the performance of the algorithm, since every time the algorithm enters a different phase, a new training needs to be performed. To show this effect, we applied the algorithm over a network monitoring application [184], where we required the algorithm to enforce $\mu = \frac{\lambda}{\rho}$ (i.e. $\rho = 0.9$). The application we consider performs Deep Packet Inspection (DPI), analyzing all the HTTP packets traveling over the network, searching for well-known patterns identifying possible security threats. To test our algorithm in a realistic case, we generated network traffic with arrival rates equal to those of a real backbone network 5.

The results are shown in Figure 3.7 where we can see that throughput has many fluctuations due to the too many re-trainings.

![Figure 3.7: Instability of the algorithm for arrival rate fluctuations. Throughput is expressed in thousands of packets per second.](image)

Such instability is common in many existing algorithms, which do not explicitly consider streaming applications. As we previously described, fluctuations in the throughput $R$ may be detected as phase changes. To address this problem, let us consider the right part of Figure 3.6 again. Intuitively, when the arrival rate changes, the $\lambda$ line moves up or down and. Consequently, the throughput curve $R$ changes and we need to do another training phase to learn how the throughput changes with the number of cores $n$. However, the service rate $\mu$ is not affected by fluctuations in the arrival rate $\lambda$. Accordingly, instead of predicting the throughput $R$ and considering as phase changes fluctuations of $R$, we could predict the service rate $\mu$ and consider as phase changes fluctuations of $\mu$. By doing so, even if $\lambda$ changes, the service rate $\mu$ will be stable.

---

and we will not need to perform a new training stage. To do that, it is sufficient to replace in the models presented in Section 3.2.2.1 and Section 3.2.2.2 the throughput \( R \) with the service rate \( \mu \).

Accordingly, we need to change function \( \text{Plan()} \) in Algorithm 2 with the one in Algorithm 4.

**ALGORITHM 4:** Online learning algorithm - Streaming application scenario

```plaintext
Function \( \text{FindLowestRhoConf()} \)
\[ \mu_{\text{max}} \leftarrow 0; \]
\[ \text{forall } n, f, p \text{ do} \]
\[ \mu_{\text{pred}}, P_{\text{pred}} \leftarrow \text{Predict}(n, f, p); \]
\[ \text{if } \mu_{\text{pred}} > \mu_{\text{max}} \text{ then} \]
\[ \mu_{\text{max}} \leftarrow \mu_{\text{pred}}; \]
\[ \text{best} \leftarrow n, f, p; \]
\[ \text{end} \]
\[ \text{end} \]
\[ \text{return best;} \]

Function \( \text{GetArrivalRate()} \)
\[ \mu_{\text{real}}, P_{\text{real}}, \rho_{\text{real}} \leftarrow \text{Monitor}(); \]
\[ \text{if } \rho_{\text{real}} < 1 \text{ then} \]
\[ \text{return } \rho_{\text{real}} \cdot \mu_{\text{real}} \]
\[ \text{else} \]
\[ c \leftarrow \text{FindLowestRhoConf}(); \]
\[ \text{Execute}(c); \]
\[ \text{Sleep(controlStep)}; \]
\[ \mu_{\text{real}}, P_{\text{real}}, \rho_{\text{real}} \leftarrow \text{Monitor}(); \]
\[ \text{if } \rho_{\text{real}} < 1 \text{ then} \]
\[ \text{return } \rho_{\text{real}} \cdot \mu_{\text{real}} \]
\[ \text{else} \]
\[ \text{return } \infty; \]
\[ \text{end} \]

Function \( \text{Plan()} \)
\[ P_{\text{min}} \leftarrow \infty; \]
\[ \lambda \leftarrow \text{GetArrivalRate}(); \]
\[ \text{forall } n, f, p \text{ do} \]
\[ \mu_{\text{pred}}, P_{\text{pred}} \leftarrow \text{Predict}(n, f, p); \]
\[ R_{\text{pred}} \leftarrow \min(\lambda, \mu_{\text{pred}}); \]
\[ \text{if } R_{\text{pred}} \geq R_{\text{req}} \text{ and } P_{\text{pred}} < P_{\text{min}} \text{ then} \]
\[ P_{\text{min}} \leftarrow P_{\text{pred}}; \]
\[ \text{best} \leftarrow n, f, p; \]
\[ \text{end} \]
\[ \text{return best} \]
```

Let us start describing the algorithm from function \( \text{Plan()} \). The function is
very similar to the previous one. The only difference is that now the `Predict()` functions returns the service rate $\mu$ instead of the throughput $R$. Then, we can obtain the throughput prediction from the service rate prediction and the arrival rate $\lambda$ by computing $R(n, f, p) = \min(\lambda, \mu(n, f, p))$.

According to queueing theory, $\lambda = \mu(n, f, p) \cdot \rho(n, f, p)$. From a practical perspective, obtaining $\lambda$ may be a non-trivial process. For example, consider a scenario where we have some sensors sending streaming data to an application for further analysis. If the data is generated at a rate $\lambda$ higher than the service rate $\mu$ of the application, the application would not be able to process all the incoming data and a fraction of data will be dropped, for example, by the network interface. This means that, eventually, the application will always receive data at a rate at most equal to $\mu$. Knowing the exact rate at which data is generated would indeed require interacting not only with the application but also (in our example) with the sensors generating the data which is processed by the application. However, to be as more general as possible we want to concentrate our interaction only with the controlled application.

Since we always measure $\lambda \leq \mu$, when $\rho = 1$, it could either be 1 or greater than one and we cannot distinguish between these two cases. This is not an issue since we do not need to know the exact $\lambda$, but just to know if $\lambda$ is lower than $\mu(n, f, p)$ (and thus we will have $R(n, f, p) = \lambda$) or if $\lambda$ is greater than $\mu(n, f, p)$ (and thus we will have $R(n, f, p) = \mu(n, f, p)$). To obtain $\lambda$, we use the `GetArrivalRate()` function. If $\rho(n, f, p) < 1$, then we have $\lambda = \mu(n, f, p) \cdot \rho(n, f, p)$ (lines 13-14). If $\rho(n, f, p) = 1$, we cannot really say anything about $\lambda$. However, before giving up, we can set the application on the configuration with the lowest $\rho$, and check if on that configuration we have $\rho(n, f, p) < 1$. If not even this configuration has a utilization lower than 1, then we can consider $\lambda = \infty$ (line 23). Indeed, independently of its real value, it will always be greater than $\mu$ for every $(n, f, p)$ triplet. To find the configuration with the lowest utilization $\rho$, it is sufficient to find the configuration with the highest service rate, and we can do that since we already computed the service rate prediction model (lines 1-10).

Eventually, we need to modify the `AnalyzeAndPlan()` function to distinguish phase changes from changes in the arrival rate, as shown in Algorithm 8.

**Algorithm 5:** Online learning algorithm - Streaming application scenario

```
Function AnalyzeAndPlan()
    if TrainingNeeded() then
        Train();
    end
    if TrainingNeeded() or ArrivalRateChanged() then
        return Plan();
    end
    return currentConfiguration;
```

We then executed the algorithm over the network monitoring application
again, and we report the results in Figure 3.8. As we can see from the bottom part of the figure, in this case, the algorithm has a much more stable behavior, since the training is only done once at the beginning of the execution. During the execution, the configuration is often changed but without the need to train the model from scratch. This leads to a reduction of 98% in the number of reconfiguration and a 34% improvement in the performance over the execution span. On the top part of the figure, we show that the algorithm satisfied the specified requirement ($\rho < 0.9$). Arrival rate is not shown since for this test it always matches the throughput (since we enforced $\rho < 0.9$).

![Figure 3.8: Time behavior of an application in presence of arrival rate fluctuations after the optimization of the algorithm. Throughput is expressed in thousands of packets per second.](image)

Eventually, we would like to point out that although we described these modifications to manage the streaming applications scenario, the very same algorithm also works for non-streaming applications. Indeed, in that case, we will usually have $\rho = 1$ and $\lambda = \infty$ and the algorithm will fallback to the one we have shown before we started describing streaming applications.

### 3.3 Summary

In this Chapter we presented two algorithms (Online Learning and Heuristic) which can be used to enforce performance and power consumption requirements on parallel applications running on shared memory multicore machines. The main novelty of the Online Learning algorithm is that it exploits machine learning techniques but without using any previously collected data, by just exploiting data collected online while the application is running. We
proposed different variations, according to the performance prediction model used, which we will evaluate in Section 4.3.

It is worth noting that, since our Online Learning algorithm is model-driven, managing other knobs would require explicitly modelling the effect of these knobs on both performance and power consumption. Moreover, to model other metrics (e.g., system temperature, memory utilization, etc...) we should introduce a separate model for each of them. In such cases, it could be better using other approaches which do not require an explicit modelling of the effect of the knobs on the different metrics [14]. However, as we described in 2.2.2.3, these solutions usually require long training phase to be performed online before the application execution, and they can only predict behaviours that were already experienced during the offline training.

In Chapter 4 we will validate our algorithms, by executing them over a wide set of real applications, by enforcing different performance and power-consumption requirements and by comparing them with some state of the art solutions.
Chapter 4

Evaluation

In this chapter, we evaluate the algorithms we presented in Chapter 3, by executing them over a wide set of real applications, enforcing different power consumption and performance requirements and by comparing them with some state of the art solutions.

The rest of this Chapter is structured as follows. In Section 4.1 we describe the testing environment. Section 4.2 will introduce the methodology we used for evaluating the reconfiguration algorithms. Then, in Section 4.3 we evaluate different choices for our online learning algorithm. This will allow us to select the best prediction models for the algorithm. Eventually, in Section 4.4 we compare the algorithms we designed with some state of the art reconfiguration strategies.

4.1 Test Environment

We conducted all the experiments on an Intel workstation with 2 Xeon E5-2695 @2.40GHz CPUs, each with 12 2-way hyperthreaded cores, running with Linux x86_64. This machine has 13 possible frequency levels: from 1.2GHz to 2.4GHz with 0.1GHz steps.

As testbed, we consider all the applications in the PARSEC benchmark. PARSEC [23] (Princeton Application Repository for Shared-Memory Computers)\(^1\) is a collection of various multi-threaded programs with high system requirements that has been used in the past for stressing shared-memory architectures [185]. One of the most interesting aspects of this benchmark suite is that it covers a wide set of application domains such as streaming, scientific computing, computer vision, data compression and so forth. For this reason,

\(^{1}\)In this thesis we refer to the PARSEC version 3.0: http://parsec.cs.princeton.edu/overview.htm
the PARSEC suite has been recently used to assess the expressive power of emerging parallel programming frameworks [186].

PARSEC consists of 13 programs from different areas of computing. Each application is provided with several input sets for each benchmark. Three datasets, with different sizes, target the execution on simulators (i.e. sim-small, sim-medium, sim-large), while the native dataset is representative of a realistic execution scenario of the application.

From the parallel programming perspective, PARSEC applications are of great interest for testing frameworks because they have different memory access behaviors, data sharing patterns, amount of parallelism, computational granularity, and synchronization frequency. This heterogeneity it was possible to validate our approach on a wide range of real-world scenarios, since these applications cover a vast spectrum of power consumption and perfor-

Figure 4.1: Scalability and power consumption of the PARSEC applications for different numbers of threads.
4.1. TEST ENVIRONMENT

mance behaviors, as shown in Figure 4.1. Table 4.1 reports the official name of the benchmarks and their main characteristics according to PARSEC documentation. Moreover, we show the official parallel versions released within the PARSEC suite that we will use in Section 7.2 as reference implementations for our comparison.

Concerning the parallelism exploitation model, most of the applications belong to the data parallelism model, where the computation is performed on large data structures logically partitioned among multiple threads. Stream parallelism characterizes applications where a long sequence of data items is processed by a chain of threads that execute distinct computation phases on different items in parallel and in a pipeline fashion. The case of canneal is an example of applications that do not straightforwardly follow any common parallelism paradigm (in the table it is referred to as unstructured).

For all the benchmarks we considered the Pthreads version, except for frequent which only provides the OpenMP version. Among the different input sizes provided by PARSEC, we have chosen the native one, which resembles a real-world execution of the applications. The number of threads \( t \) to be used by the application have been selected with the \( -n t \) parameter provided by the parsecmgmt tool. In our experiments, we do not consider Simultaneous Multithreading (SMT), (e.g., Intel's Hyperthreading). Accordingly, we run the applications with a number of threads equal to the number of physical cores available on the machine. The only exceptions are facesim and fluidanimate, which only allow some specific number of threads. To be specific, facesim only allows 1, 2, 3, 4, 6, 8, 16 values, while fluidanimate only allows 1, 2, 4, 8, 16 values. Since the machine we used for our experiments only has 24 physical cores, and since we consider, as starting configuration, the case where we have at most one thread on each physical core, we run both facesim and fluidanimate with 16 threads. In all the presented results, we only considered the time spent in the parallel sections of the applications, without considering initialization and cleanup phases. This is a common assumption (as in [15] and [53]) motivated by the fact that the main focus of our algorithms is the optimization of the parallel phase. It is worth noting that initialization and cleanup of the algorithm are still included and that we just do not include the serial parts of the application (e.g., loading the data-set from the disk).

The algorithms we presented and the already existing algorithms that will be used in our comparison have been implemented inside Nornir, the framework we will describe in detail in Part IV. We would like to stress the fact that, thanks to the framework, it was possible to execute the self-adaptive algorithms on running applications and that the evaluation is not performed on simulated runs, as common in many self-adaptive approaches [144, 16, 52, 119, 145]. As we anticipated in Chapter 2, self-adaptive reconfiguration algorithms only work on iterative computations, and all PARSEC applications exhibit some iterative behavior. To allow the framework to monitor the actual performance of the applications, we modified them by inserting some instrumentation calls (4 additional lines of code for each application). Some of these calls will be invoked at each loop iteration, to send monitored performance
### Table 4.1: Classification and characteristics of the PARSEC v3.0 applications.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Domain</th>
<th>Parallelism</th>
<th>Parallel Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Model Grain Pthreads OpenMP TBB</td>
<td></td>
</tr>
<tr>
<td>blackscholes</td>
<td>Financial Analysis</td>
<td>data parallelism</td>
<td>coarse</td>
</tr>
<tr>
<td>bodytrack</td>
<td>Computer Vision</td>
<td>data parallelism</td>
<td>medium</td>
</tr>
<tr>
<td>canneal</td>
<td>Engineering</td>
<td>unstructured</td>
<td>fine</td>
</tr>
<tr>
<td>dedup</td>
<td>Enterprise Storage</td>
<td>stream parallelism</td>
<td>medium</td>
</tr>
<tr>
<td>facesim</td>
<td>Animation</td>
<td>data parallelism</td>
<td>coarse</td>
</tr>
<tr>
<td>ferret</td>
<td>Similarity Search</td>
<td>stream parallelism</td>
<td>medium</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>Animation</td>
<td>data parallelism</td>
<td>fine</td>
</tr>
<tr>
<td>freqmine</td>
<td>Data Mining</td>
<td>data parallelism</td>
<td>medium</td>
</tr>
<tr>
<td>raytrace</td>
<td>Computer Vision</td>
<td>data parallelism</td>
<td>medium</td>
</tr>
<tr>
<td>streamcluster</td>
<td>Data Mining</td>
<td>data parallelism</td>
<td>medium</td>
</tr>
<tr>
<td>swaptions</td>
<td>Financial</td>
<td>data parallelism</td>
<td>coarse</td>
</tr>
<tr>
<td>vips</td>
<td>Media Processing</td>
<td>data parallelism</td>
<td>coarse</td>
</tr>
<tr>
<td>x264</td>
<td>Media Processing</td>
<td>stream parallelism</td>
<td>coarse</td>
</tr>
</tbody>
</table>

(e.g., number of iterations performed per time unit) to an external manager (which executes the reconfiguration algorithm). Details about the instrumentation process can be found in Section 8.2. In Table 4.2 we show what an iteration is, according to the specific instrumentation we made. Note that this
reflects directly on the way in which the user expresses performance requirements, since throughput requirements will be expressed in terms of \( \text{iterations second} \). Consequently, it would be possible for example for \texttt{blackscholes} to express requirements about the number of stock options to be processed per second.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{blackscholes}</td>
<td>1 Stock Option</td>
</tr>
<tr>
<td>\texttt{Bodytrack}</td>
<td>1 Frame</td>
</tr>
<tr>
<td>\texttt{Canneal}</td>
<td>1 Move</td>
</tr>
<tr>
<td>\texttt{Dedup}</td>
<td>1 Chunk</td>
</tr>
<tr>
<td>\texttt{Facesim}</td>
<td>1 Frame</td>
</tr>
<tr>
<td>\texttt{Ferret}</td>
<td>1 Query</td>
</tr>
<tr>
<td>\texttt{Fluidanimate}</td>
<td>1 Frame</td>
</tr>
<tr>
<td>\texttt{Freqmine}</td>
<td>1 Call of the \texttt{FP_growth} function</td>
</tr>
<tr>
<td>\texttt{Raytrace}</td>
<td>1 Frame</td>
</tr>
<tr>
<td>\texttt{Streamcluster}</td>
<td>1 Evaluation for opening a new cluster center</td>
</tr>
<tr>
<td>\texttt{Swaptions}</td>
<td>1 Simulation</td>
</tr>
<tr>
<td>\texttt{Vips}</td>
<td>1 Image Tile</td>
</tr>
<tr>
<td>\texttt{X264}</td>
<td>1 Frame</td>
</tr>
</tbody>
</table>

Table 4.2: \textit{Iteration} meaning in PARSEC benchmarks, according to the instrumentation we performed.

As we will show in Section 8.1, in principle we can control the applications without instrumenting them, by relying on hardware performance counters (e.g. \textit{Instructions Per Second} (IPS)). However, in that case, performance requirements should be expressed by the user in IPS. We decided to focus our evaluation on instrumented applications since we believe that is much more intuitive for the application user to express requirements regarding \textit{frames per second} or \textit{query per second} than expressing them in terms of IPS. These modified PARSEC applications have been added to the P\textsuperscript{3}ARSEC benchmark suite (which will be described in Section 7.1) and can be seamlessly executed by using the standard tools already provided by PARSEC. For example, to run the \texttt{blackscholes} application with specific power consumption and performance requirements, the user should create, in the PARSEC root directory, an XML configuration file containing, among others, the requirements and the self-adaptive reconfiguration algorithm to be used\footnote{More details about the configuration file will be provided in Section 8}. Then, the application can be executed by using the standard \texttt{parsecmgmt} tool as follows:

\[
\text{parsecmgmt run blackscholes --size 1 --threads 1 --version gcc-pthreads-nornir}
\]

Since we do not want to modify the original ap-

application (except for the instrumentation calls to gather performance data), we will adjust the number of cores allocated to the application by applying thread packing. We will show in Part III how, if the application is appropriately designed, the algorithms could exploit concurrency throttling, improving the performance/power tradeoffs of the configurations they find.

4.2 Evaluation Methodology

We divide the application into the following two sets, according to the regularity of their runtime behavior:

**Stable** This set includes Blackscholes, Bodytrack, Canneal, Facesim, Raytrace, Streamcluster, Swaptions, Vips applications. These applications are characterized by a throughput more or less stable during their execution. Fluctuations in the throughput occur but their amplitude is limited, as shown in Figure 4.2a for the Raytrace application. Some applications in this set are characterized by large throughput changes during their execution, but such changes are not too frequent and will be considered as phase changes, as shown in Figure 4.2b for the last part of Streamcluster execution.

**Unstable** This set includes Dedup, Ferret, Fluidanimate, Freqmine, X264 applications. These applications are characterized by large and frequent fluctuations, as shown in Figure 4.2c for the Dedup application.

We need to perform this distinction since, as we will show later, online learning algorithms may have difficulties in managing unstable applications, while they will work better than other solutions when applied on stable applications.

In our evaluation we consider two different type of requirements:

**Throughput Requirement** Given a throughput requirement, the self-adaptive algorithm should find the configuration with the lowest power consumption among the configurations with a throughput higher than the specified one.
4.2. **EVALUATION METHODOLOGY**

**Power Consumption Requirement** Given a power consumption requirement, the self-adaptive algorithm should find the configuration with the highest throughput among the configurations with a power consumption lower than the specified one.

Other requirements are possible since once we derived prediction models for throughput and power consumption, we can easily predict other metrics derived from these two, such as execution time, energy consumption, energy-delay product (EDP) and others.

To avoid biases in the results due to specific requirements choices, we test the algorithms over a wide range of different requirements. For example, if we need to specify a power consumption requirement for an application that has a minimum power consumption of 20 Watts and a maximum of 220 Watts, then we will test the algorithms by setting the power consumption requirement to 40, 60, ... and 200 Watts. We slice the range in 10 equal intervals, to cover the entire power consumption spectrum, avoiding biases due to specific choices. The same technique has also been adopted for throughput requirements. The algorithm will be executed 5 times for each requirement and, for each application, we will show the average and standard deviation obtained over all these requirements. For all the algorithms we set a control step of 1 second. We set the control step during the training phase (or search phase for the heuristics) equal to 100 milliseconds. On our online learning algorithm, the training phase will terminate when both the performance and power consumption prediction errors are lower than 10%.

When evaluating the reconfiguration algorithms, we will consider the following metrics:

(a) Throughput of the Raytrace application.

(b) Throughput of the Streamcluster application.

(c) Throughput of the Dedup application.

**Figure 4.2: Stable and Unstable Applications**
CHAPTER 4. EVALUATION

Requirements Violation The percentage of test cases for which the algorithm failed in finding a configuration with the required throughput or power consumption.

Training Steps The number of configurations visited by the algorithm during the training phase (or during the search phase for heuristics). If multiple training stages are executed (for example because different phases are present in the application), this metrics will consider the sum of the duration of all the training stages.

Training Time The time spent in the training phase (or during the search phase for heuristics). As we anticipated while describing the online learning algorithm, having a short training time is a desirable property since during the training no guarantees on performance and power consumption are provided. This is particularly critical for power consumption requirements since, as recently highlighted in [11], a key factor to consider when designing power capping techniques is how fast they can respond to power overdraw, to guarantee protection from tripping circuit breakers. To provide a measure of the maximum duration the training phase should have, [11] reports that current equipment can sustain power overdraws for at least 30 seconds. Moreover, it is worth noting that we are considering the worst case scenario where the requirements are violated for the entire duration of the training phase. However, during the training, the algorithm may also visit some configurations with acceptable performance and power consumption. If multiple training stages are executed (for example because different phases are present in the application), this metrics will consider the sum of the duration of all the training stages. It is worth noting that the training time is not simply the number of training steps multiplied by the length of the control step. Indeed, when collecting monitoring data from the application (e.g., number of iterations performed per time unit to compute the throughput), we need to collect data for at least one iteration. Accordingly, the actual length of the control step depends on the length of one application iteration.

Suboptimality Even when a proper configuration is found, it may be not the optimal one. For example, consider the case where the user sets a power constraint of 50 Watts. Let us assume that the configuration $C_1$ is the one with the highest throughput, among those consuming less than 50 Watts. However, due to mispredictions, the algorithm could predict the power consumption of $C_1$ to be 52 watts, thus discarding it from the set of possible configurations. Then, most likely it will select a configuration with a lower throughput. With suboptimality we consider the percentage difference between the true optimal configuration and the configuration selected by the algorithm. As optimal configuration, we consider the one found by an ideal oracle, which knows exactly the power consumption and performance of all the possible configurations. However, in reality, algorithms could not find the optimal configuration due to prediction
inaccuracies. Moreover, even the overhead introduced by the algorithm execution could lead to choosing suboptimal configurations. Indeed, to take into account this effect as well, we will consider as optimal reference implementations the original, non-instrumented applications.

Intuitively, for all these metrics the lower is the value, the better is the algorithm.

4.3 Online Learning Algorithm Alternatives

Before comparing the algorithms we described in Chapter 3 with existing state of the art solutions, we would like first to understand how some decisions such as the choice of the prediction model and the different threads’ placements impact the quality of the algorithm itself. We briefly recall that the algorithm is composed of a training phase, in which throughput and power consumption of different configurations are collected. These data are then used to build a prediction model, which will be used in the steady phase to select the optimal configuration according to the requirements expressed by the user. This structure is general enough, and different prediction models could be plug into this algorithm. We proposed three different performance prediction models:

- **Amdahl’s Law** Described in Section 3.2.2.1. For brevity, we will denote it as Amdahl.

- **Universal Scalability Law** Described in Section 3.2.2.2. For brevity, we will denote it as Usl.

- **Universal Scalability Law - Optimized** Like the previous one, but with the additional measurement of \( R(1, f_{min}) \) which, in principle, should improve the accuracy of the model. For brevity, we will denote it as Usl\(_{\text{opt}}\).

Concerning the power consumption, we have considered one single prediction model (Section 3.2.2.3).

We first consider in Section 4.3.1 the impact of the different prediction models. Then, in Section 4.3.1 we evaluate the advantages and disadvantages of considering multiple threads placements.

For brevity’s sake, in this section we only consider results obtained on applications in the *Stable* set. We will consider both *Stable* and *Unstable* applications in Section 4.4, when comparing our algorithm with state of the art ones.

4.3.1 Performance Prediction Models

**Requirements Violations** First of all, in Figure 4.3 we analyze the percentage of requirements which are violated when using each prediction model, considering the average among power consumption and for performance requirements. In the figure, we report the average over all the requirements and all the *stable* benchmarks.
Figure 4.3: Percentage of tests for which it was not possible to satisfy the requirement specified by the user, due to inaccuracies in the prediction models. The lower the better.

As we can see from the data, all the performance prediction models perform well since they only fail on 3% of all the possible cases, with \text{Uslopt} performing slightly worse than the others.

Training Steps After we assessed the capacity of the prediction models used by our online learning algorithm in enforcing the user requirements, we now want to evaluate how many steps the algorithm will take before converging to such solution, i.e., how many configurations are visited by the algorithm in the training phase. We show the results of this evaluation in Figure 4.4, where we report the data separately for each benchmark. We average the results over all the throughput and power consumption requirements.

Figure 4.4: Steps required by the online learning algorithm to terminate the training phase. The lower the better.

In general, we can observe that, for all the benchmarks, all prediction models we proposed are accurate enough, since they converge after visiting less
than 15 configurations (over the 312 available on our target architecture). This means that, after visiting less than 5% of the total possible configurations, the models can predict, with an accuracy > 90%, the performance and power consumption of the application in all the other 297 remaining configurations. We believe that this is an relevant result if you consider that usually, machine learning techniques use between 60% and 80% of all the available data to train the model while we were able to achieve a good accuracy with only 4% of the data available. This was possible since we started from a rough analytical prediction model, which we then refined at runtime by collecting monitoring data on the application.

In Figure 4.5 we report the same results but averaged over all the stable benchmarks. Again, there are no significant differences between the different performance prediction models, with Ulsopt performing slightly better than the other two models.

Figure 4.5: Steps required by the online learning algorithm to terminate the training phase, averaged over all the stable benchmarks. The lower the better.

Training Time We now analyze the absolute time spent in the training phase. Indeed, the relationship between the number of steps and the time spent training the algorithm is different for different benchmarks. Let us consider the results reported in Figure 4.6.

In all the cases except facesim the algorithm can perform the training in less than 10 seconds, independently of the performance prediction model we use. Despite being characterized by a comparable number of training steps with respect to the other benchmark, Facesim spend a significantly longer time in the training phase, since the time spent in each training step is much higher. As we anticipated earlier, to complete a step of the MAPE loop we need to collect data for at least one iteration of the application. In the facesim case, 1 iteration may take up to 13 seconds to be completed. Accordingly, even if the algorithm needs only to execute few training steps, the training phase will last for tens of seconds. The length of the training phase is longer for the Ulsopt case since, as described in Section 3.2.2.2, this prediction model needs
to collect data about the configuration where the application uses 1 core run-
ing at minimum frequency. This will most likely be the slowest configuration possible, thus significantly increasing the length of the training phase. When we described the Training Time metric, we said that a desirable property is to have a training time lower than 30 seconds, since (for power consumption re-
requirements) in the worst case this could lead to power outages. However, for facesim, only the Amdahl model will be able to converge in less than 30 sec-
onds.

In Figure 4.7 we report the average training time over all the benchmarks.

Even in this case, there are no particular differences among the performance prediction models, except for Uslopt that this time performs slightly worse than the other two models, also due to the excessive length of the training phase in facesim.
4.3. ONLINE LEARNING ALGORITHM ALTERNATIVES

Suboptimality  Lastly, we evaluate the suboptimality of the different performance prediction models, i.e., how much the configuration found by the online learning algorithm is far from the optimal configuration, regarding performance or power consumption (according to the requirement).

We report the result of this evaluation in Figure 4.8

![Figure 4.8: Suboptimality of the online learning algorithm. The lower the better.](image)

In almost all the cases all the algorithms select, in average, a configuration which is no more than 5% worst than the optimal one. As depicted in Figure 4.9, in average, $U_{Sl}$ and $U_{SLOPT}$ perform better than $Amdahl$, since they can predict more accurately the performance, due to a better modelling of contention effects.

![Figure 4.9: Suboptimality of the online learning algorithm, averaged over all the benchmarks. The lower the better.](image)

Summary  To summarize the result of the comparison of our prediction models, we found out that $U_{SLOPT}$ performs slightly better than the other two performance prediction models, despite being characterized by a slightly longer training time. When referring to online learning algorithm in the rest of this
thesis, we will refer to the particular case where $Uslopt$ is used to predict the performance of the parallel application in different configurations. The only exception will be for $facesim$, for which we will consider the Amdahl prediction model, since it is the only model which terminates the training phase, in average, in less than 30 seconds.

4.3.2 Threads Placement Impact

Another important evaluation which needs to be done concerns the different threads placements we described in Section 3.2.2.4. Specifically, we want to evaluate if the extra training time required to predict the effect of threads placement will lead to improved predictions and to solutions closer to the optimal one. Firstly, in Figure 4.10 we show the number of training steps performed by the algorithm in the two cases.

![Figure 4.10: Training steps when using only one thread placement (Linear) and when using automatic discover of optimal thread placement (Automatic Placement). For each benchmark, results are averaged over all the performance and power consumption requirements. The lower the better.](image)

As we could expect, the number of steps is increased for almost all the benchmarks, since some extra configurations need to be visited to derive models for the interleaved thread placement as well. This, of course, will have a direct effect on the training time, which we do not show here for brevity’s sake.

On the other hand, in Figure 4.11 we show the suboptimality results for the two cases. Despite there are some small advantages in some benchmarks, we believe that these modest gains are not worth the extra time required to train the models. For these reasons, we will not consider different threads placements in the rest of this thesis.

4.4 Comparison with State of the Art Strategies

We will now compare the heuristic and the online learning algorithm we designed in this thesis (denoted as $Heuristic$ and $Online Learning$), with the
4.4. COMPARISON WITH STATE OF THE ART STRATEGIES

Figure 4.11: Suboptimality when using only one thread placement (Linear) and when using automatic discover of optimal thread placement (Automatic Placement). For each benchmark, results are averaged over all the performance and power consumption requirements. The lower the better.

Following state of the art reconfiguration strategies:

**LiMartinez** is a well-known heuristic presented in [16], which uses a combination of hill climbing and binary search algorithms to find the lowest power consuming solution under a performance constraint. However, since it doesn’t explicitly model power consumption, it is not possible to specify any power consumption requirement.

**Leo** is a recent reconfiguration algorithm presented in [14]. It uses an offline learning approach based on previous profiling of the applications. These profiling data is integrated with information collected online while the application is running. This algorithm explores a fixed number of configuration during the online phase, set to 20 by its designers. This algorithm is more general than those we presented, and it could be used on any knobs.

**Rapl** is an hardware-enforced solution available on newer Intel’s processors [187], which can be used to limit the maximum power consumption of the CPU over a time window. We set a time window of 1 second, equal to the one we considered for the other algorithms. This algorithm cannot be used to enforce performance requirements.

We decided to pick these algorithms so to have one heuristic (**LiMartinez**) and one machine learning algorithm (**Leo**). **Rapl** have been included since it is the standard *de-facto* for power-capping solutions.

4.4.1 Requirements Violations

First of all, we want to evaluate the percentage of tests for which the different algorithms failed in finding a configuration satisfying the user requirements
regarding performance or power consumption.

**Stable Applications** In Figure 4.12 we show these results for *stable* applications.

![Graph showing requirements violations (%) for stable applications](image)

Figure 4.12: Requirements violations (%) of the different algorithms for *stable* applications. For each benchmark, results are averaged over all the requirements. The lower the better.

Concerning performance requirements, we can notice that the Online Learning algorithm violates less than 10% of requirements, and only for some applications. The percentage is similar also for the other algorithms, except for Leo which, due to mispredictions, in some case violates up to 40% of the requirements.

Concerning power consumption requirements, the first thing we notice is the high number of requirements that Rapl is not able to enforce. Indeed, since it only operates on the clock frequency, it is not able to decrease too much the power consumption of an application, thus violating low power consumption requirements. This is not an issue for the other techniques, since they also operate on the number of cores, thus being able to explore a broader range of possibilities. The only other algorithm violating some requirements is Leo, which however violates less than the 10% of all the possible power consumption requirements.

There are usually less power consumption requirements violations than performance requirements violations, since power consumption is generally
more straightforward to predict. Moreover, in Figure 4.13 we analyze the amplitude of such violations, i.e., when the algorithm is not able to find a configuration satisfying the user’s requirements, by how much (in percentage) this requirement is violated.

In Figure 4.13 In general, all the algorithms except for Rapl, even if they violate the requirements, they usually find a solution with performance (or power) within 10% than the required one.

To summarize, for stable applications, the algorithms we proposed in this thesis (Online Learning and Heuristic) violate less than 5% of all the possible requirements. Moreover, even when a requirement is violated, it is not violated for more than the 5%.

Unstable Applications In Figure 4.14 we report results for unstable applications.

It quickly becomes clear that algorithms that exploit some form of training now significantly suffer from the instability of the application. For example, for the Online Learning algorithm, fluctuations would lead to numerous retrainings of the algorithm (since these fluctuations will be detected as phase changes), impairing its effectiveness. Indeed, due to workload fluctuations, the configuration found by the algorithm could not be correct anymore, and could now violate the requirement. This is not the case for Heuristic algorithm which, when the contract is violated, jumps in another configuration, without performing any training phases. As we will see in Section 4.4.3, for stable applications Heuristic algorithm cannot find configurations as good as those found by Online Learning algorithm. However, thanks to its simplicity, for unstable application, it is more suitable than the Online Learning algorithm.
Figure 4.14: Requirements violations (%) of the different algorithms for unstable applications. For each benchmark, results are averaged over all the requirements. The lower the better.

**Summary** To summarize the results about requirements violations, the two algorithms we proposed violates less than 5% of requirements for stable applications, less than the violations of the other algorithms we are considering. Even if the requirement is violated, it is usually not violated by more than 5%. Concerning the unstable applications, online learning violates much more requirements since the training phase should be re-executed every time a fluctuation in the throughput is detected as a phase change, impairing its effectiveness. On the other hand, the Heuristic violates only a small percentage of the requirements, and only for the dedup and x264 applications.

### 4.4.2 Training Steps and Training Time

We now analyze how long does it take for the different algorithms to find a proper configuration according to the user requirements.

**Stable Applications** First of all, we evaluate the number of training steps for all the algorithms when executed over stable applications. For Leo, we only consider the number of configurations explored in the online training part. For Rapl, we always consider both training steps and training time to be zero since, when it can enforce the specified power cap, it will usually do so in
few milliseconds. We report the results of this evaluation in Figure 4.15, where we show the average number of configurations visited by each algorithm before they find a proper configuration, averaged over all the requirements.

![Figure 4.15: Average number of configurations visited by each algorithm before they find a proper configuration, averaged over all the requirements. Stable applications considered. The lower the better.](image)

The **Online Learning** algorithm can usually find a proper configuration after visiting less than 10 configurations, while the **Heuristic** is faster and can converge after visiting less than 5 configurations. As specified when we introduced the Leo algorithm, in [14] the designers suggest collecting online data about 20 configurations. It is worth recalling that, besides these 20 configurations, the Leo algorithm also uses a broad training set containing data collected offline, consisting of 312 configurations per 12 applications, for a total of 3744 additional configurations. Lastly, the LiMartinez algorithm usually needs few more steps than the **Online Learning** algorithm before finding a suitable configuration.

![Figure 4.16: Average number of seconds required by each algorithm to find a proper configuration, averaged over all the requirements. Stable applications considered. The lower the better.](image)
In Figure 4.16 we report the same data but regarding absolute time, expressed in seconds. Both the Online Learning algorithm and the Heuristic can find the best configuration in less than 5 seconds. The only two exceptions are facesim, which as we described earlier is characterized by slow iterations, and streamcluster, for which the algorithm needs to collect power consumption and performance data of a slightly higher number of configurations. Lastly, LiMartinez algorithm finds a good configuration in a time comparable to that of the Online Learning and Heuristic algorithms.

Unstable Applications For unstable applications, the situation is quite different, as shown in Figure 4.17. For brevity’s sake we only report data about training time.

![Figure 4.17: Average number of seconds required by each algorithm to find a proper configuration, averaged over all the requirements. Unstable applications considered. The lower the better.](image)

We recall that for the Online Learning algorithm we are considering the sum of the duration of all the training stages performed during the application execution (one for each phase). The Online Learning algorithm doesn’t perform well, due to re-trainings, which increase the time spent in the training phase. On the other hand, both the Heuristic and the LiMartinez algorithms can usually converge in less than 10 seconds, since they do not require sophisticated training to be executed for every different phase.

Summary To summarize, for stable applications the Online Learning algorithm can quickly find a proper configuration, usually faster than the Leo algorithm but not faster than the Heuristic and LiMartinez heuristic algorithms. When a requirement can be enforced by RAPL, it will usually be done immediately. For unstable applications, Online Learning spends too much time performing re-trainings, while LiMartinez and Heuristic algorithms are not affected by fluctuations in the workload, thanks to the simplicity of their (heuristic) prediction algorithms.
4.4. COMPARISON WITH STATE OF THE ART STRATEGIES

4.4.3 Suboptimality

Eventually, we evaluate the suboptimality of the configurations selected by the different algorithms, i.e., once a proper configuration has been found, how far this configuration is from the optimal one.

**Stable Applications** In Figure 4.18, for simplicity’s sake, we show the average among the tests we did for both power consumption and throughput requirements.

![Suboptimality example](image)

Figure 4.18: Suboptimality of the different algorithms when executed on stable applications. The lower the better.

Both Online Learning and Leo selected configurations which were no more than 5% distant from the optimal ones. The only exception is streamcluster for which performance and power consumption are slightly more difficult to be predicted, as we saw when discussing requirements violations (Section 4.4.2). Also, the remaining algorithms can find configurations within 5% from the optimal one, except for facesim and streamcluster.

**Unstable Applications** Similar results have been obtained for unstable applications, and we report the results in Figure 4.19.

Even in this case, the Online Learning algorithm selects better configurations than the Heuristic algorithm. Moreover, all the algorithms are usually within 10% from the optimal, except for the freqmine case.

**Summary** We have shown in this comparison that all the considered algorithms can find configurations characterized by a power consumption (for the performance requirement) or performance (for the power consumption requirement) within 5% from the optimal one. Differently from the previous metrics, suboptimality does not truly change between stable and unstable applications.
CHAPTER 4. EVALUATION

4.5 Summary

In this Chapter, we evaluated the new self-adaptive algorithms we propose in this thesis and which we described in Chapter 3. In Section 4.3 we analyzed the different performance prediction models we proposed for our Online Learning algorithm, as well as the impact of various threads placements. Then, in Section 4.4 we compared our algorithms with some existing state of the art strategies. We executed all the algorithms over the applications of the PARSEC benchmark, by requiring them to satisfy several requirements on performance and power consumption.

In Figure 4.20 we report a summary comparison through a radar chart, where we normalized the results we already showed with values between 0 and 1. On each radius, the highest the value, the better the algorithm is. Accordingly, if we consider for example the Training Time radius, a higher value
doesn’t mean that the algorithm has an higher training time but a better (i.e., shorter) training time. The same applies to the other metrics as well. We added a Supported Requirements radius, to also consider the types of requirements supported by each algorithm. We assigned a value of 0.5 to algorithms supporting only power consumption or performance requirements and a value of 1.0 to algorithms supporting both types of requirements.

**Stable Applications** For stable applications, Online Learning, Heuristic and LiMartinez violate a comparable number of requirements. Rapl performs poorly since, by acting only on voltage and clock frequency, cannot decrease the power consumption too much. Regarding optimality, configurations found by Online Learning algorithm are very close to those found by an efficient and hardware-enforced solution such as Rapl. On the other hand, Online Learning and Leo supports both performance and power consumption requirements, while Heuristic and LiMartinez support only performance requirements and Rapl only supports power consumption requirements. Concerning the training, Online Learning can find a proper configuration in a time comparable to Heuristic, LiMartinez and Rapl. In a nutshell, we succeeded in the goal of designing an algorithm that, as heuristics, do not exploit any previously collected data but that achieves an accuracy comparable to that of complex offline learning solutions.

**Unstable Applications** For unstable applications the performance of the Online Learning and Leo algorithms decrease, concerning training time, training steps and requirements violations. However, our Heuristic algorithm still performs well and, concerning requirements violations, better than the other heuristic (LiMartinez) we considered in our comparison.

We would like to highlight the following main considerations which came out from this evaluation:

- The first important result is that, despite not using any offline data, our Online Learning algorithm achieve an accuracy comparable to that of more complex strategies (such as Leo) which mix both offline and online data and requires a long data collection phase to be performed before running the application. By collecting (in average) data about less than 10 configurations, we were able to achieve the same accuracy of an algorithm which collected data about more than 3000 configurations (offline plus online). In addition to that, our algorithm is not sensitive to the choice of specific training sets, since it is trained on the application itself.

- Online Learning and Heuristic algorithms complement each other, depending on the application. Indeed, for stable applications the Online Learning algorithm works fine and can find better configurations than the Heuristic algorithm. However, for unstable applications, due to the numerous re-trainings, the Online Learning algorithm spend too much
time in the training phase. For this reason, even if less optimal the the Online Learning algorithm, we suggest using the Heuristic when many workload fluctuations characterize the application. Moreover, it is worth recalling that Heuristic can only be used to enforce performance requirements.

- Despite RAPL is the current de-facto standard solution for enforcing power caps in datacenters, and despite being efficient and requiring no training, it cannot be used to enforce aggressive power caps.

It is worth noting that the applications considered in this Chapter may have different characteristics with respect to classical streaming applications, since the latter are usually characterized by finer-grain parallelism. However, as we also shown in Section 3.2.4, our results still holds for the streaming case. In particular, the training time is significantly reduced since, due to fine grain nature of the application, the application performs more iterations per time unit and reacts more promptly to reconfigurations. Similarly, the accuracy of the predictions is improved as well, since in a single monitoring step much more iterations are executed and more information will be collected with respect to applications with coarse-grain parallelism. Moreover, it is worth noting that the other state of the art strategies we considered for our evaluation would require re-training the model each time there is a fluctuation of the input rate, significantly worsening the performance of the application, as we shown in Section 3.2.4.
In this Chapter we take a different approach and, instead of operating on the number of cores, clock frequency or threads’ placement, we dynamically change the algorithm used by the threads of an application to access shared data structures, i.e. the concurrency control algorithm\(^1\). This mechanism could also be used together with the ones used for the other algorithms, although we do not consider such possibility in this thesis.

In Section 5.1 we will introduce the problem providing some background concepts. Then, in Section 5.2 we describe the design of our algorithm and, eventually, in Section 5.3 we show some experimental results which validate this new concurrency control algorithm. Eventually, in Section 5.4 we summarize this chapter.

Parts of this chapter have been published in [29].

5.1 Problem and Background

In a shared-memory system, a standard approach to synchronize producer/consumer interactions between pairs of threads relies on a concurrent FIFO queue that supports push and pop operations atomically and efficiently. This approach is particularly relevant in streaming applications, where applications are modeled as arbitrarily complex graphs of modules/operators exchanging infinite sequences of data items through channels implemented by concurrent queues [188].

A naive implementation consists in protecting the access to a concurrent queue by using mutex mechanisms. If the thread that currently holds the mutex is delayed, all the other threads attempting to access the data structure are delayed too. Furthermore, acquiring the mutex implies passive waiting, that

\(^1\)Please note that this must not be confused with concurrency throttling, which indicates the possibility to dynamically change the number of threads composing a parallel application.
CHAPTER 5. A CONCURRENCY CONTROL ALGORITHM

is the suspension of all the threads waiting for the mutex acquisition. The suspended threads are moved in a waiting queue, and their core/hardware contexts are released to the OS. Also, passive waiting can be used as the underlying mechanism to handle synchronization events during the usage of the queue, like when the producer tries to push a data item in a full-size queue or when the consumer attempts to pop from an empty queue. Under the assumption of having at most one thread per core (which is common for HPC application), suspending a thread would leave the corresponding core idle. By doing so, the CPU can place the core into a deeper C-State (Section 2.1.1.1), reducing the overall power consumption of the CPU. However, suspension and main restart mechanisms may impair application performance due to many factors such as the waiting time in the ready queue, context switch overhead, compulsory cache misses or core migration [45]. In general, a concurrent algorithm that may force the calling thread to be blocked waiting that a given condition holds is defined as blocking.

Concurrent queues can be implemented in an efficient and scalable way by using nonblocking algorithms. Although many definitions exist and have been utilized in the literature, in this thesis, we consider a concurrent algorithm as nonblocking when thread suspension cannot be caused by synchronizations related to the data structure usage (of course a thread can still be de-scheduled by a time-sharing scheduler or due to preemption). Therefore, a solution that replaces passive waiting phases with a busy-waiting spin-loop (e.g., by replacing mutexes with spin-locks) is accepted as nonblocking according to this definition. One of the drawbacks of non-blocking algorithms is that, since the CPU is executing instructions, the busy-waiting phase would not allow the CPU to enter in a deep C-State (Section 2.1.1.1). This means that the waiting thread is wasting CPU cycles and power without doing any useful activity. Moreover, busy-waiting may impair application throughput if more threads share the same hardware context/core.

An interesting and widely studied class of nonblocking algorithms are the ones classified as lock-free. This term refers to the fact that the failure or the suspension of a thread in any arbitrary point during its execution cannot prevent at least one thread in the system to make progress [48]. As an example, a spin-lock based queue is not lock-free because if a thread fails after the lock acquisition, none of the other threads will be able to complete its operation on the queue correctly. In a lock-free concurrent queue, the producer and the consumer threads can push and pop elements concurrently by working on different positions of the queue. This does not necessarily mean that threads do not have to take into account their mutual interference. For instance, a thread may start a push/pop operation and, if the queue state changes due to the access by another thread, this must be detected and the operation restarted until it is correctly executed. Such actions (occurring in while-loops) may consume CPU cycles and power although the degree of concurrency inside the queue’s code is maximized.

While lock-free algorithms are mainly chosen for their progress guarantees, they are also employed for their higher throughput and lower latency [48] [49].
Furthermore, by avoiding the threads to be de-scheduled in the synchronization phases, they contribute in reducing the so-called OS noise which may be a source of scalability problems in many High-Performance Computing applications [189]. Unfortunately, as said, the nonblocking approach is not power-efficient due to the busy-waiting loop executed when a given operation cannot be immediately concluded (e.g., CAS retry loop). Although several methods have been proposed to reduce the power consumption during busy-waiting loops, for example using pause, memory barriers or monitor/mwait instructions [45], none of them revealed utterly successful on the off-the-shelf commodity multicores, and busy-waiting is de-facto considered not power efficient.

A way to improve power saving is to delay the busy-waiting phases with short phases of passive waiting obtained by executing micro-sleep system calls. This technique is called backoff and has been widely adopted in the implementation of spin-locks [113]. Theoretically, if the micro-sleep phases are correctly regulated each time by an oracle, the achievable performance/power trade-off can be optimal (same performance of the pure busy-waiting approach with almost the same power saving of passive waiting). However, this is unlikely in real situations, and the use of wrong values could have dramatic effects on the reactivity of the system or its power consumption. Furthermore, the tuning phase can hardly be automatized, and the solutions are in general not portable since the sleeping intervals should be regulated for each machine and application. Indeed, it might be even impossible to use sufficiently fine-grained sleeps in some OSs.

Instead of optimizing the backoff technique, we propose a different approach that breaks the dichotomy between blocking vs nonblocking techniques that have often been used as mutually exclusive alternatives in the past. Our solution is designed for data stream processing scenarios, where thread synchronization happens on the use of lock-free concurrent queues. To exemplify the problem, we show in Figure 5.1 the analysis we performed on a network streaming application where a continuous flow of data packets is analyzed in real-time. We tested the application with two different input rates: the first is of 350K packets per second while the second has about 1M packets per second. In both cases, we collected the throughput and power consumption measures obtained by two configurations of the application, the first using a blocking concurrency mode, and the second uses a nonblocking mode.

Interestingly, with the slowest input rate, the two versions achieve almost the same throughput, whereas the power consumption is significantly in favor of the blocking concurrency mode. This result is obtained because in that case the application is fast enough to process the packets at their arrival speed, and it often idle waiting for new packets. The blocking version optimizes power consumption in those phases by suspending the application threads. Furthermore, the wake-up overhead is negligible compared with the packet inter-arrival time.

\[\text{For example, on Windows OSs is challenging to put one thread to sleep for less than one millisecond.}\]
Figure 5.1: Motivating example: throughput and power consumption of the blocking vs nonblocking concurrency control modes in a network application with different input rates.

In the case of the high input rate, the pressure to the system is intensive, and on average there are always packets to be processed. Hence, the idle time is minimized, and the two versions have a comparable power consumption. The extra-overhead of the blocking version (e.g., for waking up suspended threads) impairs throughput because the inter-arrival time interval is short and a slight improvement of the thread processing speed immediately delivers better throughput.

The solution we propose consists in automatically switching between the two concurrency modes according to the actual properties of the incoming workload. Each queue supports both the concurrency modes that can be modified transparently to the user. This strategy is used to adapt arbitrary graphs of streaming operators working on single-producer single-consumer (SPSC) queues. Although the switching decision requires some tuning, we design our algorithm so that the tuning is done only once per architecture by executing some micro-benchmarks, being independent of specific workloads and applications. To confirm the effectiveness of our approach, we validate the algorithm using two real streaming applications.

5.2 Algorithm Design

In this section, we describe the design of a shared-memory message queue that can be used both in blocking and nonblocking concurrency control modes for a generic data streaming computation. Then, we describe how to use it to optimize the power consumption and performance of a generic streaming graph.
5.2. ALGORITHM DESIGN

5.2.1 Base Mechanisms

An effective way for implementing pipeline parallelism between two threads on multicores is to use a lock-free Single-Producer Single-Consumer (SPSC) FIFO queue [190, 191]. As discussed in Section 5.1 this approach is not power-efficient if a thread is performing busy-waiting because the underlying hardware context remains active so that the OS cannot set the core in a low-power state.

In case of variable arrival rates, the producer (P) or the consumer (C) threads might spent some time in a busy-waiting loop because the message queue is full or empty. Rather than spinning, to reduce power consumption, we want to put the threads to sleep waking them up as soon as they can make useful work. The only portable way for doing this is to use POSIX mutexes and condition variables (or equivalent C++1x features).

We consider FastFlow [47] as the reference parallel framework. FastFlow is an open-source, parallel programming framework supporting highly efficient stream parallel computation on heterogeneous multi-core platforms. It is realized as a C++11 header-only template library that allows the programmer to simplify the development of parallel applications modeled as directed graphs of processing nodes. We extended the FastFlow concurrency mode by associating to each FastFlow channel a POSIX mutex and a condition variable and by changing the push and pop operations as described in the following pseudo-code.

Let us describe the Algorithm 1. To push a message into the output queue the runtime first pushes the data pointer into the lock-free SPSC queue Q (line 1); if it succeeds, depending on the current concurrency mode of the node (called CCM in the algorithms), two different actions are taken. In case of non-blocking mode, the operation has been successfully completed without any further step (returning success at line 11). If CCM=blocking, the runtime checks if the consumer node (C) has to be woken up because it has been previously put to sleep waiting for a new message (testing C.waiting at line 4). In that case, C will be awakened by an explicit signal on its input condition variable C.cond_in (line 5) and the operation returns with success (line 11). If the push...
on the lock-free queue fails and $\text{CCM}=\text{nonblocking}$, the push operation is executed again until it will complete with success (line 10). If $\text{CCM}=\text{blocking}$, then the runtime checks if the queue is full (line 8) and, in that case, the thread is put to sleep on its output condition variable $P_{\text{cond\_out}}$ (line 9). If it is not full (this is a spurious condition that may happen with lock-free data structure), or if the thread is woken up by a signal, the operation is restarted from the beginning (line 10).

Let us now consider the pop operation described in the Algorithm 2. The runtime pops a new data pointer from the lock-free queue (line 1). If the operation succeeds and $\text{CCM}=\text{nonblocking}$, the process has been successfully completed (line 3 and line 11). If $\text{CCM}=\text{blocking}$ then the runtime checks if the producer ($P$) has to be woken up because it is waiting for a new free slot in the queue (line 3 and 4). This case can only happen if the input queue has a bounded size. In that case, $P$ receives a signal on its output condition variable, and the operation completes with success (line 5 and 11). If the pop fails and $\text{CCM}=\text{nonblocking}$ the operation will be repeated until it ends with success (line 10). If $\text{CCM}=\text{blocking}$ then the runtime checks if the queue is empty (line 8) and puts the thread to sleep on its input condition variable (line 9), otherwise the operation is restarted from the beginning (line 10).

By switching the $\text{CCM}$ variable between $\text{blocking}$ and $\text{nonblocking}$ concurrency mode, it is possible to control the throughput and the power consumption of the nodes using the queue, as we will explain in the next section.

### 5.2.2 Power-Aware Data Pipelining

To describe the algorithm, we first consider a streaming network structured as a pipeline, a connected graph where each node has at most one input queue and one output queue. Then, in Section 5.2.3 we extend the algorithm to generic streaming networks.

To simplify the exposition, from now on we consider that all message queues are $\text{unbounded}$ in size. Consequently, each node would never need to do busy-waiting or to suspend itself when doing a push operation. This decision may cause an uncontrolled growth of memory usage, and we will discuss this aspect in Section 5.2.3.

As described in Section 5.2.1, by changing the $\text{CCM}$ variable it is possible to change the concurrency mode of the producer and consumer thread. As typical in self-adaptive algorithms and as we already saw in Chapter 3, this decision is taken by the manager during the $\text{Analyze}$ and $\text{Plan}$ phases. The manager will ensure that the $\text{CCM}$ variable is modified consistently, so that both consumer and producer always use the same concurrency mode, to avoid critical cases like having the consumer blocked on its input queue while the producer is already in the $\text{nonblocking}$ mode.

Each concurrent activity in the pipeline will execute the following operations in a loop:
5.2. ALGORITHM DESIGN

1. Reads an element from its input queue by executing a **pop**. The average latency of this operation is $L_{\text{pop}}^{b}$ for blocking queues and $L_{\text{pop}}^{nb}$ for non-blocking queues. If no data is present in the queue (the **pop** fails), the node waits for new data to arrive by suspending itself or by doing busy waiting. Let us denote this average waiting time with $L_{\text{idle}}$.

2. Executes some processing (with a latency $L_{\text{proc}}$) on the data element;

3. Sends the computed result(s) on its output queue through a **push**. This operation has an average latency of $L_{\text{push}}^{b}$ in case of blocking queues and $L_{\text{push}}^{nb}$ in case of non-blocking queues. Since the output queue is unbounded in size, this operation will always succeed.

The breakdown of a single loop iteration of a node is sketched in Figure 5.2. Timing values (e.g., $L_{\text{idle}}$, $L_{\text{proc}}$) are stored by the single node in its internal variables that can be accessed (read-only) by the manager in the Monitor phase of the MAPE loop (without any extra synchronization).

In the rest of this section we will describe the two possible cases for the dynamic switching (in the **Execute** phase of the MAPE loop) of the concurrency control mode: i) from blocking to nonblocking; ii) from nonblocking to blocking.

### 5.2.2.1 From blocking to nonblocking

Suppose that when the application starts, it uses all the message queues in blocking mode. To improve the throughput of the application, we have to improve the throughput of its slowest node, i.e., the one with the highest latency. We call this node $S$. If it has $L_{\text{idle}} > 0$, despite being the slowest node in the application, it is still fast enough to process the incoming data, so there is no need to improve the throughput of the application at all. Otherwise, we can improve the throughput of $S$ by reducing the latency of both the pop and push operations. Let us start with the pop operation. Switching the input queue to nonblocking mode would have no impact on the power consumption since $L_{\text{idle}} = 0$ and $S$ will not do busy wait. Now let us consider the push operation. We could switch the output queue of $S$ to the nonblocking mode and reduce $L_{\text{push}}$ as well. Let us call $T$ the successor of $S$. Since $S$ is slower than $T$, $L_{\text{idle}}(T)$ is greater than zero. If the message queue is in blocking mode, while idling $T$ is suspended on the condition variable. However, after switching to nonblocking mode, $T$ will start doing busy-waiting, thus increasing the power consumption of the application. Differently from the algorithms we described in Chapter 3, instead of predicting the changes in performance and power consumption, we adopt a **rollback-based** approach. When a potential performance improvement for the output queue is detected, the algorithm switches the queue from blocking to nonblocking. Then, the performance and the power consumption are monitored for the next control step. If the results of the switching do not comply with the user requirements, the decision is reverted. Otherwise, it is kept. If the decision is not rollbacked, we then
improved the performance of $S$, and the slowest node might now be a different one. If this is the case, the algorithm is executed on the new slowest node. Otherwise, it terminates. To avoid too many rollback operations, if a node was involved in a rollback operation, it is marked, and it is not re-evaluated for a time interval that can be specified by the user.

![Diagram](image)

Figure 5.2: Different kind of latencies in the node operations.

5.2.2.2 From nonblocking to blocking

Due to workload fluctuations, the system could start receiving data slower. In such a case, the message queues will become empty, and some nodes will start doing busy-wait on their input queues. By switching a queue to blocking mode, the nodes using the queue would suspend on the condition variable instead of doing busy-wait. However, we would also increase the latency of $\text{push}$ and $\text{pop}$ operations. To ensure that this switch does not decrease the throughput of the nodes, it is sufficient to ensure that the increase in the $\text{push}$ and $\text{pop}$ latency is "absorbed" by the idle latency. Accordingly, even if these operations will last longer, the nodes will still have enough time before receiving the next data element, thus not reducing their performance. To do so, it is sufficient to find the pairs of nodes $P$ (producer), $C$ (consumer) such that the following condition is true:

$$L_{idle}(C) > L_{\text{pop}}^b(C) - L_{\text{pop}}^{nb}(C) \text{ and } L_{idle}(P) > L_{\text{push}}^b(P) - L_{\text{push}}^{nb}(P)$$

If this condition is satisfied, we will have $L_{idle} > 0$ for both nodes after switching to blocking, thus not reducing their throughput.

5.2.3 Generic Streaming Graphs

Here we discuss how to apply to a generic connected graph, the algorithm previously described for the pipeline graph.

We may observe that a data element, flowing from a source to a sink of the streaming network, will cross different processing nodes and different message queues. Since each node may have multiple output channels towards various nodes, the path followed by an input element depends both on the
5.3. EXPERIMENTAL RESULTS

scheduling policies adopted by these multi-output nodes and by the data element itself. However, all the possible paths are statically known (see Figure 5.3).

Figure 5.3: A data stream processing graph and all its possible routing paths.

Since each of these paths is a pipeline, we can apply the algorithm described for the pipeline separately on each path. When computing all the possible paths, we remove the backward edges, i.e., those forming a loop in the graph. Indeed, a loop simply replicates a sub-paths multiple times ($N$ Times in Figure 5.3) in the pipeline. However, it is sufficient to optimize each node of the path just once. Indeed, the algorithm will still optimize the throughput of the application even if we do not consider these duplicate nodes. For example, two bottom paths in Figure 5.3 (right) will be the same path for the algorithm.

5.2.3.1 Message queues’ memory utilization

In the algorithm, we considered all the message queues used by the application to be unbounded in size. However, if the application receives data at a faster rate than the one it has been designed for, data would accumulate in the message queues, leading to an uncontrolled growth of memory utilization and to catastrophic effects on the application.

The application, however, has been designed to sustain a given maximum input data rate or to do not exceed a specific memory utilization. Accordingly, the source nodes would throttle itself by not injecting data into the application at a rate faster than the one the application was designed for. In this way, the total amount of memory of the system is kept under a maximum value.

5.3 Experimental Results

In this Section we validate our concurrency control algorithm over two real streaming applications.

We conducted all the experiments on the same workstation we used in Chapter 4, an Intel workstation with 2 Xeon E5-2695 @2.40GHz CPUs, each with 12 2-way hyper-threaded cores, running with Linux x86_64. We did not
use hyper-threading, and we ran all the experiments by selecting the maximum CPU clock frequency available through the performance scaling governor.

Since we this algorithm requires operating on low-level mechanisms of the runtime system, this approach cannot be adopted for any application, but only on applications implemented with a framework providing such possibility. As anticipated earlier, we modified the FastFlow runtime so to have the possibility of dynamically switching between blocking and nonblocking concurrency modes. For this reason, we consider two streaming applications implemented in FastFlow: the Protocol Identification and Malware Detection.

In these experiments the user requirement was to have a throughput that matches the data arrival rate, i.e., to have a utilization $\rho < 1$.

We used a control step of one second, and since at each control step the algorithm execution only takes few milliseconds, the overhead of the algorithm is less than 1% both regarding performance and power consumption.

To compute $L_b^{push}$, $L_{nb}^{push}$, $L_b^{pop}$, and $L_{nb}^{pop}$, (needed to decide when to switch from nonblocking to blocking mode) we run a micro-benchmark composed by a producer-consumer pair of nodes (i.e. a simple 2-stage FastFlow pipeline), considering the average latency over 200 thousands messages exchanged when the queue between the producer and the consumer is empty, which represents a worst case scenario for the latency. On the target architecture we obtained the following average values: $L_b^{push} = 27\text{ usec}$, $L_{nb}^{push} = 0.4\text{ usec}$, $L_b^{pop} = 0.8\text{ usec}$ and $L_{nb}^{pop} = 0.01\text{ usec}$.

These values include the cost of the push and pop operations and the cost introduced by the FastFlow runtime for the message management.

### 5.3.1 Protocol identification application

The first application we use for validating our algorithm is a network monitoring application [184].

This application is implemented as a three-stage pipeline. The source node receives the network packets and assigns to each of them a key, such that packets belonging to the same “application flow” have the same key. The packets are then forwarded to the second stage through the message queue. For each packet, the second stage stores packet information into a hash table by using the key. This information is used to correlate packets belonging to the same “application flow” to detect the application protocol (e.g., HTTP). If the node receives a packet belonging to a flow for which the protocol has been already identified, no additional processing is performed.

This behavior creates a situation where for each logical “application flow” we have a high latency on the first packets, but then, after the protocol has been identified, the latency drops down to almost zero. This is a typical scenario in many data stream processing applications [188], where the communication overhead may have a significant impact on the overall application performance.
5.3. EXPERIMENTAL RESULTS

Eventually, the second node will forward each packet to the third node, which injects the packets again on the network. To analyze the application in a realistic environment, we sent the packets to the application at variable rates, equal to those characterizing a modern Internet Service Provider. For this purpose, we used the dataset available at http://bit.ly/1RY7fEt.

We ran the application once for 24 hours, and we show the results in Figure 5.4. The spikes in the plot of the power consumption can be explained considering that the application uses only three threads pinned on three CPU’s cores. During the application execution, the OS executes on the other CPU’s cores others threads/processes (i.e., demons, services, etc.) which temporarily increase the power consumption of the CPU. The algorithm we are proposing (automatic) can provide, at every time, the best performance and the lowest power consumption among the three concurrency modes. When there is no need to improve the throughput (because there is not enough data to process), the algorithm switches the queues to blocking concurrency mode, thus reducing the power consumption. However, when the input arrival rate increases, it switches some queues to nonblocking, thus improving the performance to be able to sustain the input arrival rate. For this application, the automatic algorithm leads to a maximum performance improvement of 33.28% with respect to the blocking case and to a maximum power reduction of 11% with respect to the nonblocking case.

5.3.2 Malware detection application

This is the same application we used in Section 3.2.4. Logically, the Malware Detection Application is structured as a 3-stage pipeline where the middle stage computes the most expensive part and can be conveniently replicated a number of times. In the middle stage each replica, after having identified the
protocol, searches for a predefined set of “signatures” (representing malware binaries) inside each HTTP packet. Even in this case the packets are aggregated in “flows” and they are scheduled to a specific node according to the value of the flow key, computed by the first logical node of the pipeline.

In this experiment, the application is composed by 24 nodes (one for each core of the machine). As in the previous test, the arrival rate of the packets to the application is variable. In our test, we used the rate that characterize a modern Internet Exchange Point network\(^3\). For the malware detection part, we used a subset of the database used by the ClamAV antivirus\(^4\), containing 2000 signatures.

The results of our test are sketched in Figure 5.5, showing that the automatic policy can achieve the maximum performance while having the optimal power consumption (i.e., the same values obtained by the blocking concurrency mode). Between 15 and 22 the blocking concurrency mode has a lower power consumption but it cannot sustain the same arrival rate of the nonblocking mode.

In Figure 5.6 we show another interpretation of the result, by plotting the efficiency of the different concurrency control techniques, expressed as the ratio between the performance and the power consumption. As we can see from the plot, the automatic strategy is always characterized by the highest efficiency between those of the other two techniques.

Summarizing, the nonblocking strategy is the most performing one and the most power-consuming. Its power consumption depends on the number of cores used and not on the input data rate. The blocking strategy is the most

---

\(^3\)https://stats.linx.net/, (IXManchester), 24 hours data between 02/01/2016 and 03/01/2016. We scaled it down by a 3x multiplicative factor to match the maximum performance achievable on our target architecture.

\(^4\)https://www.clamav.net/.
5.4. **Summary**

In this chapter we described a concurrency control strategy which automatically selects the best algorithm to be used by threads when accessing a shared *Single-Producer, Single-Consumer* (SPSC) queue. In a nutshell, we considered applications implemented with a *threads-based* programming model, and structured as a graph of operators communicating through shared message queues. We considered two common situations that may occur when the consumer thread finds no data in one of its input queues: either it will suspend itself on a condition variable (blocking) or it will actively do busy-waiting until one new message is received (nonblocking). We have shown that both of these solutions are inefficient, either on the performance or power consumption side and that the best solution depends on the arrival rate of data to the application. For these reasons, we designed a new algorithm which can select the most appropriate concurrency control mode according to the current data arrival rate. We validated the algorithm over two real streaming applications, showing that we can take the best from the two worlds, i.e., having a power consumption lower than the nonblocking mode and a throughput higher than the blocking mode.

![Figure 5.6: Comparison of efficiency of blocking, nonblocking and automatic concurrency control strategies on the Malware Detection application.](image)

power-efficient for low to medium input rates, but its associated overhead does not allow to reach the maximum throughput. The new automatic strategy we designed is a good trade-off allowing to exploit both benefits of the previous two approaches.
Part III

Parallel Patterns Based Application

In Part II we showed how the self-adaptive reconfiguration algorithms we designed can enforce requirements on performance and power consumption of parallel applications. In this part of the thesis we show that, if the application has a well-defined structure, it would be possible for the algorithm to exploit better knobs such as *concurrency throttling*, which could potentially lead to configurations with better performance/power tradeoffs. As anticipated in Section 2.2.2.2, *concurrency throttling* implies dynamically changing the number of threads used by the application. This knob, however, cannot be applied to any application, since it requires the application to have some well defined structural properties.

Parallel applications designed by using *parallel patterns*-based methodology satisfy such properties and are good candidates for the exploitation of *concurrency throttling*. For these reasons, in Chapter 6 we show the potentiality of the *concurrency throttling* knob and we introduce parallel design patterns. Then, in Chapter 7 we describe P³ARSEC, a benchmark we built during this thesis by modeling and implementing the applications in the PARSEC benchmark suite with parallel patterns. By doing so, we achieved two main goals. On one side, we proved that parallel patterns programming model is a good candidate for implementing many real application, reducing the programming effort and in some cases even improving the performance, thanks to a simpler and linear design. On the other side, we provide to the scientific community a wide set of parallel patterns based applications, which can be used to validate techniques targeted towards applications designed through parallel patterns. This could be used, for example, to assess in the future the real effectiveness of *concurrency throttling*. 
Chapter 6

ON THE RELEVANCE OF PARALLEL DESIGN PATTERNS

In this Chapter, we introduce parallel design patterns. Despite, as shown in Chapter 4, our algorithms work on generic iterative parallel applications, some control knobs can only be exploited when the application has a clear and well-defined structure. Up to now, we always showed results where the number of cores used by the application has been reduced by applying thread packing to the application. Although being a non-intrusive and powerful reconfiguration mechanism, there are situations where it might not be the best solution. Indeed, due to contention on computing resources, having at most one thread per core could be more efficient than running more threads on the same core. However, this implies dynamically changing the number of threads that compose the application (i.e., applying concurrency throttling).

In Section 6.1, we compare concurrency throttling and thread packing, to motivate the focus we put on concurrency throttling and, consequently, on parallel patterns-based programming. Then, in Section 6.2 we present some common patterns. To provide a clearer idea of how parallel patterns typically work, in Section 6.3 we show a brief example of parallel pattern-based code. Eventually, in Section 6.4 we summarize the content of this chapter.

Parts of this chapter have been published in [30, 32, 33].

6.1 On Concurrency Throttling

In this Section, we try to motivate our interest in concurrency throttling showing that, despite being more complex to be used, there are situations where it leads to a higher performance given a specific power budget or lower power consumption given a specific performance requirement.

As shown in Table 6.1 both concurrency throttling and thread packing are
widely used by self-adaptive algorithms. However, to the best of our knowledge, no extensive and systematic comparison between these two techniques have been performed. We try to partially address this problem, by comparing these two techniques and by showing advantages and disadvantages of both of them. Part of this section has been published in [32].

<table>
<thead>
<tr>
<th>Self-Adaptive Algorithm</th>
<th>Thread Packing</th>
<th>Concurrency Throttling</th>
</tr>
</thead>
<tbody>
<tr>
<td>[192]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[193]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[194]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[116]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[119]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[17]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[141]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[195]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[52]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[15]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[117]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[16]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[26]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>[145]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[142]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[118]</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.1: Usage of Thread Packing and Concurrency Throttling in existing solutions.

Before moving forward with the discussion, we need to briefly recall the difference between thread based programming and task based programming. In both cases, the application is describing as a set of concurrent activities interacting with each other. The main difference between the two approaches concerns the execution model, i.e. how such activities are executed. In thread based programming, each logical concurrent activity corresponds to an execution thread, while in task based programming the logical activity is a task, which is decoupled from the runtime entity that will execute it. Accordingly, in task based programming models there is not a one-to-one equivalence between tasks and threads and tasks are scheduled to a pool of threads. The difference between these two models is depicted in Figure 6.1.

In the following, we will assume to have a thread based programming model. Then, in Section 6.1.5 we adjust our considerations for the task based programming model. The algorithms we designed are independent of the specific programming model. We recall that the results we shown in Chapter 4 considered Pthreads versions of the PARSEC applications (so a threads based model). In Section 8.4.2 we will show some results for the task based case as well. The
6.1. ON CONCURRENCY THROTTLING

Figure 6.1: *Thread based vs task based* programming model. Arrows represent control dependencies and/or data dependencies between activities.

comparison will be made regarding programming effort required, achievable performance and overhead.

6.1.1 Programming effort

An important aspect that we would like to analyze concerns the programming effort required to implement and use *thread packing* and *concurrency throttling*.

*Thread Packing* is usually implemented by changing threads-to-core affinity (e.g., by using the *taskset* Linux utility) to let threads run on specific cores or contexts of the CPU. The application structure is not changed; thus this approach does not require any additional effort to the application developer.

On the other hand, since *Concurrency Throttling* modifies the number of the threads (or processes) used by the application, a support from the programmer may be required. Indeed, except for pure functional computations which doesn’t have any internal state, in all the other cases the programmer must specify the action to be taken when a reconfiguration occurs, to correctly rearrange the internal state. In general, this cannot be done automatically since the actions to be taken depend on the specific structure and characteristics of the application. In Figure 6.2 we show a trivial example of an application processing in parallel a one-dimensional array, by partitioning it among the active threads.

When two threads are active, the array is divided so that each thread processes 3 elements of the array. When a new thread is added, we need to change the partitioning of the array. In this case, we need to split the array among 3 threads, by assigning a partition composed of 2 elements to each thread. Despite in this trivial example it was possible to easily reorganize the inter-
CHAPTER 6. ON THE RELEVANCE OF PARALLEL DESIGN PATTERNS

Figure 6.2: Example of state management when concurrency throttling is applied.

Figure 6.2: Example of state management when concurrency throttling is applied.

nal state, in general this process is time-consuming, error-prone and in some cases it may even be unfeasible. In [33] we tried to exploit some characteristics of the structure of the application to automatically perform state reconfigurations without any programmer support. However, this is still a preliminary investigation and at the moment can only be applied to a restricted set of applications. To be as general as possible and to do not impose any additional constraint, we consider in this thesis the case where the state rearrangement to be performed after a concurrency throttling operation is manually managed by the application (or runtime) programmer.

Is now clear that the programming effort required to effectively use these techniques is different. For thread packing the programmer can rely on external mechanisms (e.g., provided by the operating system) and the reconfiguration is entirely transparent to the application. On the contrary, concurrency throttling requires proper run-time support, a deep understanding of the parallel structure of the application and some intrusive modification to the application code.

6.1.2 Overhead

We now analyze the overhead introduced on the application execution when thread packing or concurrency throttling are applied. Thread packing implies moving the threads on a different (larger or smaller) set of cores. This in turns leads to migration of data between caches, introducing some performance drops during the migration of the threads. Concurrency throttling also introduces an additional delay caused by the insertion or removal of running threads and to the reorganization of the internal state, which may be significantly bigger than that caused by thread packing.

6.1.3 Achievable Performance

We now show the advantage of using concurrency throttling, by considering the common scenario where the application uses a number of threads equal to the number of available cores $n$, which is what happens by default in many runtime systems like OpenMP. We then analyze how the performance changes when the number of allocated cores is changed. For thread packing, this cor-
6.1. ON CONCURRENcy THROTTLING

responds to placing \( n \) threads on a set of \( m \) cores, with \( m \leq n \). For concurrency throttling, that corresponds in running the application with a number of threads \( m \). As performance indicator, we consider the speedup \( s \), defined as:

\[
s(x) = \frac{T_{seq}}{T(x)}
\]

i.e., the execution time of the sequential version of the application divided by the execution time of the parallel version using \( x \) threads. In Figure 6.3 we show the result of this comparison over the applications of the PARSEC benchmark. All the experiments have been conducted on an Intel Xeon Phi Knights Landing architecture, with 64 cores, each of them having 4-way SMT, for a total of 256 hardware contexts. The number of threads has been selected by using the \(-n\) parameter provided by the parsecconf tool and all the experiments have been performed by using the native input set. On the x-axis we have the number of used cores and on the y-axis the speedup. For example, if we consider canneal performance at \( x = 128 \), for thread packing, this corresponds to running the application with 256 threads running on 128 cores, while for concurrency throttling, it corresponds to running the application with 128 threads on 128 cores. Facesim and swaptions have been run up to 128 threads/cores due to limitations of the applications.

Intuitively, for the same value on the \( x \) axis, we should have a similar power consumption. However, as we can see from the figure, in some cases (bodytrack, canneal, dedup, facesim, fluidanimate, streamcluster and vips) there is a significant performance advantage in using concurrency throttling with respect to thread packing. This means that, given a power budget, by using concurrency throttling it is possible to achieve higher performance than by using thread packing. For example, in the fluidanimate, facesim and canneal cases by using concurrency throttling, for a fixed power budget, we could have performance more than two times higher than the thread packing case. On the other hand, there are applications for which there are no significant differences. This depends on the characteristics of the specific application, concerning contention of shared resources.

This analysis was done on postmortem data, i.e. we didn’t actually apply concurrency throttling, but we changed the number of threads between successive runs of the same application. We plan in the future to evaluate the effectiveness of this mechanism by actually implementing concurrency throttling on these applications to assess its real impact.

6.1.4 Concurrency Throttling Requirements

After we described advantages and disadvantages of concurrency throttling, we now analyze the characteristics an application should have in order to successfully apply concurrency throttling. The effectiveness of concurrency throttling significantly changes according to the structure of the application. In general, the more heterogeneous the threads of the application will be, the more will be difficult to predict how performance will change when the number of threads
CHAPTER 6. ON THE RELEVANCE OF PARALLEL DESIGN PATTERNS
is increased or decreased. For example, let us consider an application composed of 5 threads, where 2 threads read from the disk, 2 threads perform the computation, and one thread sends the results over the network. Since not all the threads perform the same computation, removing a thread executing I/O operations leads to a different performance degradation (and power consumption decrease) than removing one of the computing thread. This would significantly complicate the performance and power consumption prediction models since we should have one different model for each different types of threads.

**Requirement 1**

To simplify the application of concurrency throttling, the majority of application’s threads should perform a similar computation.
However, this constraint alone is not sufficient to apply concurrency throttling. Consider for example the application depicted in Figure 6.4, composed by 3 threads. Each thread applies on each received element $x$, the function $f(x) = x^2$ and sends the result to the next threads.

![Figure 6.4: Example of a non-parametric application with homogeneous threads.](image)

Although all the threads perform the same computation, it would not be possible to remove a thread from the application, since this would alter its semantic. In some particular cases, the functionality of one thread could be moved on one of the remaining threads, by using techniques like pipelined fission [172]. This method is however outside the scope of this thesis.

**Requirement 2**

To apply concurrency throttling, the application needs to be *parametric* with respect to the number of its threads, i.e. it must be possible to add or remove threads without altering the semantics of the application.

Moreover, as shown in Section 6.1.1, if the application has an internal state, such state must be rearranged to reflect the new structure of the application and to preserve the correctness of the computation. Of course, the more linear and well-defined the structure of the computation is, the more comfortable will be to perform state rearrangement.

**Requirement 3**

To apply concurrency throttling, the application should have a clear and well-defined structure, so that it would be possible for the programmer to easily rearrange the internal state of the application when threads are dynamically added or removed.

To efficiently use *concurrency throttling* we introduced strong requirements which, however, are present in most works exploiting concurrency throttling [145, 196]. As we will show in Chapter 7, these requirements are not particularly limiting since they are satisfied by many existing applications. Moreover, even if some application doesn’t have such characteristics, it would still be possible to apply our reconfiguration algorithms by exploiting the other available reconfiguration mechanisms (e.g., *thread packing* and *Dynamic Voltage and Frequency Scaling* (DVFS)). In the rest of this thesis’ Part, we will concentrate on
6.1. ON CONCURRENCY THROTTLING

Parallel patterns-based applications, since they satisfy these three properties in most cases.

6.1.5 Task-Based Models

Up to now, we described why the requirements we introduced are needed on threads-based programming. We now analyze these requirements in the context of task-based programming.

Requirement 1 In iterative applications, we may have the same tasks spawned over and over during the application execution. Let us consider the example shown in Figure 6.1. Even if the 5 tasks are heterogeneous activities with different latencies, the actual the behavior of the threads will be homogeneous in the long run, as shown in Figure 6.5, where in Period 1 and Period 2 they execute the same set of tasks. This however depends from the scheduler ability in keeping the workload well balanced among execution threads.

Requirement 2 In this case is not the application that needs to be parametric with respect to the number of threads but the runtime (since the application does not have the concept of threads). This is usually true in task-based environments, where tasks are scheduled to a pool of threads which execute them. Changing the number of threads usually only implies scheduling the tasks to a larger or smaller set of threads.

Requirement 3 In task-based execution, there will be no need to rearrange the internal state of the application. Indeed, with concurrency throttling we will change the number of threads, not the number of tasks. Threads could be added or removed transparently to the application since the application structure is decoupled from the runtime. Concurrency throttling would only be a concern of the designer of the execution support, which needs to keep the internal state of the runtime consistent. However, it is important to notice that we merely shifted the problem from the application level to the runtime level.

For example, to avoid issues related to state redistribution, OpenMP allows the programmer to change the number of threads only between two successive parallel sections, i.e. between two distinct and independent parallel regions of the application (e.g., between two different parallel loops). This limitation has an impact on techniques using concurrency throttling on OpenMP applications [195, 119, 17, 117, 118], since in all these cases concurrency throttling can only take place in some specific points of the execution, thus limiting its effectiveness. Moreover, if the application is composed of a single parallel region, concurrency throttling could not be applied at all. To partially address this problem, recently the MAESTRO project [197] exploited the Qthreads runtime [198] for OpenMP to provide alternative thread scheduling mechanisms and to
allow the application of concurrency throttling even inside a single parallel phase.

![Diagram of task-based execution](image)

Figure 6.5: Example of a homogeneity in tasks-based execution.

Despite significantly simplifying the adoption of concurrency throttling, task-based programming is not the holy grail. For example, in some cases it is not possible to express stateful computations [199] or there may be performance issues when synchronizations are present inside the tasks [200]. Furthermore, thread-based execution models are usually characterized by a more predictable performance since they remove an abstraction level, thus removing overhead and unpredictable effects introduced by the tasks scheduler. In Section 7.1 we will show a comparison we made regarding programming effort and performance between some threads-based and tasks-based runtimes showing that concerning performance, in general, there is no a solution better than the other and the choice depends on the specific scenario. Moreover some issues related to concurrency throttling, like redistribution of the internal state, would be simply shifted from the application level to the runtime level.

In Section 8.4.2 we will describe the design of a tasks-based runtime, which we will use to show the effectiveness of our reconfiguration algorithms (also) on a tasks-based dataflow applications.

To conclude, despite concurrency throttling is an intrusive approach, requiring additional effort to the programmer and possibly introducing a more considerable overhead during the reconfigurations, it allows the exploitation of configurations with better performance/power tradeoffs. Moreover, the application should satisfy specific requirement which, as we will see in the next section, are usually met by parallel patterns-based applications. This section motivated the focus we pose on concurrency throttling and, consequently, on parallel patterns.

### 6.2 Common Patterns

Applications designed by using parallel patterns usually satisfy the requirements we introduced in Section 6.1.4 for using concurrency throttling. For this reason, in this section we describe some of the most common parallel patterns.

Pattern-based frameworks provide a set of parallel patterns that solve recurrent problems in parallel programming. Some notable examples are: map,
6.2. COMMON PATTERNS

In this section we review some common parallel patterns, which we will use to build the benchmark we will describe in Chapter 7.

Several past papers have described parallel patterns by providing a formal semantics that allows patterns to be composed and nested according to specific rules [201, 202]. Rewriting rules have been derived to transform a pattern expression into an equivalent one (i.e., a different pattern composition that preserve the computation correctness), possibly able to achieve better performance. Such analysis and formalism are out of the scope of this thesis. In this section, we recall the most common patterns, some of which have been used to build our P3ARSEC benchmark suite 7. To represent the patterns, we use a synthetic syntax that simplifies the description of alternative implementation schemes.

Sequential (seq). This pattern encapsulates a portion of the “business logic” code of the application that can be used in this way as a parameter of other more complex patterns. The implementation requires to wrap the code in a function \( f : \alpha \rightarrow \beta \) with input and output parameter types \( \alpha \) and \( \beta \), respectively. For each input \( x : \alpha \) the pattern \((\text{seq } f) : \alpha \rightarrow \beta\) applies the function \( f \) on the input by producing the corresponding output \( y : \beta \) such that \( y = f(x) \). The pattern can also be applied when the input is a stream, i.e. a sequence possibly of unlimited length of items with the same type. Let \( \alpha \) stream be a sequence \((x_1, x_2, ..., x_i, ...)\) where \( x_i : \alpha \) for any \( i \). The pattern \((\text{seq } f) : \alpha \text{ stream } \rightarrow \beta \text{ stream}\) applies the function \( f \) to all the items of the input stream, which are computed in their strict sequential order, i.e. \( x_i \) before \( x_j \) iff \( i < j \).

Pipeline (pipe). The pattern works on an input stream of type \( \alpha \) stream. It models a composition of functions \( f = f_n \circ f_{n-1} \circ ... \circ f_1 \) where \( f_i : \alpha_{i-1} \rightarrow \alpha_i \) for \( i = 1, 2, ..., n \). The pipeline pattern is defined as \((\text{pipe } \Delta_1, ..., \Delta_n) : \alpha_0 \text{ stream } \rightarrow \alpha_n \text{ stream}\). Each \( \Delta_i \) is the \( i \)-th stage, that is a pattern instance having input type \( \alpha_{i-1} \) stream and output type \( \alpha_i \) stream. For each input item \( x : \alpha_0 \) the result out of the last pipeline stage is \( y : \alpha_n \) such that \( y = f_n(f_{n-1}(...f_1(x)...)) \). The parallel semantics is such that stages process in parallel distinct items of the input stream, while the same item is processed in sequence by all the stages, as depicted in Figure 6.6.

![Diagram of Pipe parallel behaviour.](image)

From an implementation viewpoint, a pipeline of sequential stages is im-
implemented by concurrent activities (e.g., threads) passing items through cooperation mechanisms (e.g., via shared buffers), as depicted in Figure 6.6.

**Farm** (farm). The pattern computes the function \( f : \alpha \rightarrow \beta \) on an input stream \( \alpha \) stream where the computations on distinct items are independent. The pattern is defined as \((\text{farm } \Delta) : \alpha \text{ stream} \rightarrow \beta \text{ stream}\) where \( \Delta \) is any pattern having input type \( \alpha \) stream and output type \( \beta \) stream. The semantics is such that all the items \( x_i : \alpha \) are processed and their output items \( y_i : \beta \) where \( y_i = f(x_i) \) computed. From the parallel semantics viewpoint, within the farm the pattern \( \Delta \) is replicated \( n \geq 1 \) times (\( n \) is a non-functional parameter of the pattern called \textit{parallelism degree}) and, in general, the input items may be computed in parallel by the different instances of \( \Delta \).

In case of a farm of sequential pattern instances, the run-time system can be implemented by a pool of identical concurrent entities (worker threads) that execute the function \( f \) on their input items. In some cases, an active entity (the \textit{emitter} thread in FastFlow [47]) can be designed to assign each input item to a worker, while in other systems the workers directly pop items from a shared data structure. Output items can be collected and their order eventually restored by a dedicated entity (a collector thread) that produces the stream of results. An example of such implementation is graphically depicted in Figure 6.7.

This pattern satisfied all the requirements we outlined in Section 6.1.4 since most of the application threads perform the same computation and since it is parametric with respect to the number of workers. Moreover, since it is (in general) a stateless computation, it would be possible to add or remove threads during application execution easily.

**Master-worker** (master-worker). This pattern works on a collection (\( \alpha \) collection) of type \( \alpha \), i.e. a set of data items \( \{x_1, x_2, \ldots, x_n\} \) of the same type \( x_i : \alpha \) for any \( i \). There is an intrinsic difference between a stream and a collection. While in a collection all the data items are available to be processed at the same time, in a stream the items are not all immediately available, but they become ready to be processed spaced by a certain and possibly unknown time interval. The pattern is defined as \((\text{master-worker } \Delta, p) : \alpha \text{ collection} \rightarrow \alpha \text{ collection}\) where \( \Delta \)
is any pattern working on an input type $\alpha$ and producing a result of the same type, while $p$ is a boolean predicate. The semantics is that the master-worker terminates when the predicate is false. Different items can be computed in parallel within the master-worker.

A master-worker of sequential pattern instances consists of a pool of concurrent workers that perform the computation on the input items delivered by a master entity. The master also receives the items back from the workers and, if the predicate $p$ is true, reschedules some items.

Similarly to the farm case, even in this case the requirements we described in Section 6.1.4 are satisfied and, usually, state re-arrangement would just involve a redistribution of the items of the input collection $\alpha$ collection.

Map ($\text{map}$). The pattern is defined as $(\text{map } f) : \alpha \text{ collection} \rightarrow \beta \text{ collection}$ and computes a function $f : \alpha \rightarrow \beta$ over all the items of an input collection whose elements have type $\alpha$. The output produced is a collection of items of type $\beta$ where each $y_i : \beta$ is $y_i = f(x_i)$. The precondition is that all the items of the input collection are independent and can be computed in parallel.

The runtime of the map pattern is similar to the one described for the farm pattern. The difference lies in the fact that since we work with a collection, the assignment of items to the worker entities can be performed either statically or dynamically. Depending on the framework, an active entity can be designed to assign input items to the workers according to a given policy.

Concurrency throttling can be easily applied on applications structured as map for the same reasons we outlined for the master-worker pattern.

Map+reduction ($\text{map+reduce}$). It is defined as $(\text{map+reduce } f, \oplus) : \alpha \text{ collection} \rightarrow \beta$, where $f : \alpha \rightarrow \beta$ and $\oplus : \beta \times \beta \rightarrow \beta$. The semantics is such that the function $f$ is applied on all the items $x_i$ of the input collection (map phase). Then, the final result of the pattern $y : \beta$ is obtained by composing all the items $y_i$ of the output collection result of the map phase by using the operator $\oplus$, i.e., $y = y_1 \oplus y_2 \oplus \ldots \oplus y_n$.

A typical implementation is the same of the map where the reduction phase can be executed serially, once all the output items have been produced, or in parallel according to a tree topology by exploiting additional properties on the operator $\oplus$ (i.e., if it is associative and commutative).

Concurrency throttling can be easily applied on applications structured as map+reduce for the same reasons we outlined for the master-worker pattern.

Composition ($\text{comp}$). This pattern is the composition of two pattern instances that work either on single items, on streams or collections. In case of collections, the composition is $(\text{comp } \Delta_1, \Delta_2) : \alpha \text{ collection} \rightarrow \gamma \text{ collection}$ where $\Delta_1$ is any pattern (e.g., map or master-worker) working on input $\alpha$ collection and that produces an output $\beta$ collection, while $\Delta_2$ is a pattern working with input type $\beta$ collection and transforming it into a type $\gamma$ collection. The semantics is that the first pattern is executed, and when its execution has finished (i.e., all the items in the input collection have been computed) the second pattern can be started by processing the collection produced by the first pattern. In case of streams, the composition semantics is applied on an item-by-item ba-
sis, i.e. each item in the input stream is processed first by $\Delta_1$ and then by $\Delta_2$ before starting to compute the next item.

The run-time system of a pattern-based framework must ensure that the two patterns within the comp instance are executed serially. In the case of collections, a barrier can be added after the call to the first pattern and before starting the second one.

**Iterator** (iterator). In its basic form this pattern iterates a pattern $\Delta$ working on a single input item (seq or comp) or on a collection of items (map, master-worker). In case of collections, the pattern is defined as $(\text{iterator } \Delta, p) : \alpha \text{ collection } \rightarrow \alpha \text{ collection}$, where $p$ is a boolean predicate. The inner pattern $\Delta$ is iterated until the predicate is true.

At the implementation level, the runtime executes the pattern for a certain number of times determined statically or at run-time. At the end of each iteration, there is an implicit barrier, since the output collection computed at iteration $i - 1$ may be used as input for the iteration $i$.

**Divide and Conquer** (dac). It can either work on a collection or a stream. It can be used to solve recursive problems by exploiting three steps: i) a divide step, in which the problem is decomposed in smaller sub-problems; ii) a conquer phase where the sub-problems are solved recursively. If the problems are not large enough, they are directly resolved by using a pattern $\Delta$ without further recursion; iii) a combine step in which the solutions of the sub-problems (produced by the pattern $\Delta$) are merged to obtain the solution of a bigger problem.

**Stencil** (stencil). The pattern works on a collection $x : \alpha \text{ collection}$ and operates in subsequent timestep. In each timestep, the stencil code updates all the elements in the collection. The new value of the element is computed as a function of the element itself and of a set of its neighbors.

### 6.3 Examples of Parallel Patterns-Based Code

In this Section we provide some examples of patterned code written in FastFlow and SkePU, to give an idea of the interface and the programming abstractions offered by some of the existing frameworks.

Over the last twenty years, many parallel programming models and frameworks based on parallel patterns and algorithmic skeletons have been proposed. In [203] it is possible to find a review of several of them. Some of these frameworks, as P$^3$L [204], ASSIST [205] and SAC [206], provide a new language used to introduce pattern abstractions already in the early phases of the software development process. More recent approaches like FastFlow [47], SkePU [159], GrPPI [207], SPar [169] and PACXX [208], rely on new features of modern C++ language. Patterns are introduced by instantiating class objects at any place in the code or by using suitable C++11 attributes as in SPar. We now describe FastFlow and SkePU since they are the frameworks we used to implement our parallel patterns-based benchmarks (which will be described in Chapter 7). Overall, we decided to use these frameworks because both of
them are well-known and currently maintained projects. In addition to this, FastFlow offers all the patterns we need for our modeling, while SkePU provides a significant subset of them.

FastFlow [47] is a C++11 header-only template library that allows the programmer to build directed graphs of streaming computations. It provides the application programmer with a variety of ready-to-use stream and data parallel patterns that may be freely composed and customized to implement complex parallel applications. The patterns provided are: pipeline, farm, map, map+reduce, master-worker, feedback-loop and sequential. Patterns are implemented with threads which communicate by using non-blocking lock-free synchronization, enabling efficient processing in high-throughput streaming scenarios [191]. Parallel patterns can be used by instantiating proper objects from the FastFlow classes. The framework has been originally designed to target shared memory multi/many cores with two main goals in mind: performance and programmability.

SkePU [209] provides a multi-backend framework for heterogeneous parallel systems. The framework is composed of a source-to-source compiler and a runtime library. Data-parallel patterns are implemented as C++ objects whose instances with their input/output arguments are called skeletons. When the SkePU compiler recognizes a C++ construct that represents a data-parallel skeleton, it rewrites the source code and generate backend-specific versions of the user functions to execute the skeleton on the selected backend. SkePU version 2.0 provides backends for sequential C++, multi-core OpenMP, GPU with CUDA and OpenCL. The following patterns are provided: Map, Reduce, MapReduce, MapOverlap and Scan.

The iterator pattern is not natively provided by these frameworks. However, by knowing that the pattern is iterated, we can still exploit this design information to optimize the code, for example by keeping the threads alive between two successive iterations of the pattern instead of destroying and creating them at each iteration.

As examples of parallel pattern-based code, we present in the following the implementation of two PARSEC benchmarks: i) Ferret that is a stream-parallel benchmark and, ii) Swaptions that is a data-parallel benchmark. The first one is implemented using the FastFlow streaming patterns, while Swaptions has been implemented with the SkePU map pattern.

As sketched in Figure 6.8, Ferret can be modelled as a single pipeline pattern of six stages where the first and last one are intrinsically sequential while the other four stages are internally concurrent. Code 6.1 shows the FastFlow parallel code. The business logic code of each pipeline stage is encapsulated in a sequential FastFlow node (ff_node_t) by implementing the svc method (a pure virtual method of the ff_node_t class). Then, each node is added to the ff_Pipe pattern respecting the pipeline order (lines 26-30). The four middle stages are further parallelized using the farm pattern created with the utility function make_Farm, which creates n replicas of the sequential ff_node_t passed as template parameter.

Code 6.2 shows the code of the SkePU version of Swaptions, which has
1 // First stage
2 struct Load: ff_node_t<long, load_data> {
3     load_data* svc(long*) { <business logic code> };
4 } In;
5 // Second stage
6 struct Segment: ff_node_t<load_data, seg_data> {
7     seg_data* svc(load_data* in) { <business-logic code> };
8 };
9 // Third stage
10 struct Extract: ff_node_t<seg_data, extr_data> {
11     extr_data* svc(seq_data* in) { <business-logic code> };
12 };
13 // Fourth stage
14 struct Index: ff_node_t<extr_data, vec_query_data> {
15     vec_query_data* svc(vec_query_data* in) { <business-logic code> };
16 };
17 // Fifth stage
18 struct Rank: ff_node_t<vec_query_data, rank_data> {
19     rank_data* svc(vec_query_data* in) { <business-logic code> };
20 };
21 // Sixth stage
22 struct Output: ff_node_t<rank_data> {
23     void* svc(rank_data* in) { <business-logic code> };
24 } Out;
25 // Creating the pipeline with farm stages
26 ff_Pipe<> pipe(In,
27     make_Farm<Segment, n>(),
28     make_Farm<Extract, n>(),
29     make_Farm<Index, n>(),
30     make_Farm<Rank, n>(),
31     Out);
32 // Pipeline execution
33 pipe.run_and_wait_end();

been parallelized with a single map pattern. First of all, the programmer must specify the function to be executed on each element of the collection (lines 2-4). The function can have an arbitrary number of parameters. In this specific case, it only takes the single element of the collection (s). Then, input and output collections must be instantiated, by using SkePU smart data containers (lines 7-8). For the input collection (line 7), we wrapped an already existing C array, by specifying the pointer to the first element and the length. For the output one (line 8) we directly use the SkePU vector. Then, the map object is created by providing the map function that encapsulates the business logic code for the computation of the single element of the input data collection (line 11). The object takes as template argument the number of additional parameters accepted by the function mapFunction (the element of the collection is implicitly provided). All the additional arguments must be specified when creating the map object. For example, if we need to provide an integer x to mapFunction, we need first to add it to its parameters. Then, line 11 would become:

```cpp
auto map = skepu2::Map<1>(mapFunction, x);
```

If needed, a specific backend runtime and a parallelism degree can be selected for the map pattern (lines 14-16). Finally, the data parallel computation is executed, inserting in the output data collection the computed results (line 19).

6.4 Summary

In this Chapter, we outlined the reasons why we are interested in applying concurrency throttling when using a self-adaptive reconfiguration algorithm. To apply concurrency throttling, applications need to have some structural characteristics, i.e. to be parametric in the number of threads and to have homogeneous threads. These requirements are satisfied by many classes of applications included, among others, parallel pattern-based applications. We introduced parallel patterns and we described the most widely used parallel patterns, together with some brief code examples showing the expressiveness and intuitiveness of pattern-based parallel programming. In the next Chapter, we will design and implement a benchmark for pattern-based applications.
Source Code 6.2: SkePU implementation of the Swaptions benchmark.

It is worth noting that when Concurrency Throttling can not be applied due to the nature of the application, or if it significantly degrades its performance, we can always rely on Thread Packing. This is the solution we adopted in Chapter 4 for the PARSEC applications to evaluate the algorithms we proposed, since in some cases Concurrency Throttling could not be applied. In a nutshell, Concurrency Throttling is something which could further improve the quality of the configurations found by our algorithms but which is not a requirement of this work.
Chapter 7

A Benchmark for Parallel Patterns-Based Applications

In this chapter, we describe the design and implementation of P³ARSEC (Parallel Patterns PARSEC), a benchmark suite for parallel patterns-based applications. We devote Section 7.1 in describing how we designed P³ARSEC. We implemented these applications by using two different pattern-based frameworks, and we use them in Section 7.2 to prove that pattern-based applications: i) are expressive enough to model a wide set of real applications; ii) that by using parallel patterns is possible to reduce the programming effort required to implement parallel applications; iii) that, despite introducing an additional abstraction level with respect to a hand-written parallel implementation, they are characterized by the same (or even better) performance than traditional programming models.

Parts of this chapter have been presented in [31, 30].

7.1 Design of the P³ARSEC Benchmark Suite

In this Section, we describe the design of the applications of the P³ARSEC benchmark suite, a parallel pattern-based version of the PARSEC benchmark suite. For a brief description of PARSEC and for a discussion of its relevance, please refer to Section 4.1.

Starting from the PTHREADS implementations available in the PARSEC suite, we designed and implemented a parallel version of each application by composing and nesting the patterns described in Section 6.2. To provide an immediate view of the patterned scheme, we use the syntax introduced in Section 6.2. While in most of the applications this description matches precisely

1https://github.com/ParaGroup/p3arsec
the structure of the implementation, for other complex benchmarks the executed patterns depend on conditions evaluated at runtime. In those cases, the description has been simplified by focusing on the most relevant computational kernels. The exact structure can be found in the P^3ARSEC source code.

Furthermore, some of the PARSEC applications have a quite complicated structure and semantics that often exploits lock-based synchronizations. To be conservative in the porting of such applications, in some cases we maintained the lock primitives that cannot be easily eliminated in the sequential portions of code passed as input parameter to the patterns instantiation.

**Blackscholes** The application belongs to the Intel RMS benchmark suite [210] (Recognition, Mining and Synthesis). It performs pricing for a portfolio of European options by numerically solving the Black-Scholes partial differential equations [211]. The Pthreads implementation divides the portfolio into work units, one for each available thread. Then, each thread calculates the prices for the options in its work unit. This algorithm is iterated multiple times to obtain the final estimation of the portfolio. This benchmark is an iterative data-parallel computation. We model it as an iterator pattern where the internal pattern is a map whose input is the collection of items composing the portfolio. The pattern scheme is, therefore:

```plaintext
iterator(map)
```

**Bodytrack** The application is aimed at tracking the body pose of a human subject by analyzing videos collected by multiple cameras. A frame contains one image from each camera. Bodytrack has two phases that are executed for each frame. In the first phase, three kernels are executed for each image. After this phase, two additional kernels are applied a number of times on the frame. Before applying a kernel, we need to ensure that the previous kernel is terminated. Accordingly, we can exploit parallelism only within each kernel.

The Pthreads version is implemented by using a thread pool, which can execute different kernels. The execution starts in the main thread and, for each frame, when a kernel needs to be executed the main thread sends a command to the pool with an identifier corresponding to the kernel type. The threads in the pool will then start to process chunks of the frame with the specified kernel. To keep the load balanced, the chunks are not statically partitioned. Each thread, after the processing of the current chunk, accesses a shared variable (using a lock) to get the identifier of the next chunk, and updates such variable.

In our pattern-based implementation, we remove the thread pool, parallelizing each kernel as a map. During the execution, every time a kernel is found the corresponding map is executed. Within the pattern runtime, load balancing is achieved by using a dynamic scheduling policy without any synchronization among the workers of the map. The structure of the benchmark is the following:
iterator(iterator(map1; map2; map3); iterator(map4; map5))

To simplify the notation, we use the symbol ";" to represent the \texttt{comp} pattern. As an example, the syntax \texttt{map1;map2;map3} is a shortcut to write

\[
\texttt{comp(map1,comp(map2,map3))}
\]

Furthermore, it is possible that between the composition of two patterns some piece of plain sequential code is executed after the completion of the first pattern and before starting the second one. In the sequel, the presence of sequential code regions between the composition of two parallel patterns will be considered implicit with the symbol ";".

\textbf{Canneal} The application minimizes the routing cost of a chip design. The algorithm applies random swaps between nodes and evaluates the cost of the new configuration. If the new configuration increases the routing cost, the algorithm performs a rollback step by swapping the elements back. While the evaluation of the elements to be swapped can be performed in parallel, swaps are executed atomically through a CAS instruction (compare-and-swap). After each iteration, a convergence condition is checked, and eventually, the benchmark is terminated. The workload is memory-intensive because the resulting memory accesses are irregular and not easily cacheable.

The \texttt{Pthreads} version follows an unstructured interaction model among threads that execute atomic instructions on shared data structures. At the end of each iteration, a barrier is executed, and each thread checks the termination condition.

We model this application as a single master-worker pattern, where the workers are sequential pattern instances executing the swaps, the evaluation and eventually the rollback actions. At the end of each iteration, the workers notify the master which in turn: \textit{i)} implements the barrier between two iterations by waiting for all the notifications by the workers; \textit{ii)} evaluates the termination condition; \textit{iii)} (re-)starts the workers’ computation if the condition is false.

\textbf{Dedup} It is a streaming application that compresses a data stream with a combination of global and local compression phases called “deduplication”.

The \texttt{Pthreads} version implements a pipeline with five stages, where each middle stage is implemented with a thread pool (the first and last stages are single-threaded). To lower the contention on communication channels, cooperation between two consecutive stages is implemented using multiple queues of fixed size. Each queue is assigned to a subset of threads in the same pool. Figure 7.1 shows a representation of the dedup pipeline. Interestingly, results out of the third stage may be transmitted directly to the last stage by-passing the fourth stage. Furthermore, the second stage can generate more output items per input item.

The first stage (\textit{Fragment}) reads the data stream from the disk and then partitions the data at fixed positions; then, it produces in output a stream of data
CHAPTER 7. A BENCHMARK FOR PARALLEL PATTERNS-BASED APPLICATIONS

Figure 7.1: General scheme of the Dedup pipeline.

chunks. Each chunk can be processed independently from the other chunks. The second stage (Refine) further partitions the input chunk into smaller fine-grained chunks generating a nested stream. The third stage (Deduplication) checks if the chunk has already been compressed in the past by accessing a hash table. If so, the chunk is marked as duplicate. The fourth stage (Compress) compresses all the chunks that are not marked as duplicate and updates the corresponding table entries. To ensure correctness in the access to the table performed by the Deduplication and the Compress stages, each bucket in the hash table is protected with a lock. Finally, the Re-order stage writes the final compressed output data into the output file. If the input chunk was marked as duplicate, it stores a "reference" to the corresponding chunk. This stage reorders the data chunks as they arrive to match the original order of the uncompressed data. This stage represents the main bottleneck of the dedup pipeline, both due to data reordering and to I/O.

The dedup benchmark can be modeled using different nestings of pipe and farm patterns. The composition is possible even though some of the stages keep an internal state which is accessed concurrently. Such state is lock-protected using the same schema used in the native Pthreads implementation. The first solution is the one with a structure closest to the original Pthreads implementation. We model the application as a pipeline, where the first stage and the last stage are seq patterns, while the three middle stages are instances of the farm pattern. We implement the by-passing mechanism between the Deduplication stage and the Compress stage by adding a flag to each data element. The flag is set if the data element must be transmitted directly to the last stage. In that case, the Compress stage only forwards the element to the final stage without any further processing. The synthetic scheme of this parallelization is the following:

1. pipe(seq1, farm(seq2), farm(seq3), farm(seq4), seq5)

By using well-known rules about farm and pipe pattern compositions that preserve the semantics [202], we can also provide an alternative implementation described as follows:

2. pipe(seq1, farm(pipe(seq2, seq3, seq4)), seq5)

As we can see, all the middle stages can be replicated within a farm pattern, i.e. each farm worker is a nested pipeline of three sequential stages. Alternatively, we can execute the stages of the inner pipeline sequentially, by replacing the pipe with a comp, thus obtaining:
3. pipe(seq1, farm(seq2; seq3; seq4), seq5)

Finally, it is possible to derive a fourth version that exploits a specialization of the farm pattern available in some frameworks (e.g., FastFlow). The ofarm pattern is a farm that preserves input/output order. When available, the use of this pattern allows to lighten the computational burden to the last stage (denoted by seq5'), that now will just write the already ordered results on disk:

4. pipe(seq1, ofarm(seq2; seq3; seq4), seq5')

Section 7.2 will show a comparison among these different versions.

**Facesim** It is an Intel RMS application simulating the motion of human faces. It applies the iterative Newton-Raphson algorithm over a sparse matrix. At every time step, different kernels are executed on a mesh (some kernels are executed multiple times within a single time step).

The **Pthreads** version uses a thread pool which, at every time step, executes different kernels on the mesh. Every time a kernel is found during the execution, it is executed by the thread pool, where each thread works on a statically assigned portion of the mesh.

In our pattern-based design, each kernel is parallelized with a map pattern. We report only a synthetic view of the overall structure of the application since there are 19 different map kernels, some of them repeated multiple times at a single time step. We focus on the seven most time-consuming kernels (the remaining 12 map kernels are parallel operations on arrays invoked multiple times during the execution):

```
iterator(map1; map2; map3; map4; map2;
  iterator(map5; map6; map7); map1; map4; map2)
```

**Ferret** It is based on a toolkit used for content-based similarity search of feature-rich data such as audio, images, video, and 3D shapes [212]. The toolkit is configured for image similarity search.

The **Pthreads** parallel implementation decomposes the application into six pipeline stages. The first and last stages are single-threaded while the other stages are configured with a thread pool each. Communication channels between pools are implemented using queues of fixed size. The ferret pipeline does not have by-passing links as in dedup (see Figure 6.8).

We model the application as a pipe pattern. Differently from dedup, all the stages access only private data. The four middle stages are instances of the farm pattern, whereas the first and last stage, in charge of I/O operations, are seq instances. As for dedup, we identified three possible nested schemes of patterns:

1. pipe(seq1, farm(seq2; seq3; seq4; seq5), seq6)
2. pipe(seq1, farm(pipe(seq2, seq3, seq4, seq5)), seq6)
3. pipe(seq1, farm(seq2; seq3; seq4; seq5), seq6)
Moreover, seq1 is composed of two phases: seq1.1 that iterates over the files in the input folder and seq1.2 that, for each file in the folder, loads the image contained in the file in the main memory. Since seq1.2 can be performed in parallel over different files, we can move it inside the farm\(^2\). This leads to the following patterns’ composition:

\[
4. \text{pipe(seq1.1, farm(seq1.2; seq2; seq3; seq4; seq5), seq6)}
\]

Also in this case in Section 7.2 we will show a comparison among such patterned schemes.

**Fluidanimate** It is another Intel RMS benchmark that uses an extension of the Smoothed Particle Hydrodynamics method to simulate an incompressible fluid. At every time step, the application executes nine kernels to compute the position of the fluid particles at the next time step. As in other benchmarks, the sequence of kernels is sequential while parallelism can be safely exploited within each kernel region.

In the Pthreads implementation, the three-dimensional space is statically divided among the threads. Each thread applies each kernel on its space partition. A barrier is executed by all the threads between two successive kernels.

In our pattern-based implementation, we design each kernel as a map pattern. Since the kernel sequence is iterated a number of times (one for each time step), the overall structure can be represented as follows:

\[
\text{iterator(map1; map2; \ldots ; map9)}
\]

**Freqmine** It is a data mining program that finds the most frequent items within a transactional dataset. It is based on the Frequent Pattern Tree data structure and executes the Frequent Pattern Growth algorithm [213]. This data-mining application uses a compact tree data structure to store information about frequent patterns of the transaction database. Seven kernels are identified in the application, where the last kernel is executed multiple times.

The Pthreads parallelization is not present in PARSEC while the standard version is an OpenMP one. Each kernel is parallelized using the OpenMP 2.0 \texttt{parallel-for} construct.

In our version, each kernel corresponds to a map pattern, the last one iterated a number of times:

\[
\text{map1; map2; \ldots ; map6; iterator(map7)}
\]

**Raytrace** This application consists in a graphical render aimed at generating animated 3D scenes by using a hierarchical grid raytracing algorithm. A kernel is executed at each frame.

In the Pthreads version, the kernel is parallelized by partitioning the 3D scene among the threads. The work is dynamically partitioned and, similarly

\(^2\)The same technique can be applied to the other two alternative pattern compositions.
to the bodytrack Pthreads implementation, once a thread finishes to process a partition, it gets another one in order to keep the load balanced.

The application can be modeled as a map iterated a fixed number of times. Differently from blackscholes, the computation is extremely unbalanced, and a proper dynamic scheduling of the partitions is of great importance. Furthermore, the computational weight of each map iteration is low while the number of iterations is high. The patterned scheme can be expressed as:

\[ \text{iterator(map)} \]

**Streamcluster**  It is an application that solves the online clustering problem over incoming streaming data. The program consists of a sequence of loops whose iterations can be executed in parallel. Different loops are executed sequentially by using barriers, and they are interleaved by serial regions of code whose length impacts the overall speedup.

The computational kernel consists of two phases. The first iterates a composition of a map+reduce and a number of map instances (that are in turn iterated multiples times). The second phase, working on different data, repeats the same steps exactly one time. The simplified patterned structure can be expressed as follows:

\[
// \text{Phase 1:} \\
i \text{iterator(map+reduce; map1; iterator(map2; map3; map4);} \\
\text{iterator(map5; map6; map7));} \\
// \text{Phase 2} \\
\text{map+reduce; map1; iterator(map2; map3; map4);} \\
\text{iterator(map5; map6; map7)}
\]

**Swaptions**  This application is based on the Heath-Jarrow-Morton (HJM) method [214] to price a portfolio of financial options.

The Pthreads parallel version divides the data structures of the program into blocks equal to the number of threads and assigns one block to each thread. The threads are in charge of applying the method to the options within their partition.

This benchmark has a simple structure that can be modeled as a single map pattern, where the input is a collection of items representing the swaptions portfolio.

**Vips**  It is based on the VASARI Image Processing System [215] and includes basic image processing kernels such as affine transformations and convolutions. This benchmark is a domain-specific runtime system that can be used for image manipulation.

In the Pthreads version, the user specifies a function to get the next partition. Each thread executes a loop where at each iteration: i) it gets a new partition of the image by calling the function specified by the user; ii) the partition is processed by using another function specified by the user; iii) the end
of the processing on the current partition is notified to the main thread by using a POSIX semaphore. The main thread calls a user-defined function at each notification.

Although it may look like a data parallel computation, vips can also be modeled as a stream parallel computation. Indeed, since the function to get the next image partition is specified by the user, we can not access the entire image at once and decide how to partition it. For this reason, we model this benchmark as a pipe, where the first stage is a farm where each worker retrieves a partition and processes it by using the functions provided by the user. The last stage of the pipeline is sequential and calls the progress function specified by the user. The structure is expressed as follows:

\[
\text{pipe} \left( \text{farm(seq1)}, \text{seq2} \right)
\]

**x264** This application has been considered stream parallel, although it has a complex structure and interaction among its stages. In [216] the authors have presented this application as a wavefront algorithm instead of a stream parallel one. In P³ARSEC we do not implement this application since, due to its complexity, is not possible to easily separate the parallelism management from the functional code. Moreover, besides requiring domain-specific knowledge, this application cannot be easily expressed by using the available patterns.

### 7.2 Evaluation

The new suite P³ARSEC is provided as an extension of the original PARSEC suite and can be executed with the same tools used to run the native suite, e.g. the parsecmgmt tool. All the benchmarks have been implemented in FastFlow [217], while some data-parallel applications have also a SkePU [159] implementation. We verified the correctness of all the implemented benchmarks with the corresponding original sequential and Pthreads implementations.

In the analysis, we focus both on the programming effort and on the performance achieved. The comparison is made with the parallel versions already available in PARSEC (see Table 4.1), and with the task-based implementations written in OmpSs and presented in [186]³. Their work covers most of the PARSEC applications except raytrace and vips. For x264 the authors provided an implementation that maps one to one the pthreads version (i.e., thread creations are replaced with task spawns and thread joining with task waiting). Results regarding performance and code complexity are the same of the pthreads version and are not reported in the remaining part of this section. Furthermore, the authors declared a performance improvement compared with pthreads up to 42% in bodytrack and dedup. By studying their

³We would like to thank the authors for making their source code publicly available at https://pm.bsc.es/gitlab/benchmarks/parsec-ompss. We used the most recent version available at the time of writing this thesis (commit ea319e57)
implementations, we found that this advantage is mainly due to some optimizations and code rewriting that changed the PARSEC sequential semantics, i.e. their output is different from the one produced by the original sequential and Pthreads versions. To be more precise, in their implementation of bodytrack consecutive frames are processed in parallel, while according to the algorithm semantics the parallelism can be exploited inside a frame but not between frames, since the computation of a frame depends on the result of the previous one. This produces an output which is different from the original one. In dedup, the output produced by the OmpSs version is not deterministic, and the dedup decompressor (provided with the original PARSEC benchmark) is not able to decompress it. Since we want to strictly preserve the original semantics of the applications, we do not consider these implementations. In our evaluation, we decided to do not modify the original reference implementations (Pthreads, OpenMP and TBB) since the purpose of this work is not to optimize the PARSEC benchmarks but to show that they can be parallelized using parallel patterns obtaining similar performance figures with lower lines of code and lower code churn.

In the following, we first evaluate the programming effort and then the performance results.

7.2.1 Programming Effort

To analyze the programming effort required to parallelize each benchmark with different parallel programming approaches, we use as metrics the Lines-Of-Code (LOC) and the Code Churn. Evaluating the programming effort objectively is a difficult task, and no universally accepted metrics exist. We decided to use the LOC and Code Churn metrics since they are often used as proxy metrics to evaluate programmability [186, 218, 219].

7.2.1.1 Lines of Code

This metric is commonly used in software engineering to measure code and programming complexity [218]. For each benchmark, we considered only the source files required to implement the parallelization or modified during the parallelization (the other files are the same in all the versions). These files include the definition of data structures used for thread communications, synchronization mechanisms and the files containing calls to the different parallel programming frameworks. To have a fair comparison, these files have been normalized by formatting them according to a fixed programming style (e.g., brackets on the same line of the statement, single-line if, and so forth). After that, we removed empty lines, comments, and sections of code that are not executed due to inactive macros. The measures have been normalized with respect to the Pthreads version (i.e. Pthreads is always 1, a value higher than 1 means more lines of code and lower than 1 means fewer lines of code).
7.2.1.2 Code Churn

A useful metric to estimate software complexity is the code churn [219, 220], defined as the number of lines modified and added with respect to a previous version. In our case, we consider the code churn of each parallel version with respect to the original sequential code. Starting from the sequential code, two different parallel implementations may have a similar number of code lines. However, if an implementation needs to modify and introduce a higher number of lines, this means that the effort required is likely higher than the one needed to implement the other versions. This metric is computed on the files normalized with the same process described for Lines of Code.

7.2.1.3 Discussion

We analyze the two metrics over all the benchmarks and over all the parallel versions. The results are shown in Figure 7.2 and 7.3. Note that, when a bar is missing in the plot, it means that the implementation with the corresponding framework is not available.

![Normalized Lines of Code](image)

Figure 7.2: Lines of Code (LOC) of the different parallel implementations, normalized between 0 and 1 with respect to the Pthreads version (the lower the better).

On freqmine and swaptions there are no particular differences between the implementations. In canneal, FastFlow and Omr5s versions have a slightly higher code churn, since Pthreads code is very similar to the sequential one (the same functional part is executed by \( \text{n} \) threads).

Concerning blackscholes, bodytrack, facesim and raytrace, the Pthreads implementation has a higher LOC and code churn, because of thread pools implementations in the different benchmarks, which for blackscholes is simply a wrapping of Pthreads calls to simplify threads management. In blackscholes all the other implementations are equivalent, with Omr5s having a slightly higher code churn. The TBB implementation of bodytrack has around double the code churn of FastFlow and OpenMP. This happens because our FastFlow

\footnote{For reproducibility of results, we provide the script used to compute the metrics in the P3ARSEC repository, under the scripts/ folder.}
Figure 7.3: Code Churn (i.e. number of modified and added lines of code) of the different parallel implementations with respect to the original sequential implementation (the lower the better).

implementation widely exploits C++ lambda expressions that simplify code development. On the contrary, the TBB version available in the PARSEC suite does not exploit this C++ feature, forcing the programmer of the TBB version to move and rewrite code which would not have been necessary if lambda were used. As discussed earlier, we did not change the code of the reference applications since this is not the purpose of this work. In facesim, despite the LOC of the FastFlow version is slightly higher than OmpSs, the code churn of OmpSs is much higher since FastFlow version modified only a minimal part of the sequential code.

In the FastFlow versions of dedup and ferret, we implemented different pattern compositions. We show the metrics of the version characterized by the best performance (discussed in Section 7.2.2). The other alternative FastFlow versions have similar measures. For both dedup and ferret, the FastFlow code has significantly lower LOC and code churn, since the Pthreads version also needs to implement the threading support and all the data structures required to let the threads communicate and synchronize with each other. Furthermore, in dedup, the advantage is even more significant since we were able to remove all the code lines related to data reordering, which in our case is implicit in the ofarm pattern (pattern composition number 4.). The TBB code of ferret is slightly longer, and more lines have been modified.

The Pthreads version of fluidanimate has a higher LOC and code churn because of a hand-written synchronization primitive (a spin-wait barrier) which has been implemented and used to separate the different parallel kernels. This is not needed in FastFlow since this is implicit at the end of the map pattern. The OmpSs implementation has a higher code churn as well, due to a routine which is used to create a data structure used to enforce non-trivial dependencies between the parallel tasks. The TBB code has a higher code churn due to the specific parallelization design.

In streamcluster, the pattern-based implementations (FastFlow and SkePU) have the lowest LOC and code churn. These metrics are higher for Pthreads since, also in this case, a spin-wait barrier implementation is provided. OmpSs
has a higher code churn as well, due to the rewriting of some processing routines, but also to the introduction of additional parallelizations of some sections with respect to Pthreads and FastFlow implementations.

In vips, the FastFlow version has a slightly higher LOC and code churn (~20 lines). This happens because this benchmark is a framework which can be customized with code specified by its users. It has been designed to be parallelized with Pthreads and has some stringent constraints and assumptions on the code provided by its users. However, being able to design a different parallelization by only modifying few tens of lines of code, while still preserving the same design and semantics, is a significant result.

### 7.2.2 Performance Evaluation

In the following, we describe the performance results achieved on three different multi-core architectures. They are described in Table 7.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Server</td>
<td>Dual-socket NUMA machine with two Intel Xeon E5-2695 Ivy Bridge CPUs running at 2.40GHz featuring 24 cores (12 per socket). Each hyper-threaded core has 32KB private L1, 256KB private L2 and 30MB of L3 shared with the cores on the same socket. The machine has 64GB of DDR3 RAM.</td>
<td>Linux 3.14.49 x86_64 shipped with CentOS 7.1. Available compiler gcc version 4.8.5.</td>
</tr>
<tr>
<td>Intel Xeon Phi</td>
<td>Machine with the Intel Xeon Phi model 7210 (codename Knights Landing, KNL). The KNL is equipped with 32 tiles (each with two cores) working at 1.3 GHz, interconnected by an on-chip mesh network. Each core (4-way Hyper-Threading) has 32 KB L1d private cache and a L2 cache of 1 MB shared with the sibling core on the same tile. The machine is configured with 96 GB of DDR4 RAM with 16 GB of high-speed on-package MCDRAM configured in cache mode.</td>
<td>Linux 3.10.0 x86_64 shipped with CentOS 7.2. Available compiler gcc version 4.8.5.</td>
</tr>
<tr>
<td>IBM Power8 Server</td>
<td>Dual-socket IBM server 8247-42L with two Power8 processors each with ten cores (total 20 cores) working at 3.69GHz. Each core (8-way SMT) has private L1d and L2 caches of 64 KB and 512 KB, and a shared on-chip L3 cache of 8 MB per core. The machine has 64 GB of RAM.</td>
<td>Linux 4.4.0-47 ppc64 shipped with Ubuntu 16.04. Available compiler gcc version 5.4.0.</td>
</tr>
</tbody>
</table>

Table 7.1: Multi-core machines used in the performance evaluation.

### 7.2.2.1 Experimental settings

In all the three architectures we used FastFlow version 2.1, SkePU version 2 and OmpSs version 16.06.3. The source codes of all the parallel versions have
been compiled with the -O3 flag. For the evaluation we used the PARSEC native input set, to obtain results representative of real-world program executions. The parsecmgmt tool has been used for launching the original PARSEC benchmarks and the FastFlow and SkePU implementations. For the OmpSs implementations, we used the scripts released by the authors. In the parallel versions of the benchmarks, we need to specify the concurrency degree \( n \) to use, which, except for dedup and ferret, corresponds to the number of threads executed. We used different values for \( n \), ranging from 1 to the number of threads contexts available in the used architecture (i.e., 48 in the Intel Xeon Server, 256 in the Intel Xeon Phi and 160 in the IBM Power 8 Server). The only exception to this rule is swaptions, which cannot be executed with more than 128 threads, due to limitations in the input set provided. The canneal, raytrace and vips benchmarks cannot be compiled on the IBM Power architecture due to architecture specific assembler instructions used in the original implementations.

### 7.2.2.2 Discussion

The time measured is the one spent in the so-called region of interest (ROI), which includes all parts sensitive to the parallelization. This approach is commonly adopted when comparing different parallelizations of the same application [186]. Each program has been run multiple times, and the average results are shown (the standard deviation is always negligible and it is not shown for readability reasons). All the benchmarks have been executed with the original parameters provided by PARSEC. The results are shown in Figure 7.4, where the best execution times of the benchmarks for each version, obtained by varying the \( n \) parameter, have been normalized to the PARSEC reference implementation (i.e., OpenMP for freqmine, Pthreads for all the others benchmarks). Accordingly, values lower than 1 represent cases with execution time lower than the one of the reference PARSEC implementation. For completeness, Table 7.3 reports the values of the best execution times of the various versions on the different architectures. Detailed performance results are reported in Appendix B.

Small differences and discrepancies in the results (between different versions of the same benchmark and between different architectures) are reasonably due to differences in the compiler, architecture and by the intrinsic differences and optimizations in the runtime of the frameworks used. Concerning architecture differences, the IBM Power 8 implements an 8-way Simultaneous Multi-Threading (SMT), whereas the Intel Xeon Phi and the Intel Xeon server implement a 4-way and a 2-way Hyper-Threading, respectively. The OmpSs implementations of blackscholes, canneal and fluidanimate executed on the Intel Xeon Phi give poor performance results. This is because currently the OmpSs runtime has not been optimized for this new kind of platform. In the

5Other benchmark specific flags are those specified by default in the Makefiles distributed by PARSEC.
CHAPTER 7. A BENCHMARK FOR PARALLEL PATTERNS-BASED APPLICATIONS

Figure 7.4: Best execution times normalized with respect to the PARSEC reference (i.e. OpenMP for freqmine, Pthreads for the remaining benchmarks)

In conclusion, we will discuss the most remarkable differences between the analyzed versions.

For dedup, we show in Fig 7.4 only the best FastFlow version (the one with scheme pipe(seq1, ofarm(seq2, seq3, seq4), seq5')). This version is significantly faster than the Pthreads one since it removes all the logic related to data reordering from the seq5 stage, leaving only the writing of the data on disk. The improvement is less evident in the Intel Xeon Phi architecture since the writing part of the seq5 stage is slower than on the other architectures due to the much lower clock, thus reducing the impact of this optimization. As shown in Table 7.2, the performance of this patterned version is 26% higher than the one of the other patterned versions that are more similar to the original Pthreads implementation. This is an interesting case...
where patterns composition allows the programmer to prototype alternative versions that are more efficient than the initial one, by changing just a few lines of code (less than 10).

Despite different versions could also be implemented with other programming models, this would require expressing again from scratch all the communications and the data dependencies between different parts of the parallel application. This is an error-prone task and could significantly increase the code length. On the contrary, in pattern-based model dependencies and communications are implicitly coded in the pattern.

Table 7.2: Best speedups of different parallel patterns for dedup and ferret. Numbers refer to the different patterns compositions described in Section 7.1.

<table>
<thead>
<tr>
<th>ARCH.</th>
<th>BENCH</th>
<th>1.</th>
<th>2.</th>
<th>3.</th>
<th>4.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Server</td>
<td>dedup</td>
<td>9.25</td>
<td>7.36</td>
<td>8.74</td>
<td>9.26</td>
</tr>
<tr>
<td></td>
<td>ferret</td>
<td>25.44</td>
<td>24.48</td>
<td>25.89</td>
<td>25.89</td>
</tr>
<tr>
<td>Intel Xeon Phi</td>
<td>dedup</td>
<td>6.22</td>
<td>6.54</td>
<td>6.32</td>
<td>6.6</td>
</tr>
<tr>
<td></td>
<td>ferret</td>
<td>51.13</td>
<td>52.9</td>
<td>55.69</td>
<td>92.6</td>
</tr>
<tr>
<td>IBM Power 8</td>
<td>dedup</td>
<td>10.79</td>
<td>12.07</td>
<td>12.61</td>
<td>13.59</td>
</tr>
<tr>
<td></td>
<td>ferret</td>
<td>25.53</td>
<td>23.79</td>
<td>25.32</td>
<td>35.2</td>
</tr>
</tbody>
</table>

In facesim both the FastFlow and the OmpSs versions outperform the Pthreads parallelization (up to 40% faster). This is mainly due to implementation choices adopted in the different versions. In Pthreads, when a parallel kernel is found during the execution, one abstraction of a mesh partition is inserted for each thread in a shared queue, accessed by all threads and protected by locks. Instead, in the other two implementations a partition is statically assigned to each thread without any need to access any shared data structure, thus achieving better speedup. Indeed, as shown in Figure 7.5, while the different versions are equivalent with low concurrency levels, the Pthreads version starts to perform poorly when more threads are used, due to the high contention on this shared queue. A parallelization strategy similar to that of FastFlow could be probably used in the other programming models as well. However, as described earlier, we decided not to modify the original reference implementations since it is not the purpose of this work.

For the pattern-based implementation of ferret, we report in Table 7.2 the results of all the alternative pattern implementations. The pipe(seq1.1, farm(seq1.2; seq2; seq3; seq4; seq5), seq6) version is 80% more performing than the first pattern-based implementation, closer to the design of the Pthreads version, showing again the importance of and flexibility of patterns composition to introduce optimizations. OmpSs and FastFlow versions of ferret produce the best performance gain over the Intel Xeon Phi and IBM Power 8 server. This happens because, differently from Pthreads and TBB, both versions parallelize the load of the images from the file (by separating seq1.1 from seq1.2). The same effect does not occur on the Intel Xeon Server.
Figure 7.5: facesim speedup, for different versions and on different architectures.

due to the lower number of cores, since the images loading stage becomes a bottleneck only when using a high number of threads.

On fluidanimate, we measured significant improvements of the pattern-based implementation with respect to the Pthreads one on the Intel Xeon Phi and IBM Power 8 Server. This is mainly due to the different implementation of the barrier provided by the various frameworks. A barrier (implicit in the pattern-based approach) is executed after each parallel kernel. The one implemented in FastFlow is more efficient than the one used by Pthreads, thus leading to this performance gap. This performance difference is remarkable only at high concurrency levels, as shown in Figure 7.6 and does not occur on the Intel Xeon Server, since it has just 48 threads contexts.

On streamcluster, by parallelizing the kernels with map patterns, we greatly simplified the code. This made it possible to remove some unnecessary synchronizations (e.g., in the pspeedy function), leading to a performance improvement up to 40% on the Intel Xeon Phi. Such inefficiencies in the Pthreads implementation occur because of an intricate design, which led to a non-optimized implementation. On the other hand, the pattern based design of streamcluster is more straightforward and more effective. Moreover, we did not parallelize some tiny functions which were parallelized in the Pthreads and OmpSs version. For such functions, the overhead of the parallelization is not worth and slows down the entire application.

The original TBB implementation of swaptions produced poor performance results with respect to other parallel implementations. We were able to reduce this gap by changing the size of the block scheduled to the different threads.
## 7.2. EVALUATION

### 7.2.2.3 Summary

To summarize the results, we achieved an average reduction of 26% in the lines of code (in both FastFlow and SkePU) compared with the original Pthreads implementation, and an average reduction of 3% with respect to the OmpSs implementations. In the best case, we reduced the lines of code up to 87% with respect to Pthreads, and 14% compared with the OmpSs versions. The code churn is in average 58% lower than Pthreads and 34% lower than OmpSs version. Concerning the performance, the FastFlow implementations obtained an average performance gain of 14%, with a maximum gain of 42% and a maximum loss of 9% with respect to the Pthreads one. Considering the benchmarks implemented with SkePU, we obtained an average performance gain of 7% (maximum gain of 45%, maximum loss of 11%). Finally, OmpSs implementations obtained an average gain of 2% (maximum gain of 37%, maximum loss of 23%) with respect to the Pthreads implementation.

This evaluation confirmed that the pattern-based parallel programming approach reduces the lines of code and code churn without impairing performance. Also, in several cases we were able to improve the performance by rapidly prototyping alternative pattern compositions.

---

For the sake of fairness we did not consider in this comparison the results of OmpSs implementations of blackscholes, canneal and fluidanimate on the Intel Xeon Phi.
Table 7.3: Best execution times (sec.). For the parallel versions, they are obtained by varying the concurrency degree. BS (blackscholes), BD (bodytrack), CN (canneal), DD (dedup), FC (facesim), FR (ferret), FL (fluidanimate), FQ (freqmine), RT (raytrace), SC (streamcluster), SW (swaptions), VP (vips). Concerning the missing data, in the OmpSs implementation of the benchmark suite, raytrace and vips are not available. Moreover, the output produced by dedup and bodytrack is different from the one produced by the original PARSEC implementation and therefore their related results are not shown. Finally, canneal, raytrace and vips benchmarks cannot be compiled on the IBM Power architecture due to some architecture specific assembler instructions.

7.3 Summary

In this chapter, we described the parallel patterns-based benchmark suite we designed and implemented. We first described how we model application of the PARSEC benchmark suite by using parallel patterns. The implementation of these pattern-based applications led to the creation of Parallel PARSEC, a benchmark suite for parallel pattern-based applications. We first proved that parallel patterns are a good candidate for implementing parallel applications since they are flexible enough to model a wide set of real applications. Moreover, we demonstrated that implementing an application designed by using parallel patterns requires a lower programming effort than implementing it with other programming models as Pthreads or tasks-based models. In addition to that, performances are usually comparable with those of more commonly used model. In some cases, thanks to a simpler design, performances are even better than those achieved by using Pthreads, OpenMP, TBB or OmpSs. Despite being a relevant contribution per se, the benchmark could be used in the future to prove the effectiveness of some reconfiguration techniques such as concurrency throttling.
Part IV
Framework Design

In this Part of the thesis, we describe the design of a new customizable C++ framework to enforce explicit performance and power consumption requirements on parallel applications. We called this framework Nornir$^7$ and we released it under open source license. Current implementation targets applications running on shared memory multicore machine and is publicly available at http://danieledesensi.github.io/nornir/. To reach our target we built an entire software stack, achieving three main goals:

- We used Nornir to implement the algorithms we described in Chapter 3 and 5 and to evaluate them by actually enforcing performance and power consumption requirements on real applications. This is an important step since usually self-adaptive algorithms are validated through simulations.

- We built a software which could be used in real scenarios to enforce power consumption and performance requirements on parallel applications.

- We provide monitoring and actuation tools which could also be used different contexts from the one we are considering in this thesis.

The framework can be used by different actors:

- Application users can use the framework to enforce performance and power consumption requirements on their applications.

- Application programmers can use the tools provided by Nornir to interface an application to the manager (i.e., the entity executing the self-adaptive reconfiguration algorithm).

$^7$In Norse mythology, the nornir are female beings who decide and rule the destiny of men. Perceived as background figures, they stand at the feet of Yggdrasill (the tree of life), pouring water over it so that its branches will not rot. We chose this name for our framework since we liked the analogy with an external entity controlling the destiny and the behavior of parallel applications.
Despite some state of the art self-adaptive strategies are already available in NORNIR, it is possible for the self-adaptive strategy designer to implement new strategies or to customize existing ones.

In the following chapters, when referring to designers, programmers or users using italic, we will intend the different actors using NORNIR.

In Chapter 8 we describe how users and programmers can use the framework. Then, in Chapter 9 we show how it can be customized to add new reconfiguration policies, new actuators or to support different monitoring infrastructures.
Chapter 8

**Using the Framework**

In this chapter, we will first describe how Nornir can be used to express requirements on an application. Then, we will describe the possible ways to interface an application to the Nornir manager, which is in charge of executing the MAPE loop and reconfiguring the application. The manager runs in a separate thread/process and interacts with the application to gather monitoring data and to apply reconfiguration decisions (e.g., changing the number of threads), to enforce the user’s requirements.

The application user can express performance requirements on the metrics reported in Table 8.1 and energy and power consumption requirements, on the metrics shown in Table 8.2.

The requirements will be specified by the user through an XML configuration file. Moreover, it is possible to express constraints on more than one metric at the same time. For example, code snippet 8.1 shows how to require Nornir to find a configuration characterized by a power consumption lower than 50 Watts and a latency lower than 30ms. Since more than one configuration could have such characteristics, the user wants Nornir to select the one characterized by the highest throughput. The requirements are not static and could be dynamically changed by the user while the application is running.

The user can also choose the self-adaptive algorithm to be used to enforce such requirements (some of them have been shown in Part II). Moreover, by using the configuration file, the user can also specify other parameters, such as the actuators to be used when reconfiguring the application. Eventually, it is possible to monitor in real-time how Nornir is reconfiguring the application by using the dashboard shown in Figure 8.1. The dashboard is based on Grafana\(^1\) and is fully customizable, allowing to specify the metrics to monitor, the time span, the refresh rate and also to analyze aggregate data. It is

\(^1\)https://grafana.com/
## Metric S Description (Streaming) Description (Batch)

<table>
<thead>
<tr>
<th>Metric</th>
<th>S</th>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>( R )</td>
<td>Number of stream elements processed per second.</td>
<td>Number of iterations executed per second.</td>
</tr>
<tr>
<td>Latency</td>
<td>( L )</td>
<td>Time required to process a single stream element.</td>
<td>Time required to perform a single iteration.</td>
</tr>
</tbody>
</table>

**Completion Time**

Time required to process all the elements on the stream. The user needs to specify the expected length of the stream. By doing so, Nornir can estimate the completion time as the ratio between the number of the number of elements still to be processed and the current throughput. For this reason, this requirement can only be specified when the length of the stream is known a priori.

Time required to perform all the iterations. Similarly to the streaming case, the number of iterations need to be known a priori.

**Utilization Factor** \( \rho \)

Represents the utilization of the application, i.e. the fraction of time spent processing stream elements (between 0 and 1). \( 1 - \rho \) is the fraction of time wasted by the application waiting for new data to arrive from the stream. A low \( \rho \) means that the resources allocated to the application are not fully utilized. It must not be confused with efficiency, which measures the ability of the parallel application in making a good usage of the available resources. Indeed, while efficiency is a property of the application and depends on its structure, utilization is not a property of the application itself but it only depends on the rate of data arriving at application. Accordingly, an application could have a high efficiency but a low utilization. Moreover note that, as explained in Section 3.2.3, this definition of \( \rho \) is different from the one used in queuing theory. Moreover note that, as described in Section 3.2.4, the definition of \( \rho \) we are using is different from the standard queuing theory definition since in our case we always have \( \rho \leq 1 \).

Not applicable on batch applications since \( \rho \) is always 1 (because the data is always available to the application and it never needs to wait for new data to arrive).

### Table 8.1: Performance requirements that can be specified by the user.

Moreover possible to set alerts if some condition occurs, for example, to receive e-mails when some metric exceeds a specified threshold.

We briefly recall that the MAPE loop is composed (among others) by a Monitor phase, in which collects data about the application and by an Execute phase, in which the configuration found by the Plan phase is applied. The MAPE loop is executed by an external entity, provided by Nornir, called manager. Monitoring some metrics (e.g., power consumption, temperature or current clock frequency) do not require an explicit interaction between the manager and
Table 8.2: Power consumption requirements that can be specified by the user.

<table>
<thead>
<tr>
<th>Metric</th>
<th>S</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Consumption</td>
<td>$P$</td>
<td>Instantaneous power consumption. Since current operating systems don't provide mechanisms to monitor the individual power consumption of each application, this may correspond to the system power consumption.</td>
</tr>
<tr>
<td>Energy</td>
<td>$E$</td>
<td>Power integrated over time. It can be both specified as energy required to process a single stream element (or iteration in batch applications) or to process all the elements. In the latter case, the number of elements to be received (or the number of iterations to be executed) must be known a priori, since energy will be estimated as $E = P \times T$.</td>
</tr>
</tbody>
</table>

Source Code 8.1: Example of user requirements.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nornirParameters>
  <requirements>
    <throughput>MAX</throughput>
    <latency>30</latency>
    <powerConsumption>50</powerConsumption>
  </requirements>
</nornirParameters>
```

the application. For example, power consumption will be collected by measuring that of the entire system (or part of it) by using the available internal counters or external power meters. The collection of other metrics, like performance and application specific data, may require an explicit interaction of the manager with the application. Nornir offers different possibilities to application programmers for realizing this interaction, allowing to chose the desired trade-off between programming effort and flexibility. In general, more intrusive approaches collect more precise metrics and leads to better solutions, while requiring a higher effort to the programmer. However, such solutions allow Nornir to access some actuators which may be specific to the application or to the runtime system, extending the range of possible configurations and allowing the self-adaptive algorithm to take better decisions. On the other hand, some approaches will enable the user to directly interface his application to Nornir, without any programmer intervention, despite this may lead to suboptimal decisions during the plan phase.

In Figure 8.2 we depict a flowchart showing the different possibilities available to the programmer and the sections where they are described. In the following, we will discuss the different opportunities provided by Nornir to attach a manager to the application, starting from the less optimal and less intrusive
CHAPTER 8. USING THE FRAMEWORK

Figure 8.1: Example of the dashboard that can be used by the user for realtime monitoring of his application.

Figure 8.2: Flowchart describing the possible choices for the application programmer to interface Nornir manager to an application.

ones. The solutions we will present in Section 8.1, Section 8.2 and Section 8.3 are the most appropriate for legacy applications, while in Section 8.4 we provide support for new applications.

Parts of this chapter have been published in [35, 28, 25].

8.1 Black-Box Interaction

The simplest solution for the user is to use Nornir on his existing application, without any modification to the source code and without any programmer intervention. In some cases this may be the only feasible solution since the programmer may not have the possibility to modify and recompile the applica-
8.2 INSTRUMENTATION

If the user wants to specify some constraint on his foobar application, he can run the application by using the application launcher provided by Nornir and by specifying the XML configuration file, as shown in Code 8.2.

```
manager-blackbox --parameters parameters.xml
   --application ./foobar
```

Source Code 8.2: Example of attachment of a Nornir manager to an already existing application.

The Nornir manager will run in a separate process and will not interact directly with the application. Nornir will monitor the application only by relying on performance counters, for example by monitoring the number of assembler instructions executed per time unit (i.e., instructions per second (IPS)). Since it is not possible for Nornir to monitor detailed metrics like latency and throughput, the user can only express his performance requirements for the application regarding IPS. Correlating the IPS to the actual application throughput is not an easy task and not so intuitive from the user perspective. Moreover, as shown in [67, 84, 101] performance counters may not be a good performance proxy since they are not always strictly correlated to the actual application-level performance. The following approaches will solve this issue, requiring, however, an explicit intervention from the application programmer.

8.2 Instrumentation

Alternatively, if the programmer can modify the application, he can explicitly interface it to a Nornir manager running in a separate process. This is the solution we adopted in Chapter 4 to allow all the self-adaptive algorithms we considered in the evaluation to control the PARSEC applications.

As motivated in Chapter 2, self-adaptive algorithms mainly work on iterative applications. For streaming applications, an iteration would correspond to the processing of an element received from the stream. To be homogeneous in the following, even when considering streaming applications, we will usually refer generically to the term iteration.

The underlying idea is to insert few instrumentation calls in the existing application. These calls will be invoked at each iteration of the application, to collect all the performance data needed by the manager to take reconfiguration decisions (e.g., latency and throughput). Instead of sending data to the manager at each iteration, data is stored locally, and it is sent to the manager only when the manager explicitly requests such data. All these operations are implicitly done when executing the instrumentation calls, as shown in Code 8.3.

On the left, we have the original, already existing, streaming application and on the right the same application after it has been instrumented. Let us focus on the right snippet. In line 2, Nornir opens a connection towards the
StreamElement* s;
while(s = receive()){
    process(s);
}

using nornir::Instrumenter;
Instrumenter r("parameters.xml");
StreamElement* s;
while(s = receive()){
    r.begin();
    process(s);
    r.end();
}
    r.terminate();

Source Code 8.3: Example of streaming application instrumentation.

manager (which is running in a separate process) and sends to it the configuration file (containing, among others, user’s requirements). Then, for each stream element, after receiving it from the stream (line 4), the processing is wrapped between 2 calls (lines 5 and 7). Timestamps collected during the r.begin() and r.end() calls are stored and used by the Instrumenter to derive performance metrics.

When the manager needs to collect monitoring data about the application (i.e., once per control step), it will send a request to the Instrumenter. The r.begin() call verifies if there is any pending request and, if this is the case, the monitored data is sent to the manager.

To understand how throughput, latency and utilization factor are derived from the timestamps, let’s focus on a single control step. We define with $K_{\text{begin}}(i)$ and $K_{\text{end}}(i)$ the timestamps associated to r.begin() and r.end() calls over the $i$-th iteration. Moreover, let’s suppose that at iteration $\alpha$ the Instrumenter finds a pending monitoring request and that no other monitoring requests are received until iteration $\beta$. The distance between $\alpha$ and $\beta$ depends on the length of the control step used by the manager. We define with $s = \beta - \alpha$ the number of iteration performed by the application during a control step. The relationship between these quantities is clarified in Figure 8.3.

Nornir derives the metrics in Table 8.1 in the following way:
Throughput is computed as
\[ R = \frac{\beta - \alpha}{K_{\text{begin}}(\beta) - K_{\text{begin}}(\alpha)} \]
i.e., the number of iterations performed by the application between two successive monitoring requests, divided by the time elapsed between such requests.

Latency The latency monitored for the \( i \)-th iteration is
\[ L(i) = K_{\text{end}}(i) - K_{\text{begin}}(i) \]
To get the average latency per iteration, we need to sum all the latencies computed between two requests and to divide it by the number of performed iterations, i.e.
\[ L = \frac{\sum_{i=\alpha}^{i=\beta} K_{\text{end}}(i) - K_{\text{begin}}(i)}{\beta - \alpha} \]

Utilization Factor We recall that this metric is only meaningful for streaming applications. The time between the start of processing of two successive elements is \( K_{\text{begin}}(i+1) - K_{\text{begin}}(i) \). This includes both the time to process element \( i \) (i.e., the latency) and the time spent waiting for element \( i + 1 \) to appear on the input stream\(^2\). The ratio between the latency and the time elapsed between the start of the processing of two successive elements is
\[ \rho(i) = \frac{K_{\text{end}}(i) - K_{\text{begin}}(i)}{K_{\text{begin}}(i+1) - K_{\text{begin}}(i)} \]
and corresponds to the time spent doing useful work when processing element \( i \). To get the average utilization, is sufficient to sum these quantities over the elements received between two successive monitoring requests and to divide them for the number of elements, i.e.
\[ \rho = \frac{\sum_{i=\alpha}^{i=\beta} \rho(i)}{\beta - \alpha} \]
In batch applications, since the data is already available, we have \( K_{\text{begin}}(i+1) - K_{\text{end}}(i) = 0 \), i.e. we never wait for arrival of new data. Accordingly, \( K_{\text{begin}}(i+1) - K_{\text{begin}}(i) = K_{\text{end}}(i) - K_{\text{begin}}(i) \), which leads to \( \rho = 1 \), as we would expect in batch applications.

---
\(^2\)Actually it also includes the time required to perform the actual reading of the element once it appears on the stream. To be more accurate, we should remove this quantity from the computation of the utilization factor. However, this would require application-specific modifications, leading to a more intrusive approach. We prefer to have a slightly approximated factor to keep this solution as less intrusive as possible.
Eventually, in line 9, the connection with the Nornir manager is closed. The advantage of this approach is that despite requiring the insertion of only 4 instrumentation calls in the already existing application, it provides Nornir with all the needed information. Differently from the Black-Box case, it is now possible for the user to express requirements on all the metrics presented in Table 8.1. To use begin() and end() in multiple threads at the same time, it is sufficient to specify the identifier of the calling thread. In Code 8.4 we show how they can be used to instrument a parallel OpenMP application.

```
Instrumenter r("parameters.xml");
StreamElement* s;
while(s = receive()){
    #pragma omp task
    int threadId = omp_get_thread_num();
    r.begin(threadId);
    process(s);
    r.end(threadId);
}
```

Source Code 8.4: Example of OpenMP application instrumentation.

Moreover, besides predefined performance metrics, it is also possible to store custom values (e.g., application specific metrics). Such values will be provided to the self-adaptive algorithm, which can use them together with standard performance metrics to perform its decisions (Section 9.3). In Code 8.5 we show how to store two application specific metrics (frameSize and imageQuality). Each custom metric must be associated to an identifier (0 and 1 in this case).

```
Instrumenter r("parameters.xml");
StreamElement* s;
while(s = receive()){
    r.begin();
    process(s);
    r.storeCustomValue(0, frameSize);
    r.storeCustomValue(1, imageQuality);
    r.end();
}
```

Source Code 8.5: Example of custom values storage during instrumentation.

The instrumentation process has been designed to be as lightweight as possible. Despite being a lightweight computation, for applications characterized
by a low latency per iteration, the insertion of these two instrumentation calls may have a significant impact on application performance. For example, on the blackscholes application this could lead to a performance degradation up to 25% on an Intel Xeon machine with a clock frequency of 2.4GHz. Indeed, since at each iteration blackscholes performs a straightforward computation, the overhead introduced by the instrumentation calls have a significant relative impact on performance.

A possible way to mitigate this effect would be skipping some calls, for example by calling begin() and end() every $m$ iterations instead of calling them at each iteration. In [66] this task is delegated to the application programmer, which must select an appropriate value for $m$. This is a critical choice since $m$ is application- and input-specific. If $m$ is too low, it will not reduce the overhead while if $m$ is too high samples would be sent to the manager rarely, impairing the reactivity of the manager. Moreover, letting the programmer selecting this parameter would contradict the goal of this thesis, i.e. being able to automatically control applications without any information about previous application runs and without manually tuning of parameters.

To solve this issue, we introduced an adaptive sampling mechanism. The idea is that we can find out the time spent in the execution of the begin() and end() calls by executing a microbenchmark when the library is compiled and installed on the system. Let’s call this latency $c$. Let us pick the interval $K_{\text{end}}(\beta) - K_{\text{begin}}(\alpha)$ between the start of the iteration with index $\alpha$ and the end of the iteration with index $\beta$. Since the begin() and end() calls are executed at each iteration, the total time spent in these calls will be $\beta - \alpha \cdot c$. If we divide this quantity by the time interval and we multiply it by 100 we get the estimated performance degradation introduced by the instrumentation, i.e.

$$\frac{(\beta - \alpha) \cdot c}{K_{\text{end}}(\beta) - K_{\text{begin}}(\alpha)} \cdot 100$$

Then, we need to find how many times at most we want to execute the instrumentation calls over the interval $K_{\text{end}}(\beta) - K_{\text{begin}}(\alpha)$ to keep this overhead below 1%. This means that we want to find a value $n$ such that:

$$\frac{n \cdot c}{K_{\text{end}}(\beta) - K_{\text{begin}}(\alpha)} \cdot 100 < 1$$

i.e.

$$n < \frac{K_{\text{end}}(\beta) - K_{\text{begin}}(\alpha)}{c \cdot 100}$$

This implies that, to have an overhead lower than 1%, we should not execute the instrumentation calls more than $\frac{K_{\text{end}}(\beta) - K_{\text{begin}}(\alpha)}{c \cdot 100}$ times over the interval $K_{\text{end}}(\beta) - K_{\text{begin}}(\alpha)$. This can be easily enforced by skipping instrumentation of iterations $i$ such that $(i/n) \neq 0$. This process is completely transparent to the programmer, which just needs to instrument his code as before. The begin() and end() calls are still executed but they will not perform (almost) any computation unless $(i/n) = 0$. 
Since we are performing sampling, we may now collect performance data on samples which are not representative of the application behavior. For example, we may collect latency measurement for an iteration which is particularly faster (or slower) than the average, thus making wrong assumptions on the application behavior. We can detect this situation by comparing the actual length of the control step with the estimated length, computed as:

$$\frac{L}{\rho} \cdot (\beta - \alpha)$$

It can be proved that these two quantities match when sampling is not applied\(^3\). If the estimation is more than 5% different from the actual length, we mark latency and utilization factor data as inconsistent. Throughput is not affected by sampling since it is merely computed as the number of iterations executed divided by the actual length of the control step.

To summarize, in this section we described how to instrument iterative applications, so that they can export performance monitoring data to Nornir. Since in some isolated cases instrumentation may introduce a significant overhead we designed an adaptive sampling mechanism, which automatically selects the appropriate sampling length so to have a controlled overhead below 1%. However, since sampling may introduce inaccuracies in the measurement of latency and utilization factor, it should not be used when requirements on these two metrics need to be enforced by Nornir. Despite being an integrating part of Nornir, the instrumentation part has been implemented as a separate library, which we called Riff\(^4\). We made this choice since Riff could also be useful as a standalone library to be used for performance monitoring of applications. We did not rely on existing monitoring tools [66, 105] since they don’t provide the possibility to monitor utilization factor, to communicate custom monitoring values or to select the appropriate sampling rate automatically. Moreover, the instrumented applications we used in Chapter 4 to validate our algorithms have been released as open-source and integrated into the P\(^3\)ARSEC framework. This is an important step since if a strategy designer uses Nornir to implement a new self-adaptive strategy, he can quickly and easily validate it on a comprehensive set of applications already interfaced to Nornir.

### 8.3 Interaction with Supported Runtimes

In some cases, having runtime-specific information and being able to access runtime-specific actuators may help the planning algorithm in finding better configurations. For example, in Section 5 we showed how, by accessing framework-specific actuators, it has been possible to dynamically change and optimize the concurrency control algorithm used by the threads to access some shared data structures. At the time being, the only supported runtime is FastFlow [217],

\(^3\)It is not done here for brevity’s sake.

\(^4\)https://github.com/DanieleDeSensi/riff
which we already described in Section 6.3. We decided to focus on a framework providing parallel patterns since, as we saw in Chapter 6, this will simplify the exploitation of some reconfiguration knobs such as concurrency throttling, potentially leading to better performance/power tradeoffs. Nornir currently support only the FastFlow farm pattern. We briefly recall that an application structured as a farm is an embarrassingly parallel application, where multiple elements are processed in parallel by concurrent entities (usually threads) called workers. In the FastFlow case, elements are scheduled to the workers by an active entity called emitter and the results may optionally be gathered by another thread called collector.

To show how the manager can be attached to an already existing FastFlow farm application, let us consider the dummy application shown in Code 8.6.

```cpp
struct Emitter: public ff::ff_node{
    void* svc() {
        for(uint i = 0; i < N; i++) {
            ff_send_out((void*) new int(i));
        }
        return NULL;
    }
};

struct Worker: public ff::ff_node{
    void* svc(void* t) {
        int* i = (int*) t;
        cout << "Square of " << *i << " is " << pow(*i, 2) << endl;
        delete i;
        return GO_ON;
    }
};

int main(int argc, char** argv){
    ff::ff_farm<> f;
    vector<ff::ff_node*> w;
    for(uint i = 0; i < 5; i++) {
        w.push_back(new Worker());
    }
    f.add_emitter(new Emitter());
    f.add_workers(w);
    f.run_and_wait_end();
    return 0;
}
```

Source Code 8.6: Dummy application implemented by using FastFlow.
This application simply computes, in parallel, the square of the numbers between 0 and \( N \). In FastFlow, we first need to define the emitter (lines 1-8), which send the element to be processed to the workers by using the \texttt{ff\_send\_out} function. Eventually, the emitter returns a \texttt{NULL} pointer to indicate that no more elements will be produced. Workers are defined in lines 10-17, with the \texttt{svc} function containing the code to be executed for each element received from the emitter. In this case, for each received element \( t \), a generic worker prints the element and its square\(^5\). Workers return a special value (line 15) to indicate that the computation is not yet finished. After the definition of emitter and workers, in the main function the programmer defines the farm (line 20), adds 5 workers and an emitter to it (lines 21-26) and runs the parallel farm computation (line 27).

To attach this application to the \texttt{Nornir} manager, the programmer should execute the following steps:

1. Modify lines 1 and 10. Instead of extending the \texttt{ff\_node} class, emitter and workers should now extend the \texttt{nornir::AdaptiveNode} class.

2. Line 27 should be replaced with the following code:

```c++
Parameters p("parameters.xml"); // Load Nornir parameters.
ManagerFastFlow<*> m(&f, c); // f = FastFlow Application.
m.start(); // Start application.
m.join(); // Wait for application end.
```

Source Code 8.7: Embedding of \texttt{Nornir manager} inside an existing \texttt{FastFlow} application.

The manager will be executed in a separate thread, interacting with the FastFlow runtime to collect performance measurements and to apply concurrency throttling (i.e., to dynamically change the number of threads) and concurrency control (Chapter 5) if this has been required by the user. Note that changes in the number of threads occur on the fly while the application is running, with no need to stop or restart the application. As we shown in Section 6.1.1 this is a delicate operation, since both the computation and the internal state of the application must be kept consistent despite modifications in the number of running threads. We tackle this problem by allowing the programmer to specify in both emitter and workers (and also the collector, if present) the member function shown in Code 8.8.

In this function, the programmer should define the code to be executed to rearrange the internal state, after the number of threads have been changed\(^6\).

\(^5\)Note that explicit casts are needed to and from \texttt{void*}. FastFlow also allows to specify data types explicitly, removing the need of using explicit casts. This interface, however, is still not supported by \texttt{Nornir} at the moment.
8.3. INTERACTION WITH SUPPORTED RUNTIMES

void manageThrottling(uint oldNumThreads, uint newNumThreads)

Source Code 8.8: Member function to be implemented to manage concurrency throttling.

from oldNumThreads to newNumThreads. To ensure that this does not interfere with the application execution, possibly creating inconsistencies, the manager acts in the following way:

1. When the self-adaptive algorithm decides that the number of active threads needs to be modified, we freeze the entire application by using appropriate mechanisms provided by FastFlow. Freezing the application implies suspending the threads on a condition variable, forcing them to do not execute the code specified in the svc functions. Threads will be suspended as soon they exit the svc function or before returning from the ff_send_out function. This has the effect that threads are suspended between the processing of two successive elements.

2. The manageThrottling function are called on each component of the application. In this way, all the operations can be performed when the application is not executing anything, avoiding inconsistencies.

3. A number of threads equal to the one required by the self-adaptive algorithm are activated. The remaining ones will be still suspended on the condition variable.

In this thesis, we adopt a very conservative approach to concurrency throttling, since we do not assume any other additional property about the application and its state. We are currently working [33] on identifying common patterns in the structure of the internal state of applications. This would allow us to exploit such information to optimize the dynamic change in the number of threads, avoiding suspension in some specific situations.

In this section, we described how existing FastFlow applications could be interfaced to Nornir. Despite being a less general solution than instrumentation (since it is targeted towards specific programming frameworks) it would allow the manager to access more actuators. NORNIR can be easily extended to support other parallel programming frameworks, by customizing the Monitor and Execute stages of the MAPE loop, as we will show in Section 9.1 and Section 9.2. Lastly, we would like to recall that the study we did in Chapter 4 to evaluate our algorithm, has been performed on applications which were implemented with Pthreads and OpenMP, by relying on the instrumentation techniques we described in Section 8.2 and without any interaction with the runtime support. Accordingly, interacting with the runtime or, as we will describe in Section 8.4, using the runtime provided by NORNIR, is
something which could improve the quality of the solutions found by a generic self-adaptive algorithm but which is not strictly necessary.

8.4 Nornir Programming Framework

Lastly, if the application has not been implemented yet, the programmer may consider implementing it by using the parallel programming interface provided by Nornir. We decided to provide a custom programming interface to enable, in the future, access to additional reconfiguration knobs.

We provide a programming interface for both the threads-based programming model (Section 8.4.1) and the tasks-based, dataflow programming model (Section 8.4.2). In the first case, it will be possible to implement applications by using parallel patterns while in the second case it will also be possible to implement unstructured applications, expressed as generic graphs of concurrent activities.

8.4.1 Threads-Based, Parallel Patterns Interface

In the following we describe the interface we developed for building threads-based parallel patterns applications. Currently, our parallel pattern interface provides two patterns: farm and map. As we shown in Section 7.1, these two patterns are commonly used in many existing applications.

8.4.1.1 Farm

The farm interface is a wrapper over the FastFlow interface and exploits the FastFlow runtime system. With respect to the FastFlow interface, it is characterized by a higher abstraction level and by a better integration with the Nornir manager.

Code 8.9 shows an example of a streaming application that receives data from the network, analyzes them and send them back on the network. First, the programmer needs to specify the computation to be performed by the emitter (lines 1-10), which is in charge of receiving data from the network and dispatching them to the workers. The type of produced data needs to be specified (lines 1 and 3). Then, the code that will be executed by the worker (lines 12-17) and by the collector (lines 19-24) is defined and the farm is created (line 28), by providing the Nornir configuration file containing, among others, the user’s requirements. Eventually, the application is started (line 29), by specifying how many workers should be activated. By default, data will be scheduled with a round-robin policy. However, it is possible to use an on-demand scheduling policy or to implement custom policies, by replacing the send call with a sendTo call, specifying the identifier of the target worker. Moreover, due to the parallel processing of data, by default data could be received by the collector in any random order. Some applications, however, require the data to be received and processed by the gatherer in the same order they were received by the scheduler. This can be enforced by calling the preserveOrder function on
8.4. NORNIR PROGRAMMING FRAMEWORK

```cpp
class Emitter: public nornir::Scheduler<Packet>{
public:
    Packet* schedule(){
        while(!stop){
            Packet* p = readFromNetwork();
            send(p);
        }
        return NULL;
    }
};

class Worker: public nornir::Worker<Packet, Packet>{
public:
    Packet* compute(Packet* p){
        return process(p);
    }
};

class Collector: public nornir::Gatherer<Packet>{
public:
    void gather(Packet* p){
        writeToNetwork(p);
    }
};

int main(int argc, char * argv[]){
    unsigned int numWorkers = 10;
    nornir::Farm<Packet, Packet> farm("parameters.xml");
    farm.start<Emitter, Worker, Collector>(numWorkers);
    farm.wait();
    return 0;
}
```

Source Code 8.9: Example of parallel application implemented by using the `farm` provided by the Nornir programming framework.

The rearrangement of the internal state following a dynamic change in the number of threads must be manually managed by the programmer, similarly to what we showed in Section 8.3.

Alternatively, instead of setting up the entire farm, it is possible to use it as a software accelerator, i.e. to offload parts of a sequential computation to a parallel component, similarly to what happens when offloading computation to a hardware accelerator. In that case, the input will be provided by the main control flow and the output will be gathered by the main control flow. This
concept has been borrowed from the FastFlow framework and integrated inside our interface.

```c
1. class Processor: public normir::Worker<Packet>{
2. public:
3. Packet* compute(Packet* p){return process(p);}
4. };
5. int main(int argc, char * argv[]){
6. .normir::Parameters p("parameters.xml");
7.  normir::FarmAccelerator<Packet, Packet, Packet, Packet> acc(6p);
8.  acc.start<Processor>(NUM_THREADS);
9.  for(size_t i = 0; i < 100; i++){
10.    Packet* p = receive();
11.    if(p->srcAddress == TARGET_IP_ADDRESS){
12.      acc.offload(p);
13.    }
14.  } acc.shutdown();
15.  Packet* r;
16.  while(r = acc.getResult()){
17.    send(r);
18.  }
19.  return 0;
20. }
```

![Diagram](image.png)

**Figure 8.4: Example of execution with software accelerator.**

We clarify this concept in Figure 8.4, where we show an application that reads 100 packets from the network and processes those sent from a given IP address in parallel. In lines 1-4 we express the function which the workers will compute, in parallel, on each packet. In the main function we first load the Nornir parameters (line 7) and we create the accelerator (line 8). We need to express 4 types in the template instantiation:

1. One type for the data flowing from the application to the accelerator (Packet type).
2. One type for the data flowing from the emitter to the workers. In this case, there is no explicit emitter, so the type is equal to the previous one.
3. One type for the data flowing from the workers to the collector. In this case, the workers produce data of type Packet.
4. One type for the data flowing from the collector back to the application. In this case, there is no explicit collector, so the type is equal to the previous one.

The accelerator is started (line 10) and some packets are offloaded to it (lines 11-16). Eventually, we explicitly indicate that no more elements will be offloaded to the accelerator (line 17). We then collect the packets processed by the workers and we send them back over the network (lines 19-22). Eventually, we wait for the termination of the accelerator (line 23). If needed, the collection...
of the computed results can be interleaved with offloading of new data to the accelerator.

All the considerations about concurrency throttling, ordering of data and scheduling also holds for accelerator-based computations.

### 8.4.1.2 Map

We briefly recall that in a computation structured as a *map* the same function is applied to each item in a collection (e.g., an array). Computation over the different items can be performed in any order and in parallel. Conceptually, the main difference with respect to the *farm* case is that, while in the *farm* different elements are processed in parallel, but each element is processed sequentially, in the *map* case each element (which must be a collection) is split among the threads and processed in parallel. Moreover, while the *farm* works on a (potentially infinite) stream of elements, the *map* works on a collection with a fixed and known size. A more formal definition of *farm* and *map* was provided in Chapter 6.

In our case, we implemented the map pattern as a *parallel loop*, since the two models almost match one-to-one [159]. We describe the interface through the example in Code 8.10, which represents the computational part of the blackscholes application, which we require to terminate in less than 20 seconds.

```c++
std::vector<float> prices(numOptions);
...
// Setup all the vectors sptprice, strike, etc...

nornir::parallel_for_execution_time(0, numOptions, numThreads, 20,
[](const long i) {
    /* Calling main function to calculate option*
    * value based on Black & Scholes's equation. */
    prices[i] = BlkSchlsEqEuroNoDiv(sptprice[i], strike[i],
                                  rate[i], volatility[i],
                                  otime[i], otype[i], 0);
});
```

Source Code 8.10: Example of parallel application implemented by using the parallel loop provided by the Nornir programming framework.

After the setup of the input and output data (lines 1-2), the parallel loop is executed (lines 3-11). Its first two parameters are the start and end indexes of the loop, respectively. Then, the number of threads is specified, as well as a number (20) indicating the maximum execution time in seconds that Nornir needs to enforce for this loop. Eventually, by using C++11 lambda functions, the function to be executed at each loop iteration is specified. Additional parameters can be used to define the loop increment or other types of require-
ments such as energy, power consumption, etc.... Moreover, an additional lambda function can be provided to specify the actions to be executed before dynamically modifying the number of active threads.

The implementation uses the accelerator described in the previous section, with the emitter scheduling loop sub-intervals to workers, which will execute in parallel the lambda function on different ranges of the loop. The size of the iteration ranges to be sent to the workers can be specified manually by the programmer. This is a critical choice since long ranges can potentially cause load imbalance. On the other hand, short ranges can increase the overhead of the computation, since the time required to compute the function on such small ranges could be comparable or lower than the time needed to schedule them to the workers. As a future work, we are planning to let Nornir automatically find the optimal size of the ranges without any programmer intervention.

8.4.2 Tasks-Based Interface

The interface for tasks-based applications is based on a dataflow programming model. In Nornir, we used the Macro Data Flow (MDF) parallel programming model [221], that allows the user to specify parallel computations by expressing them as direct acyclic graphs (DAG). In such graphs, each node represents a sequential code fragment, while the edges represent the flow of the data computed inside the graph. The nodes are also called “Macro Data Flow instructions” (MDFi, also known as operations in other similar programming models), with the Macro term underlining the fact that each instruction actually represents a consistent part of the computation. Such structure is depicted in Figure 8.5, where A, B, C, D and E represent sequential code fragments, the arrows represent the data dependencies between such code blocks, and i* and o* represent the inputs and the outputs of each instruction.

Each instruction may receive/send data from/to one or more instructions, except for the first and last instruction of the graph. Indeed, the first instruction can only have one input, corresponding to the input data, while the last instruction has only one output, i.e. the result of the computation. By adding this constraint, we provide the possibility to seamlessly compose the graphs between each other, i.e. to use a graph in place of an instruction. This also allows the programmer to build complex graphs incrementally.

When an instruction is executed, it produces one or more results, called tokens. These tokens will be used as inputs for the instructions that depend on the current one. For example, in Figure 8.5, when instruction A terminates its execution, two output tokens will be produced, one used as input for instruction B and one used as input for instruction C. An instruction become fireable (i.e., it is ready to be executed), when it receives all input tokens. For example, instruction E can be executed only after it has received the results from instructions C and D.

The execution progress is orchestrated by an instruction scheduler, that works as follows:
1. A fireable instruction is located in the graph and sent to one of the interpreters in a pool of interpreters in other similar programming models. The interpreters can execute different instructions in parallel, thus allowing to exploit the parallel execution between instructions that do not have data dependencies. Each of the interpreters in the pool is capable of executing any fireable instruction. This is possible because the instruction code is stored inside the instruction itself.

2. The results of instructions execution are received from the interpreters and used by the scheduler as input tokens for the corresponding destination instructions. As soon as one of these instructions becomes fireable, it is sent to the interpreters.

These two steps are iterated until there are no more fireable instructions available (i.e., up to program termination).

In streaming applications for each received element, a new instance of the graph is created and stored into a graph pool, as shown in Figure 8.6. When the last instruction of a graph is executed, the final result is sent over an output stream and the corresponding graph instance is destroyed. Accordingly, there is a 1-to-1 association between stream elements and graphs instances.

To implement the dataflow interpreter we took inspiration from the implementation of the Muskel [222] framework. We relied on the farm pattern provided by Nornir, with the emitter running the scheduler of the instructions and the workers running the interpreters, as depicted in Figure 8.7.

When one of the interpreters in the pool finishes the execution of an instruction, it sends the computed results back to the scheduler. The results will then be read by the scheduler and used to update the corresponding dataflow
instructions stored in the graph pool. If no elements are present on the input stream, the scheduler must not wait for new items to arrive. Indeed, meanwhile new results could arrive on the feedback channels from the interpreters and those results need to be managed. Similarly, the scheduler must not wait on the feedback queues if they are empty. To implement a non-deterministic read from these channels, the interactions with both channels has been implemented through non-blocking mechanisms. Accordingly, if no elements are present on the input stream the receive call will immediately return, and the scheduler will check the feedback channels (and vice versa).

Moreover, we need to find a tradeoff in the priority given to the different scheduler’s input channels. Indeed, if the priority is always given to the input stream, then the computations of the graphs already present in the system will never advance in their execution. On the other hand, if we read from the input stream only when there are no more instructions on the queues from the interpreters, then we are not entirely exploiting the available parallelism. In our implementation, we face this problem by putting a limit on the maximum number of graphs that can be present in the graph pool at any moment. The higher the limit, the more we shift towards a solution where we always prefer to read from the input stream.

Algorithm 6 shows the full scheduler pseudocode.

Having a centralized scheduler is a common choice in many dataflow environments [223, 142]. In principle the centralization of the scheduling could be a bottleneck, thus limiting the scalability of the system. However, the scheduler implementation we used has been proven efficient and feasible in similar contexts for state-of-the-art multicore architectures up to 32/64 cores using a variety of different applications [224]. This is also confirmed in Section 8.4.2.4 where we show that, even for applications performing fine-grained computation, our dataflow framework achieves performance comparable to those obtained by using other frameworks. Moreover, having an active entity performing the scheduling enables definition of custom scheduling policies, improving the flexibility of the approach.

As described in Section 6.1.5, one of the advantages of task based runtimes
Figure 8.7: The runtime architecture. Node 'S' is the scheduler. Nodes 'I' are the interpreters.

**ALGORITHM 6: Macro Data Flow Scheduler Algorithm**

```plaintext
Function Scheduler()
numGraphs ← 0;
while true do
    if numGraphs < maxNumGraphs then
        if t ← receiveFromStream() then
            numGraphs ← numGraphs + 1;
            g ← pool.createGraph(t);
            sendToInterpreters(g.getFirst());
        end
    end
    if r ← receiveResult() then
        if isOutputResult(r) then
            numGraphs ← numGraphs - 1;
            sendToOutputStream(r);
        else
            instructions ← pool.updateTokens(r);
            for i in instructions do
                if isFireable(i) then sendToInterpreters(i);
            end
        end
    end
end
```
is that the user does not need to explicitly manage state rearrangement after a dynamic change in the number of threads (i.e., concurrency throttling), since the execution model is decoupled from the programming model. This means that we, as designers of the dataflow runtime, need to ensure that the runtime keeps working correctly even after the number of threads is dynamically modified. However, concurrency throttling will be utterly transparent to the application user.

8.4.2.1 Scheduling Issues

In general, the scheduler sends fireable instructions to the interpreters (workers) by using an on-demand strategy (i.e., the instruction is sent to an interpreter if that interpreter has no other instructions to process). However, in some cases, different scheduling strategies may be more appropriate. For example, consider the case depicted in Figure 8.8 where we have 3 interpreters, a graph composed of two pipelined instructions (A and B). An element arrives on the input stream, and the scheduler generates a new graph instance. Since the first instruction (A₁) becomes fireable, it will send it to the first available interpreter (I₁). Meanwhile, a new element arrives on the input stream, the corresponding graph is generated and its first instruction A₂ is sent to interpreter I₂. When A₂ terminates its execution, B₂ became fireable and it is sent to I₃. Note that, since execution of A₁ takes too long to terminate, B₂ is executed before B₁.

Albeit this can be acceptable in some applications, there are situations where the application programmer needs instructions belonging to different stream elements to be processed in the same order they are received (we will show an example of such application in Section 8.4.2.4). Note that this cannot be enforced by data dependencies alone since they ensure the correctness of the execution for a specific input but they do not constrain instructions associated with different stream elements. To solve this problem, we can schedule the instructions with the same identifier to the same interpreter. Since the enqueued instructions are processed by the interpreter in FIFO order, we guarantee that
instructions with the same identifier and associated to different stream elements are processed in the same order they are received. Moreover, since the feedback channels are FIFO as well, even the next instruction in the graph will be processed in the correct order. However, this type of scheduling may lead to workload imbalance between the interpreters, since some of them may receive more (or more costly) instructions. To clarify this, let’s consider the case where we have 3 interpreters, a graph composed of 7 instructions with the same average latency and a scheduling function that assigns the instructions with identifier $x$ to the interpreter $I_{x \% 7}$. This scheduling function assigns all the instructions with the same identifier to the same interpreter. However, for each element received from the input stream, the interpreter $I_1$ will process 3 instructions while interpreters $I_2$ and $I_3$ will process 2 instructions each. This workload imbalance could be present even if different scheduling functions are used and also if the instructions have a different latency. Indeed, since a given instruction can only be executed by a specific worker, an optimal scheduling could not always be found, leading to inefficiencies. Note that the FIFO constraint on the queues between the scheduler and the interpreters is only needed in this case and it could be relaxed in scenarios where the input stream elements do not need to be processed in the same order they are received.

8.4.2.2 Order of Produced Results

Another problem concerns the order in which the results are sent on the output stream. When instructions associated with different stream elements can be executed in any order, the last instruction of the graphs (which will produce the output stream elements), could be executed in any order as well. Therefore, elements could appear in the output stream in the wrong order. Since an increasing identifier is assigned to the graphs when they are created, we can keep track of the last output sent over the stream. When a new result to be sent appears, if the identifier of his graph is the next to be sent on the output stream, then it is sent. Otherwise, the result is stored in a priority queue, where the priority is the identifier of the result (lower identifiers represent higher priority). The head of the queue is periodically checked and sent on the output stream only if it is the next element to be sent.

8.4.2.3 Interface

In this Section, we describe the steps to be performed by the programmer to transform an existing sequential application into a parallel dataflow and to specify performance constraints, taking as an example the Strassen algorithm for multiplying two matrices.

First, the programmer analyzes the sequential code (Code 8.9), extracting the dataflow dependencies in the algorithm (Figure 8.10). After that, he can implement the corresponding MDF graph, which instructions will be executed in parallel by the runtime (Code 8.11). First of all, the input stream is de-
Matrix A, B, C,
P1, ..., P7,
A11, ..., A22,
B11, ..., B22,
C11, ..., C22;

while(!in.end()){
    // Receive two matrices from
    // the stream.
in.receive(A, B);
    A.split(A11, A12, A21, A22);
    B.split(B11, B12, B21, B22);

    // Compute Strassen algorithm
    P1 = (A11+A22)*(B11+B22);
P2 = (A21+A22)*B11;
P3 = A11*(B12-B22);
P4 = A22*(B21-B11);
P5 = (A11+A12)*B22;
P6 = (A21-A11)*(B11+B12);
P7 = (A12-A22)*(B21+B22);
C11 = P1+P4-P5+P7; C12 = P3+P5;
C21 = P2+P4; C22 = P1-P2+P3+P6;
    
    // Send result matrix to
    // the stream.
    C = compose(C11, C12, C21, C22);
    out.send(C);
}

using nornir::dataflow::InputStream;
using nornir::dataflow::OutputStream;
using nornir::dataflow::Computable;
using nornir::dataflow::Data;
using nornir::dataflow::Mdfg;
using nornir::Parameters;

Computable *INInstr, ..., *C21Instr, ...;

class InputStreamApp: public InputStream{
public:
    void* next() {...}
    bool hasNext() {...}
};

class OutputStreamApp: public OutputStream{
public:
    void put(void* a) {...}
};

... /** Code for other instructions. **/

class C21Code: public Computable{
public:
    void compute(Data* d){
        Matrix *P2, *P4;
P2 = (Matrix*) d->receive(P2Instr);
P4 = (Matrix*) d->receive(P4Instr);
d->send(OUTInstr, (*P2)+(*P4));
    }
};

int main(){
    // Create all the instructions.
    ...;
    C21Instr = new C21Code(...);
    // Link all the instructions.
    Mdfg g;
g.link(P2, C21); g.link(P4, C21);
    ...
    // Create streams.
    InputStreamApp is;
    OutputStreamApp os;
    Parameters p("parameters.xml");
    Dataflow d(&p, &is, &os);
    d.start();
d.wait();
    return 0;
}
8.4. NORNIR PROGRAMMING FRAMEWORK

fined, by extending the InputStream class and by implementing two member functions (lines 10-14). The function next() must return the stream element if available, or NULL if no elements are currently present on the stream. The function hasNext() returns true if there are still elements to receive from the stream and false when there are no more elements on the stream. Then, the output stream should be defined (lines 16-19), by implementing the function put() to manage the computed results and to send them on the actual stream. To define the instructions, the application programmer needs to wrap each business logic fragment (i.e., the actual code performing the computation) into a class that extends the Computable class and implements the compute() function (lines 25-29). In the compute() function the application programmer can use the data received from the linked instructions (lines 27-28) and send the results to the output linked instructions (line 29). After that, the graph is created by instantiating the instructions (lines 35-36) and by linking them together (lines 40-41). Then, Nornir configuration file is loaded (line 47) and the runtime is created (line 48) and started (line 49). As described in Chapter 8, the configuration file contains, among others, the user requirements on performance and power consumption. Moreover, for dataflow applications, it is possible to use the configuration file also to require the runtime to preserve the order of computed results. Eventually, wait call (line 50) will wait for the termination of the dataflow application, which will occur when false is returned by the hasNext() call (lines 13).

In addition to arbitrary graphs, it is also possible to specify parallel pattern based applications, which will be automatically translated into the corresponding Macro Data Flow graphs. Currently, we support pipeline, farm, map and reduce patterns [78].

Patterns (but also unstructured graphs) can be nested. This is a common requirement in many existing applications, as we shown in Section 7.1. Moreover, it is possible to mix the thread-based parallel patterns interface with the tasks-based dataflow one. For example, we could have a 3-stage pipeline defined with the thread-based API, where the middle stage uses the dataflow execution. In this case, the input stream will correspond to the output channel of the first pipeline stage, and the output stream will correspond to the input channel of the last pipeline stage. This is possible because we can guarantee the order of the stream elements flowing through the dataflow stage, as discussed in Section 8.4.2.2.

8.4.2.4 Evaluation of the Dataflow Runtime

To evaluate our dataflow approach, we use the self-adaptive algorithm we described in Section 3.1 to enforce some performance requirements on some applications we implemented by using the dataflow runtime. In this way, we also prove that the self-adaptive and power-aware algorithm we proposed work both on applications implemented by using a thread-based model (Chapter 4) and on applications implemented by using a tasks-based, dataflow model. All the experiments have been executed on the same workstation we used in Part II
to evaluate the self-adaptive algorithms, i.e. an Intel workstation with 2 Xeon E5-2695 @2.40GHz CPUs, each with 12 2-way hyperthreaded cores, running with Linux x86_64. We test our dataflow runtime on different applications and scenarios.

**ffProbe** The first application we used is *ffProbe* [225], a parallel implementation of a NetFlow [226] probe, i.e. an application responsible for network traffic monitoring. Network packets are aggregated in *flows*, created when the first packet of the flow has been received, and destroyed when some *expiration* conditions is verified. In *ffProbe*, the application is logically structured as a pipeline, where each stage manages a partition of the active flows. When a packet is received by the input stream, it is inserted into a stream task and sent to the first worker of the pipeline. If the packet belongs to a flow managed by the worker, the corresponding flow is updated. After that, the worker checks if some of the flows it manages are expired and, if this is the case, these expired flows are added to the task. Eventually, the task is forwarded to the next pipeline stage. By implementing *ffProbe* by using our dataflow interface, we obtained the same peak performance obtained by the original implementation.

For our experiments, we sent data to the application by using data rates of a real network⁶, covering a 24 hours span. To model future network scenarios, this rate has been linearly scaled up to increase the parallelism need. In this application, the packets belonging to the same flow are always processed by the same Macro Data Flow instruction and must be processed in the same order they are received. As we described in Section 8.4.2.1, this implies that the scheduling of the instructions to the interpreters is done according to their identifier. Since the instructions are characterized by a very similar execution time, the best scheduling strategy is to partition the graph into equal parts among the interpreters to keep the load perfectly balanced. However, this is only possible if the number of graph instructions is a multiple of the number of active interpreters. Moreover, since we instantiate a new graph for each element received from the input stream, this unbalance would accumulate as more elements are received, thus leading to the impossibility to process all the elements of the stream. In a static scenario, this could be solved by forcing the number of interpreters to be a divisor of the number of instructions in the graph, to have an perfectly balanced scheduling. However, this solution is not feasible in our scenario since we are dynamically changing the number of interpreters at runtime. To solve this problem, we forced the runtime to use only a number of interpreters which is a divisor of the number of instructions in the graph. Since we have a graph composed of 20 instructions, the runtime can activate 1, 2, 4, 5, 10 or 20 interpreters. In this application, the instructions have an average latency of 3 milliseconds. We set as requirement a *utilization* lower than the 90%.

In Figure 8.11 we show the throughput of the application and the amount

---

⁶[http://bit.ly/1RY7fEt - The used rate is the one collected on 08 Jun 2005](http://bit.ly/1RY7fEt)
of used resources\textsuperscript{7}. We plot the product between the number of used cores and their frequency to show that, as anticipated in Section 3.1, the performance is proportional to this quantity. The labels on the right y-axis report the number of cores and the clock frequency used by the runtime. We would like to point out that around 18 hours from the application start, the runtime starts to oscillate between 10 cores running at 2GHz and 20 cores running at 1.2GHz. This happens since the optimal solution falls in an intermediate value that cannot be used because we restricted the possible choices for the number of interpreters (i.e., cores) that can be used by the runtime. The final effect of the resources scaling is sketched in Figure 8.12, where we show both the throughput and the power consumption of the application. The power consumption ranges from 20 to 100 Watts according to the workload conditions.

**Streaming Blackscholes** This second application is a dataflow streaming implementation of the blackscholes application, which we described in Section 7.1. In this case, instead of loading the data from the disk, it receives a (potentially infinite) stream of data from the network. Differently from the

\textsuperscript{7} Arrival rate is not shown since the algorithm was always able to guarantee a throughput equal to the arrival rate.
CHAPTER 8. USING THE FRAMEWORK

<table>
<thead>
<tr>
<th>Dataflow</th>
<th>Pthreads</th>
<th>OpenMP</th>
<th>Intel TBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.87</td>
<td>0.83</td>
<td>0.95</td>
</tr>
</tbody>
</table>

Table 8.3: Maximum performance achieved by different streaming blackscholes implementations. Results are normalized between 1 (best) and 0.

ffProbe case, this application does not require the tasks to be processed according to any precise order. In this case, the instructions will be scheduled according to an on-demand policy, avoiding the unbalancing problems that characterize the ffProbe application. In our experiments we used the PARSEC native input and as input rates, those of a trading day of the NASDAQ market\(^8\). In this application, the instructions have an average latency of 0.3 milliseconds. Firstly, we evaluated the maximum performance achieved by our dataflow solution with those obtained by using pthread, OpenMP and Intel TBB implementations that are all distributed with PARSEC. In Table 8.3 we show the average normalized maximum performance (between 0 and 1, the higher, the better) obtained over 10 different executions of the experiment.

The dataflow implementation obtains a slightly better performance with respect to the other solutions. This is mainly because the dataflow runtime pins each interpreter on a specific core. On the contrary, pthread, OpenMP and Intel TBB implementations let the operating system manage the threads allocation.

Moreover, we execute the application by requiring even in this case a utilization lower than 90%. As we can see from Figure 8.13, the algorithm can enforce this requirement on the application. This leads to the effect depicted in Figure 8.14, i.e. the self-adaptive algorithm scales the number of resources allocated to the application (and consequently its power consumption) according to the arrival rate of the stock options to the application.

\(^8\)http://www.nyxdata.com - The used trading day is 30 Oct 2014
8.4. NORNIR PROGRAMMING FRAMEWORK

Figure 8.14: Throughput (Millions Opt. Per Second) and power consumption for blackscholes.

Figure 8.15: Resources allocated to Strassen application with a required execution time of 200 seconds.

**Strassen** The last application we used to validate our solution, consists in computing the Strassen matrix multiplication algorithm [227] over a stream of matrices. This algorithm divides both the input matrices in 4 parts and applies several sums, differences and multiplications to these submatrices to get the final result. The application can be expressed as a non-structured dataflow graph (Figure 8.10). In this application, the instructions have an average latency of 60 milliseconds.

In this experiment, we have a stream of 4000 matrices, and we required a maximum execution time of 200 seconds. Moreover, after 100 seconds from the beginning, an external application is started (doing parallel data compression with 24 threads).

As shown in Figure 8.15, around 20 seconds from the beginning, the algorithm forces the application to run in a configuration satisfying the user requirements. After 100 seconds from the start, due to the co-running application, the performance of Strassen application decreases and the self-adaptive algorithm increases the number of resources used by the application, allowing it to terminate its execution within the target.
8.4.2.5 Summary

Dataflow model has been around since the earliest days of computer science research activities [228] and many programming environments currently use dataflow concepts in the implementation of different parallel programming frameworks. For example, MuSkel [222], S-Net [199], StarPU [229] and STREAMIT [230] are programming frameworks using different flavours of the dataflow programming model. Moreover, dataflow is also present in widely used frameworks like OpenMP [231, 232], Microsoft TPL⁹ and Google’s TensorFlow¹⁰.

We implemented a new dataflow interface to be used in Nornir, instead of relying on an existing one since understanding the actual implementation of existing frameworks and modifying it is not an easy task. By implementing our framework, we can quickly tune it and add and prototype new features.

Of course, this is a prototype which we need to validate self-adaptive algorithms and to enable further research in the future, and it was not meant to become a competitor of mature and widely supported dataflow programming framework like TensorFlow or OpenMP. For this reason, at the moment there are some limitations with respect to other dataflow programming frameworks.

8.5 Summary

To summarize, in this chapter we described the different solutions provided by Nornir to interface new or existing application with the manager. In general, the higher is the effort the programmer is willing to put into this task, the higher will be the benefit regarding optimality of the configuration chosen by the manager.

The optimal solution would be to program the application with the provided programming API (either parallel pattern-based or dataflow). Despite the programming API is not particularly novel (since we mostly relied on existing concepts), we integrated all these different programming models in one single framework, allowing many different applications to exploit the self-adaptive strategies provided by Nornir. We also showed how a single powerful pattern (farm) enable the user to express various types of computations, including parallel loops and task-based applications.

If the application was already existing and it was implemented with one of the supported frameworks (at the moment only FastFlow), it is possible for Nornir to access framework-specific executors that would not be available otherwise (e.g., concurrency control or dynamic concurrency throttling), thus improving the quality of the selected configuration, as we shown in Chapter 5 and Section 6.1.

If the application is not implemented with one of the supported programming frameworks and if it is not possible to rewrite the application by using

---


¹⁰https://www.tensorflow.org/
one of these, the programmer can just insert few instrumentation calls inside the application, allowing Nornir to monitor it. This is the technique we used in Chapter 4 to control all the application of the PARSEC benchmark.

Eventually, if even the instrumentation is not feasible (e.g., because the programmer can’t or doesn’t want to change the application code and recompile it), Nornir can still manage the application, not requiring any programming effort. However, we can only monitor performance through system’s hardware counters, losing useful information about the application. Moreover, expressing performance constraints in this scenario could be not intuitive from the user perspective.

We provided a wide range of solutions to enable the use of self-adaptive algorithms in as many scenarios as possible. We would like to remark that legacy applications can be interfaced to Nornir by simply adding two instrumentation calls in the code, as we did on PARSEC applications to collect the results we showed in Chapter 4. The alternative solutions we considered (interacting with the runtime support or using the Nornir runtime) could be useful to access more executors and to improve the quality of the selected configuration but it is not a requirement of Nornir.
Chapter 9

CUSTOMIZING AND EXTENDING THE FRAMEWORK

In this chapter, we describe how self-adaptive strategies designers can add new algorithms to the framework, by exploiting the monitoring infrastructure and the actuators (knobs) already provided by Nornir.

Code 9.1 shows a simplified version of the main parts of Nornir implementation:

The meaning of this code snippet will become more evident after the end of this section. For the moment, we can focus on the MAPE loop (lines 9-24). In the remaining part of this Chapter, we describe how each step is designed and how they can be customized by the self-adaptive strategy designer.

First, in Section 9.1 we describe how the monitoring infrastructure can be customized. Then, in Section 9.2 we show how to implement new actuators inside Nornir. Eventually, in Section 9.3 we show how to build new reconfiguration strategies.

Parts of this chapter have been published in [35, 34].

9.1 Monitor

As we described in Chapter 8, the user can get the highest benefits from interfacing Nornir to an application implemented with the Nornir parallel programming interface or to an application implemented by using one of the supported frameworks. At the moment, this only includes FastFlow. To enable framework-specific monitoring for additional frameworks, the designer needs to define a new manager for the new runtime support, by defining a subclass of the Manager class and implementing the getSample function (Code 9.1, line 26). In this function the designer should implement the code to retrieve a new

\footnote{Actual implementation consists of approximately 18000 lines of code}
typedef enum{
    KNOB_VIRTUAL_CORES = 0,
    ...
    KNOB_NUM
}KnobType;

class Manager{
...
    void run(){
        while(isRunning()){
            sleep(_configuration.samplingInterval);
            // Monitor (Section 9.1)
            ApplicationSample s = getSample();
            storeSample(s);
            // Analyze & Plan (9.3)
            KnobsValues k = _selector->getNextKnobsValues();
            // Execute (Section 9.2)
            for(uint i = 0; i < KNOB_NUM; i++){
                _knobs[i]->changeValue(k[i]);
            }
        }
    }
    virtual ApplicationSample getSample() = 0;
};

class Knob{
...
    std::vector<double> _knobValues;
    virtual void changeValue(double v) = 0;
};

class Selector{
...
    virtual KnobsValues getNextKnobsValues() = 0;
}

Source Code 9.1: Simplified version of the main parts of Nornir implementation.
monitored sample from the runtime. This function will be called by the MAPE loop (line 14) and the sample will be stored (line 15), to be accessible from the Analyze and Plan phases. Beside storing the metrics we introduced in Table 8.1 and 8.2, it is possible for the designer to store additional metrics. As an example, let us consider the case where the designer wants to add the support for TBB to Nornir. In this case, he simply needs to implement in the getSample function the code to extract from TBB runtime performance metrics such as number of tasks processed per second, average latency of a task, etc...

On the other hand, system’s metrics (e.g., power consumption, temperature, etc...) can be monitored by using any existing monitoring tool. To simplify the process, we developed Mammut [34], an object-oriented C++ framework allowing a transparent and portable monitoring of system sensors as well as management of several system knobs. Mammut will be described in Section 9.2.1.

9.2 Execute

To implement a new executor, the designer must define a subclass of the Knob class (Code 9.1, lines 29-35). In the constructor, the _knobValues vector must be populated with the set of values that the knob can assume. When the planning phase terminates, the function changeValue will be invoked by the manager on all the available knobs (lines 21-23), by providing as parameter the value that the specific knob must assume according to what has been decided by the Plan phase. By implementing the function changeValue, the designer specifies the actual code to be executed to change the value of that knob. For example, if the designer wants to implement a knob to set the DRAM frequency, in the changeValue function he will insert the code to perform this action. The new Knob object must then be created and added to the _knobs array (used in line 22). Moreover, a new enumeration value must be assigned to this knob (lines 1-5). Currently, the following knobs are implemented:

Number of Cores  Turns off (or on) some cores. Threads will be allocated to cores through the Threads Mapping knob, while this knob only enforces the specified number of cores to be active.

Hyperthreading Level  On systems supporting Simultaneous MultiThreading (SMT, e.g. Intel’s Hyperthreading technology), this knob sets the number of hardware threads to use on each physical core.

Threads Mapping  Once the number of cores to use has been decided, this knob can be used to apply a specific allocation of threads on the physical cores. For example, to place them on a set of cores sharing some resources (e.g., last level caches) to minimize power consumption, or to

[^3]: http://danieledesensi.github.io/mammut/
place them on a set of cores with the minimum amount of shared re-
sources, wasting more power but improving performance. We exploited
this knob in our online learning algorithm (Section 3.2.2.4).

**Clock Frequency** Operates on the clock frequency (and voltage) of the cores,
allowing to trade a decreased performance for a lower power consump-
tion.

**Concurrency Control** For applications using FastFlow as runtime support,
this knob operates on the algorithm to be used when two threads access
their shared message queue, as we described in Chapter 5.

We showed in Section 6.1 that, when reducing the number of active cores,
by using Concurrency Throttling (dynamically changing the number of threads
used by the application) it is possible to achieve a higher performance for unit
of power with respect to Thread Packing (forcing more threads to run on the
same core). The **Number of Cores** knob uses Concurrency throttling or Thread
Packing, according to the type of interaction between the self-adaptive man-
ger and the application. If this interaction occurs by means of a black-box
approach (Section 8.1) or through instrumentation (Section 8.2) Thread Pack-
ing is applied, since Nornir does not have a direct access to the runtime. If
the application has been implemented by using the Nornir programming API
(Section 8.4) or if it is developed by using one of the frameworks supported by
Nornir (Section 8.3), Concurrency Throttling can be applied. Currently, we pro-
vide Concurrency Throttling only for FastFlow, but it can be applied to other
frameworks as well. To do that, is sufficient to implement a new knob, as
we already described, by specifying in the changeValue member function the
framework-specific code for adding/removing a thread at runtime.

Concerning the other knobs, they have been implemented by using Mam-
mut, which could also be used to implement new actuators. In the remaining
part of this Section, we describe how we designed Mammut and what it offers
to self-adaptive strategy designers.

### 9.2.1 Mammut Design

As outlined in Section 2.2.2.2, existing tools for tuning the hardware related
control mechanisms have some limitations:

1. They can usually monitor only the same computing node on which they
   are executed. However, a standard requirement [120, 121] in modern
   computing devices, like the ones that are becoming increasingly popular
   in the IoT and Fog computing systems [233], is the possibility to integrate
   remote monitoring functionality and dynamic management of resources
   capabilities.

2. They are often targeted towards one specific architectural aspect, without
   any possibility to interact and extract information obtained with other
tools, hampering their effectiveness. For example, to implement the *Threads Mapping* knob we could use the `taskset` Linux command, which maps application threads to a particular set of cores. However, other tools (e.g., `lstopo`) should be used to select the correct cores identifiers accordingly to the chosen mapping type. Implementing such interaction is time-consuming and requires converting the data representation between different tools.

3. They usually do not provide a sufficiently high abstraction level, forcing the designer of the self-adaptive algorithm to deal with low-level technical issues, instead of focusing on the algorithm itself.

To address such limitations, we designed *Mammut*, an object-oriented, open-source C++ framework, that provides a high-level interface for the management of hardware related mechanisms on local and remote Linux systems. It aims to solve the issues mentioned earlier, by simplifying application and software runtime system development. *Mammut* provides an easy-to-use API to correlate and integrate data coming from different architectural sources, hiding portability issues and providing a uniform interface to the user. By using *Mammut*, the programmer is relieved of the burden of dealing with the details of the specific platform and tools, since these aspects are handled transparently by the framework. Thanks to its modular design and its open-source nature, *Mammut* can be easily extended to add new features and mechanisms. It currently targets multi-core machines and has been successfully tested on different modern Intel, ARM and PowerPC architectures. *Mammut* is about 10 thousand lines of C++ code structured as a set of modules, each of them managing a given set of functionality. Currently, the following modules are available: Topology (Section 9.2.1.1), Energy (Section 9.2.1.2), CpuFreq (Section 9.2.1.3) and Task (Section 9.2.1.4).

Before starting using *Mammut*, the programmer must instantiate a *Mammut* object, which will be used to obtain handles to the different available modules via a `getInstance[ModuleName]` call. Indeed, since different implementations of the same module may exist (in principle one implementation for each supported architecture or operating system), the *Mammut* object will provide the most suitable implementation according to the underlying system. Each module can provide its functionality by a direct interaction with the OS, with the hardware, with other modules or by exploiting third-party libraries, as shown in Figure 9.1. This allows *Mammut* to expose a homogeneous interface to the application programmer independently from the underlying system.

We now briefly describe the structure of the modules and their main functionality. Due to their criticality, some of the following features (e.g., changing the clock frequency) can only be used by users with privileged rights.

---

3https://www.open-mpi.org/projects/hwloc/
9.2.1.1 Topology Module

This module allows the management of the physical topology of the underlying architecture. We hierarchically organized the hardware components: CPUs, Physical Cores (i.e. the basic computation unit of a CPU) and Virtual Cores. Virtual Cores represent the different hardware threads of a Physical Core. Indeed, when Simultaneous MultiThreading (SMT) is present, modern CPUs provide multiple abstract (virtual) cores (also known as hardware contexts) for each physical core present on the CPU. Each hardware component instance is represented by an object, which can be used to retrieve its sub-components, to turn it off or to get information about the specific version and capabilities of that component. Moreover, Virtual Cores allows to explicitly set C-States (Section 2.1.1.1). This is a commonly used feature to reduce the energy consumption of applications [151].

9.2.1.2 Energy Module

By using the energy module, it is possible to read the power consumption of the system or some of its components. It is possible to check which types of energy counters are available on the system and to retrieve one of them by querying the module. Each counter provides a reset() call to set the cumulative counter to 0 and a getJoules() call to retrieve the energy consumed since the last reset() call. Some counters may provide additional calls. For example, energy counters available on Intel SandyBridge, IvyBridge and Haswell provide specific functions to read energy consumption of each CPU, of the Physical Cores on the CPU, of the DRAM (Dynamic Random Access Memory).
controller, and of the integrated graphics card. Thus, by combining information coming from this module and from the Topology module, it is possible to monitor energy consumption of specific hardware components.

Currently, the following architectures are supported:

- Intel SandyBridge, IvyBridge and Haswell. For these architectures, it is possible to read energy consumption of each CPU, of the cores on the CPU, of the DRAM controller and of the integrated graphics card. On some of these systems, DRAM or integrated graphics counters may not be available.

- Any system connected to a SmartPower\textsuperscript{4} external power meter. In such case, it is possible to read the total energy consumption of the system.

- IBM Power8 architectures, by exploiting the Amester tool [234] and by abstracting its interface.

9.2.1.3 Cpufreq Module

With this module, the user can read and change the frequency and the governors of the CPU cores. The governors are algorithms used by the OS to manage the clock frequency of the CPUs. If the user needs to implement custom frequency scaling policies (as those we saw in Chapter 3), it is possible to use Mammut to change the clock frequency manually. In general, it is not possible to change the frequency of each core individually. For this reason, we provide the concept of Domain, i.e., a set of Virtual Cores that must run at the same frequency. Therefore, the user may read and change the governor, the frequency or the voltage of a Domain. Accordingly, by leveraging on the data provided by the Topology module, it is possible to know which cores will be influenced by such operations.

9.2.1.4 Task Module

This module allows managing processes and threads running on the system. For example, it is possible to move threads between Physical or Virtual cores, to change their priority or to read statistics about their execution.

9.2.1.5 Remote Management

As we anticipated, Mammut allows the monitoring of remote computing systems. To do so, a mammut-server should run on that system. From the client side, it is sufficient to specify the address and the port on which the server is listening to when the Mammut object is created. All the other calls and interactions do not require any modification. In this way, the user, by simply changing a single line of code, can seamlessly reuse the code written for local system management to manage a remote one. This is possible because

\textsuperscript{4}\url{http://odroid.com/dokuwiki/doku.php?id=en:odroidsmartpower}
the getInstance[ModuleName] call, in this case, will return an object with the same interface of the one used for local management but that will act as a client towards the remote server. The interaction with the server is performed via libprotobuf\(^5\) library. Note that this type of interaction can also be used to provide access to privileged features (e.g., changing clock frequency) to users with non-privileged rights. For example, we could start a mammut-server with privileged rights on a system and run a non-privileged client on the same system to access privileged features. For this reason, mammut-server can be executed with a limited set of modules. We are planning to include a more fine-grained capabilities control in the next versions of the library.

9.2.1.6 Illustrative Example

In the following code snippet, we provide a full running example, showing how it is possible to leverage on the information provided by the different modules to shut down unneeded CPUs (lines 19-23), move the application on a specific CPU (lines 25-2), change its governor and frequency (lines 29-33) and read its power consumption (lines 35-38).

To perform these same operations on a remote computing system, is sufficient to replace Mammut m in line 13 with Mammut m(new CommunicatorTcp(ipAddress, port)) where ipAddress is the address of the remote system and port is the port on which the mammut-server is listening.

9.2.1.7 Comparison with Existing Tools

To better understand the advantages of using Mammut, we implemented the same code snippet by using standard tools provided by the operating system. Such implementation is not shown here since it is composed of almost 700 lines of code (against the 40 of Mammut). Despite being a basic example, this allowed us to estimate the effectiveness of Mammut and we can expect this gap to increase significantly for scenarios using the more complex features. Moreover, modifying the code using standard OS tools to monitor remote machines would introduce additional complexity and would lack flexibility, since the user should explicitly implement the serialization and transfer of the data. On the other hand, monitoring a remote computing node in Mammut requires only specifying the remote address during the initialization of the library. Mammut allows the designer of self-adaptive strategies to focus on the actual part of his algorithm, minimizing the effort required to monitor or control the computing architecture.

To prove that memory footprint and performance of Mammut are comparable to those of existing tools, we show some results in Table 9.1. We compared the memory usage and execution time of the Mammut API calls with

\(^5\)https://github.com/google/protobuf
#include <mammut/mammut.hpp>
#include <iostream>
#include <unistd.h>

using namespace mammut;
using namespace mammut::energy;
using namespace mammut::task;
using namespace mammut::topology;
using namespace mammut::cpufreq;
using namespace std;

int main(int argc, char** argv){
    Mammut m;
    Energy* energy = m.getInstanceEnergy();
    CpuFreq* frequency = m.getInstanceCpuFreq();
    Topology* topology = m.getInstanceTopology();
    TasksManager* tasks = m.getInstanceTask();

    Cpu* cpu = topology->getCpu(0);
    vector<Cpu*> cpus = topology->getCpus();
    for(size_t i = 0; i < cpus.size(); i++){
        if(cpus[i] != cpu){ cpus[i]->hotUnplug(); }
    }

    ProcessHandler* process = tasks->getProcessHandler(getpid());
    process->move(cpu);
    tasks->releaseProcessHandler(process);

    vector<Domain*> domains = frequency->getDomains(cpu);
    for(size_t i = 0; i < domains.size(); i++){
        domains[i]->setGovernor(GOVERNOR_USERSPACE);
        domains[i]->setHighestFrequencyUserspace();
    }

    CounterCpus* c = (CounterCpus*)
    energy->getCounter(COUNTER_CPUS);
    if(c){ c->reset(); }
    // Do work
    if(c){ cout << "Joules: " << c->getJoulesCpu(cpu) << endl; }
}

Table 9.1: Execution time (in seconds) and memory footprint (in KiloBytes) comparison of different tools. N.A. = Not available (because the functionality is not provided by the tool).

<table>
<thead>
<tr>
<th>Module/Functionality</th>
<th>Mammut</th>
<th>Likwid</th>
<th>Lstopo</th>
<th>Cpupower</th>
<th>Turbostat</th>
<th>Taskset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Energy</td>
<td>0.022 (2012KB)</td>
<td>1.3 (11176KB)</td>
<td>N.A.</td>
<td>N.A.</td>
<td>0.018 (776KB)</td>
<td>N.A.</td>
</tr>
<tr>
<td>Topology</td>
<td>0.011 (2004KB)</td>
<td>0.055 (11128KB)</td>
<td>0.046 (1664KB)</td>
<td>N.A.</td>
<td>N.A.</td>
<td>N.A.</td>
</tr>
<tr>
<td>CPU Frequency</td>
<td>0.029 (1784KB)</td>
<td>0.232 (11088KB)</td>
<td>N.A.</td>
<td>0.008 (1004KB)</td>
<td>N.A.</td>
<td>N.A.</td>
</tr>
<tr>
<td>Process /Thread</td>
<td>0.020 (2652KB)</td>
<td>0.056 (11076KB)</td>
<td>N.A.</td>
<td>N.A.</td>
<td>N.A.</td>
<td>0.002 (680K)</td>
</tr>
</tbody>
</table>

The tool most similar to Mammut is Likwid. However, it does not have any means to monitor remote architectures. This is an important feature due to the capillary diffusion of computing devices, like in IoT and Fog systems. Moreover, Mammut provides a flexible API, that can be used by the programmer to enhance his application by exploiting information about the underlying architecture. On the contrary, Likwid was mainly designed for system administrators, since it provides a set of tools to be used from a command line interface. Although an API has been later added to Likwid, it has a low abstraction level and, differently from Mammut, it is not object-oriented. Providing an object-oriented abstraction is of paramount importance, since it leads to improved maintainability and understandability of the code written by Mammut’s users [235].

9.3 Analyze and Plan

After we described how to customize (if needed) the Monitor and Execute phases, we now show how to customize the Analyze and Plan phases, i.e. how to define new self-adaptive strategies. First, the designer must define a subclass of the Selector class (Code 9.1, lines 35-38) and implement the getNextKnobsValues...
function. In its own \texttt{Selector} the \textit{designer} can access different information provided by the superclass, like: parameters specified by the \textit{user}, the current configuration of the application and statistics about the previous monitored samples, to be used during the \textit{Analyze} phase. The type of monitored data available depends on how the manager has been attached to the application. For example, if the \textit{black-box} interaction was used, the only available information is the throughput (expressed as IPS). Monitored data is kept consistent and updated by \texttt{Nornir} and should be exploited by the algorithm \textit{designer} to select the configuration to be applied at the successive control step to enforce user requirements. Once the decision is made, the next value of each knob needs to be stored into a \texttt{KnobsValues} object, an array of values (one for each knob) which can be accessed by using the enumeration values identifying the type of the knob (lines 1-5). The returned object will then be used to set the appropriate values on the available knobs (lines 9-11).

For example, Code 9.3 shows how to implement a simple selector that, when the monitored latency is lower than 100 ms, will force the application to run on the 25\% of the available cores, setting them to work at 50\% of their maximum clock frequency. When the latency is higher (or equal) than 100 ms, it will run the application on the 80\% of the available cores and will set them to work at 100\% of their maximum frequency.

\begin{verbatim}
class SelectorDummy: public Selector{
  ...
  KnobsValues getNextKnobsValues(){
    KnobsValues k(KNOB_VALUE_RELATIVE);
    if(_samples->average().latency < 100){
      k[KNOB_VIRTUAL_CORES] = 25; k[KNOB_FREQUENCY] = 50;
    }else{
      k[KNOB_VIRTUAL_CORES] = 80; k[KNOB_FREQUENCY] = 100;
    }
    return k;
  }
};
\end{verbatim}

Source Code 9.3: Example of \texttt{Selector} implementation.

\texttt{Nornir} will then automatically translate the percentage values for number of cores and frequencies in real values, according to the availability of resources on the target architecture. Alternatively, it is possible to express absolute values for the knobs. By replacing \texttt{KNOB\_VALUE\_RELATIVE} with \texttt{KNOB\_VALUE\_REAL} in line 4, \texttt{Nornir} will interpret line 6 as “Run the application on 25 cores and set their frequency to 50”. The object \_samples contains the moving average (simple or exponential) of the monitored data. The type of moving average as well as the size of the moving window (or the exponential parameter) can be specified through the \texttt{XML} configuration file. Moreover, when imple-
menting new strategies, it is possible to support only a subset of the available performance and power consumption requirements. For example, the algorithm we described in Section 3.1 only supports performance requirements.

All the strategies we used to validate the self-adaptive algorithms we proposed in this thesis (Chapter 3 and 5) have been implemented in separate Selector classes, by following the process we just described. Being able to implement a broad spectrum of different techniques, ranging from heuristics to online machine learning proves the generality and flexibility of our design. To better highlight the advantages of using Nornir, in Appendix A we will show a more complex example describing the implementation of the algorithm we described in Section 3.1. In some cases, instead of implementing from scratch self-adaptive strategies, it is possible to customize some existing algorithms. For example, it is possible to customize the prediction model used by the online learning algorithm we described in Section 3.2. We briefly recall that the algorithm was composed by an online training phase, in which data about different configuration is collected and used to train a machine learning model. We proposed some regression models for predicting performance and power consumption. However, by extending and defining some C++ classes, it would be possible to maintain the same algorithm structure and only to change the model to be used to perform prediction (e.g., by replacing regression with neural networks). For brevity’s sake, we do not further describe this process here.

Simulator Nornir also provides a simulator, which can be used to prototype and test new algorithms rapidly. In this case, monitoring data can be loaded from log files of previous executions. This allows the designer to optimize and tune his algorithm in a controlled environment. Since the execution of the application is simulated, it is possible, for example, to reproduce 24 hours of execution in just a few minutes. This will allow the designer to quickly analyze the quality of his algorithm and to make the appropriate corrections, without any need to execute it entirely.

9.4 Summary

In this Chapter, we described how a self-adaptive strategy designer can customize Nornir to implement his reconfiguration algorithms. The designer can customize every part of the MAPE loop.

The Monitor phase can be customized by adding the support for other programming frameworks besides FastFlow, allowing also to collect framework specific information, which can be later exploited during the Plan and Analyze phases.

We described how the Execute phase has been designed and how it can be customized by adding new control knobs, to extend the range of possible reconfigurations, improving the granularity of self-adaptive algorithms. We then described Mammut, a high-level C++ framework which we designed and that we used to implement the knobs currently available in Nornir and to col-
lect some architecture monitoring data such as power consumption. By using Mammut, it is possible to interact with system knobs in an intuitive and portable way and to correlate information coming from different parts of the system. Mammut can also be used to add new executors to Nornir or to collect additional information about the computing system on which the application is running. A relevant property of Mammut is that it can be seamlessly used to monitor remote machines, which is a first step towards extending Nornir, in the future, to operate on multiple computing nodes.

Eventually, we described how to add new Analyze and Plan strategies. This is probably the most important characteristic of Nornir, which would enable a quick and easy prototyping and implementation of new self-adaptive and power-aware algorithms, by exploiting the monitor and execute phases already provided by Nornir. Indeed, this made it possible to easily implement all the algorithms we used in Chapter 4 and to execute them over a wide set of real applications, allowing us to validate the new algorithms we presented in Chapter 3 and 5.
Part V

Conclusions and Future Research Directions
Conclusions

In this thesis, we addressed the problem of providing explicit performance and power consumption guarantees on parallel applications running on shared memory multicore machines, using self-adaptive reconfiguration solutions. We addressed this problem both on the scientific and engineering side, by tackling the problem from three different directions:

- We designed some new self-adaptive algorithms, which monitor the application throughout its execution, selecting time to time the best amount of resources to be allocated to the application to enforce user’s requirements on performance and power consumption. In doing so, we built a simple online learning algorithm which, without any previous knowledge, can achieve the same accuracy of complex solutions which exploit data collected offline about many different applications. Similarly to a heuristic, we will only use data collected online while the application is running, while achieving at the same time an higher accuracy than heuristic solutions. Moreover, differently from offline learning solutions, our algorithm it is not affected by biases due to the choice of the training set, and can better react to new and unseen applications since it will train the model on-the-fly on the application itself.

Then, we proposed a concurrency control algorithm, which can be used to optimize the access to shared data structures in multithreaded applications.

- By analyzing post-mortem data about applications in the PARSEC benchmark we showed that in some cases, applying concurrency throttling could be much more profitable than using thread packing. This means that, for example, given a maximum power budget, by using concurrency throttling it would be possible to achieve higher performance. However, concurrency throttling is a complicated technique and cannot be applied to
any parallel application. Indeed, we identified some structural requirements the application should have to be a possible candidate for concurrency throttling application. We show that these requirements are usually satisfied by parallel patterns-based applications. For this reason, we designed and implemented a benchmark for parallel patterns-based applications. On one side, we proved that parallel patterns programming is a good alternative to more common programming methodologies since it requires a lower programming effort while providing comparable (or even better) performance. On the other side, since we provided a comprehensive set of pattern-based applications, it would be possible in the future to analyze in detail the impact of concurrency throttling on parallel applications.

- We designed and implemented Norniir, a new C++ framework for enforcing performance and power consumption requirements on parallel applications. On one side, we believe that this could be a useful tool to be used in real scenarios. Indeed, we built the framework so that it can be used for any iterative application with minimal programming effort, as we did for legacy PARSEC applications to evaluate the algorithms we proposed. On the other side, the framework is customizable and can be used by strategy designers to rapidly prototype new self-adaptive algorithms and to easily compare them with existing solutions. By doing so, the algorithm designer would just focus on the algorithm itself, without any need to deal with issues related to monitoring and execution of the reconfiguration decisions.

We believe that providing some open source tools will enable further research in the future, even in fields not directly related to self-adaptive computing. For example, Mammut (the tool we designed to interact with system’s knobs and sensors) have been recently used in the RePhrase EU H2020 project\textsuperscript{1} as low-level runtime tool for collecting power consumption and other statistics (e.g., intensity of memory accesses) of parallel applications\textsuperscript{2}. The tool has been used for a completely different use case from those we considered in this thesis. Indeed, in this specific project, information collected by using Mammut are used by the runtime system for deciding which architecture is most suited to execute a specific parallel application. Mammut has also been recently used to measure and optimize power consumption of query processing in web search engines\textsuperscript{3}, and OCaml and C bindings of Mammut have been recently implemented and released as open source\textsuperscript{3} by researchers at University of Orleans.

Some interesting challenges are waiting for us shortly.

\textsuperscript{1}http://rephrase-ict.eu/
\textsuperscript{2}http://rephrase-eu.weebly.com/uploads/3/1/0/9/31098995/d4-1.pdf
\textsuperscript{3}https://github.com/mathiasbourgoin/ocaml_mammut
Firstly, we would like to investigate the possibility of applying our online learning algorithm to a scenario with multiple co-running applications, where each application has its performance and power consumption requirements. Despite some existing works allow specifying requirements for multiple applications, they have some limitations. For example, [135] and [105] can only enforce performance requirements, while [149] can only provide guarantees on power consumption. Moreover, these solutions usually target distributed memory machines, and they usually do not exploit all the knobs available on the machine. To the best of our knowledge, providing explicit performance and power consumption guarantees on multiple applications, on multicores architectures, is still an open problem. Considering such scenario would pose significant extra challenges to be solved (e.g., coordination between different MAPE loops), which are clearly outside the scope of this thesis. It is worth to remark that, the proposed algorithm allowed other applications to run on the system at the same time. The only limitation was that only one application at a time is allowed to have requirements on performance or power consumption.

Another step involves the exploitation of additional emerging control knobs such as \textit{uncore frequency scaling} [95] and \textit{voltage scaling} on DRAMs. Another interesting possibility could be the design of online learning algorithms for predicting the impact of \textit{Simultaneous MultiThreading} (SMT) on performance and power consumption of parallel applications. Indeed, most solutions either disable SMT or simply use all the hardware threads available on the machine. However, recent computing architectures like \textit{Intel Xeon Phi KNL} and \textit{IBM Power8} provides many hardware threads for each physical core (4 and 8 respectively). Accordingly, the optimal solution could lay somewhere in the middle (e.g., using only 3 hardware threads of the 4 available). This, of course, depends on the specific characteristics of the application and even on the application’s input. Despite some solution exists, they are usually evaluated on computing systems with few hardware threads per core. On a machine like the \textit{IBM Power8}, we would have more than 8000 possible configurations of number of cores, clock frequency and number of hardware threads. Being able to find the optimal configuration according to the user requirements in such a big search space is a much more challenging problem and would require further investigation.

Eventually, we would like to actually apply concurrency throttling on the applications in the P3ARSEC benchmark suite. By doing so, we would get useful insights on some practical issues which may arise when using concurrency throttling on real and complex parallel applications. Moreover, this would also allow us to evaluate the actual impact of using concurrency throttling during application execution.
Part VI

Appendices
HEURISTIC IMPLEMENTATION IN NORNIR

To appreciate the programmability of Nornir, we show in Listing A.1 the code required to implement the self-adaptive algorithm described in Section 3.1. We briefly recall that this algorithm can be used to enforce a throughput higher than a given threshold on an application. Since more configurations may have such throughput, the algorithm will select the configuration with the lowest power consumption among them.

The algorithm predicts the throughput of any configuration starting from the throughput of the current configuration, supposing that it scales linearly with both the frequency and the number of cores. Accordingly, if \( R(n, f) \) is the throughput of a configuration using \( n \) cores with a clock frequency \( f \), we have:

\[
R(n, f) = R(\bar{n}, \bar{f}) \cdot \frac{n}{\bar{n}} \cdot \frac{f}{\bar{f}}
\]

where \( \bar{n} \) is the number of cores currently used and \( \bar{f} \) is the current clock frequency. In lines 4-5 we retrieve the \( n \) and \( f \) for the configuration \text{pConf} \ we want to predict, while in lines 6-7 we get the current number of cores, the current clock frequency and the current throughput. Eventually, in lines 9-11 we perform the prediction.

The power consumption \( P(n, f) \) is predicted as:

\[
P(n, f) \propto v^2 nf
\]

with \( v \) being the voltage associated to a specific number of cores and clock frequency. The voltage depends only on these two factors and it is precomputed when installing Nornir and stored to a configuration file, which can be accessed at runtime through the \text{getVoltage} function. Lines 18-21 predicts the power consumption of a specific configuration \text{pConf} by using this model.
Source Code A.1: Implementation of the algorithm presented in [26] (denoted as Heuristic in this thesis.)
Then, as described in Section 9.3 we need to extend the Selector class and define its getNextKnobsValues function (lines 44-51). In lines 45-48 we check if the monitored throughput is higher or equal to the one required. The monitoring data can be accessed through the _samples variable provided by the Selector class\(^1\). If this is the case, the configuration we are using satisfies the requirements expressed by the user and we simply return it. Otherwise, the algorithm searches a better configuration (line 49), by calling the getBestConfiguration function. This function scans all the configurations (line 34) and, for each of them, checks if the throughput is higher than that required by the user, and if the power consumption is lower then the lowest found up to that moment. If this is the case, both the best configuration and its power consumption are updated (lines 35-39). Eventually, the best configuration found is returned (line 40). As discussed in Section 9.3, this configuration will eventually be enforced by Nornir by using the appropriate actuators.

Although being a simple example, it clearly shows the advantages using Nornir. Indeed, we only focused on the algorithm, without dealing with complex issues related to monitoring the application and interfacing with the underlying hardware, which is automatically managed by Nornir.

\(^1\)For brevity’s sake, we did not show the constructors of the different classes. Constructor of SelectorAnalytical class simply creates the two predictors objects, by providing them the _configuration and _samples variables, which it gets from the Selector base class.
In this Appendix we will provide detailed results about the speedup of the applications used in the P³ARSEC benchmark suite (Chapter 7). For each architecture and for each application, we will show the speedup of both the original PARSEC implementations and of the parallel patterns-based implementations.

B.1 Intel Xeon Server
APPENDIX B. P³ARSEC SPEEDUPS

Dedup - Intel Xeon Server - Alternative Version 2

Dedup - Intel Xeon Server - Alternative Version 3

Dedup - Intel Xeon Server - Alternative Version 4

Facesim - Intel Xeon Server

Ferret - Intel Xeon Server

Ferret - Intel Xeon Server - Alternative Version 2

Ferret - Intel Xeon Server - Alternative Version 3

Ferret - Intel Xeon Server - Alternative Version 4
Figure B.1: Speedup of PARSEC applications varying the number of used cores on Intel Xeon Server architecture.

B.2 Intel Xeon Phi
APPENDIX B. $P^3$-ARSEC SPEEDUPS

![Graphs showing speedup for different applications and parallelism degrees.](image-url)
Figure B.2: Speedup of PARSEC applications varying the number of used cores on Intel Xeon Phi architecture.

B.3 IBM Power 8
B.3. IBM POWER 8

![Graphs of different benchmarks on IBM Power 8](images)

- **Blackscholes** - IBM Power 8
- **Bodytrack** - IBM Power 8
- **Dedup** - IBM Power 8
- **Dedup - Alternative Version 2**
- **Dedup - Alternative Version 3**
- **Dedup - Alternative Version 4**
- **Facesim** - IBM Power 8
- **Ferret** - IBM Power 8
APPENDIX B. \textit{P}^3\textit{ARSEC SPEEDUPS}

Figure B.3: Speedup of PARSEC applications varying the number of used cores on IBM Power 8 architecture.


