POLITECNICO
MILANO 1863

POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

ENABLING POWER-AWARENESS FOR MULTI-TENANT SYSTEMS

Doctoral Dissertation of:
Matteo Ferroni

Supervisor:
Prof. Marco Domenico Santambrogio

Tutor:
Prof. Donatella Sciuto

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

2016 – Cycle XXIX

Abstract

POWER consumption has become a major concern for almost every digital system: from the smallest embedded devices to the biggest data centers, energy and power budgets are always constraining the performance of the system. Moreover, the actual power consumption of these systems is strongly affected by their current "working regime" (e.g., from idle to heavy-load conditions, with all the shades in between), which depends on the guest applications they host, as well as on the external interactions these are subject to. It is then difficult to make accurate predictions on the power consumed by the whole system over time, when it is subject to constantly changing operating conditions: a self-aware and goal-oriented approach to resource allocation may then improve the instantaneous performance of the system, but still the definition of energy saving policies remains not trivial as far as the system is not really able to learn from experience in real world scenarios.

In this context, this thesis proposes a *holistic power modeling framework* that a wide range of energy and power constrained systems can use to profile their energy and power consumption. Starting from the preliminary experience developed on power consumption models for mobile devices during my M.Sc. thesis, I designed a general methodology that can be tailored on the actual systems features, extracting a specific power model able to describe and predict the future behavior of the observed entity. This methodology is meant to be provided in an "as-a-service" fashion: at first, the target system is instrumented to collect power metrics and workload statistics in its real usage context; then, the collected measurements are sent to a

remote server, where data is processed using well known techniques (e.g., Principal Components Analysis, Markov Decision Chains, ARX models, etc.); finally, an accurate power model is built as a function of the metrics monitored on the instrumented system. The generalized approach has been validated in the context of power consumption models for multi-tenant virtualized infrastructures, outperforming results from the state of the art.

Finally, the experience developed on power consumption models for server infrastructures led me to the design of a *power-aware* and *QoS-aware* orchestrator for multi-tenant systems. On the one hand, I propose a performance-aware power capping orchestrator in a virtualized environment, that aims at maximizing performance under a power cap. On the other hand, I bring the same concepts into a different approach to multi-tenancy, i.e., *containerization*, thus moving the first steps towards power-awareness for Docker containers orchestration, laying the basis for further research work.

Contents

CHAPTER 1

Introduction

1.1 Context and problems definition

Power consumption has become a major concern for almost every digital system: from the smallest embedded devices to the biggest data centers, energy and power budgets are always constraining the performance of the system.

On the one hand, this is the case of battery-powered devices like smartphones and tablets: these have brought a disruptive change in the way we work and live [31] [27], as they allow the user to surf the Web, take pictures, track sport performance and health status using the same embedded device. Unfortunately, their batteries have to be small and lightweight, as the device has to be portable, but this of course constrains their potentials. Moreover, once the hardware has been designed and produced to deal with those constraints, any further energy optimization is left to the runtime, from the firmware to the application, throughout the whole software stack.

On the other hand, power consumption remains an open issue also in those contexts that do not involve batteries: for instance, data centers providers aim to reduce it as much as possible to decrease operating costs and to improve system reliability. Even though the performance-per-watt ratio has

been constantly rising, the total power drawn is hardly decreasing and recent trends suggest that the cost of the energy consumed by a server during its lifetime will probably exceed the hardware cost in the near future [20]. Given the strong correlation with live operating costs, power consumption *consolidation* through applications colocation and migration becomes critical for the Cloud Computing paradigm, which delivers computing services as a utility in a "pay-as-you-go" manner [16].

These systems share an important characteristic: their actual power consumption is strongly affected by their current ***"working regime"***, from idle to heavy-load conditions, with all the shades in between. This depends on the guest applications they host, as well as on the external interactions these are subject to.

In the case of a mobile device, its usage is often spiky (e.g., check emails, make a fast phone call, quickly take a picture, etc. [52]) and its battery is highly influenced by its internal hardware state (e.g., which network connectivity interface is turned on, which sensors are being monitored and at which sampling frequency, etc.) and by external conditions (e.g., wireless network signal strength in a certain location [32]). Moreover, we can not disregard the user's behavior, as extensively discussed in the literature [89, 151].

For what concern data centers, *virtualization* allows multiple tenant applications to share physical resources while fulfilling needs for Quality of Service (QoS), security, and strong isolation [10, 99, 141]. It provides a clean separation of software development concerns from the underlying hardware platform, fostering *heterogeneity* from both the virtual tenants and the underlying infrastructure. On the one hand, virtual tenants may be intrinsically different from one another due to different workload limitations (i.e., they can be memory-bound, I/O-bound and/or CPU-bound) and evolving load patterns (e.g., algorithmic phases). On the other hand, each physical host may exhibit different performance and power characteristics from other hosts (even supposedly identical ones), given the same configuration of tenants. It is easy to see how a virtualization infrastructure requires sophisticated approaches to resource allocation and accounting, a requirement that can quickly become intractable as the number of virtual tenants per host increases [140].

Even though these use cases are quite different from multiple perspectives (e.g., computational purposes, domain fields, target users and so on), it is easy to notice what they have in common: they need to deal with (1) hardware *heterogeneity*, (2) software *multi-tenancy* and (3) input *variability* over time. Given this context, it is difficult to make accurate predictions on

the power consumed by the whole system over time, as it is subject to constantly changing operating conditions: under this perspective, any attempt to optimize these system then becomes far from trivial.

The aforementioned scenarios will be extensively discussed throughout this thesis work, addressing the following two questions:

A. *how much power is a system going to consume, given certain working conditions?*

B. *is it possible to control a system to consume less power, still satisfying its functional requirements?*

An important aspect we need to face is the fact that heterogeneity, multi-tenancy and time variability make a *comprehensive profiling* of these systems unfeasible: in fact, it is not reasonable to explore all the possible system's *"working regimes"* before the final deployment of the system, given the huge number of hardware features available, the possible combination of concurrently running application, together with their input fluctuations.

However, we can tackle these uncertainties starting from the following simple but significant assumption: *the system will probably behave in the future as it did in the past*. This suggests that the system can *learn from experience*, trying to improve its performance in its real working conditions, as presented in the next sections.

1.2 Proposed approach and requirements

A preliminary analysis of the context leads to the following outcomes:

A. the best strategy is to *observe* the behavior of the system at *runtime*, during its real working conditions;

B. we can then *learn* from the experience which are the variables that mostly affect the system's power consumption;

C. this knowledge can then be used to *decide* what to do and *actuate* the best strategy with respect to performance goals and power constraints.

These steps must be performed in *automation* throughout the whole lifetime of the system.

In literature, this approach is known with the name of (O)bserve-(D)ecide-(A)ct (ODA) [80] control loop: here, we introduce an additional (L)earning phase to decouple the knowledge learnt (i.e., self-awareness) and the decision phase (i.e., self-expression). The architecture style can then be

described as an OLDA control loop, as suggested in other works as [54]. The same approach is often called *MAPE-K* [93], where the (K)nowledge component is shared by the (M)onitor, (A)nalyser, (P)lanner and (E)xecutor components. The O and A components in ODA are equivalent to M and E components in MAPE-K respectively, while analysis and planning tasks are subsumed in the Decide component. As these formulations are equivalent, the next chapters will refer to one or the other indistinctly.

In this thesis work, I want to exploit this approach to enable *power-awareness* for a wide range of autonomous agents, relying on the following:

> **definition**: a system is ***power-aware*** when it is *aware of how much power a certain behavior will be consuming, given a certain working condition*.

On the one hand, in the context of battery-powered systems like smartphones and tablets, it is important to predict how long the battery will last in order to avoid energy waste and achieve both short-term objectives (e.g., find the fastest route while driving to the office in the morning) and long-run ones (e.g., make an important phone calls in the evening). On the other hand, in a multi-tenant virtualized server, it may be important to precisely estimate how much each tenant is contributing to the actual system power consumption, in order to limit it, still guaranteeing the performance requirements of the other tenants.

Given the research opportunities in the field, this work proposes a *holistic power modeling framework* that a wide range of energy and power constrained systems can use to learn and predict their energy and power consumption. I designed a general methodology that can be tailored on the actual systems features, extracting a specific power model able to describe and predict the future behavior of the observed entity.

This methodology is meant to be provided in an as-a-service fashion: at first, the target system is instrumented to collect power metrics and workload statistics in its real usage context (*Observe* phase); then, the collected measurements are sent to a remote server, where data is processed using well known techniques (e.g., Principal Components Analysis, Markov Decision Chains, ARX models, etc., *Learning* phase); finally, an accurate power model is built as a function of the metrics monitored on the instrumented system. This information will support a power-aware Operating System (OS) in the estimation of the best tradeoff between global performance and power consumption (*Decide and Act* phases), still providing the required QoS to the guests applications towards an adaptive and power-aware multi-tenant system.

Each phase has its own requirements and trade-offs:

A. *accuracy* and *precision* are strong requirements when monitoring a system in its real working conditions; however, the *overhead* introduced should be as low as possible;

B. the analysis performed on the acquired data must produce *precise* models of its behavior, but the computation required to build the model should not be in charge of the system itself; this is the reason why we aim at defining a *model-a-service* approach, thus leaving the burden of models' estimations to a more specialized and efficient third-party;

C. finally, the actuation on the system's knobs should be *effective*, thus leading to a more *efficient* system configuration, still considering the *performance* of the tenant applications.

These requirements will be discussed throughout the whole thesis, while the next section gives an overview of the entire work.

1.3 Thesis outline: a journey towards power-awareness

I moved my first steps in the field of power-aware systems with my M.Sc. thesis: it consisted in the *MPower* project, a mobile application able to predict how long the battery of a smartphone is going to last, given the current hardware configuration of the device (i.e., CPU utilization, LCD brightness, WiFi state, and so on) [27, 58, 120]. Chapter 2 gives a brief overview of the previous work, as it is necessary to lay the basis for this thesis, then focusing on the novel contributions developed during my first year of Ph.D.: (1) the formal description of the improvements on the power model estimation methodology for mobile device, taking into account both the device modeling and the user behavior; (2) an extensive experimental campaign, that allowed to compare the predictions obtained by the *MPower* application with the ones provided by the Android L OS; (3) a set of "in–lab" experimental tests to assess the negligible overhead introduced by the monitoring application.

Chapter 3 discusses how it is possible to generalize the same concepts towards a comprehensive and general methodology: the observed system does not need to be a smartphone but it could be a generic system, i.e., an *"agent"*, that wants to become *power-aware* or, in a wider sense, *"resource-aware"*. The generalized data-driven methodology for resource consumption modeling has then been implemented into MARC, a Cloud-service platform designed to Model and Analyze Resource Consumption trends (MARC), supporting a "Model-as-a-Service" paradigm. In order to validate

the proposed methodology, a custom simulator has been set up to generate a wide spectrum of controlled resource consumption traces: this allowed to verify the correctness of the framework from a general and comprehensive point of view. Moreover, regression tests show how it is able to reproduce the same precision of the results obtained in Chapter 2, thus showing how the *MPower* methodology has been generalized and abstracted consistently. Then, the following chapters show how this generalization allows to bring power-awareness into a completely different context: power consumption models for virtual machines in a multi-tenant virtualized system.

As already discussed in this chapter, the first step is to *observe* the behavior of the system at runtime, during its real working conditions: Chapter 4 describes the design and the implementation that lead to *XeMPower*, a lightweight monitoring solution for the Xen hypervisor. It precisely accounts hardware events to guest workloads, enabling attribution of CPU power consumption to individual tenants. Results show that *XeMPower* introduces negligible overhead in power consumption, aiming to be a reference design to monitor power-aware virtualized environments.

The approach to power consumption attribution presented in Chapter 4 is trivial, as it represents a mere example to show the tool's potential. Chapter 5 then presents how to improve power-awareness, using *XeMPower* and *MARC* to build data-driven power consumption models for multi-tenant virtualized infrastructures. Results show a modeling relative error of around 2% on average, and under 4% in almost all the cases and on different workload classes, outperforming previous research in the field. Moreover, the chapter discusses model portability across similar architectures, showing how they can be also used to evaluate tenants colocation in a multi-tenant infrastructure.

Up to now, we just focused on power modeling. However, in order to thoroughly explore the topic, workloads' *performances* needs to be included in the loop. The last two chapters will then explore how a power-aware system should *plan* future decisions and *execute* the best actions with respect to performance goals and power constraints, i.e., the last two steps of the OLDA control loop introduced in Section 1.2.

Chapter 6 presents *XeMPUPiL*, a performance-aware power capping orchestrator for the Xen hypervisor: it implements a hybrid hardware-software power capping solution, based on the PUPiL [163] control loop, that aims at maximizing the performance of a workload under a power cap. *XeMPUPiL* has been validated with just one guest application running at a time: this condition may not be very common in a real production environment, where multiple guests may be hosted on the same node, each one

with different performance requirements.

In order to tackle these issues, a smart resource manager must be put in place to deal with resource allocation, power constraints and performance requirements. These issues are addressed in Chapter 7, that discusses preliminary results and opportunities of *containerization*, i.e., a different approach to multi-tenancy: the proposed solution is called *DockerCap* and represents the first step towards power-awareness and QoS-awareness for Docker container orchestration, laying the basis for further research work. These and other future works are discussed in Chapter 8, thus concluding this thesis work.

Most of this work has been part of a collaboration between Politecnico di Milano and the University of California, Berkeley (CA, USA), and it has been supported by a HiPEAC collaboration grant that I won on my second year of Ph.D.

CHAPTER 2

Preliminary steps: power models for Android devices

2.1 Introduction

Smartphones and tablets have brought a disruptive change in the way we work and live [31] [27]: we can now surf the Web, take pictures, track sport performance and health status using the same embedded device. Even though dozens of brand new devices are released every year [64], all of them have to deal with a limited energy budget: their batteries have to be small and lightweight, as the device has to be portable, but this of course constrains their capacity. Once the hardware has been designed and produced to deal with those constraints, any further energy optimization is left to the developers, from the firmware to the application, throughout the whole software stack.

Many energy saving solutions have been explored in the field: some of them involve the display (e.g., dimming the screen brightness, reducing the frame rate or the resolution, etc.), while others involve the network connectivity (e.g., limiting the amount of background data transmitted) or the applications running (e.g., revoking *wakelocks* for background applica-

tions), and so on. However, all these solutions often set some constraints on the functionalities of the system, thus introducing a new important tradeoff between energy optimization and the features provided to the user, the last stakeholder that comes into play: from her point of view, the device should not save energy regardless the context, but it has to provide all the functionalities she needs until she can recharge its battery again. This is the reason why it is important to know exactly how the battery drains given a certain usage of the device, in order to save energy with respect to both the users current needs and final goals: therefore, an accurate power model of the device is fundamental to predict its energy behavior over time.

Many attempts have been made in the field, trying to create accurate power models describing the discharge level as a function of the time, and they all have to deal with the tradeoff between the precision and the flexibility of the power modeling methodology. On the one hand, most of them provides solutions for a controlled environment and for a single device [14, 30, 128, 135, 143, 165]. Therefore, they hardly generalize to the wide variety of devices currently available on the Market. On the other hand, it is possible to provide a more flexible methodology, performing the estimations on the device itself in its real world condition: an interesting contribution is provided by Google within the last Android release, codename *Lollipop*, as they introduced a prediction of the battery lifetime given the current level of charge and the recent activity of the smartphone. These prediction may be accurate under the assumption that the system will behave in the same way as it did in the last short period of time, but it is not the case of a mobile device: its usage is often spiky (e.g., check emails, make a fast phone call, quickly take a picture, etc. [52]) and its battery is highly influenced by its internal hardware state (e.g., which network connectivity interface is turned on, which sensors are being monitored and at which sampling frequency, etc.) and by external conditions (e.g., wireless network signal strength in a certain location [32]). Other solutions take into account the user's behavior [89, 151], using real data to build a customized power model. These solutions overcome the single device modeling limitation and are more flexible, but they do not consider device specific power consumption (for instance, different devices may behave differently, even if they are used by the same person).

In order to mitigate this tradeoff, this chapter presents a *data-driven* methodology for power modeling in the mobile devices' context, able to provide a good precision in prediction while it generalizes to a wide range of devices. This technique is able to both estimate the hardware components power consumption and model the users' behavior by basing on the

measurements coming from sensors embedded in the device's in its real usage context to learn how its battery behaves under different conditions. Since the model is built directly on data coming from each specific device, it guarantees a high level flexibility w.r.t. new devices and new OS versions and may apply to the burden of devices currently available on the market.

To validate the proposed methodology, I implemented *MPower*, a real system supporting a huge variety of devices running the Android OS, as it provides data through a set of system APIs common to all release and devices. The system is composed of a mobile application, implementing a low power logger, and a remote server, where the main computations are offloaded: this is done not to consume additional power on the devices, as suggested by the common cloud based *Software-as-a-Service* patterns). [1] The proposed methodology aims to be as little invasive as possible, both for the user and the device. Finally, the system makes use of these power models to make predictions of the Time-To-Live (TTL) (in terms of minutes, not just in term of remaining battery percentage) of every observed device under certain conditions: this is an important information for the user, as it allows her to choose how to use her device, with respect to her current and future needs.

This chapter summarizes the most recent and final developments of the ideas presented in [27, 58, 120]. In [27] we just presented the concept at the beginning of its development, while in [120] we described the preliminary results obtained on a limited set of devices and in [58] we presented the part of the methodology taking into account only the device modeling. The novel contributions presented in this chapter are:

- the formal description of the improvement of the power model estimation methodology for mobile device, taking into account both the device modeling and the user behavior, by means of a set of parametrized linear model and an Hidden Markov Models (HMM) techniques, respectively;
- an extensive experimental campaign performed on 600 devices data coming from the aforementioned Android App, from which the model was estimated and where we asses its effectiveness compared with the one embedded in the Android L OS release;
- "in–lab" tests have been performed to show how the implemented logging application does not affect the energy behavior of the observed mobile devices.

[1] The application is available for free on the Google Play Store (`https://play.google.com/store/apps/details?id=org.morphone.mpower`)

The chapter is structured as follows: Section 2.2 provides the analysis of the state of the art about power models for mobile devices; Section 2.3 gives a bird's-eye view of the proposed methodology, stating some fundamental concepts; Section 2.4 presents the theoretical background on which the power model is based; Section 2.5 discusses the most remarkable implementation details of the developed logging system; Section 2.6 presents results on the accuracy of the produced power models and on the efficiency of the logging application; Section 2.7 provides some final remarks.

2.2 Related work

Several power modeling methodologies for mobile devices have been proposed in literature and used as a support for smart power management systems or to compare power performances of different devices. In this section, I provide a brief overview of the most noteworthy approaches, while an interested reader may find a thorough discussion in [57].

I propose a categorization based on the approach used to observe the system, since it affects different features of the model (e.g., the modeling theory that can be exploited, the level of detail achievable, the degree of adaptivity to changes in the working conditions and over time, etc.). I identified three big families of methodologies that build power models: using *external measurement* (Section 2.2.1), *internal measurement* (Section 2.2.2) or by relying on *power-related variables* available through software APIs (Section 2.2.3).

2.2.1 External measurements

External measurements provide an accurate inspection of the mobile device with predetermined and controlled conditions [14, 30, 135, 143]. Usually, they are performed by replacing the battery with an external power supply able to profile the energy consumption. Despite the precision of the measurements (generally, the estimation error is under 2%), the main problem with this method is the lack of flexibility. In fact nowadays there is a great variety of mobile devices, each one of them with a different hardware/-software configuration, thus it is unpractical to generate an offline power consumption model for each of them [110]. Moreover, this analysis does not evolve with the running life of the device. On the other hand, these kind studies help understanding the breakdown structure of mobile devices power consumption. For instance, from the pioneering studies in [30] emerged that the majority of power consumption can be attributed to the GSM module and the display (including the LCD panel and touchscreen),

the graphics accelerator/driver, and the screen backlight, while other subsystems showed negligible power consumption.

2.2.2 Internal measurements

The second approach takes into account measurements provided by interfaces already available on a specific device or OS, e.g., the Advanced Configuration and Power Interface (ACPI) or the Nokia energy profiler. It is slightly more flexible than the first one [48,69,97,155] and allows to gather precise information about the battery status (e.g., voltage, current and temperature), whose precision is determined only by the available internal sensors sensitivity. However, these models are not fully portable: not every Android-powered devices can take advantage of these power models, because they may be unable to provide the required low level physical sensors, e.g., most of the Motorola and Samsung devices do not provide current and temperature sensors. Nonetheless, one of the most famous work in this area is PowerBooter [164], where each component is analyzed separately, using external tools. Interestingly, their studies concluded that the power consumption of major component affects the system independently, e.g., the device power consumption when 3G and WiFi are active is the sum of 3G and WiFi contributions. They also noticed that different devices does not share the same power model, which justifies the development of device-customized power models. Their method was modified to be able to create a power model based on data coming from sensors within each device and the results indicate that the power model built with PowerBooter is accurate within 4.1% of measured values for 10-second intervals. However, this model presents two main limitations: it has to know beforehand the discharge curve of each device and the application requires root privileges to be run. With this second measurement methodology, it is possible to obtain models with error from the 5% [164] to the 10% [155].

2.2.3 Energy-related variables observation

The third possible approach (the one we adopted) relies on important energy-related variables to estimate the battery discharge curve and the correspondent Times-To-Live (TTLs). The energy related data are accessed by means of OS Application Programming Interfaces (APIs), thus having some limitations in terms of precision, but allowing these power models to be applied on every device. Even if many works follows this paradigm, at the best of my knowledge, the methodology proposed in this chapter is the first public available methodology (and application) actually working on the large

variety of the Android ecosystem.

Xiao et al. [157] present a methodology for building a system-level power model, without requiring laboratory measurements. They develop a linear regression model with non-negative coefficients, describing the aggregate power consumption of the processors, the wireless network interface and the display. They discovered that a linear regression model is sufficient to model the relation between the variables they choose and the power consumption. To estimate the model, they used five different workloads: idle with different brightness levels, audio/video players, audio/video recorders, file download/upload at different network data rates, and data streaming. The power estimation, based on their model, exhibits a median error of 2.62% in real mobile internet services. Moreover, they provide a model independent from usage scenarios providing reasonable accuracy for runtime estimation. Differently from the work proposed here, in [157] a single device has been used to build the model. Moreover, they built one single model for all the configurations, without taking into consideration that the discharge curves presents are influenced by the device configurations. Finally, they used measurements gathered in a controlled environment: this is an important point since, as already mentioned, the environment of mobile devices cannot be fairly reproduced in laboratory.

In [165], a context-aware system to accurately predict the battery lifetime is proposed. The energy consumption of each system component is considered dependent on its operational state and the amount of time the component remains in that state. As a consequence, the system power consumption is modeled as the sum of the system components. Data about the discharge rate were collected in several system contexts, where a single system context is a combination of CPU utilization, LCD brightness, WiFi state, IO idle rate and volume of transferred data. Multiple linear regression technique was used to build a model able to describe the battery discharge rate, based on the system component states. The system was tested using only a single device, i.e., a HTC G1 smartphone, running the Android OS. The generated model predicted the battery remaining lifetime with a relative error of 10%.

In [128], Pathak et al. propose a new power modeling approach based on tracing system calls of the applications. Their scheme consists of a Finite State Machine (FSM), to model the power states, and state transitions. Some of the states have constant power consumption, to describe non-utilization power consumption, while other states leverage a Linear Regression (LR) model (the second major component of the model) to capture the power consumption due to system calls generating workload.

Moreover, a testing application is used to systematically uncover the FSM transition rules. Tests were performed on a HTC Touch Pro and a HTC TyTn 2, powered by Windows Mobile 6, and an HTC Magic running Android. This new approach improves the accuracy of fine-grained energy estimation compared to utilization-based model. Indeed, their model have a 80th percentile error of less than 10% estimating the power consumption over time of a generic application execution that lasted 50ms, while the utilization-based model have an error that varies between 16% and 52%. The whole application error, with 1 sec granularity, varies between 0.2% and 3.6%, compared to the error 3.5-20.3% given by utilization-based models.

Finally, the last and most significant contribution is provided by Google, that introduced a suite of enhancements to boost smartphones' battery life within the *Android Lollipop* release, developed in 2014.[2] One of the most anticipated features of this release was the ability of the operating system to provide a prediction of how long a certain level of battery charge will last, i.e., what we previously defined as TTL. An example of this feature is provided in Figure 2.1. Even though other applications with the same goal (i.e., TTL prediction) are available on the Google Play store [7, 8, 66, 131], none of them provides information about its precision in the TTL estimation or details about the modeling methodology.

On the contrary, we believe that the aforementioned feature of Android Lollipop has been inspired by the already cited work of [164], then extended and made available inside the Android OS. As this approach is similar to the one proposed in this thesis and it is the most recent and official solution to the problem of power models for mobile devices, I decided to consider the results provided by Android Lollipop as a baseline to compare our methodology with. Nonetheless, I still want to stress that our system provides more flexibility, as it is designed to work on all the available Android devices, giving the possibility to support already existing devices, and it is able to consider the users' habits too, as discussed in the next sections.

2.3 Problem definition

As discussed in Section 2.2, the vast majority of the existing methodologies makes use of measurements performed on a limited number of devices under a controlled environment, thus leading to non extendible power modeling techniques. Conversely, this chapter aims at designing a system which:

[2] An exaustive description of the android lollipop release can be found at `http://www.android.com/versions/lollipop-5-0/`

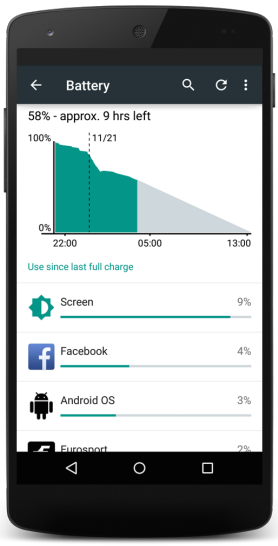


Figure 2.1: *TTL prediction in Android Lollipop*

- is able to estimate power models in the most flexible way while having a reasonable precision for energy saving purposes;
- is able to adapt to the behavior of each different user;
- does not depend either on the mobile device model or *OS*;
- does not influence the power behavior of the device itself.

Most of the devices available on the market provide instantaneous information about their state and the variable influencing their power discharge by resorting to a set of built–in sensors. By relying on these data it is possible to monitor the power discharge of the device, as well as to adapt to changes occurring on the hardware (e.g., caused by a degradation of the battery or the physical components) or the user behavior (e.g., as soon as she modifies her daily routine). Given a minimum amount of sensor, common to all devices, it is possible to gather these information to design a general purpose methodology for power estimation. Thus the only assumption required to develop the proposed methodology is the availability of a set of features providing information on the power state and discharge of the device. [3]

[3]The choice of the set of feature considered is out of the scope of this chapter. While for applicative purposes one can refer to [30] for the ones most influencing the power state of a device, for a more theoretical point of view one can refer to [11].

The problem of predicting the power discharge on a set of F mobile devices is here formulated as the estimation of a set of unknown dynamic processes $\{\mathcal{P}_1, \ldots, \mathcal{P}_F\}$, each of which is a dynamic process corresponding to a single device. We assume to have a dataset coming from the each one of the aforementioned dynamic process $\mathcal{P}_i$:

$$Z_N = \{z_t = (x(t), y(t))\}_{t \in T}$$

where $x(t)$ is the set of the measurement provided by sensors at time t, $y(t) \in \mathbb{R}$ is the energy discharge in a given time interval $(t-1, t)$, $T = \{t_1, \ldots, t_N\}$ is the set of the time instant when samples are available and $N = |T|$ is the cardinality of the dataset. Starting from this dataset, we want to estimate a predictive model for the power discharge of the device which takes into account the hardware drain and user behavior.

We can logically distinguish among two different kind of measurements $x(t)$ available for power estimation purposes: the one which determines the *"working regime"* of the device and controllable by the user/by the software, and those which are related to the interaction of the device with the environment, which can not be directly modified. Based on this distinction, we introduce the following definition:

> ***working regime***: vector of the states of some of the device components:
>
> $$\mathbf{c} = (c_1, \ldots, c_S),$$
>
> where $\mathbf{c} \in \mathcal{C} \subset \mathbb{R}^S$, $S \in \mathbb{N}$ is the dimension of the regime space $\mathcal{C}$.

Intuitively, a working regime c gives information on the state of a set of modules in the device. For instance, let us consider a very simplified device with only three components: a screen, a CPU and a WiFi module; a working regime is $c_1 = \{$screen = ON; cpu = 300MhZ; WiFi = OFF$\}$, while another one is $c_2 = \{$screen = ON; cpu = 200MhZ; WiFi = ON$\}$.

All the other information is here considered as the exogenous **input** of the dynamic process. Thus the dataset Z_N can be seen as:

$$Z_N = \{z_t = (c(t), u(t), y(t))\}_{t \in T}$$

where c(t) is the working regime at time instant t and $u(t) = (u_1(t), \ldots, u_H(t)) \in \mathcal{U} \subset \mathbb{R}^H$, being $H \in \mathbb{N}$ the dimension of the input space $\mathcal{U}$. We would like to remark that by relying on a data-driven methodology, we are able to model only behaviors that has been observed in the past and explain only dynamics in the input we observe in past. From now on, for sake of concision, we will identify the set of working regimes $\mathcal{C}$ with the set of those

working regimes which has been experienced in the past history of the device and the set of the input $\mathcal{U}$ as the set of those input which has been experienced in the past. At last, for modeling purposes, we consider that mild assumption that the input u generated by the real usage of the device are persistently exciting.

2.4 The proposed methodology

In this section, we formalize the process of building a power model for mobile device based on data Z_N provided by single device. A twofold goal is here required: estimate the model for each dynamic process $\mathcal{P}_i$ corresponding to a working regime c and estimate the transition model of working regimes $c \in \mathcal{C}$. While the former model is taking into account the power consumption of a specific hardware status of the device, the latter is able to provide an approximation of the user behavior, since each change in the working regime corresponds to an action of the user or the software.

The high level description of the procedure for the power model estimation is provided in Algorithm 1. The process starts by dividing data corresponding to each regime c, after that a *Multiple-Input-Single-Output (MISO) linear time-invariant model* $\mathcal{M}_c$ is estimated for each working regime and added to the overall model $\mathbf{M}$. This way it is possible to approximate the energy behavior of the device. Since in the simulation of the model $\mathcal{M}_c$ we are not provided with the values of the input $u(t)$, we also compute an estimated value for the input $\bar{u}_c$ in working regime c, which will be used to approximate the real input. Once a model $\mathcal{M}_c$ has been estimated for each working regime, we provide a model for how the working regimes are changing over time by means of a *HMM* $\mathcal{H}$ regulating the transition among working regimes $c \in \mathcal{C}$.

Algorithm 1 High level algorithm

Input: Dataset Z_N;
Output: Hardware model , Transition model Dataset Z_N;
Set $\mathbf{M} = \emptyset$;
for regime $c \in \mathcal{C}$ **do**
  Extract data Z_c corresponding to c from Z_N;
  Compute a MISO model $\mathcal{M}_c$;
  Compute the average value for the input $\bar{u}_c$ from Z_c;
  Add the new model $\mathcal{M}_c$ to $\mathbf{M}$;
end for
Compute the transition model $\mathcal{H}$ among working regimes $c \in \mathcal{C}$ by relying on data Z_N;

2.4.1 Hardware model estimation

Here we consider the indexed set of the working regimes $\mathcal{C} = \{c_1, \ldots, c_{|\mathcal{C}|}\}$ as the different modes $\mathcal{P}_{c_1}, \ldots, \mathcal{P}_{c_{|\mathcal{C}|}}$ of a single process $\mathcal{P}$. The decomposition of the process $\mathcal{P}$ is here also motivated by the fact that we are willing to remove, as much as possible, any source of unnecessary non-linearity from the estimation procedure. Given a fixed working regime $\bar{c} \in \mathcal{C}$, considered model $\mathcal{M}_c$ estimation either MISO linear time-invariant predictive models [105], Extreme Learning Machines [82], Reservoir Computing Networks [87]. For sake of simplicity, we describe the procedure with the prediction function of an Autoregressive with Exogenous variables (ARX) model [105]:

$$\hat{y}(t|\alpha_{\bar{c}}) = \alpha_{\bar{c}}^T(y(t-1), \ldots, y(t-na), \phi_1(u(t-nk)), \ldots, \phi_{nb}(u(t-nk-nb+1))),$$

where $na \in \mathbb{N}$ and $nb \in \mathbb{N}$ are the orders for the autoregressive and exogenous part of the prediction function, respectively, $nk \in \mathbb{N}$ is the delay order, $\phi_i : \mathbb{R} \to \mathbb{R}$ are base function on input $u(t)$, $\alpha_{\bar{c}} \in \mathbb{R}^p$ is the parameter vector estimated from the data and $p = na + nb \cdot H$ is the dimension of the parameter vector. The introduction of the base function $\phi_i(\cdot)$ allows to manage the residual non-linearity present in each process $\mathcal{P}_{\bar{c}}$. We would like to point out that, despite this model is nonlinear w.r.t the input $u(t)$, it is linear w.r.t. the parameters $\alpha_{\bar{c}}$. This modeling choice is due to theoretical results which allows to characterize the distribution of the parameter vectors, thus providing a tool for model comparison both among different working regimes and different devices. More specifically, by relying on the system identification theory [105] we are able to say that, if we consider equally long batches Z_M to compute an estimate $\tilde{\alpha}_{\bar{c}}$ of the optimal parameter vector α^o in the family of the approximating functions, we have that asymptotically (when $M \to \infty$):

$$\sqrt{M}\Sigma_M^{-1}(\tilde{\alpha}_{\bar{c}} - \alpha^o) \sim \mathcal{N}(0, I_p) \tag{2.1}$$

where Σ_M is a properly defined covariance matrix and I_p is the identity matrix with order p. Since the parameter vector distribution is known, we could perform statistical tests (in the specific multivariate normal distribution hypothesis tests [117]) to compare different regimes and different mobile devices.

We now want to focus on the problem of estimation of working regime c parameter vector $\tilde{\alpha}_{\bar{c}}$. A closed form solution to this problem is provided by the Least Square method which, given a dataset $Z_M^{\bar{c}} = \{(u, y) \in Z_N, z = (c, u, y) \text{ s.t. } c = \bar{c}\}, M \in \mathbb{N}$, provides an estimation of the optimal (i.e., the

one minimizing the mean square error) parameter vector as:

$$\tilde{\alpha}_{\bar{c}} = \left(\Phi^T \Phi\right)^{-1} \Phi y$$

where j-th row of the matrix Φ is $\Phi_j = (y(j-1), \ldots, y(j-na), \phi_1(u(j-nk))\phi_{nb}(u(j-nk-nb+1))), j \in \{t_1, \ldots, t_M\}$, $y = (y(t_1), \ldots, y(t_M))$ is the battery level and $\{t_1, \ldots, t_M\}$ are the time instants corresponding to samples in $Z_{M,\bar{c}}$ defined as a sequence of M consecutive samples from Z_N where we have the same working regime $\bar{c}$. The estimation of the matrix Φ would then require the knowledge of M consecutive measurements, which is hardly possible due to the non-consecutiveness of the data retrieved by the logging system (this problem is better described in Section 2.5). However, once the rows of the features matrix Φ_j are computed from a sequence of data $Z_L \subset Z_M$, with $L \geq \max(na, nb + nk)$, we may sample randomly M of those rows to built the features matrix Φ. This procedure overcomes the problem of missing data, since we only need batches of $L \ll M$ consecutive data to build Φ_j. Indeed, by relying on the aforementioned procedure, we are able to sample only well conditioned features matrix Φ, i.e., not containing highly correlated input. [4] The adopted procedure allows to obtain a set of parameter vectors $\{\tilde{\alpha}_{\bar{c}}^{(1)}, \ldots, \tilde{\alpha}_{\bar{c}}^{(Q_{\bar{c}})}\}$, each of which has been estimated on a M independent samples, thus following the distribution in Equation (2.1). Finally, the parameters we use for estimation are computed by taking the average over each parameter vectors set, i.e., for each working regime c we have:

$$\bar{\alpha}_c = \frac{1}{Q_{\bar{c}}} \sum_{q=1}^{Q_{\bar{c}}} \tilde{\alpha}_{\bar{c}}^{(q)}.$$

2.4.2 User model estimation

Once a model is estimated for each working regime $c \in \mathcal{C}$, the transition occurring during the daytime among different working regimes is here modeled by means of a HMM modeling approach. A HMM is a statistical model represented by a tuple (S, A, λ), where $S = s_1, \ldots, s_{|\mathcal{C}|}$ is the set of the possible states, A is the transition matrix which specifies the probability to move from one state to another, given the current state, i.e., $[A]_{ij} = a_{ij} = P(s_j|s_i)$ and λ is the initial probability distribution over the states set S. In our case, each state corresponds to a working regime $s_i = c_i, c_i \in \mathcal{C}$ and the initial probability distribution is deterministic, since

[4] Other viable solutions could consider techniques based on Tikhonov regularization [147].

we know the working regime c_{init} of the device at the beginning of each prediction period.

To estimate the transition matrix A, we consider data from Z_N and we estimate the transition by computing:

$$n_{ij} = |\{t \text{ s.t. } (c_i, u(t), y(t)) \in Z_N \wedge (c_j, u(t+1), y(t+1)) \in Z_N\}|$$

i.e., the number of times the working regime changes from c_i to c_j. The estimated transition probabilities are:

$$a_{ij} = \frac{n_{ij}}{\sum_{i=1}^{|\mathcal{C}|} n_{ij}}.$$

Thanks to the estimated HMM, we are able to simulate a sequence of working regimes $G = (c_{init}, c_{i_1}, \ldots)$ of arbitrary length. The presented model is valid as long as the Markovian assumption holds, i.e., if the transition model is time-invariant (see [25] for more details). If it does not hold, we should analyze the longest period where it holds and train a HMM for each of these time periods, for instance by computing the transition matrix over different time period and by checking their similarity [5]

2.4.3 Discharge curves/traces prediction

After the estimation process has been performed, the model for the hardware $\mathbf{M}$, the user behavior model $\mathcal{H}$ and the mean inputs $\bar{u}_c$ are available to perform a prediction of the discharge process on the device. We are then able to simulate the entire discharge curve of a mobile device, given an initial working regime c_{init} and an initial battery level y_{init}.

Once the parameter is computed, if we consider a working regime c we may predict the battery level at a time instant t by starting from y_{init} and iteratively using:

$$\hat{y}(t|\alpha_{\bar{c}}) = \alpha_{\bar{c}}^T(y(t-1), \ldots, y(t-na), \phi_1(\bar{u}), \ldots, \phi_{nb}(\bar{u}))),$$

by resorting to the computed average input $\bar{u}$. However, this procedure does not incorporate a mechanism to switch from a working regime to another. In the following sections, the previously developed methodology is extended to consider also this possibility.

At last, by relying on the estimated set of models $\mathbf{M}$, we are able to simulate the system in the state corresponding to parameter α_{c_i} according to the sequence G. We will stop simulating the system as soon as the battery level reaches 0, i.e., the mobile device is completely discharged. This

[5] The experimental analysis of the Markovian assumption on real data will be provided in Section 2.6.

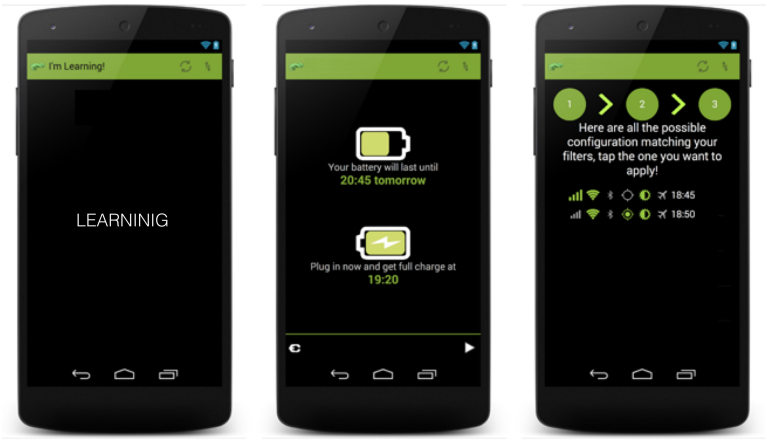


Figure 2.2: *Mobile application different screenshots: (A) learning stage, (B) TTL prediction and (C) device state constraint selection.*

methodology provides a prediction of the battery level at each time instant and, as a byproduct, it also gives an estimation of the TTL of the device too.

2.5 Implementation

Two main phases are needed to generate the power models defined in this chapter: a *observe phase* and a *learn phase*. These have been discussed in Section 1.2, highlighting their requirements and trade-offs. This section discusses how they have been implemented in the current use case scenario.

To achieve high adaptability and minimum impact to the usage context during the data gathering phase, I implemented a client application that logs information about the Android device it is installed on, to which we refer as the **mobile app**, optimized to save as much energy as possible but still able to adapt to different working conditions. I then decided to perform the computation phase on a **remote server**, in order not to waste energy on the mobile device itself: this is a common pattern to adopt while designing distributed systems composed of energy-constrained nodes.

The interaction flow between the mobile app and the remote server is briefly described as follows. After the identification of the user and the device by means of a *pseudo-authentication* [119] mechanism (based on the

Google OAuth2 protocol), the mobile application starts collecting and storing data about the device's energy consumption (e.g., battery level) and its internal status (e.g., CPU frequency, screen brightness and so on). Once a certain amount of data has been locally collected, a log file is compressed and encrypted[6] before being sent to the remote server. As soon as enough information has been gathered (usually 2-4 days of continuous usage of the device), the model computation process is performed on the server by means of the statistical learning approach discussed in Section 2.4. As far as no power model is available (the application is in learning phase), a message inviting the user to wait is shown (Figure 2.2.A). As soon as the power model has been computed, the mobile app retrieves it from the remote server and the estimated TTL for the current working regime is finally provided to the user (Figure 2.2.B), which may eventually choose a different device working regime to achieve the corresponding TTL (Figure 2.2.C). In order to reach a huge ecosystem of different mobile devices, the proposed application has been developed in Java using the standard Android SDK and it is currently available for free on the Google Play Store: it can then be easily installed (i.e., no device rooting is required) and it is compatible with almost any Android device.

Given the context of this work, energy efficiency is of course the requirement that constrained the most the design and the implementation of the application. On the one hand, I wanted to introduce no power consumption related to data transmission: information is sent only when the device is plugged into an external power source and connected to a WiFi network, not to consume battery power and to avoid unnecessary 3G/4G data transfer too. On the other hand, no computation is performed on the device: it actually retrieves from the server a lookup table containing the precomputed TTLs for each battery level (from 0% to 100%) and for each observed working regime, produced by means of the power model estimated for the specific device. This technique is energy efficient, since providing the current battery TTL boils down to a simple query on the lookup table instead of requiring a simulation of the power model.

Then, the same data collecting phase had to be the most unobtrusive as possible from an energy point of view, but custom and hardware-specific solutions are in contrast with the high compatibility requirement stated before. Things are made difficult by the fact that modern mobile devices have several hardware sensors which can be used to get information about their internal and external environment. Generally, the operating system implements an Hardware Abstract Layer (HAL) for every hardware platform it

[6]The encryption is performed using a *secret token* exchanged during the first authentication

Table 2.1: *Data gathered from the device.*

Screen	**Battery**	**CPU(s)**	**Mobile**
is on	on charge	max freq.	state
brightness mode	temperature	min freq.	activity
brightness value	voltage	current freq.	net type
width	percentage	max scaling freq.	signal strength
height	technology	min scaling freq.	tx bytes
refresh rate	health	governor	rx bytes
orientation		usage	call state
		cpu id	airplane mode
WiFi	**Audio**	**Bluetooth**	**GPS**
is on	music active	is on	is on
is connected	speaker on	state	status
signal strength	music volume		
link speed	ring volume		

can run on. Even if the Android OS already provides a HAL and some best practice to access power related information [65], the implementation of the proposed logging infrastructure is not trivial, given the high hardware and software fragmentation of the Android ecosystem [70, 71]. In fact, different devices and even different OS versions have partially different Hardware Abstract Layers (HALs) and APIs. Moreover, each hardware information has to be retrieved sparingly, since each reading operation consumes additional energy [167]. In such a context, I developed an highly optimized *Sense Library*, i.e., a set of Java classes meant to provide a single access point to the device hardware information. The *Sense Library* encapsulates all the logic needed to support different Android versions, following the Android best practices for an effective, efficient and maintainable source code. To save as much energy as possible, these libraries implement a *cache layer* whenever possible: they record the status of the underlying hardware and update it only when it changes, providing a "cached" value to the caller (i.e., the logger daemon) instead of continuously polling the state of the device to gather state information. The cached value is updated with an asynchronous approach by means of *Android Intents*, *Intent Filters* and *BroadcastReceivers*. The hardware features that can be accessed by means of the *Sense Library* are: screen, battery, CPUs, Audio, mobile connection (3G/4G), Wi-Fi, Bluetooth and GPS connections. Table 2.1 lists all the data gathered for each device component. The source code of this library is released under the GNU LGPL license: it can be downloaded from [122], while documentation and an extended description is available on [123].

The sampling frequency of the monitoring activity is then a crucial as-

pect from an energetic point of view. On the one hand, the higher is the frequency, the more accurate is the information about the device status and its battery discharge curve estimation, thus providing uniform information to increase the effectiveness of the computed power models. On the other hand, a high sample frequency would impact on the device battery life because of the power consumed to access sensors' data [167], even if this effect is mitigated by the aforementioned *Sense Library*. I then developed an Android Service, called *LogService*, which automatically starts as soon as the device is turned on and retrieves the status of the device through the aforementioned Sense Libraries. To provide a good adaptability to different working conditions, it is meant to be restarted if killed: this may happen as soon as Android needs to free resources for applications with a higher priority [111]. The logged data are stored in a plain text file using the JSON representation [7], which provides enough flexibility to support devices with different hardware features (e.g., an higher number of CPU cores).

Following the idea presented in [164], I decided to log the device status every 10 seconds. This is an upper bound from the power model perspective since choosing higher time intervals might cause a loss of information about system dynamics (e.g., the user turning on the screen for a limited amount of time, a small data transfer operation). More details about this choice are presented in Section 2.6.

Another notable design choice regards the use of *wakelocks*: on the one hand, this software mechanism prevents Android from killing our service during the logging period, while, on the other hand, such a solution influences the system behavior, i.e., it forces the operating system to stay awake until the lock has been released, thus potentially implying an unnecessary energy consumption. Therefore, we avoided the acquisition of a wakelock except for the exact logging instant: it is acquired and quickly released as soon as the atomic operation is completed. As a consequence, we allow the Android system to fall into a *deep sleep*: in such an operating mode, all the running services are suspended, thus the mobile app does not log any data. This is the reason why we have discontinuous registrations of the system behavior, since the mobile app is resumed only when the system leaves this low power mode: again, the energy efficiency and adaptability requirements lead me to deal with the missing data problem in the power modeling phase.

[7] The JSON structure of the log file is available on [123]

2.6 Experimental results

This section analyzes if the requirements defined in Section 1.2 are met by the proposed methodology and its implementation, for what concerns the current use case scenario: on the one hand, the analysis performed on the acquired data must produce *precise* models of the system's behavior, while, on the other hand, the *overhead* introduced on the mobile devices should be as low as possible.

2.6.1 Power model estimation

Experimental Setting D1

Since the goal of the proposed system is to build precise power models for real-world mobile devices, this section wants to provide evidence on how every specific model $\mathbf{M}$ produced is effective in predicting the specific device's energy behavior. Data has been collected from about 600 devices which are continuously sending data to our server: this set is composed of more than 30 different devices models, which are mainly Samsung, HTC and LG. The 75% of the data were used for training, while the remaining was saved for validation purposes.

The linear time-invariant ARX model family has been used to satisfy the assumption required by the methodology described in Section 2.4. The ARX orders and the input variables have been selected by basing on the validation error and exploring models with $n_a = \{0, \ldots, 3\}$ and $n_b = \{1, \ldots, 5\}$, using as input variables $u(t)$ CPU frequency and screen brightness (25% of the training set was used for model selection). After this phase, batches of $M = 100$ samples (selected as described in Section 2.4) were extracted and used to compute the parameter vectors α_cs. For each training batch, the parameter vector has been computed through a *non-positive least square* algorithm[8], which guarantees to reach the least square solution under the constraint of having only negative parameters. A model with only negative coefficients has been chosen since, within this context, if a variable increases its value we expect an higher discharge rate, e.g. the more the screen brightness is high, the more the battery level will drain. At last, in a real scenario, no information about the future value of the uncontrollable inputs u is available; hence, estimated average values are used for prediction purpose of future inputs.

Three different models have been considered for the HMM transition, by considering different assumption on the Markov property: this allowed

[8]This algorithm is a trivial modification of the non-negative least square presented in [100].

me to deal with potentially different behaviors (i.e., different transition matrices) of the same user in different moments of her daily routine. In the first *hourly model* HM we assume that the Markov assumption is true for each hour slot in the day (e.g., the user's behavior is constant, for instance, from 1.00pm to 1.59pm); we can then partition the data and estimate different transition matrices A_h for each hour of the day, i.e., $h \in \{1, \ldots, 24\}$). In the *single model* SM, I assume a single transition model A, thus assuming the Markov assumption holds for the whole dataset. At last, I considered a *dynamic model* DM, where we identify the longest period of the day where the estimated transition matrices are reasonably similar; the similarity is computed by considering a threshold on distance induced by the Frobenius norm computed on the difference of two matrices. The previously defined models are compared with the one without any transition M, i.e., where the working regime is considered constant and equal to c_{init} during the whole discharge.

The performance of the proposed methodology have been evaluated using the following figures of merit:

- the mean square error of the value of the battery level, i.e., $mse = \sum_{t \in T} \frac{|y(t) - \hat{y}(t|\bar{\alpha}_c)|}{|T|}$
- the mean square error on the slope of the discharge curve, i.e., $mpe = \sum_{t \in T} \frac{\tan|\hat{\gamma}(t) - \gamma(t)|}{|T|}$

where $\hat{\gamma}(t)$ is the slope of the discharge curve induced by the approximated model at time t, $\gamma(t)$ is the slope of the real discharge curve at time t and T is the set of time instants in which the prediction is performed. The average value $\bar{e}$ and the standard deviation s are provided over the possible realizations (since $\mathcal{H}$ is a probabilistic model) for each one of the figure of merit considered, as well as the percentage of improvement $I_{\%}$ w.r.t. the basic model M.

The performance have been also analyzed on different prediction times. I considered at first the performance on the totality of the testing set TOT, then I also consider the prediction of discharge curves from where the total discharge was between 30% and 50% ($30D$), between 50% and 80% ($50D$) and more than 80% ($80D$), to evaluate the capabilities of the considered methods with tasks with increasing difficulty (i.e., larger time windows).

Results for D1

The experimental results are provided in Table 2.2. It is possible to see that the introduction of a transition model is able to improve the performance of

Table 2.2: *Results for different models without transition model M, with hourly transition model HT, with single transition model SM and dynamically chosen one DM (average values are reported $\pm$ standard deviations).*

		M	SM		HM		DM	
			$I_{\%}$	$\bar{e} \pm s$	$I_{\%}$	$\bar{e} \pm s$	$I_{\%}$	$\bar{e} \pm s$
TOT	mse	11,1	4,4 %	$10,61 \pm 0,065$	3,42 %	$10,72 \pm 0,0075$	3,6 %	$10,7 \pm 0,014$
	mpe	8,1	4 %	$7,77 \pm 0,078$	0,86 %	$8,03 \pm 0,004$	0,74 %	$8,04 \pm 0,0047$
30D	mse	7,42	2,29 %	$7,25 \pm 0,023$	2,69 %	$7,22 \pm 0,0082$	2,69 %	$7,22 \pm 0,013$
	mpe	8,09	5,56 %	$7,64 \pm 0,086$	2,1 %	$7,92 \pm 0,0073$	1,97 %	$7,93 \pm 0,0066$
50D	mse	15,04	9,9 %	$13,55 \pm 0,24$	4,58 %	$14,35 \pm 0,021$	4,85 %	$14,31 \pm 0,021$
	mpe	8,31	4,33 %	$7,95 \pm 0,07$	1,08 %	$8,22 \pm 0,0049$	1,2 %	$8,21 \pm 0,0064$
80D	mse	19,2	13,75 %	$16,56 \pm 0,51$	4,375 %	$18,36 \pm 0,026$	5 %	$18,24 \pm 0,029$
	mpe	8	3,25 %	$7,74 \pm 0,05$	0,625 %	$7,95 \pm 0,0053$	0,875 %	$7,93 \pm 0,006$

the power estimation methodology, both in terms of mse and mpe. While the single model SM has the highest performance in terms of mpe, it has also standard deviations of the errors one order of magnitude greater than HM and DM, thus it may provide more variable results than the others. While for short discharge curves $30D$ we have slightly better results for the HM and DM in terms of mse, for the longer discharge curves $50D$ and $80D$ we have evidently better performance by the SM. This would suggest to use a DM in the case of a low values for the initial battery level y_{init} (favored on HM due to its potential lower complexity) and SM for longer discharge curves. Moreover, these results strengthen the idea that the Markov assumption is holding for the whole day in the setting we considered.

Experimental setting D2

In this second experimental setting, I want to compare the predictions provided by the proposed power modeling methodology with the ones provided by Android Lollipop [63].

Unfortunately, at the time of writing, no official Android API is provided to monitor the TTL predictions of Android Lollipop, strongly limiting this experimental setup. An additional mobile application has then been implemented to gather this information as follows: it records a screenshot of the display each 10 minutes, until the complete discharge of the battery; then, the obtained images are processed with an OCR application to extract the TTL information that Android Lollipop shows on the screen of the monitored devices. This operation has been performed on 4 different devices owned by 4 different users, for a total of 20 complete discharge curves produced as the sequences of predictions extracted by the OCR al-

Table 2.3: *Results given from the comparison the proposed system with different transition matrix schemes SM, HM, DM and Android Lollipop AL.*

	AL	SM		HM		DM	
		$I_{\%}$	$\bar{e} \pm s$	$I_{\%}$	$\bar{e} \pm s$	$I_{\%}$	$\bar{e} \pm s$
mse	118.59 ± 24.22	9.84 %	16.76 ± 10.76	9.89%	16.75 ± 10.69	9.86 %	16.75 ± 11.22
mpe	7.08 ± 4.89	15.81%	5.96 ± 1.96	16.80%	5.89 ± 1.85	19.49%	5.7 ± 2.08

gorithm. Again, this has been performed on devices in their real operating conditions.

Results on D2

Results for the experiments comparing the proposed methodology with Android Lollipop estimations are presented in Table 2.3. It is possible to see that all the proposed transition models present an improvement w.r.t. the estimations provided by Android Lollipop, both in terms of mse and mpe. While the improvement in terms of mse is almost the same over the different transition models, slightly better results can be achieved using multiple transitions models for different hours of the day if we consider the mse metric.

These results finally show how a data-driven power model is able to provide a better prediction of the energy behavior of a mobile device in its real usage context, thus validating the proposed power modeling methodology. Moreover, the model can be retrained and updated when its prediction error starts increasing and new data is available for the mobile device.

2.6.2 Energy consumption of the mobile application

In this last section, I want to show how the implemented power-aware logging application has a negligible impact on the device's power consumption. The procedure proposed here consists in comparing the average power consumption of the device in two different *testing conditions*, to see if any significant difference can be observed: on the one hand, I measured the power consumption of the device with a stock Android OS installation (i.e., without the proposed monitoring application), while, on the other hand, I repeated the same measurements with the logging application installed on the same device.

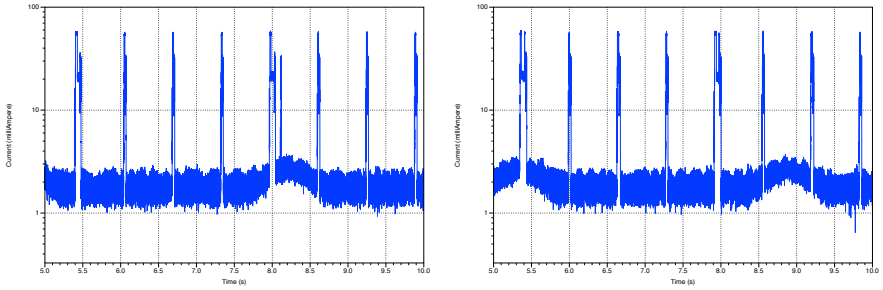
Experimental setting D3

To obtain comparable results, the environmental conditions should be maintained as much as possible unaltered during the experiment (e.g., in different locations the mobile network signal strength may be different, thus affecting the consumption of the mobile network hardware module), while the discharge curve of the device's battery has to be sampled with a reasonable accuracy. I used the Monsoon Power Monitor [83] and its related software, since it provides a robust and reliable solution to measure the current drained (mA) or the power consumed (W) by any device that can be powered by a 4.5 V battery (or less), with a maximum current of 3 A. However, I had to wire the Monsoon to the device, thus I was able to perform only in-lab experiments. The target device for these tests is a Samsung Galaxy SIII, since is the most common device in our dataset of real users provided by our logging application and, thus, is somehow representative of a wide range of devices.

For the whole duration of the tests, the device was left in an *idle configuration*: this represents the case in which the user is not interacting with her smartphone. Display is left off, like Wi-Fi, GPS and Bluetooth network interfaces, while the mobile data connection (over the 3G network) has been left enabled. This configuration has been chosen to reduce as much as possible any additional contribution to the power consumption caused by other applications. We then considered three different conditions: the first one is the idle configuration without the logging application installed ($IDLE$), the second with the logging application installed ($LOGGING10$) and a third condition with the logging application installed and a logging frequency of 1Hz ($LOGGING1$)[9]. In this set of experiments, we considered the third condition ($LOGGING1$) to explore the trade-off between information *accuracy* and *overhead* introduced, in terms of power consumption. The measurements have been performed with a sampling frequency of 5KHz, lasted 5 minutes, recording around a million and a half samples for each test and were repeated over 30 independent run for every testing condition. An example of the results of the measurements is presented in Figure 2.3.

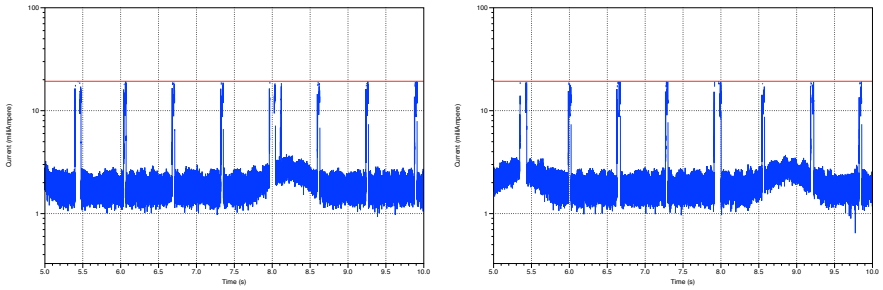
Even tough the most possible conservative settings have been considered, it is possible to notice some sporadic peaks around 50 mA and, in rare occasions, up to 300 mA, while normally the current drained was around 5 mA. The average number of peaks was approximately the same in all the

[9]Please note that the application released on the Google Play Store has a logging frequency of 1/10 Hz, i.e., the condition $LOGGING10$.



(a) *Idle* **(b)** *Logging*

Figure 2.3: *Sample of instantaneous power consumption with and without the logging application: original data.*



(a) *Idle* **(b)** *Logging*

Figure 2.4: *Sample of instantaneous power consumption with and without the logging application: filtered data. The red line represents the threshold corresponding to the 95th percentile of the idle data.*

Table 2.4: *Pvalues for the statistical tests between different configurations.*

	IDLE vs. MPOWER10	IDLE vs. MPOWER10
T_1	0.6264	0.0431
T_2	0.3132	0.0215

tests performed. We think that these spikes are relative to the stock Android OS, thus not dependent on the logging application. To mitigate the effect of these peaks we filtered data above the 95th percentile, measured on idle data. The result of this filtering is shown in Figure 2.4.

I finally computed the mean current drained for each run and I performed a statistical hypothesis test for two populations with same unknown variance [117] for two different hypothesis tests:

$$T_1 := \begin{cases} H_0: & \text{the mean power consumption is the same in the two testing conditions} \\ H_1: & \text{the mean power consumption is different in the two testing conditions} \end{cases}$$

$$T_2 := \begin{cases} H_0: & \text{the mean power consumption when the logger is installed is less or equal than the case in which it is not present} \\ H_1: & \text{the mean power consumption when the logger is installed is greater than the case in which it is not present} \end{cases}$$

The assumption that both populations related to tests with and without the logging application have same variance is here reasonable, as we are observing the same discharge phenomenon.

Results on D3

Results of the tests are presented in Table 2.4. These statistical tests showed no strong evidence (do not refuse H_0 with high confidence) of a greater power consumption due to the presence of our logger $LOGGING10$, thus proving the negligible impact of our application on the device's energy behavior. In the case we considered measurements coming from $LOGGING1$, the statistical tests showed strong evidence (accept H_1 with high confidence) that a higher sampling frequency would have affected the power consumption of the device.

2.7 Final remarks

This chapter presented a preliminary data-driven methodology for power modeling in the mobile devices' context, able to both estimate the hardware components power consumption and model the users' behavior. This is done starting from the measurements coming the device in its real usage context, to learn how its battery behaves under different conditions.

The *MPower* mobile application implements a low power logger (*monitoring* phase), offloading models computation to a remote server (*analysis* phase). Then, the application makes use of these power models to make predictions of the TTL (in terms of minutes, not just in term of remaining battery percentage) of the device under certain conditions: this is an important information for the user, as it allows her to choose how to use her device, with respect to her current and future needs (*plan* and *execute* phases). In this context, the final decision on how and when save battery life is left to the user, that becomes part of the OLDA loop: however, we can bring power-awareness one step further, as discussed in the next chapters.

CHAPTER 3

Generalization: Model and Analysis of Resource Consumption (MARC)

3.1 Introduction

The experience developed on power models for Android devices, described in Chapter 2, can be brought to a wider set of *power-constrained* systems. Even though the same Android OS can run on a plethora of different platforms and embedded devices, the proposed methodology can be extended to deal with a generic system, hereafter called *"agent"*, that wants to become *"power-aware"* or, in a wider sense, *"resource-aware"*: in fact, every agent that aims to be autonomous must be endowed with the necessary tools to measure and learn how its actions affect the surrounding environment and the resources available in the system.

Many recent works that exploit autonomous agents highlight how *resource modeling* is fundamental in many decision making processes, as resources are often the main constraint agents need to consider when evaluating adaptation strategies. For example, [50] and [103] succeeded in supporting adaptation inside a Cloud environment, exploiting the agent model; nevertheless they assume the models to be known a priori. [50]

identified this shortcoming and suggested a data-driven methodology, based on statistical models, to remove such a limitation. An even more explicit call for precise resource modeling methodologies is found in [76], where the authors assumes that a resource consumption model is available for every metric presented for characterizing and testing the elasticity of an Infrastructure-as-a-Service (IaaS) platform.

It is then clear how modeling techniques have to be put in place if we want the agents to be able to gaze beyond their sensors and to be *adaptive*, i.e., to become always better at reaching their goals by learning from past experience [107]. Unfortunately, data mining and machine learning techniques are usually complex and computationally intensive, thus making their integration with agents far from trivial.

Given the opportunities in the field, the main contributions of this chapter are twofolds: (1) I propose a general methodology to model resource consumption trends and (2) I implemented it into *MARC*, a Cloud-service platform that produces *Models-as-a-Service*, thus relieving self-aware agents from the burden of building their custom modeling framework.

After an extensive exploration of different types of resources and their consumption trends, I realized that the phenomenon itself can be described independently from the specific application fields. The proposed methodology then tackles the resource modeling problem relying on three fundamental pillars: (1) generality, (2) extensibility and (3) precision. As of (1), we want to avoid any field-specific assumption, apart from the ones that are intrinsic to resource consumption. Then, we need to define integration points where our methodology can be extended to include different modeling techniques (e.g., the ones proposed in recent works as [152] and [126]), thus ensuring (2). These first two goals usually increase the level of abstraction, paying wider applicability with longer computation time to obtain precise results; in order to guarantee a reasonable level of (3), the methodology can be tuned to produce coarse-grained results quickly or more precise results in a wider time span. In order to meet these three requirements, I tackle the modeling problem with the same solid mathematical techniques introduced in the previous chapter, i.e., ARX models, Discrete Markov Models (DMM) and Time Series analysis, specifically combined to produce a much more complex and detailed characterization of the modeled trend.

The methodology has then been implemented into *MARC*, a platform designed to Model and Analyze Resource Consumption trends (MARC). Any self-aware agent can simply send a raw dataset to this service, together with additional metadata on the system's working regime; *MARC* will then provide the agent with model's parameters estimated on the input dataset,

allowing it to have a better understanding on how its resources are depleted. This approach then perfectly fits the Learning stage of any OLDA loop, i.e., a runtime control loop that is emerging as the principal solution to agents' life cycle definition. For example, recent works as [21, 62, 168] would be able to remove many of their stricter assumptions by simply relying on *MARC*, without any severe modification.

The proposed platform has been developed with the scalability requirement in mind: models computation parallelism and distribution is backed by functional programming and the actor-based design of *MARC*. Moreover, a centralized approach to model generation allows knowledge sharing and reuse: in fact, when an agent requests a model for a certain phenomenon, the platform can take advantage of previously cached information that another agent of the same ecosystems may have requested previously. This implicitly increases agents cooperation: the more agents use our platform, the bigger this shared knowledge becomes.

All the *MARC* source code has been released open source [1] and it aims to be a reference for future work in the field of autonomous agents and self-aware systems.

This chapter is organized as follows: Section 3.2 introduces the resource consumption problem, giving a hint on how we want to tackle it; Section 3.3 gives a complete overview of the proposed methodology, from the high level flow to theoretical considerations; Section 3.4 presents the system design, together with the most relevant implementation details; results that validate the proposed methodology are then discussed in Section 3.5, while Section 3.6 points out future directions and opportunities.

3.2 The resource consumption problem

Resource consumption is the main focus of this chapter. We define a ***resource*** as an asset, either material or immaterial, subject to finite availability, that has to be exploited by a process to function effectively. It can be the energy stored in a battery, the capacity of a subway station, or even time: in all these cases, it is fundamental to notice how a resource is limited in availability.

The subject directly relating with a resource has been specialized into a ***process***, i.e., a systematic sequence of well-defined operations that are performed to transform resources into products. This very broad definition states that a process is just a systematic transformation of resources into

[1] Source code available at: https://bitbucket.org/paperblindauthor/2017-powermodels

some resulting outcome; indeed, the adjective "systematic" is crucial for our entire definition structure.

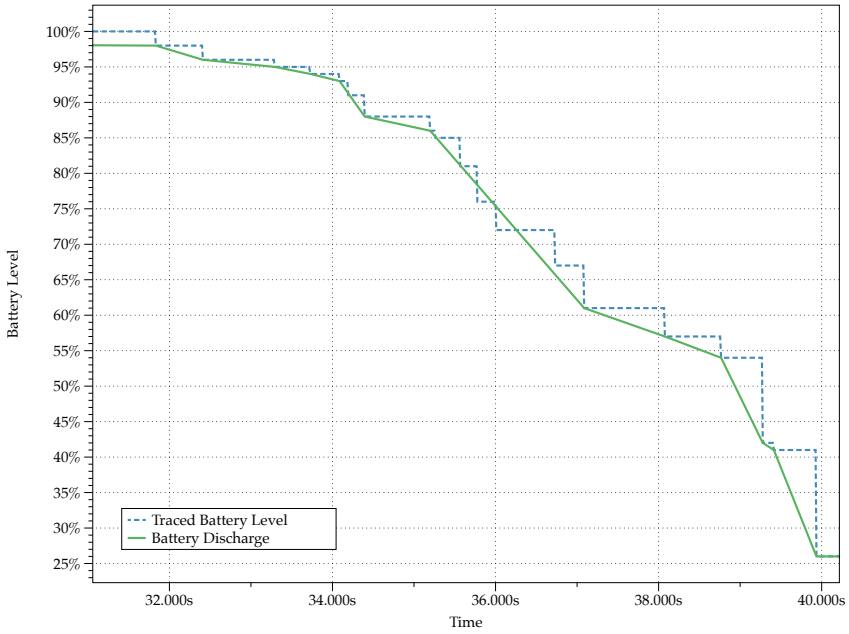
Apart from the process concept, that defines what is enacting on a resource, ***utilization*** is another critical term to be clarified within our specific context: it is the trend at which a process transforms a resource into a product. It represents the concept that bridges resources to their availability. Then, the generic concept of ***budget***, defined as the amount of resource that its owner is willing to make available to a specific process, helps us to specialize utilization into consumption. We thus define ***consumption*** as the trend at witch a process reduces its resources' budget to obtain a product.

Even though a plethora of highly specialized techniques have been proposed to ***model resource consumption*** in different specific fields of application, there is still a lack of generalization of these efforts towards a unified methodology. The goals of this chapter are then: (1) the study and the evaluation of an approach to resource consumption modeling, built on top of the best techniques validated in the state of the art, and (2) its implementation in a distributed, Cloud-ready platform to enable an *as-a-Service* interaction.

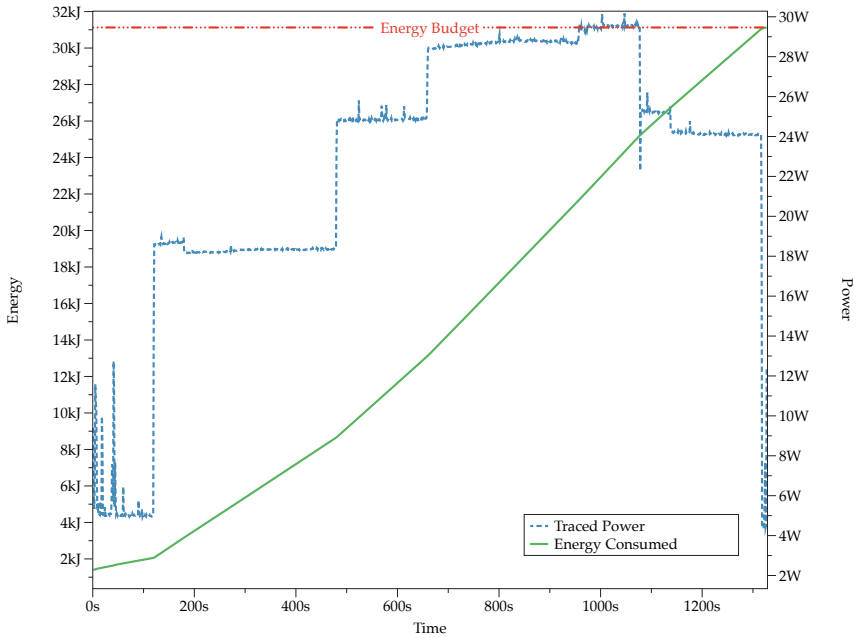
The problem of analyzing and modeling resource consumption is a composite one. In this section, we want to highlight the features that different consumption trends have in commons. Figure 3.1 presents two different use cases: on the one hand, Figure 3.1a shows the discharge of the battery of a smartphone, while, on the other hand, Figure 3.1b shows the energy budget allocated to a virtual machine over time. In the first case, we deal with a *naturally constrained* resource, i.e., the amount of energy that can be stored in the battery, while the second case consists in a *budgeted* resource, i.e., we have a total amount of energy that can be consumed in a certain time span.

Even if the whole consumption trend in both the plots in Figure 3.1 is far from being linear, there are wide sections in which the behaviour can be easily approximated using a linear trend. If we consider, for instance, the power plot (dashed blue) in Figure 3.2b, it is evident that the process can be subdivided in sub-processes having almost constant power consumption. This suggests to model each of these sections with a different consumption model.

Then, we can see how the process is switching from one linear section to another: as this happens, we experience a sudden slope change on the consumption trend (*e.g.*, see Figure 3.2b). We then need to devise a methodology to predict these events, in order to combine the different consumption models identified for each sub-process.

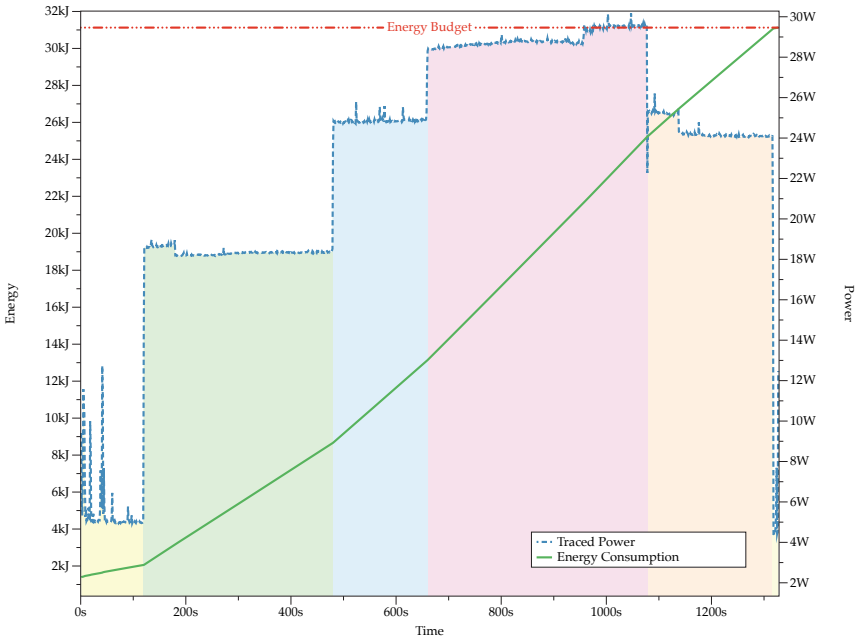


(a) *Discharge of a smartphone - Naturally constrained resource*

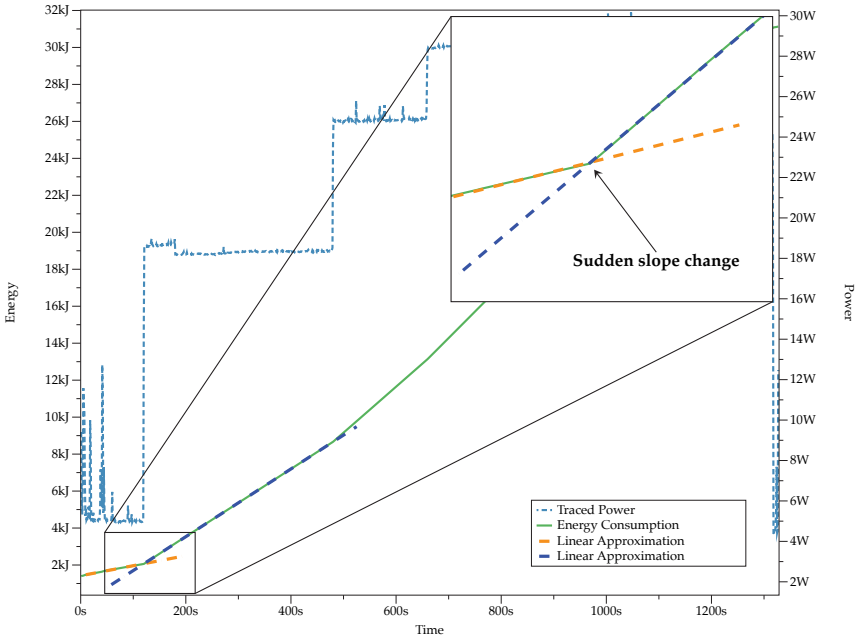


(b) *Energy consumption of a virtual machine - Budgeted resource*

Figure 3.1: *Different resource consumption behaviours*



(a) *Highlighted linear sections*



(b) *Highlighted sudden slope change*

Figure 3.2: *Piece-wise linearity of a generic resource consumption trend*

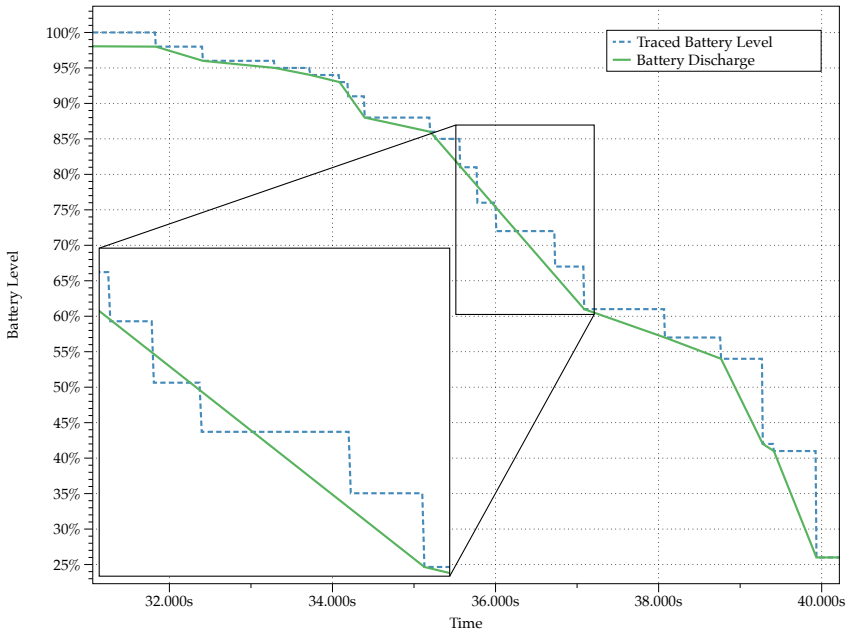


Figure 3.3: *Highlighted local deviation from linearity*

Finally, a local amount of non-linear behaviour still remains inside each model of the sub-process, as shown for instance in Figure 3.3. Even if these deviations cause a negligible error for an important part of consumption trends, this component cannot be simply ignored in some *"noisy"* cases. A methodology to identify and predict this residual non-linearity is then needed.

The combination of these three observations lead to the generation of a *piece-wise linear model* of the phenomenon: we observed that it is a behaviour that is shared by the vast majority of the engineering processes, which usually rely on different, well-defined, phases. A formal definition of the proposed approach, as well as some details on the system design and implementation, are provided in the next section.

3.3 Methodology generalization

The proposed methodology is essentially a non linear workflow from raw consumption traces to valuable resource consumption models, and represents a generalization of the approach proposed in Section 2.4.

Before describing this flow, it is important to stress the definition of some fundamental terms:

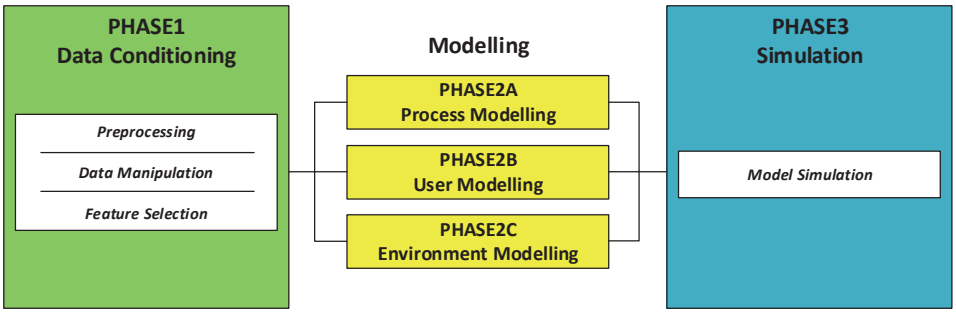


Figure 3.4: *MARC workflow block diagram*

Process: a systematic sequence of well-defined operations that are performed to transform resources into products.

User: short for *resource's user*, the entity – either a human or another agent – that is deciding on what and how the *process* implemented by the system has to produce, thus impacting on the resource consumption trend.

Environment: the group of conditions that influence both the *process* and the *user* but that can not be controlled by either of them.

Working regime: an identifiable working regime where either *process*, the *user* or the *environment* presents a set of steady properties characterizing its behaviour.

These terms will be referenced with a very context-specific meaning throughout the whole chapter.

Figure 3.4 lays out the high level structure of our methodology in three main phases. The first phase – `PHASE1`: **Data Conditioning** – is in charge of cleaning the raw traces in order to maximize the accuracy that the subsequent phases can achieve, e.g., by dealing with missing or corrupted samples, excluding non interesting features, and so on.

Once the data set has been properly conditioned, it is ready to go through the modeling processes required to extract knowledge from raw traces. These are gathered in the second *macro-phase* – `PHASE2`: **Modeling**. We consider this a *macro-phase* as it is composed of three independent modeling sub-phases, one for each aspects of a generic resource consumption trend: first of all, `PHASE2A` produces a set of models representing the characteristics that are intrinsic to the *process* itself, building a sort of *"fingerprint"* of the resource consumption trend in different *working regime*s, i.e., the different *sections with linear behaviour* identified in Section 3.2; then,

PHASE2B is in charge of modeling how the *user* behaves with respect to the resource under study by capturing how it acts on the controllable *knobs* of the *process*, causing those *sudden slope changes* identified in the previous section; finally, PHASE2C models the *environment* in which the *process* takes place, by capturing the *noise* it causes, Iidentified in Section 3.2.

The modeling workflow then concludes with a last phase – PHASE3: **Simulation** – that gathers all the models produced, combining them to estimate how they jointly contribute to the description of the resource consumption trend under study.

Each phase of the MARC methodology can be seen as a black-box with well-defined input and output interfaces. Every phase produces three outcomes: (1) a *result*, (2) a *mathematical performance report* and (3) a *computational performance report*. The first consists in the actual output expected by the phase, e.g., the whole cleaned data set, the slopes of the sections with linear behaviour, and so on. Then, the second comprises additional metrics that are useful to understand the quality of the *result* (e.g., the number of samples discarded or edited by the cleaning process, the modeling errors, etc.). Finally, the last one gives a feedback on the computational effort required by the phase. It is easy to see how both (2) and (3) are interesting to understand and balance the trade-off between precision and timeliness of (1).

The following sub-sections provide some details on each different phase, explaining how the three pillars of *generality*, *extensibility* and *precision*, introduced in Section 3.1, are met.

3.3.1 PHASE1: Data conditioning

The **Data Conditioning** phase is the one that mostly influences in practice the quality of the results produced by the entire methodology. In fact, it conducts all the tasks that are preparatory for the *data mining* steps – viz., selection, preprocessing and transformation. As widely shown in the state of the art, these tasks are fundamental when dealing with real-world data that may be noisy, inconsistent and incomplete.

Input The input of this phase is the raw data set, defined as the $s \times f$ matrix

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1f} \\ x_{21} & x_{22} & \dots & x_{2f} \\ \vdots & \vdots & \ddots & \vdots \\ x_{s1} & x_{s2} & \dots & x_{sf} \end{bmatrix} \tag{3.1}$$

where each row represents a **sample** $\boldsymbol{x_i}$, i.e., the representation of the state of the system in a certain time instant, while each column represents a **feature**, i.e., a single aspect of the system state; each cell is defined as **feature** f**'s value at time instant** t. It is fundamental that one feature represents time – feature f_t – and one feature represents the resource to be modeled – feature f_r.

Each feature f is defined by a set of properties:

- *Label*: it is a string used as unique identifier for the feature;
- *Type*: it describes how the feature values must be treated; a value can be:
  - *Instantaneous* if it represents a continue punctual state variable (*e.g.*, the mobile signal level or the electrical power absorbed by a server),
  - *Categorical* if it represents a variable with a finite countable set (*e.g.*, the type of network connection),
  - *Cumulative* if it represents a continue integral state variable (e.g., the bytes transferred over the network since system boot or the electrical energy absorbed by a server);
- *Monotony*: it describes whether the feature has a predefined monotony (either ascendant or descendant) or not;
- *Bounds*: the lower and the upper bounds of the domain of the feature (for non-categorical features, they can be either real values or $\pm\infty$).

The main goal of this phase is to condition the input data set in order to obtain a cleaner one that can be suitable for modeling purposes. Since cleaning requirements can be tightly related to each specific application model, the aim of this phase is to provide set of highly configurable and optional steps that can be selectively applied depending on the specific use case. These steps can be grouped in three subsequent tasks, divided by the scope on which each one enacts its conditioning operations:

A. **Preprocessing**: cleaning tasks at sample-level scope;

B. **Data Manipulation**: highly specialized tasks at dataset-level scope;

C. **Feature Selection**: synthesis and exclusion at feature-level scope.

On the one hand, the most common **preprocessing** tasks we identified and implemented revolves around coherency correction and incoherent sample elimination (e.g., to filter out out-of-bounds values), as well as

domain granularity reduction, to reduce the number of values a feature can take. These preprocessing rules are determined manually, once per application domain: for instance embedded devices' data will probably need a different treatment with respect to servers' traces.

On the other hand, all the operations that require a full data set scope are defined as **data manipulation** tasks. The definition of these tasks is very broad and open to future extensions, as it is deeply related to each single specific application field.

Finally, **feature selection** operations are meant to edit and choose the features that better represent the system's state.

As introduced in Section 3.2 different consumption slopes are related to different phases of the *process* we want to model. At first, we need to understand which are the *"clusters"* in which these slopes can be efficiently grouped: in order to accomplish this, we use a mono-dimensional interval generation algorithm, that may rely, for instance, on a Kernel Density Estimation (KDE) analysis [136], a non-parametric statistical method to estimate the probability density function of a sampled random variable. This algorithm takes the derivative of f_r as input – this represents all the local slopes for every sample – and returns a set of intervals grouping effectively the slopes in our data set. Each interval identified by this step represents a *working regime* of the *process* behaviour. A comprehensive example will be provided on a real use case in Section 5.3.

Then, we need to identify the features that better determine the current working regime: this can be accomplished exploiting a classification algorithm, such as *ReliefF* [98], a heuristic, distance measure based, supervised classification algorithm that presents a good trade-off between time complexity and accuracy [38]. Again, Section 5.3 will provide additional details on how to exploit this algorithm in a real context.

Once we know which are the features that can better identify different classes of slopes, we can generate synthetic features based on some combination of the ones originally present in the dataset: this operation is known as *features fusion*. Our methodology requires to specify a set:

$$F = \{f_f \mid f_f : \{\boldsymbol{x_i}\} \rightarrow \mathbb{R}\} \tag{3.2}$$

of *fusion rules*, that computes the value of the synthetic features for every sample. For each *fusion rule*, it is also necessary to define the characteristics of the resulting synthetic feature, exactly as the original ones.

Of course, it is also important to define a procedure that helps in reducing the number of features that will be crunched by the modeling phases.

This can be easily achieved exploiting a an algorithm like Principal Component Analysis (PCA), then excluding useless features from further analyses.

Before moving on to the modeling phases, it is important to understand what output this phase produces and highlight how its structure is compliant with our three main goals of generality, extensibility and precision.

Output The output of this phase is fairly simple to describe:

- **Result**: the result of this phase are the conditioned data set and the new set of features' definitions. The output data set can be defined as the $ss \times ff$ matrix

$$\hat{\boldsymbol{X}} = \begin{bmatrix} \hat{x}_{1,1} & \hat{x}_{1,2} & \dots & \hat{x}_{1,ff} \\ \hat{x}_{2,1} & \hat{x}_{2,2} & \dots & \hat{x}_{2,ff} \\ \vdots & \vdots & \ddots & \vdots \\ \hat{x}_{ss,1} & \hat{x}_{ss,2} & \dots & \hat{x}_{ss,ff} \end{bmatrix} \tag{3.3}$$

  where

  - ss is the number of samples that survived preprocessing and data manipulation, plus the ones potentially added by this last one;
    - rows represent all those **sample**s $\hat{\boldsymbol{x}}_{\boldsymbol{i}}$;
  - ff is the number of the features that survived the exclusion procedures, plus the synthetic ones added through the *fusion rules*;
    - columns represent all those **feature**s.

  It is of course fundamental that the features representing time – feature f_t – and the resource to be modeled – feature f_r have not been excluded by the feature selection tasks.

- **Mathematical Performance Report**: this report precisely accounts how many samples have been excluded due to incoherence and out-of-bound eliminations, as how many samples have been added, modified and removed by each data manipulation operation. This allows the user to quickly identify: on the one hand, highly corrupted unsatisfactory data sets and, on the other hand, if rules or operations are too restrictive for the case study.

3.3.2 `PHASE2`: Modeling

The **Modeling** macro-phase is the core component of the entire *MARC* methodology, and it revolves around the concept of ***working regime***, i.e.,

an operational state of the modeled system identified by a set of steady properties characterizing its behaviour.

Along with the working regime concept, we need to categorize each different input feature as follows:

***Working regime* features**: the working regime features f_c are a subset of the data set features that, for each combination of their values, identifies a different working regime.

Exogenous features: all the data set features f_{ex} apart from: the time feature f_t , the output feature f_r and all the working regime features f_c.

Controllable features: all the exogenous features f_{ctrl} that reflect direct actions performed by the user.

Uncontrollable features: All the exogenous features f_{uc} that are not controllable features f_{ctrl}.

The next sub-sections discuss how these concepts are used to model the piece-wise linear behaviour we are considering in all its three contributions, i.e., the process, the user and the environment.

`PHASE2A`: *Process* Modelling

Within this phase, a single model is produced for every working regime of the system and no remarkable distinction is sought within the same working regime.

Input The input of this phase is the cleaned data set $\hat{\boldsymbol{X}}$ coming from `PHASE1` (Section 3.3.1). The data set is time sorted and split into *batches* – one for each joint value of the working regime features f_{c_A}; the f_{c_A} features are then removed from every *batch*, as they are just used to identify the current working regime.

We decided to exploit a *regression modeling* technique as it is widely used in many different works in multiple application fields [12, 23, 90, 145, 149, 150, 153] due to its generality and simplicity. In fact, it is capable of capturing many different real-world behaviours and processes where the output of a system has a clear dependency on some observable inputs (or even the previous output itself). Moreover, regression models gained a huge importance in the last decades as they offer the possibility to derive a very simple Maximum Likelihood Estimator for its parameters vector [26], allowing to easily derive the regression model directly from sampled inputs.

For each batch, an ARX model is computed with the domain-specific *auto-regressive* (ar) and *exogenous* (ex) lags configured: a *lag* is the length of the time window in which we seek for correlation.

Output The goal of this phase is to provide a description of the process characteristics.

- **Result**: for each batch, the result reports the ARX models' parameters (i.e., the "alphas") for the f_r output feature from regression lag 1 to ar, and for the n f_{uc}s exogenous variables from regressive lag 0 to ex. These alphas can be used to estimate at runtime the value of the f_r output feature, given the recent past values of f_r and the present and recent past values of all the f_{uc}s, by computing the formula:

$$\begin{aligned} f_r(t) = \alpha_{r,1} f_r(t-1) \ + \ \ldots \ + \ \alpha_{r,ar} f_r(t-ar) + \\ + \ \alpha_{uc,1,0} f_{uc,1}(t) \ + \ \ldots \ + \ \alpha_{uc,1,ex} f_{uc,1}(t-ex) + \\ + \ \ldots \ + \\ + \ \alpha_{uc,n,0} f_{uc,n}(t) \ + \ \ldots \ + \ \alpha_{uc,n,ex} f_{uc,n}(t-ex). \end{aligned} \tag{3.4}$$

- **Mathematical Performance Report**: for each batch, this report accounts for the Mean Square Error (MSE) obtained computing the respective ARX model , as well as its square root, known as Root Mean Square Error (RMSE), a reliable measure that indicates the mean deviation by excess or defect that the output value estimated by the model has with respect to the real output value.

`PHASE2B`: *User* Modeling

This phase aims at producing a model able to predict *sudden slope changes*. In this context, we define the user as the entity – either a human or another agent – that is deciding when and how the process makes a transition between one working regime and another.

Input This phase receives a slightly modified version of the clean data set $\hat{\boldsymbol{X}}$ produced by `PHASE1`, containing only the time feature f_t and the working regime features $f_{cfg,B}$: in fact, only controllable features f_{ctrl} can be manipulated by the user.

This phase relies on a theoretical model called DMM [25]. A DMM can be represented as the tuple $< S,\ A,\ \lambda >$, where

- S represents the **set** of all the possible states of the model; in our case, a state is direct synonym of working regime;

- A is the **adjacency matrix**, representing the probability for the user to change from one working regime to another; this probability is estimated using

$$P(A \rightarrow B) = \frac{\#\{t \in T \mid t = A \rightarrow B\}}{\#T} \tag{3.5}$$

  where $A \rightarrow B$ is the transition from working regime A to B and T represents the trace's transitions set, i.e., the ordered set of transitions obtained comparing two subsequent samples in $\hat{\boldsymbol{X}}$;

- λ is the **initial state** from which the chain has to be built.

In order to build our DMM, we divide the time sorted input into **periods**, i.e., time spans that show the same repetitive user behaviour. Every period is then sliced in $b > 0$ time bands, where b is a domain-specific parameter that can be tuned accordingly with the user's speed. Data of every corresponding band are joined, obtaining b batches. For each batch, all the observed working regimes are considered as states of a DMM and the whole batch data set is used to estimate the probability of jumping from one state to another, thus determining the DMM adjacency matrix.

Then, the following step aims at reducing b, i.e., the number of DMMs obtained, to the minimum number possible without decreasing the precision of the whole model. This is done by comparing neighbouring bands, accounting the similarities among their respective adjacency matrices. If their *distance*[2] is below a defined threshold, the bands are merged (*cliqued*) and a new adjacency matrix is computed. This is repeated until no neighbouring bands result to be similar enough to trigger another merge.

Output The output of this phase is a model describing how the user changes its behaviour during a repetitive period in time

- **Result**: for each time band, the initial and final time of the band (relative to the starting point within the period) and the most probable Markov Chain starting from every traced working regime.
- **Mathematical Performance Report**: this report includes the distances between neighbouring time bands and the cliques tree (i.e., a tree representing how the initial b bands have been recursively *cliqued*); these are both necessary for tuning the b number of bands and the threshold for triggering similarity *cliquing*.

[2]Many definitions of distance between matrices can be given, in this case we are referring to a Mahalanobis distance [108]

`PHASE2C`: *Environment* Modeling

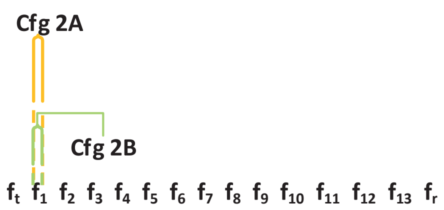
With this phase, we want to produce a model able to predict the *"noise"* from the system's environment that causes local non-linear behaviours, as discussed in Section 3.2. The environment is usually not well represented by the features of the data set; we split it in two common use cases:

- **Steady *Environment*:** the environment surrounding the system is steady, i.e., it has a single pattern that introduces *"noise"* in the system; in this trivial case, it is usually sufficient to define an empty working regime, assuming that a *catch-all model* can well represent the situation. Although this approach is suitable for simple "noise" patterns, a non-empty working regime is still needed if the user influences the environment, altering its behaviour; in this case, the `PHASE2C` working regimes will be a subset of the `PHASE2B` or `PHASE2A` working regimes.
- **Dynamic *Environment*:** in this situation, the environment surrounding the system is dynamic, i.e., it has variable *"noise"* patterns that emerge in different situations; this is the most complex case and usually requires some sort of extra instrumentation sampling additional features to identify the different contexts in which the system is immersed into.

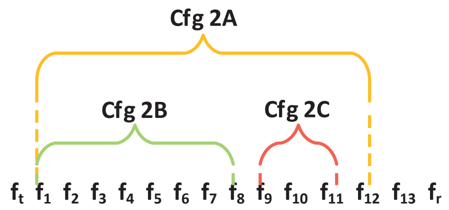
Input As the others, this phase receives a slightly modified version of the clean data set $\hat{\boldsymbol{X}}$ produced by `PHASE1`. The input of this phase is again batched according to the defined working regime (similarly to what has been described for `PHASE2A`). Moreover, it only contains all the exogenous features f_{ex} that appear in `PHASE2A` model: as we exclude the controllable features f_{ctrl}, we are explicitly neglecting the impact of the environment on the user. This is reasonable given the definition we gave for the user model: in fact, the user model implicitly accounts also for the consequences the environment causes on the user itself.

The goal of this phase is to produce a *signal generator* for every exogenous feature f_{ex}: these generators should be able to mimic the trends of the real environment and consist in software modules that encapsulates a model of the environmental *"noise"*. For example, the simplest model of a *"noise"* is a constant signal generator that emits $\overline{f}_{ex}$, the mean value of f_{ex} over the current batch.

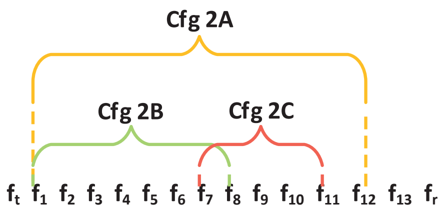
Many types of signal generators can be proposed, e.g., exploiting the representation of the f_{ex} empirical distribution or even resorting to more advanced time series analysis techniques. However, we experimentally evalu-



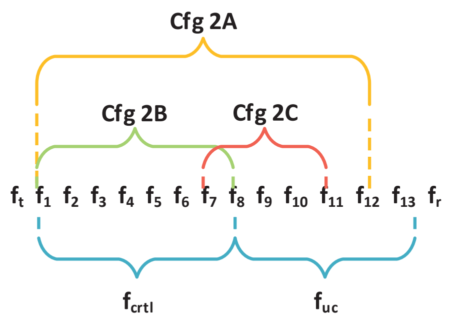
(a) *Minimal* **working regime**



(b) *Disjoint* **working regime**



(c) *Interdependant* **working regime**



(d) *Suggested* **working regime**

Figure 3.5: *All the different possible organization of* PHASE2 *phases' working regimes*

ated that even a simple constant signal generator is sufficient to obtain good quality simulations of the system, as discussed in Section 3.5.

Output The output of this phase are the signal generators able to generate a time series for each respective environmental *"noise"*.

- **Result**: for each working regime, a set of signal generators is provided, one generator for each exogenous feature f_{ex};
- **Mathematical Performance Report**: this report includes, for each working regime, the distribution (quartiles), mean and standard deviation for each exogenous feature f_{ex}.

3.3.3 PHASE3: Simulation

PHASE3 is the phase in charge of delivering a unified information about the modeled resource consumption trends. This is achieved by *simulating* the evolution of the system, exploiting the models generated so far. The simulation starts and stops according to domain-specific initial and final conditions.

This phase is highly coupled with the working regimes defined throughout PHASE2. The minimal working regime that allows a correct simulation

of the resource consumption trends is shown in Figure 3.5a. In this working regime we have only one $f_{cfg,A}$ feature for PHASE2A and this is also the only $f_{cfg,B}$ feature for PHASE2B. This would make PHASE3 useless: in fact, in a mono-regime case, no simulation is required to understand how the entire system behaves. Thus, since we are focusing on the entire methodology, the mono-regime case is here ignored.

An example on how the working regimes would mutually be set for a real case study is shown in Figure 3.5b and Figure 3.5c. In both the examples we have

$$\begin{aligned} Cfg2B \subseteq Cfg2A, \\ Cfg2C \subseteq Cfg2A. \end{aligned} \tag{3.6}$$

where $Cfg2A$, $Cfg2B$, $Cfg2C$ are the sets containing the features f_{cfg} for, respectively, PHASE2A, PHASE2B and PHASE2C. This condition is required in order to have a description of the process's behaviour in every environmental condition and for every user's behaviour. Moreover, in Figure 3.5c we show that

$$Cfg2B \cap Cfg2C \neq \emptyset \tag{3.7}$$

can be accepted, in the case of a relation between the environment and the user.

Finally, in Figure 3.5d, we show how controllable features f_{ctrl} and uncontrollable features f_{uc} should be split among the different working regimes. More precisely, f_{ctrl}s should be all part of $Cfg2B$, as they identify the variables of the system on which the user can directly operate.

Input The input of this phase is composed by:

A. *PHASE2A models*, i.e., a set of ARX models, one for each PHASE2A working regime, representing the intrinsic process characteristics;

B. *PHASE2B models*, i.e., a set of DMM, representing the evolving behaviour of the user in all the different time bands;

C. *PHASE2C models*, i.e., multiple sets of signal generators, one for each PHASE2C working regime, each one composed of one signal generator for each PHASE2A exogenous feature f_{ex}.

Give the aforementioned description of the input of this phase, it is clear how PHASE2A and PHASE2C are tightly coupled. In fact, it is important to remind that, in order to obtain a prediction of the f_r output feature's value from the PHASE2A model, a value for every f_{ex} exogenous feature is

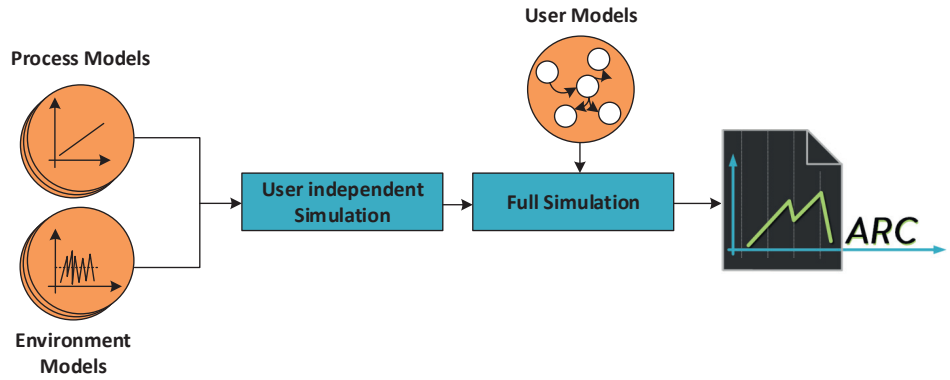


Figure 3.6: *The simulation flow, from the models of* PHASE2A, PHASE2C *(on the left) and* PHASE2B *(in the middle), to the unified resource consumption trends (on the right).*

required; these values are supplied by the signal generators created by the PHASE2C. Given this interdependence, the PHASE3 workflow is divided into two hierarchical simulation steps, shown in the block diagram in Figure 3.6: a ***User independent simulation*** and a ***User-driven full simulation***.

User independent simulation

In this step, the process models and the environment models are combined in order to obtain a set of resource consumption trends, one for each working regime of the PHASE2B.

This combination is obtained by taking each process's model and, starting from the initial conditions, using it to predict step by step the evolution of the f_r resource consumption trend, until the final conditions specified by the users are met. In order to compute the next value of f_r, the current state of every f_{ex} is needed. These values are supplied by all the signal generators produced by the PHASE2C modeling process.

When all the simulations have reached their final states, two consump-

tion trend matrices are computed:

$$\boldsymbol{T}_{discharge} = \begin{bmatrix} t_{100\%\text{-}99\%,cfg2A1} & t_{100\%\text{-}99\%,cfg2A2} & \dots & t_{100\%\text{-}99\%,cfg2Ak} \\ t_{99\%\text{-}98\%,cfg2A1} & t_{99\%\text{-}98\%,cfg2A2} & \dots & t_{99\%\text{-}98\%,cfg2Ak} \\ \vdots & \vdots & \ddots & \vdots \\ t_{1\%\text{-}0\%,cfg2A1} & t_{1\%\text{-}0\%,cfg2B2} & \dots & t_{1\%\text{-}0\%,cfg2Ak} \end{bmatrix} \tag{3.8}$$

$$\boldsymbol{T}_{charge} = \begin{bmatrix} t_{99\%\text{-}100\%,cfg2A1} & t_{99\%\text{-}100\%,cfg2A2} & \dots & t_{99\%\text{-}100\%,cfg2Ak} \\ t_{98\%\text{-}99\%,cfg2A1} & t_{98\%\text{-}99\%,cfg2A2} & \dots & t_{98\%\text{-}99\%,cfg2Ak} \\ \vdots & \vdots & \ddots & \vdots \\ t_{0\%\text{-}1\%,cfg2A1} & t_{0\%\text{-}1\%,cfg2A2} & \dots & t_{0\%\text{-}1\%,cfg2Ak} \end{bmatrix} \tag{3.9}$$

where $t_{a\%\text{-}b\%,cfg2Ah}$ are the *ticks* required for the resource to go from the $a\%$ to the $b\%$ of its budget[3], when the system is in the $cgf2Ah$ `PHASE2A` working regime. These matrices are trivially obtained by sampling the obtained simulations.

For the first time in our methodology so far, the concepts of *charge* and *discharge* appear, specializing the generic *consumption trend*. This happens because, if the chosen working regimes match the properties detailed throughout this chapter, some of them will cause the resource to be *discharged* while the others will cause a *charge*. Finally, $\boldsymbol{T}_{discharge}$ and $\boldsymbol{T}_{charge}$ simulation matrices are passed to the subsequent phase.

User-driven full simulation

This final step of the *MARC* methodology deals with the generation of a final and unified description of the modeled resource consumption trend.

We first need to compute a new version of both $\boldsymbol{T}_{discharge}$ and $\boldsymbol{T}_{charge}$ coming from the previous step, according to:

$$\boldsymbol{\Delta} = \left(\frac{(b\% \ - \ a\%) \times \ budget}{t_{a\%\text{-}b\%,cfg2Ah}} \right)_{a\%\text{-}b\%,cfg2Bh} \tag{3.10}$$

where $cfg2Bh$ is the restriction of $cfg2Ah$ to only the f features in $Cfg2B$. If multiple restrictions result in having the same $cfg2Bh$, every $t_{a\%\text{-}b\%,cfg2Ah}$ will be averaged following the same process described in the previous step; then, the resulting averaged matrix will be used for $\boldsymbol{\Delta}_{charge}$ and $\boldsymbol{\Delta}_{discharge}$

[3] for a definition of resource budget refer to Section 3.2

computation. After the computation of Δ_{charge} and $\Delta_{discharge}$, we perform a recursive simulation of the user models, i.e. each Markov chain obtained as result of PHASE2B.

Output The output of this phase is made by all the estimated times required for charging or discharging the resource budget

- **Result**: for each PHASE2B working regime, the estimated time for charging and discharging the resource between the initial and final conditions specified by the user.

These two simulation steps conclude the proposed methodology, combining the models of the process, the user and the environment that PHASE2 produced separately, into a single comprehensive model.

3.4 Implementation

The proposed methodology drove the implementation of *MARC*, a scalable platform that generates models and insights on resource consumption trends. This section presents some details on how the architecture has been implemented to achieve the requirements of (1) high parallelism and (2) high distribution of the computation, providing the service with an (3) *as-a-service* interaction paradigm.

3.4.1 Parallelism requirement

Parallelism is the first requirement of our implementation. In order to tackle the common issues that come along with parallel processing, we chose to follow the *functional paradigm* throughout the implementation of the entire business logic of our platform. More specifically, precautions for mutual exclusion and against race conditions are not required thanks to *immutability*, that enforces that two parallel tasks will only access shared values for read operations – alter operations are, in fact, denied; synchronization is implicitly enforced by data-availability, as functional programming allows to define a computation from the initial input to the final output as a data flow.

We decided to use the Scala programming language [127], as it offers a wide range of advantages that are collateral to functionality: compiled to Java Virtual Machine (JVM) bytecode, highly concise code, strong type safety and support for other paradigms.

3.4.2 Distribution requirement

Algorithm distribution is a deeply intricate problem [37, 146], involving issues ranging from communication and synchronization to fault tolerance and replication. We tackle these issues organizing the entire distribution logic following the *actor model*. Each actor is a component devoted to very narrow and precise task, and communicates with other actors only through a set of messages: this isolation constitutes a good structure for enabling distribution. Then, message routing and dispatching over the network are implemented defining a naming structure and resolving policies.

We decided to employ the Typesafe Akka framework [15], that is characterized by simple concurrency and distribution, high performance, resiliency by design, decentralization and extensibility.

3.4.3 *"As a service"* requirement

The *MARC* platform is meant to be used in a service aggregation process, to enable reasoning and self-awareness for autonomous agents and systems. We then designed each module of our platform to expose REpresentational State Transfer (REST) interfaces for both system's clients and other internal modules: this software architecture [59] is meant to ensure resource identification (using Uniform Resource Identifier (URI)) and manipulation through uniform representations and self-descriptive messages.

Moreover, a RESTful service must be:

- **Stateless**: this is guaranteed by the functional paradigm used to develop the business logic of each *MARC* module;
- **Cacheable**: this is guaranteed by the *"whiteboard approach"* that *MARC* exploits to allow phases to communicate with each other. This functionality, implemented on top of multiple Redis No-SQL databases [134], allows to cache inter-phases results along the way.

In order to avoid useless computations and allow future changes in the inter-dependencies among the modules, a *"backward activation"* pattern has been put in place.

Starting from the `ENTRYPOINT` module, that is in charge of receiving the computation requests from the outside world, the requests are sent back through the workflow, starting from the phase producing the result desired by the client. In fact, the client is not forced to require only the complete execution of the methodology, but can also request a partial result (e.g., only the process model from `PHASE2A`). This pattern, combined with the caching mechanism, also solves the problem of duplicated computation that

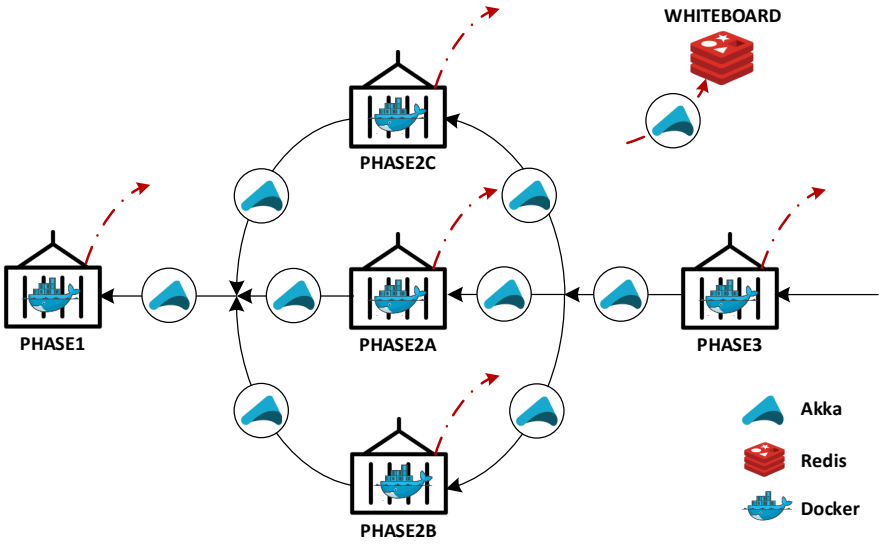


Figure 3.7: *Backward activation structure: requests are sent back through the workflow, starting from the phase producing the result desired by the client.*

a backward activation may cause when propagating the dependencies from the `PHASE2` phases back to the `PHASE1` module.

This caching mechanism is part of the `CommunicationActor` logic: it constitutes the joining link between the backbone infrastructure – in charge of dispatching the requests and the result by employing the backward activation mechanism – and each phase's internal logic – encapsulated starting from the `InternalWorker`.

The `CommunicationActor` is in charge of handling the life-cycle of a specific computation, tracking its state and possible malfunctioning behaviours. The life-cycle evolution is briefly described as follows:

- **Init State**: in this state the `CommunicationActor` is in the phase's `LoadBalancer` pool and is ready to be configured to accept a new incoming computation request.
- **Ready State**: after having received the working regime for an incoming computation, the `CommunicationActor` waits in this state for receiving a computation request from the backward activation flow.
- **Preamble State**: a computation has been requested, thus the `CommunicationActor` checks if the results are already present in

the cache.

- **Wait State**: in this state, the `CommunicationActor` is waiting for previous phases – triggered through backward activation – to compute their results.
- **Computation State**: the `CommunicationActor` instantiates an `InternalWorker` that will carry out the phase-specific computation.

3.4.4 Implementation details

In this section, we detail how some implementation issues have been tackled, solved or at least mitigated.

Computational performance

The functional paradigm and the actor model enable a high level of parallelism and distribution. However, we had to show some cares on some performance critical parts of the business logic.

On the one hand, the lack of *indexed loops* (e.g., *for* loops) in a functional language is mitigated by advanced operations based on the *"map-reduce"* pattern. When this is not sufficient, recursive functions must be employed, thus leading to JVM heap *"explosion"*: the Scala compiler is able to detect some form of problematic *tail-recursive* functions and compile them in order to avoid that problematic behaviour; however, we had to manually restructure a good amount of them to prevent heap explosion.

On the other hand, we had to deal with a potential *parallel slowdown* related to an explosion of the number of actors in the system: in fact, when the number of actors grows dramatically, message routing and balancers/-collectors become two important bottlenecks of the system, saturating the speed-up trend that happens at the beginning with an increasing number of actors. As a consequence, we studied and developed an *auto-scaling mechanism* that, based on throughput metrics, automatically scales up the size of the pool of available actors in the system without triggering the aforementioned degradation effect.

Memory management

Another significant issue that affects functional programming revolves around *garbage collection*. In fact, any operation that in a traditional object-oriented language would lead to data modification, in a functional language causes the creation of a new object representing the result of the edit operation.

This leads to at least two main problems: on the one hand, it requires to clone the object in memory[4], thus involving sensible delays if the original object is huge; on the other hand, repetitive editing (e.g., on collection iterations) can lead to a huge amount of clones being allocated in memory but becoming not referenced almost instantly.

In order to mitigate these problems, we chose to use the most effective Garbage Collector (GC) for the functional case, i.e., `G1GC`: it is highly parallel and optimized for long complex iterations – the case of almost all the manipulation primitives offered by Scala. Moreover, we inserted multiple hints for the GC in the code where a conspicuous amount of objects is about to become useless – this usually happens at the end of every *MARC* phase. These hints then cause the GC to be triggered early enough to avoid memory saturation.

Finally, we fine-tuned the balance between *young* and *old* generations, two different categories of objects that the GC maintains and treats differently. This tuning process is required to balance the trade off between fast allocation – more young generation objects – and finer garbage collection – less young generation object. In the case of Scala, where it is highly improbable that an object would last in memory enough to be transferred to the old generation, it is better to unbalance the trade off towards the first solution.

Reliability

The *MARC* platform implements the *"let-it-crash"* paradigm: every piece of code is aware that something may go wrong along the way and, when it happens, the system must be able to remain up and running. We then allowed every actor to generate, receive or relay a special `ThrowsException` message that carries the description of what went wrong. After its generation, this message climbs up the actors invocation tree and either reaches the root actor in the infrastructure or it gets intercepted by an intermediate actor that is able to solve the situation: in the first case, the computation of that specific phase is stopped and the infrastructure deallocates all the actors waiting for it, while, in the second case, the crash is just masked to the client.

Moreover, we introduced *timeout* management on every operation implemented in the system, as suggested in [37]. This is achieved by setting a timeout on every state of the `CommunicationActor`; if a timeout triggers, the infrastructure signals it to the client and deallocates all the actors

[4] In some cases the Scala compiler can understand such critical situation and avoid cloning.

that were waiting for the expired computation.

Finally, the same timeouts are also used to avoid useless recomputations. In fact, when a computation is triggered by backward activation, the results of a phase are maintained available directly from the actor for a reasonable amount of time: if more than one phase is waiting for the same results, these are distributed as soon as they are available.

Communication

We chose to use the JavaScript Object Notation (JSON) format for both the inter-phase and the client-facing communication, as it is more lightweight than XML. In order to guarantee type safety, we exploited Rapture [130], a library that allows to handle JSON objects in a type safe manner, as well as a fast on-demand serialization/deserialization.

Even though the JSON format is optimal for end-to-end communications, it may be excessively prolix for internal data representation. We then compress it using the LZ4 algorithm in a `Base64` stream, achieving a compression rate over 98%.

3.5 Validation

This section discusses the set of experimental results that support the proposed methodology, assessing the correctness, the validity and the robustness of its implementation into the *MARC* platform.

More precisely, two different experimental settings are here presented, with different goals: on the one hand, I generated synthetic and controlled traces with different patterns; on the other hand, I reproduced the results obtained in Section 2.6.1, thus showing how the general methodology performs at least as well as the *ad-hoc* modeling solution.

3.5.1 Simulator

In this section, we present a variegated set of simulations that show the pros and cons of the *MARC* methodology in a completely abstract and general validation process. We show that, under reasonable assumptions, the *MARC* methodology produces correct estimates of resource consumption trends and that, in case a precise enough estimation can not be produced, this issue can be highlighted by exploiting an extensive set of collateral figures of merit (e.g., the spectral conditioning numbers).

Parameters of the synthetic traces

Before testing and evaluating MARC on real applicative scenarios, I built a Discrete Event Simulator that is able to generate synthetic traces of an **abstract system**, according to these characteristics:

- **Number of components**: the system can be composed by one or more components – this is usually related to the number of *features* in the data set;
- **Component's impact**: each component can sum different contributions to the modeled resource consumption – these are usually related to the process's behaviour modeled in `PHASE2A` as follows:
  - **Linear**: the contribution is a linear combination of the exogenous input of the component (e.g., the impact of the screen brightness level on the battery depletion in a smartphone [58]);
  - **n-Polynomial**: the contribution is either sub-linear or super-linear with respect to the exogenous inputs of the component (e.g., the impact of voltage on dynamic power in a MOS integrated circuit [161]);
  - **Exponential**: the contribution quickly diverges (e.g., the impact of number of new nodes to process on the time for a clique problem solver [92]);
- **Component's behaviour**: each component could be in a different state in a different moment in time, and each of these states can be characterized by a different impact on the resource – usually each state corresponds to a `PHASE2B` ***configuration***;
- **Components coupling**: when two or more components are in a defined set of states, their contribution to the resource consumption can be diminished (positive coupling, e.g., bluetooth and Wi-Fi radio on at the same time have an impact on battery depletion that is lower than the sum of their separate consumptions) or augmented (negative coupling, e.g., two cores computing two highly synchronized tasks consume more than the sum of the same two computing independent tasks);
- **Exogenous signals**: different signals of different order of magnitudes can be applied to the input of each component: Constant, Square wave, Triangular wave, Random (Uniform, Poissonian, Gaussian), etc;

Table 3.1: *Error metrics used in the Simulation case study. Notice that n is the number of samples in the trace under study, Y_i and $\hat{Y}_i$ are respectively the sample output value and its estimation, while p is the number of model parameters, α_i and $\hat{\alpha}_i$ are respectively the model parameter and its estimation.*

name	symbol	formula
Mean Square Error	MSE	$\frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$
α Percent Error	α: err%	$\frac{1}{p}\sum_{i=1}^{p}(\hat{\alpha}_i - \alpha_i) \times 100$

- **Noise**: the entire sampling process can be affected by a Gaussian white noise, with the following characteristics:
  - **Low level noise**: a noise that, in mean and standard deviation, is less then or comparable to the dynamic of the original signal;
  - **High level noise**: a noise that, in mean and standard deviation, is orders of magnitude greater than the dynamic of the original signal;
- **Sampling procedure imperfection**: even in the absence of noise on the signals, the sampling procedure itself presents imperfections, such as:
  - **Time drifts**: the sampling rate is not stable;
  - **Missing data**: the procedure randomly fails to acquire either some values of or the entire sample.

Given that the number of possible combinations of the aforementioned characteristics is too high, the next sub-sections present a selection of the evaluations conducted that is sufficient to show the correctness, validity and robustness of the *MARC* approach. Error metrics we used throughout this case study are summarized in Table 3.1.

Synthetic traces: Mono component - Different impacts

This first batch of simulations is aimed to verify the basic behaviour of the *MARC* methodology while modeling a system based on the three aforementioned impact relations: *linear*, *polynomial* and *exponential*.

The resource consumption trends related to these three different situations are shown in Figure 3.8: in Figure 3.8a we see the plot of the exogenous signal that stimulates the component, while Figure 3.8b-d show the resource consumption trends for different component impacts.

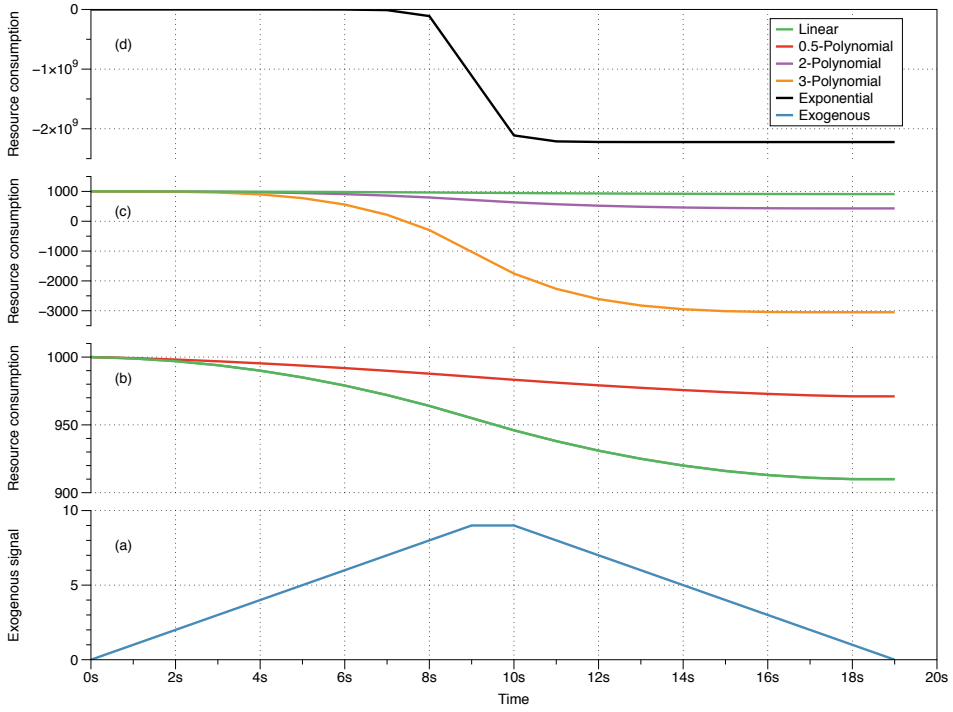


Figure 3.8: *Compared impacts on the resource consumption trends of a system based on three different impact relations: linear (b), polynomial (c) and exponential (d), with the exogenous signal reported in (a).*

Table 3.2: *Simulation results: this table shows, for different impacts of the only component present in the simulation and different input signals, the MSE and the relative estimation error over the models αs (α: err%). For both the metrics, the nearer the values to zero the better the model.*

		Constant signal		Triangular wave signal	
		MSE	α: err%	MSE	α: err%
$y = \alpha x$	single state	2.26×10^{-26}	0%	3.18×10^{-28}	0%
	multiple states	2.71×10^{-25}	0%	1.04×10^{-25}	0%
$y = \alpha\sqrt{x}$	single state	1.82×10^{-29}	0%	9.83×10^{-2}	68%
	multiple states	1.50×10^{-26}	0%	3.67×10^{-2}	66%
$y = \alpha x^2$	single state	1.76×10^{-26}	0%	5.92×10^{1}	-772%
	multiple states	4.27×10^{-26}	0%	1.51×10^{1}	-833%
$y = \alpha x^3$	single state	4.91×10^{-32}	0%	1.03×10^{4}	-7,278%
	multiple states	1.55×10^{-26}	0%	2.93×10^{3}	-7,869%
$y = \alpha \cdot 10^x$	single state	7.42×10^{-27}	-900%	6.43×10^{16}	-4,347,498,491%
	multiple states	8.41×10^{-24}	-900%	2.13×10^{16}	-4,189,333,130%

Table 3.2 shows the modeling MSE and the average relative error on the α parameters of the ARX models. We report the two more interesting batches of simulations with respect to the exogenous input signals. These results confirm the correctness of our methodology and platform. In fact, the MSE and the relative α estimation errors are almost null in the linear case, and in all the polynomial cases with a constant exogenous signal. For polynomial impacts, the error grows together with the distance of the polynomial grade from 1. Moreover, the error is consistent in monotonicity: if the impact is sub-linear, the estimation error is positive (i.e., the model underestimates it), whereas, if the impact is super-linear the estimation error is negative and grows, in absolute value, together with the grade of the impact.

Then, these results show how an exponential impact can not be modeled by our methodology, as expected and already made explicit in Section 3.3.2; however, such impacts are usually characteristic of transient states and these are out of the scope of this work.

Table 3.3: *Simulation results: this table shows, for different impacts of one of the components present in the simulation and different input signals, the Mean Square Error (MSE) and the relative estimation error over the models αs (α: err%). For both the metrics, the nearer the values to zero the better the model.*

		Constant signals		Triangular wave signals	
		MSE	α: err%	MSE	α: err%
$y = \alpha_1 x_1 +$	$\alpha_2 x_2$	2.20×10^{-21}	38%	2.16×10^{-21}	0%
	$\alpha_2 \sqrt{x_2}$	1.40×10^{-21}	76%	4.00×10^{-2}	42%
	$\alpha_2 x_2^2$	2.36×10^{-21}	-472%	4.15×10^{4}	-10,876%
	$\alpha_2 x_2^3$	1.85×10^{-19}	-5,574%	3.19×10^{8}	-766,637%

Synthetic traces: Multi component - Different impacts

In this second batch of simulations, our aim is to test a more realistic case, in which the system is composed by different components, each one characterized by its own behaviour and impact on the resource consumption trend, stimulated by a dedicated exogenous signal, independent from the other signals.

Table 3.3 reports the modeling MSE and the average relative error on the αs estimation. We report the two more interesting batches of simulations with respect to the exogenous input signals. These results confirm the correctness of our methodology with respect to complex systems and maintains the same positive characteristics that have been described in Section 3.5.1.

It is important to notice the discrepancy between the MSE and the relative error for the *"Constant signals"* case. While these data alone can be confusing, a more clear picture is given when considering also the **spectral conditioning numbers** related to the `PHASE2A` ARX models generation. In case of constant signals, the available data are not sufficient to correctly understand the impacts of components on the resource consumption; Chart 3.9 clearly shows that in the constant case the data are ill-conditioned, thus generating a model that shows high sensitivity to any alteration of the parameters, even if it is correct for the specific case – the MSE is negligible, anyway. This means that the generated model is affected by *overfitting*.

It is important, in a multi component situation, to keep into account also possible coupling impacts. We modified the simulation presented in the first row of Table 3.3 (i.e., a two-component system, each one with linear impact on the resource consumption) in order to include coupling impact for specific combinations of states. Table 3.4 shows the results of this batch of simulations. Briefly, for the *constant signals* case is evident that the model

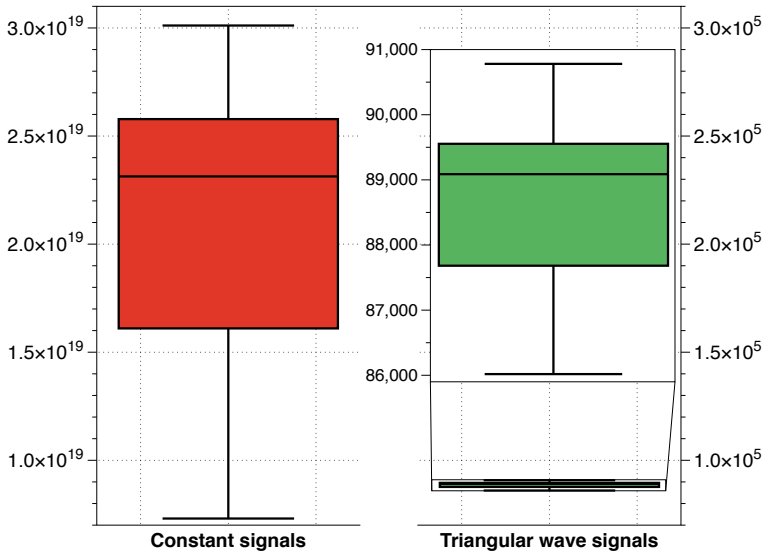


Figure 3.9: *Spectral conditioning number distributions*

Table 3.4: *Simulation results: this table shows, for different coupling impacts between components, the MSE and the relative estimation error over the models αs (α: err%). For both the metrics, the nearer the values to zero the better the model.*

	Constant signals		Triangular wave signals	
	MSE	α: err%	MSE	α: err%
linear coupling impact	6.68×10^{-03}	38%	1.70×10^{-21}	-26%
quadratic coupling impact	1.96×10^{-21}	32%	3.99×10^{-2}	-63%

is not able to generalize, as stated before and expected. By looking at the more realistic *triangular wave signals* case, the results for the linear coupling impact clearly show that our methodology is able to tackle coupling phenomena without increasing its modelling error: the MSEs are almost identical between the coupled and non-coupled case, while the estimation relative error increases, as expected, because the model has no component to represent the coupling effect, thus, it has to distribute this contribution among the coupled components. Moreover, even if non-linear coupling impacts cause greater errors than linear ones, this error is similar to the one introduced by non-linear component impacts and, as such, can be tackled by defining a more precise working regime for `PHASE2A` (as discussed in

Table 3.5: *Simulation results: this table shows the Mean Square Errors measuring how different levels of noise can impact on the model precision in a multi-component situation. The lower MSE the better the model*

		$y_2 = \alpha_2 x_2 + \ldots$	
		LNOISE	HNOISE
$y_1 = \alpha_1 x_1 + \ldots$	LNOISE	4.04×10^{-1}	1.68×10^{-21}
	HNOISE	1.69×10^{-21}	6.13×10^{0}

Section 3.3.2).

Synthetic traces: The noise impact

The system under study can be affected by *noise*: this may alter unpredictably the relationships that connect the exogenous inputs to the impact of the components on the resource consumption trend, but can also affect the sampling procedure itself, by mainly causing drifts in the sampling period and corruption of sampled data.

Since the noise characteristics can be extremely variegated, we defined two classes of noise that we use in our simulations:

$$\text{HNOISE} \sim \mathcal{N}(0, \sigma),\ \sigma \gg \sigma_{signal} \tag{3.11}$$

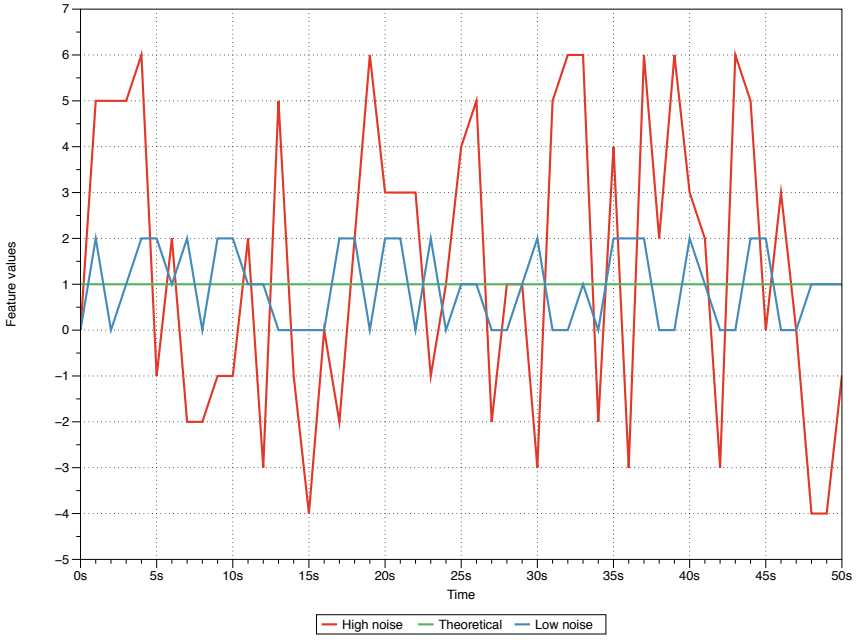
$$\text{LNOISE} \sim \mathcal{N}(0, \sigma),\ \sigma \simeq \sigma_{signal} \tag{3.12}$$

We have then modified the simulations described in Section 3.5.1 in order to perturb the impacts of each one of the components with each class of noise defined.

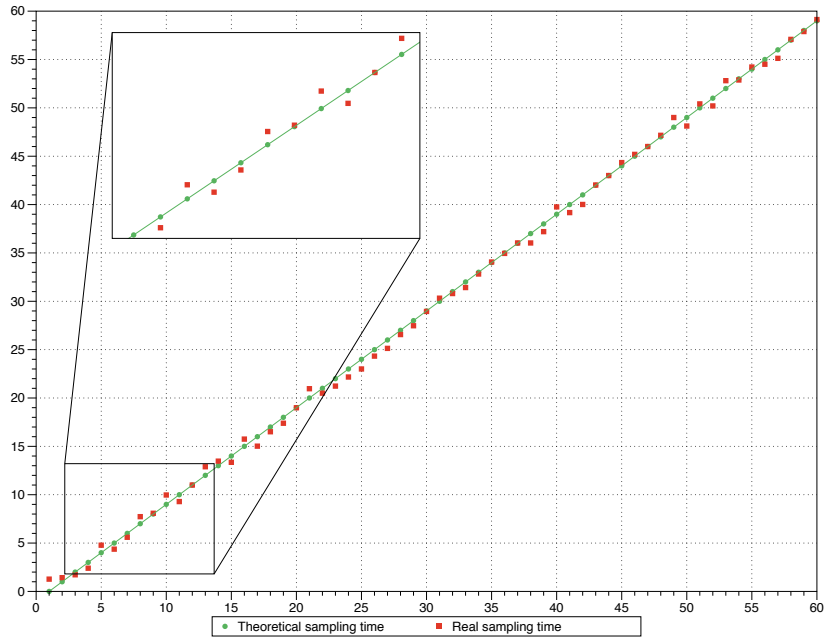
Table 3.5 shows the results of one of the simulation made, that is:

- **Number of components**: 2
- **Exogenous signals**:
  - **Component 1**: Triangular wave signal with period equals to 7 ticks and amplitude equals to 10;
  - **Component 2**: Triangular wave signal with period equals to 11 ticks and amplitude equals to 100;

In this simulation, *component 1* is much less important in determining the resource consumption trend – on average its contribution is lower than the one of *component 2* by two orders of magnitude.



(a)



(b)

Figure 3.10: *Impact of noise on: (a) the system under study with two different levels of noise: HNOISE in red, LNOISE in blue (no noise in green) and (b) on the sampling procedure - sampling period drifts shown in red*

.

Worst case scenario		
mean coverage		25.83%
mean square error		9.31
output relative error	*full scale range*	0.01%
	mean value	0.34%
	median value	0.34%

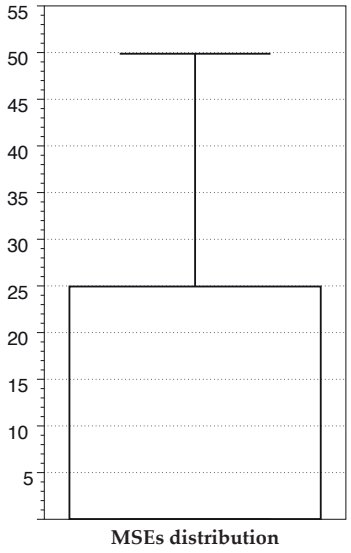


Table 3.6: *"Realistic" simulation results: on the left, the figures of merit characterizing the models produced by the simulation; on the right, the box plot of the distribution of the MSEs obtained from multiple simulation (the lower whisker is not visible). The mean coverage indicates for how many samples we had a model to estimate them (the higher the better). The output relative error compares the Root Mean Square Error to values representing the output distribution (the lower the better).*

Table 3.5 shows the results of this simulation batch: we can see that the MSE dramatically increases only in situations where an HNOISE is involved. This is not, anyway, a flaw of the methodology itself; in fact, given the definition of HNOISE, this noise is able to completely mask the original signal dynamics, and, as such, features representing this messed-up signal should be excluded because non representative (this insight can be obtained in the *feature selection* phase).

Synthetic traces: a "realistic" simulation

In this paragraph we discuss the results obtained from a batch of simulations that, from our point of view, can represent the worst case scenario that may be submitted to the *MARC* platform.

This scenario is obtained from the LNOISE-LNOISE simulation already presented in this section, plus random sampling period drifts and data corruption (random samples get lost), without extending the simulation duration to compensate in any manner.

Table 3.6 reports the most interesting figures of merit that characterize

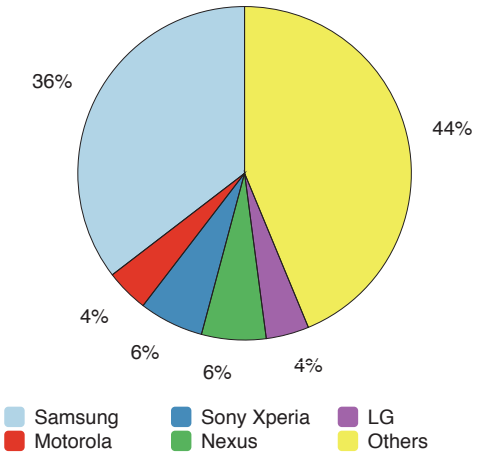


Figure 3.11: *Devices distribution by manufacturer, from the MPower database*

this batch of simulations. For the first time, we introduce the concept of **coverage**, i.e., the percentage of the samples for which a model is available. In all the previous simulations, the coverage was always equal to 100%, but, as we introduce corruption and drifts, *MARC* is no more able to compute a model for every working regime. Nevertheless, when a model is available, its MSE is negligible; this is more evident by looking also at the relative errors listed in Table 3.6, that show how distant the model estimated values are from the real ones. If we consider the full dynamic of the resource consumption trend, our error is 0.01%; with respect to the central values of the same trend it grows to 0.34%; both these errors are negligible.

With this last realistic example, we finally showed the goodness and correctness of the *MARC* methodology and implementation, meaning that it is able to produce interesting insights and to extract valuable knowledge from a non-trivial data set.

3.5.2 Regression testing: power models for Android devices

The main goal of this last section is to verify that the *MARC* methodology can reproduce the results obtained by the methodology proposed in Chapter 2.

It is important to note that data used for this purpose are not exactly the same employed in Section 2.6.1, as the data set available at the time of these tests is more than twice bigger, while part of the original data is not

available anymore. These tests then focused on 50 devices with an MPower model available and the following features:

- **Trace size**: we chose devices with traces of different sizes, ranging from 6,500 to 1,274,327, that is the maximum size currently available;
- **Device type**: there are traces coming from both smartphones and tablets; we chose traces from both of them;
- **Device model**: there are traces coming from more than 600 different models, with deeply different technical specification; we chose 50 devices that represent a similar distribution of this diversity (Figure 3.11).
- **Trace quality**: the MPower logging process can be strongly unreliable because of its power efficiency; this may lead to traces that are highly corrupted (missing samples, missing values, etc.); we chose 50 devices both from the ones with high quality traces and the ones that presented an highly noisy behavior.

Figure 3.12 shows in green that we have obtained the same MSE distributions as the one reported in Section 2.6.1, with the label *"Old"*. However, we obtained these results by considering only the devices that were present in the MPower database at the time of the study.

Considering the whole set of devices, the MSE distribution, shown in red, changes drastically. This difference in the results is caused by the data that have been added after we performed our first study: in fact, these data are related to newer versions of the Android OS, that introduced profound variation in the exposed APIs – leading to greater data corruption – and in the energy management policies – leading to highly fragmented traces.

However, these results show how we are able to reproduce the same precision of the results obtained in Section 2.6.1, thus showing how the MPower methodology has been generalized and abstracted consistently, even if it would require a logger application update in order to produce comparable results on new data.

3.6 Final remarks

This chapter discussed how it is possible to generalize the same concepts learnt from Chapter 2 towards a general methodology: the observed system does not need to be a smartphone but it could be a generic system, i.e., an *"agent"*, that wants to become *power-aware* or, in a wider sense,

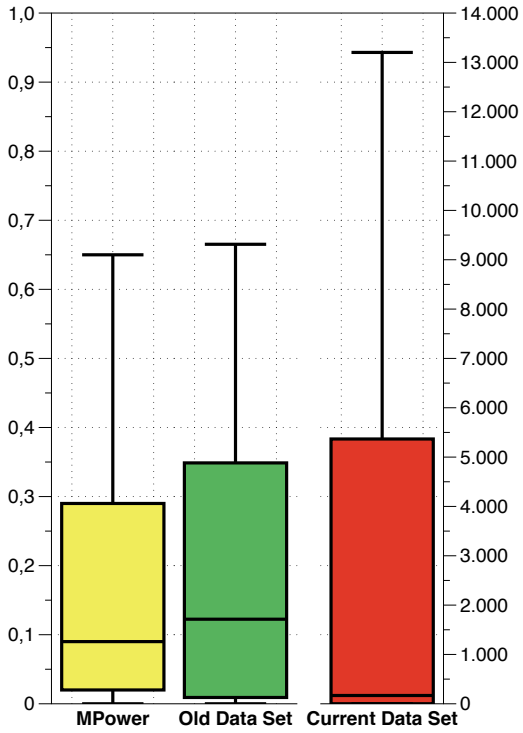


Figure 3.12: MPower *MSE distributions*

"resource-aware", of course without disregarding domain-specific assumptions on features and applications.

The generalized data-driven methodology for resource consumption modeling has then been implemented into MARC, a Cloud-service platform designed to Model and Analyze Resource Consumption trends (MARC), supporting a "Model-as-a-Service" paradigm. In order to validate the proposed methodology, a custom simulator has been set up to generate a wide spectrum of controlled resource consumption traces: this allowed to verify the correctness of the framework from a general and comprehensive point of view. Moreover, it was able to reproduce the same precision of the results obtained in Section 2.6.1, thus showing how the *MPower* methodology has been generalized and abstracted consistently.

The next chapters will show how this generalization allows to bring power-awareness into a completely different context: power consumption models for virtual machines in a multi-tenant virtualized infrastructure.

CHAPTER 4

Towards power-awareness for the Xen Hypervisor: virtual guests monitoring

4.1 Introduction

In the last few years, embedded systems have experienced a shift from microcontrollers to the adoption of multi-core processors, as these have become cheaper, smaller, and less power-hungry. This shift brings two advantages: (1) multiple embedded applications can be consolidated on the same System-on-Chip (SoC), improving the overall resource utilization, and (2) some applications can exploit concurrency and parallelism to obtain better performance.

These enabled the disruptive changes that smartphones and tablets brought in the way we work and live, as discussed in Chapter 2. However, "end-user" devices are not the only ones affected by this remarkable shift: multi-core processors empower a wide range of application fields, like automotive, Internet TV and other embedded use cases like low-power microservers for lightweight scale-out workloads [2, 84].

In the context of embedded systems, hardware-assisted and software virtualization technologies have been developed to allow colocated applica-

tions to share physical resources while having strong security and isolation [137, 141, 156].

Those technologies seek to offer a stable and predictable execution environment to make it easier for embedded applications to meet different QoS requirements.

The virtualized runtime can be a full-fledged guest OS or more suitable to embedded systems, a light-weight OS (e.g., [96, 148]), customized for a specific application. Applications executing in such runtimes generally have different performance objectives, such as: hard and soft deadlines, and peak throughput. Moreover, they are often different from one another in terms of workload limitations (i.e., memory-bound, I/O-bound, or CPU-bound) and evolving load patterns (e.g., algorithmic phases).

Unfortunately, this high *heterogeneity* comes at a price: The isolation between simultaneously resident applications, enforced by virtualization, shifts the burden of optimization from developers to the hypervisor itself because only a privileged arbiter can thoroughly observe what happens on the bare metal. Hence, it becomes clear that a smart *online* monitoring strategy is necessary to accurately observe and model applications' behavior to guarantee requirements and optimize physical resources utilization.

Since power consumption is currently a key technological limitation [75], recent works propose approaches to optimizing power [40], while maintaining Service Level Agreements (SLAs) with each hosted guest. Again, these approaches have an essential need for precise and thorough *observation* of both hardware and guests' behavior over time. Lacking appropriate tools, many of these approaches employ custom monitoring solutions or rely on outdated tools that do not provide support for the latest hardware monitoring features. Often, these ad-hoc approaches overlook the impact of measurements on the overall system's behavior.

Seeking to fill the gap, this chapter proposes *XeMPower*, a lightweight hardware and resource monitoring solution for the Xen hypervisor [17]. It is meant to be agnostic to the hosted applications, and results show it incurs negligible power consumption overhead. *XeMPower* has been released as open source,[1] and it aims to be a reference design for future works in the field of virtualized systems.

To prove its effectiveness, we present a use case in which *XeMPower* precisely accounts hardware events to virtual guests, enabling real-time attribution of CPU power consumption to each guest or "domain".[2] *XeMPower* starts with socket-level energy measurements through the Intel Run-

[1] Available at: `https://bitbucket.org/necst/xempower-4.6`

[2] Adopting Xen terminology, the remainder of this chapter will refer to virtual guests as *domains*.

ning Average Power Limit (RAPL) interface [138], and then utilizes a performance-counter-driven model to account for the proportional uses of energy by simultaneously resident domains over time. This proportional attribution of power is *XeMPower*'s secret sauce – the contribution is evaluated by measuring a subset of architectural performance counters related to each domain, but regardless of the physical core.

The chapter is organized as follows. Section 4.2 presents an overview of *XeMPower*. Section 4.3 details the implementation of the tool, while Section 4.4 shows how to attribute power to each domain. Next, Section 4.5 investigates the performance overhead of *XeMPower* while Section 4.6 presents the related work and Section 4.7 concludes.

4.2 Proposed approach and requirements

XeMPower is a lightweight monitoring solution for Xen, that perfectly fit the *Observe* phase of the OLDA loop discussed in Section 1.2. It is designed to: 1) provide *precise attribution* of hardware events to virtual tenants, 2) be *agnostic* to the mapping between virtual and physical resources, hosted applications and scheduling policies, and 3) add *negligible overhead*.

The proposed approach uses hypervisor-level instrumentation to monitor every context switch between domains. More precisely, the monitoring flow proceeds as follows:

(A) At each context switch and before the domain chosen by the scheduler starts running on a CPU, we begin counting the hardware events of interest. From that moment the configured Performance Monitoring Counter (PMC) registers in the CPU store the counts associated with the domain that is about to run.

(B) At the next context switch, the PMC values are read from the registers and accounted to the domain that was running. The counters are then cleared for the next domain to run.

(C) Steps A and B are performed at every context switch on every system's CPU (i.e., physical core or hardware thread). The reason is that each domain may have multiple virtual CPUs (VCPUs). Socket-level energy measurements are also read (via Intel RAPL interface) at each context switch.

(D) Finally, the PMC values are aggregated by domain and finally reported or used for other estimations (e.g., power consumption per domain).

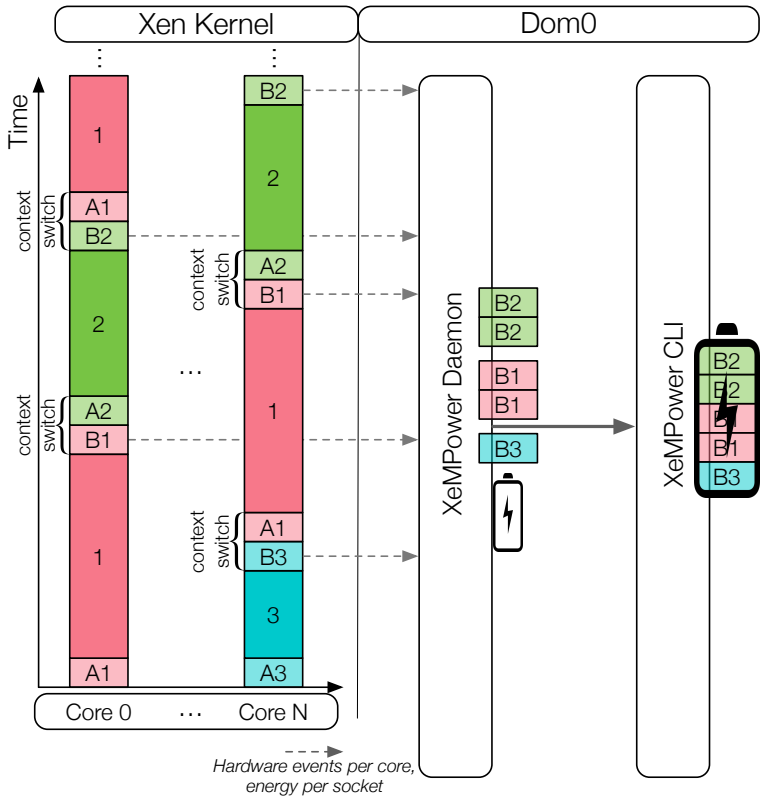


Figure 4.1: *Monitoring flow of* XeMPower.

Figure 4.1 illustrates the monitoring flow described above. Steps A and B for domains 1, 2, and 3 are shown at every context switch on the left side of the figure. On the right side, steps C and D are performed by the XeMPower *daemon* and *Command Line Interface (CLI) program*, both in Dom0.

Steps A and B allow to meet the first requirement (precise event attribution), while the second requirement (being agnostic) is satisfied by steps C and D. Regarding the third requirement of low overhead, Section 4.5 empirically confirms that *XeMPower* meets this requirement, while the technical aspects enabling that are discussed in Section 4.3.

4.3 Implementation

XeMPower implementation is inspired by *XenMon* [68], a performance monitoring tool for Xen. Unlike *XeMPower* and other works discussed in

Section 4.6, XenMon does not collect PMC reads. Nevertheless, since XenMon's authors report a maximum overhead of 1-2%, their implementation approach was an interesting starting point for this work and a reasonable baseline to compare the overhead with.

XeMPower operates at two levels (see Figure 4.1). At the first level, PMC reads are collected inside the Xen kernel and then aggregated by the XeMPower *daemon* running in Dom0, while at the second level, a *CLI program* reports aggregated values. In this section, I describe implementation details of the components forming the proposed toolchain, as these will be a reference for the community that may adopt the tool for further researches.

4.3.1 Xen kernel instrumentation

Xen runs a separate scheduler instance on each CPU, and each scheduler instance has its own queue containing runnable VCPUs of domains [33]. Xen kernel's `schedule()` function[3] preempts the currently running VCPU (scheduler-independent), chooses the VCPU that will run next (scheduler-dependent), and then makes the chosen VCPU run (scheduler-independent). Hence, this function is a suitable place to incorporate the steps A and B presented in Section 4.2.

Even though there are libraries and APIs (e.g., PAPI [29]) that give developers access to hardware events independently from the underlying architecture, I decided to directly use `RDMSR` and `WRMSR` assembly instructions to set the count of desired hardware events as well as read and clear the CPU's PMC. The reason is that these operations are performed at every context switch and we want the overhead to be as low as possible at the kernel level, in terms of execution time and memory footprint. We then accept the trade off and tie the current implementation to the Intel instruction set; however, other architectures (e.g., ARM and AMD) can be supported by modifying the registers addresses at compile time.

The current *XeMPower* implementation only counts *architectural performance monitoring events*. I made that decision because these events have consistent visible behavior across processor implementations [6]. Moreover, previous work shows that they are the most significant metrics to correlate CPU power consumption [162], which is the focus of our motivating use case in Section 4.4. Since the available PMCs are limited (e.g., 8 per core and 4 per hardware thread on Intel Sandy Bridge 2nd Gen processors), we map some monitoring events onto 4 PMCs and others are counted using auxiliary *fixed-function* counters. Table 4.1 summarizes the monitored

[3]Source code: `xen/common/schedule.c`

Table 4.1: *Monitored hardware events [6]*

Event Mask Mnemonic	Register mapping
Instruction Retired	`IA32_FIXED_CTR0`
UnHalted Core Cycles	`IA32_FIXED_CTR1`
UnHalted Reference Cycles	`IA32_FIXED_CTR2`
LLC Reference	`IA32_PMC0`
LLC Misses	`IA32_PMC1`
Branch Instruction Retired	`IA32_PMC2`
Branch Misses Retired	`IA32_PMC3`

events and their register mapping.

Regarding power monitoring, Intel RAPL interface provides dedicated read-only registers that can be accessed like standard PMCs. These are available since Sandy Bridge 2nd generation processors and provide CPU power measurements with a time granularity of 1ms approximately. *XeMPower* currently samples the register `MSR_PKG_ENERGY_STATUS`, which accumulates the actual energy consumption (in Joules) of the whole processor package; the average power consumption is then easily obtained as $energy/time$ for the time window considered. For the moment, I decided not to sample the other RAPL power planes (related to on-chip DRAM and "uncore" devices) because their availability varies across different processors.

Finally, we need to expose the collected data to a higher level. For that, I use `xentrace` [33], a lightweight trace capturing facility present in Xen that can record events at arbitrary control points in the hypervisor. Every trace record is tagged with the ID of the scheduled domain and its current VCPU, as well as a timestamp to be able to later reconstruct the trace flow.

4.3.2 XeMPower daemon

The stream of trace records produced by `xentrace` flows from the Xen kernel to the XeMPower *daemon* running in Dom0 (see Figure 4.1). The daemon, a user-space program written in C, receives the records and performs aggregation operations on them. Note that the `xentrace` user-space tool is not involved here, as it can produce a very large amount of data that may potentially cause intense disk writes. *XeMPower* daemon directly accesses `xentrace` memory buffers, to avoid any additional access to disk.

I defined two bitmasks, `TRC_POWER_PMC` and `TRC_POWER_RAPL`, to differentiate trace records with PMC and RAPL events in the `xentrace`

buffers (one per hardware thread). These buffers are constantly monitored by the *XeMPower* daemon – when a new record arrives, a callback function is invoked to process and store it.

The *XeMPower* daemon performs aggregations in three stream processing stages. First, records are grouped in tumbling windows with a configurable time interval. Second, in each tumbling window an aggregation is performed per hardware event. In this stage, the deamon also stores the difference between the values of the RAPL energy counter at the beginning and the end of the tumbling window. Finally, in each tumbling window and for each hardware event PMCs are collated per domain. Note that after aggregating the records the notions of physical and virtual CPUs disappear, bringing about a hardware-agnostic data structure.

The *XeMPower* daemon allocates a shared memory region to store a configurable number of tumbling windows in a circular buffer. Processes other than the daemon can only read from the region. Shared access to the tumbling windows allows multiple front-end applications to read and display different statistics from the same data. The tumbling window time interval, the capacity of the circular buffer of tumbling windows, and other configuration parameters can be specified at compilation time. Currently, the default value for the tumbling window interval is 100 ms and the circular buffer's capacity is 100. These values are used in our experiments reported in Section 4.5.

4.3.3 XeMPower command line interface

XeMPower CLI is a basic command line tool written in Python. It periodically scans the tumbling windows produced by the *XeMPower* deamon (in the shared memory region), and performs aggregations in two time intervals: every second and every 10 seconds. It is also in charge of converting the RAPL counter values into energy consumption values (in Joules). The conversion factor is given by the `MSR_RAPL_POWER_UNIT` register, which is architecture-specific and can be read once when the *XeMPower* deamon is started. The socket power consumption is then obtained as the ratio of the energy consumption and the considered time interval. *XeMPower* CLI is designed to show live statistics on console or to log them into a file for a later processing.

4.4 Use Case: per-domain CPU power attribution

As a first motivating use case for the tool, this section describes how *XeMPower* can perform per-domain attribution of CPU power consumption.

Zhai et al. [162] examined multiple metrics (such as instruction counts, and last-level-cache references and misses) in a wide range of microbenchmarks, including a busy-loop benchmark (high instruction issue rate), a pointer chasing benchmark (high cache miss rate), a CPU and memory intensive benchmark (to mimic virus behavior), and a set of bubble-up benchmarks that incur adjustable amounts of pressure on the memory systems. They concluded that *non-halted cycle* is the best metric to correlate power consumption (linear correlation coefficient above 0.95). Such high correlation suggests that the higher the rate of non-halted cycles for a domain is, the more CPU power the domain consumes.

I then decided to use this result along with the data produced by *XeMPower*. The approach is simple:

A. For each tumbling window, *XeMPower* CLI calculates the power consumed by the whole socket, while *XeMPower* daemon calculates the total number of non-halted cycles (one of the PMC traced).

B. Since we have the number of non-halted cycles per domain, I estimate the percentage of non-halted cycles for each domain over the total number of non-halted cycles. This percentage is adopted as the contribution of each domain to the whole CPU power consumption.

C. Finally, *XeMPower* splits the socket power consumption proportionally to the estimated contributions for each domain.

The proposed approach works well even when CPU power states (i.e., C-states and P-states) are enabled. *XeMPower* is not affected by CPU voltage and frequency scaling, as it continues to measure the actual socket power consumption and to trace and account hardware events consistently.

Note that this is just an example of how *XeMPower* enables online attribution of coarse-grained measurements to multiple tenants on a virtualized environment, thanks to per-domain accounting of hardware events.

4.5 Experimental results

XeMPower aims to be the tool of choice for any computing system demanding precise and thorough *monitoring* of hardware events attributed to domains in Xen. Since the tool is meant to continuously provide statistics at run-time, one of its key requirements is to add negligible overhead to the monitored system. Therefore, this section empirically shows that *XeMPower* monitoring components incur very low overhead under different configurations and workload conditions. Here I define the *overhead*

metric as the difference in the system's power consumption while using *XeMPower* versus an off-the-shelf Xen 4.6 installation.

4.5.1 Experimental setup and test cases

The test platform is a machine equipped with a 2.8-GHz quad-core Intel Xeon E5-1410 processor (4 hardware threads) and 32GB of RAM. I used a Watts up? PRO meter [44] to *independently* monitor the entire machine's power consumption without being influenced by the system configuration in use.

Experiments were conducted under three system configurations: 1) the *baseline* configuration uses off-the-shelf Xen 4.4, 2) the *patched* configuration uses Xen modified as described in Section 4.3 without running the *XeMPower* daemon, and 3) the *monitoring* configuration is the same as the patched configuration but with the *XeMPower* daemon actually running and reporting statistics to an attached console. In all three configurations we assign a single virtual CPU (VCPU) and 4GB of RAM to Dom0, and also dedicate physical core 0 to it. Dedicating core 0 to Dom0, besides adhering to Xen best practices [4], results in that any computational overhead introduced by *XeMPower* monitoring phase in Dom0 can be measured as an increment in power consumption on core 0 and in the whole system.

Four runtime *scenarios* are considered: an *idle* scenario in which the system only runs Dom0, and the *running-n* scenarios, where $n = \{1, 2, 3\}$ indicates the number of guest domains in addition to Dom0. Each guest domain repeatedly runs a multi-threaded compute-bound microbenchmark[4] on three VCPUs and uses a stripped-down Linux 3.14 as the guest OS. The idea in the running-n scenarios is to stress the system with an increasing number of CPU-intensive tenant applications, thus increasing the amount of data traced by the Xen kernel and collected in Dom0.

Finally, two test cases are defined for the running-n scenarios. In the *pinned-VCPU case*, each guest domain has each VCPU assigned to a dedicated physical CPU. In the *unpinned-VCPU case*, on the other hand, the guest domains are assigned VCPUs with no physical mapping (i.e., VCPUs can migrate between physical CPUs). The idea is to increase the number of context switches and thereby the amount of traces reported to Dom0.

[4]CoEVP, a simplified proxy material science application from the ExMatEx Center. It is available at https://github.com/exmatex/CoEVP.

Table 4.2: *Mean power consumption (μ), in Watts, for the pinned-VCPU test case, scenarios idle and running-{1,2,3}, and configurations baseline (b), patched (p), and monitoring (m). Mean power values are reported with their 95% confidence interval.*

	baseline (μ_b)	patched (μ_p)	monitoring (μ_m)
idle	34.10 ± 0.05	33.72 ± 0.05	33.83 ± 0.05
running-1	56.03 ± 0.09	56.07 ± 0.11	56.19 ± 0.08
running-2	66.14 ± 0.11	66.30 ± 0.06	66.56 ± 0.09
running-3	74.62 ± 0.07	74.60 ± 0.11	74.88 ± 0.29

Table 4.3: *Mean power consumption (μ), in Watts, for the unpinned-VCPU test case, scenarios idle and running-{1,2,3}, and configurations baseline (b), patched (p), and monitoring (m). Mean power values are reported with their 95% confidence interval.*

	baseline (μ_b)	patched (μ_p)	monitoring (μ_m)
idle	34.32 ± 0.30	34.14 ± 0.08	34.19 ± 0.05
running-1	70.82 ± 0.10	71.20 ± 0.09	70.78 ± 0.10
running-2	72.99 ± 0.09	73.55 ± 0.12	73.17 ± 0.10
running-3	73.68 ± 1.09	74.67 ± 0.27	74.10 ± 0.09

4.5.2 Results and discussion

We now want to compare the power that our test platform consumes for the different scenarios and test cases under the *baseline (b)*, *patched (p)*, and *monitoring (m)* configurations. Under each configuration, the idle scenario and the running-1,2,3 scenarios are run, with and without VCPUs pinned to dedicated physical CPUs (i.e., *pinned-VCPU* and *unpinned-VCPU* test cases). The system's mean power consumption (μ) is reported in Watts over a 60-second interval. I performed a set of 40 independent experiments for each [test case, scenario, configuration] combination.

Table 4.2 and Table 4.3 present the system's mean power consumption for the *pinned-VCPU* and *unpinned-VCPU* test cases, respectively, across the considered scenarios and configurations. Empirical mean power values are reported with their 95% confidence interval.

At a glance, we can see how measurements are pretty close. However, given the limited accuracy of the power meter, some of them may seem misleading, e.g., the mean power consumption of the baseline case sometimes is higher than the others. This is the reason why we estimate an upper bound ϵ for the maximum overhead by performing the following hypothesis test [117]:

$$T(\mu) := \begin{cases} H_0: & \mu \geq \epsilon + \mu_b \\ H_1: & \mu < \epsilon + \mu_b, \end{cases}$$

Table 4.4: *Estimated upper bound ϵ for the power consumption overhead, in Watts, across the considered test cases and scenarios under the patch (p) and monitoring (m) configurations. Parenthetical values are the percentage overheads with respect to the mean power consumption.*

		$\mu = \mu_p$	$\mu = \mu_m$
pinned	idle	<0.01 (<0.02 %)	<0.01 (<0.02 %)
	running-1	0.08 (0.14 %)	0.19 (0.34 %)
	running-2	0.19 (0.28 %)	0.45 (0.67 %)
	running-3	0.01 (0.01 %)	0.34 (0.45 %)
unpinned	idle	<0.01 (<0.02 %)	<0.01 (<0.02 %)
	running-1	0.44 (0.61 %)	0.02 (0.02 %)
	running-2	0.61 (0.83 %)	0.23 (0.31 %)
	running-3	1.18 (1.58 %)	0.60 (0.81 %)

where a rejection of the null hypothesis H_0 means that there is strong statistical evidence that the power consumption overhead is lower than ϵ (or equivalently, the mean μ is lower than the baseline mean μ_b increased by ϵ). We compute ϵ for the considered test cases and scenarios, ensuring average values of power consumption (μ) with confidence $\alpha = 5\%$.

Table 4.4 shows the values of ϵ across the considered test cases and scenarios for the *patched* and *monitoring* configurations. The values in parenthesis represent the percentage overheads relative to the mean power consumption (i.e., μ_p and μ_m, respectively). These results indicate (with confidence $\alpha = 5\%$) that *XeMPower* introduces an overhead not greater than 1.18W (1.58%), observed for the [*unpinned-VCPU*, *running-3*, *patched*] case. In all the other cases, the overhead is less than 1W, and less than 1% in relative terms.

This is a satisfactory result when compared to a maximum overhead of 1-2% observed for XenMon [68], which I adopted as a reference point for the *XeMPower* implementation. This overhead can then be considered a negligible and reasonable price to pay, given the high-precision information that *XeMPower* can provide at runtime.

4.6 Related work

Performance monitoring and profiling has always been crucial in every computing system over the last 30 years [67]. The need for a constant monitoring solution has then grown, especially in virtualization environments, where the same hardware is shared between multiple tenants. Unfortunately, every monitoring tool is affected by a tradeoff between *accuracy*

and *overhead*; the effective implementation of these systems is then far from trivial. In the literature, this problem has been tackled with two different approaches: code instrumentation and performance counter monitoring.

Code instrumentation solutions, as Valgrind [124] and IgProf [51], inject extra code in the applications at compile time and/or runtime, allowing complex analysis, e.g., on memory and cache accesses. These tools are excellent for an initial analysis of errors and inefficiencies in programs, but are not suitable for performing runtime analysis in production, as the overhead introduced is often high [124].

Performance counter tools, on the other hand, focus on sampling system's events at different granularity (e.g., thread level, process level, set of processors, or the entire systems). These tools provide information on hardware utilization that may not be closely related to the application domain, but their overhead can be tuned accordingly to the actual needs [114]. They differ in functionality, data granularity, level of abstraction, and interfaces they rely on.

Low-level performance counter libraries do not hide architecture-specific event types from the user and lie directly on the hardware. Perf [5] and OProfile [102] are the most popular tools available; they make use of kernel modules to access different categories of events: hardware events, software events (context switches or minor faults), and tracepoint events (disk I/O and TCP events).

Higher-level libraries (e.g., PAPI [29]) hide micro-architecture event types behind a uniform API. They support event multiplexing to compensate for the limited number of performance counter registers that can be monitored at a time: only a subset of the desired event sets is monitored during subsections of a program's execution, then results are scaled to statistically estimate rates for the entire program.

In addition, some works in the literature focus on PMC virtualization [114, 125, 158], providing low-level metrics to virtual tenants. As *XeMPower*, all these solutions require to patch Xen Hypervisor's kernel to implement operations that require privileged access, such as reprogramming counters or setting up interrupt handlers.

In the context of Xen, the most common solution is Xenoprof [114], a system-wide statistical profiling toolkit based on OProfile and specifically crafted for the hypervisor. It is a valid solution to profile a standard workload running in Dom0 or other domains in *active mode* (i.e., the domain itself collects its own hardware event counters). However, when profiling in *passive mode* (i.e., the domain is treated as a "black box"), the results indicate which domain is running at sample time but do not delve more deeply

into what is being executed. Therefore, it does not satisfy the requirement of being agnostic to hosted applications.

Another interesting tool is Perfctr-Xen [125]. It supports performance counter virtualization in Xen for: (1) paravirtualized guest kernels, using hypercalls to communicate performance counter configuration changes to the hypervisor; (2) fully-virtualized guest kernels, using the "save-and-restore" approach for all registers; and (3) a hybrid approach that offers a tradeoff between the first two. Similar to *XeMPower*, Perfctr-Xen reprograms the Performance Monitoring Unit (PMU) configuration registers (e.g., event selectors) at every context switch. Although this tool is good for workload profiling inside a domain, it is not designed as a centralized runtime monitoring solution.

4.7 Final remarks

This chapter described the design and the implementation that led to *XeMPower*, a lightweight monitoring solution for the Xen hypervisor. It precisely accounts hardware events to guest workloads, enabling attribution of CPU power consumption to individual tenants. Results show that XeMPower introduces negligible overhead in power consumption, thus fitting the *Observe* phase of the OLDA loop discussed in Section 1.2.

The presented approach to power consumption attribution to domains is very simple, as it represents a mere example to show the tool's potential. The next chapter will present how to improve power-awareness, using *XeMPower* and *MARC* to build data-driven power models for virtualized guests.

CHAPTER 5

Modeling power consumption in multi-tenant virtualized systems

5.1 Introduction

As already introduced in the previous chapter, *virtualization* has become an extremely important tool for organizing computer systems [73, 115]. It provides a clean separation of software development concerns from the underlying hardware platform, allowing multiple guest applications to share physical resources while fulfilling needs for QoS, security, and strong isolation [10, 99, 141].

Hardware-assisted and software virtualization technologies have been developed for a wide range of platforms, from embedded systems to workstations and servers. As already stressed, *power consumption* remains an open issue for all of them, also in those contexts that do not involve batteries: for instance, data centers providers aim to reduce it as much as possible to decrease operating costs and to improve system reliability. Even though the performance-per-watt ratio has been constantly rising, the total power drawn is hardly decreasing and recent trends suggest that the cost of the energy consumed by a server during its lifetime will probably ex-

ceed the hardware cost in the near future [20]. Given the strong correlation with live operating costs, power consumption *consolidation* through applications colocation and migration becomes critical for the Cloud Computing paradigm, which delivers computing services as a utility in a "pay-as-you-go" manner [16].

Much recent work [40,41] focuses on the problem of power optimization under performance constraints, defined in terms of Service Level Agreements (SLAs) between the service provider and the tenant. For most of these approaches, an accurate *model* of tenants' behavior is a first step to guaranteeing tenants' requirements and optimizing physical resource utilization.

This chapter shows how it is possible to employ the generalized methodology described in Chapter 3 to build power models for multi-tenant virtualized systems. The basic idea remains to tackle the complexity that comes from heterogeneity identifying the *working regimes* of the system, and building a model for *each* of them. This is different from the typical approach in the literature, which uses static assumptions about hardware components to attempt to build a *single* comprehensive model of the behavior of the system [20].

The input of the model consists in a set of hardware event traces, which represent the link between hardware utilization and the overall power consumption, i.e., the output of the model: these can be easily collected using *XeMPower*, presented in the previous chapter. Again, the proposed approach is completely *data-driven*, because no knowledge about the internal components of the system is required.

I demonstrate the accuracy of the methodology with experiments in a multi-tenant virtualized infrastructure based on the Xen hypervisor [17]. Results show a relative error of 2% on average, and under 4% in most cases, which is more accurate than previous results in the literature [20, 23, 160]. Results also show that the same model can be exploited to make predictions about the impact of power consumption when migrating a domain to a different hardware platform, with little loss in accuracy.

Finally, the last section shows how these power models can be used to produce a metric of power efficiency that makes it possible to compare different colocations of tenants, as long as there exists a runtime monitoring system that is able to attribute hardware events to each domain: this is the case of *XeMPower* as extensively discussed in Chapter 4. A cluster-level scheduler could use that metric to evaluate which particular colocation leads to better consolidation from a power consumption perspective.

All the code developed to build the power models, script the tests and

Table 5.1: *RMSE and mean relative error obtained building ARX models using 3 different input features sets from the State of Art: Class A, Class B and Class C. Workload types: (a) idle, (b) weak I/O intensive, (c) memory intensive, (d) CPU intensive and (e) strong I/O intensive.*

Workload	Class A		Class B		Class C	
	RMSE	Relative error	RMSE	Relative error	RMSE	Relative error
(a)	± 17.63 W	35.56%	± 16.44 W	32%	± 17.68 W	35%
(b)	± 4.7 W	9.4%	± 5.86 W	11.7%	± 7.17 W	14%
(c)	± 19.11 W	38%	± 34.54 W	70%	± 18.7 W	37%
(d)	± 0.44 W	0.08%	± 0.6W W	1.2%	± 0.42 W	0.08%
(e)	± 2.98 W	5.9%	± 38.57 W	77%	± 3.29 W	6.5%
average	± 8.97 W	17.79%	± 19.20 W	38.38%	± 9.45 W	18.52%

validate the approach has been released open source [1] and it aims to be a reference for future work in the field of virtualized, power-aware systems.

The chapter is organized as follows: Section 5.2 presents a preliminary exploration on how different workload patterns impact on power consumption, baring the limitations of the most common power modeling approaches; Section 5.3 gives a complete overview of the proposed approach, from the high level flow to the most relevant implementation details; Section 5.4 provides a comprehensive description of which experiments have been conducted to validate our methodology and how, while results are discussed in Section 5.5. Finally, Section 5.6 discusses related work and Section 5.7 concludes.

5.2 Motivational example

In order to be effective, consolidation techniques need to estimate the impact of a workload on the power consumption of the system. As many works in the field show how the impact on the overall power consumption is highly related to the workload hosted [23, 145, 160], I started this analysis with an exploration on how workload heterogeneity impacts on power consumption.

A synthetic benchmark has been built to reproduces a sequence of workloads with different characteristics, thus simulating a job composed of different phases:

[1] All source code, scripts, inputs, and patches are available at: https://bitbucket.org/paperblindauthor/2017-powermodels

A. **IDLE phase**: the benchmark has not started yet;

B. **I/O Load phase**: the benchmark loads a great amount of data into the main memory (I/O intensive task);

C. **MEM Preparation phase**: the benchmark prepares the data just loaded to be processed (memory intensive task);

D. **CPU Computation phase**: the benchmark runs an intensive computation on the prepared data (CPU intensive task);

E. **I/O Store phase**: the benchmark stores the results into persistent storage (I/O intensive task);

F. **IDLE phase**: the benchmark terminates.

Figure 5.1 shows an example of the power trace obtained from the power meter connected to the server (equipped with a 2.8-GHz quad-core Intel Xeon E5-1410 processor and 32GB of RAM), while the synthetic benchmark was running. Different phases are reported with different colors, thus highlighting strong differences of power consumption between different workload classes [2].

The experiment has been repeated multiple times, exploiting architectural Performance Monitoring Counters (PMCs) to collect traces of hardware events. These traces represent low level metrics of hardware utilization and have been extensively used in literature to build power models [20]. Since the preliminary work of [23], some works uses hardware events as input of a linear model to explain the behavior of a generic workload [53], while others realized that such a simple model is not enough to tackle more complex scenarios [45]; further works introduce the concept of "intensiveness", thus using different linear or ARX models for different classes of workload [145, 160].

Inspired by these contributions, we built a different ARX model for each prevalent "intensity" we observed in the aforementioned synthetic benchmark, thus identifying 5 different working states of the system: (a) idle, (b) weak I/O intensive task, (c) memory intensive task, (d) CPU intensive task and (e) strong I/O intensive task.

Then, we resorted to the literature to choose the subset of hardware events that better correlates with power consumption [23, 160], thus choosing the following 4 events: `INST_RET`, `UNHALTED_CLOCK_CYCLES`, `LLC_REF`, `LLC_MISS`. In order to find the most satisfactory subset of

[2]The same figure has been used in Chapter 3, while discussing the resource consumption problem.

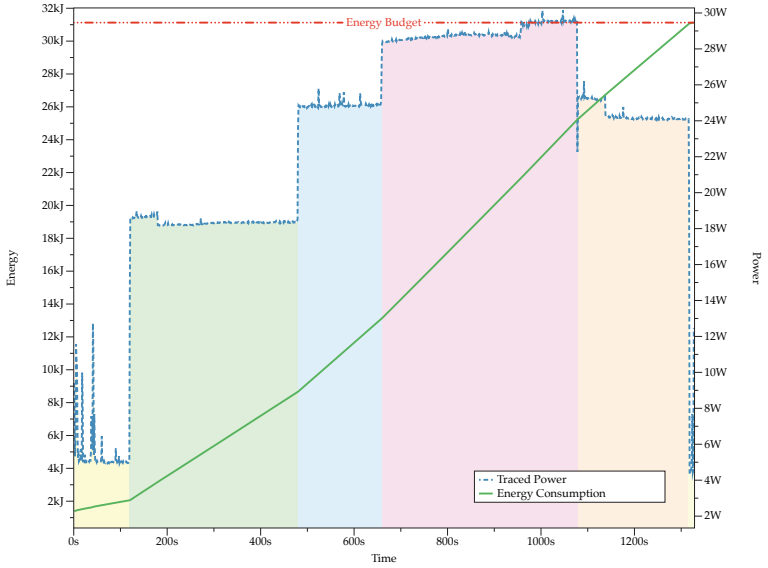


Figure 5.1: *Example of a power trace obtained by running a workload that goes through the following phases: (a) idle, (b) weak I/O intensive task, (c) memory intensive task, (d) CPU intensive task and (e) strong I/O intensive task*

input features for these preliminary models, we computed all the $2^4 - 1$ combinations of these events. The better performing and most interesting subsets were used to build the following classes of models:

A. **Class A**:
   $input \in \{$`INST_RET`, `UNHALTED_CLOCK_CYCLES`, `LLC_REF`, `LLC_MISS`$\}$

B. **Class B**:
   $input \in \{$`INST_RET`, `UNHALTED_CLOCK_CYCLES`, `LLC_REF`$\}$

C. **Class C**:
   $input \in \{$`UNHALTED_CLOCK_CYCLES`, `LLC_REF`$\}$

We built a model for each combination: *<Model Class, Working State>*, thus reporting results on models performances, in terms of RMSE and relative error, in Table 5.1.

These preliminary results led us to the following outcomes: (1) we observed a good correlation between hardware events and system power consumption, as extensively discussed in literature; it is then reasonable to use them to build power models, even though (2) a single ARX model is not

able to achieve good modeling performance with different workload types (e.g., CPU intensive, memory intensive, weak and strong I/O intensive). However, (3) the hypothesis of the existence of different working states still holds, as showed in Figure 5.1.

5.3 Proposed methodology

5.3.1 Overview

Starting from the outcomes of Section 5.2, this section discusses how to tackle the modeling complexity through *working regimes* identification. Figure 5.2 gives an overview of the steps of the modeling pipeline:

A. at first, we stress the hardware with a representative set of *benchmarks*. These have to cover the highest number of computational patterns in order to observe the different working regimes that characterize the system;

B. then, we need a data-driven approach to regimes *identification*, since results in Section 5.2 show how the definition of "prevalent intensity" proposed in the literature may lead to dramatic performance disparities while modeling different situations. We propose an approach *a posteriori*, that aims at identifying regimes from the observed power trace using a clustering technique;

C. once we identified a set of regimes, we need to correlate them with input hardware metrics, thus training a *regime classifier*. This will then be used *a priori* to identify the right regime at runtime;

D. finally, an ARX *model* is built for each working regime classified. These models, used in conjunction with the classifier, constitute the final power model of the whole system, as showed in Figure 5.3.

At runtime, we account for a certain rate of hardware events to the specific domain; these triggers the classifier, that chooses the right power model for the regime identified; finally, hardware events are given in input to the right power model, that is finally used to estimate the actual power consumption $\hat{P}_{single}$ of the system.

This section continues with some theoretical details on the techniques adopted, as well as some insights on how they have been implemented to produce the results we present in the last sections to validate the proposed methodology.

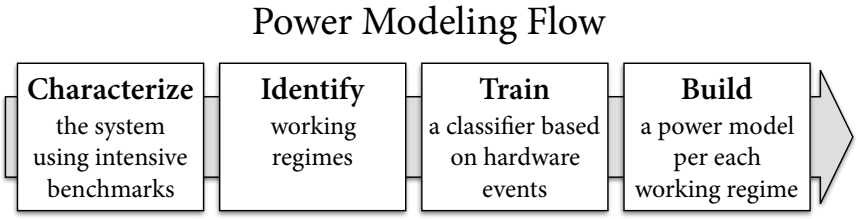


Figure 5.2: *Overview of the power models generation procedure*

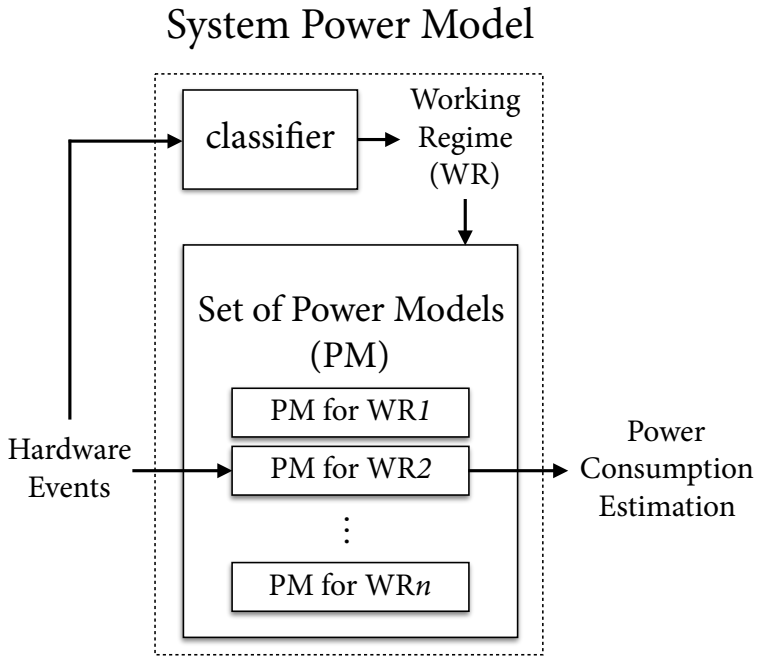


Figure 5.3: *Overview of how the final power model of the system is composed*

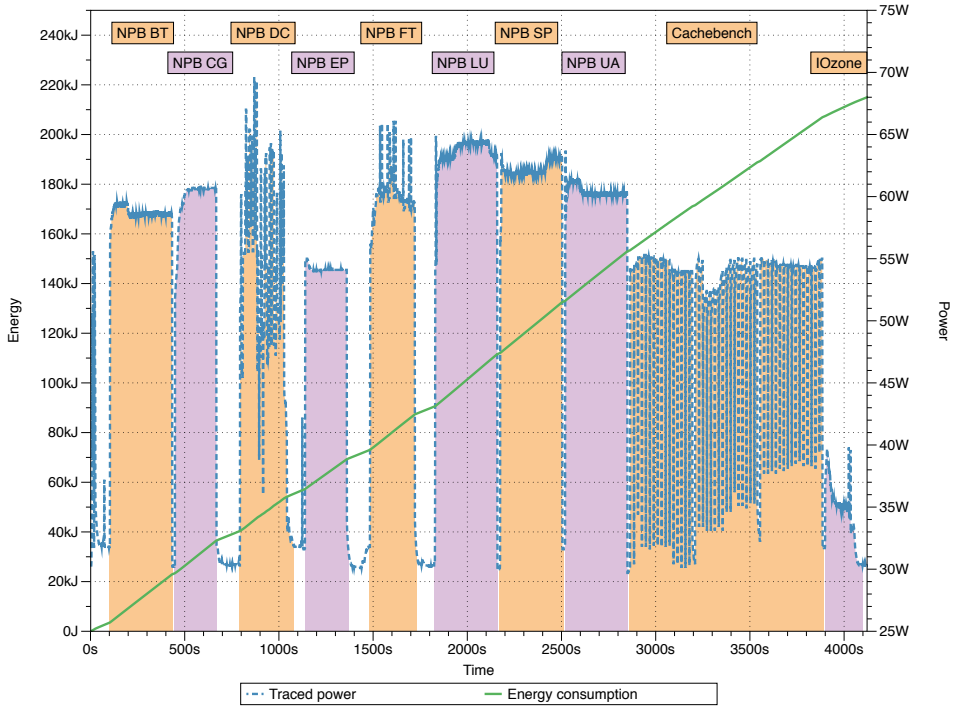


Figure 5.4: *Example of a power-energy trace obtained running the chosen benchmarks on a low-range server machine*

5.3.2 System benchmarking

We introduce a set of micro-benchmarks to stimulate the hardware, narrowing down to single aspects (e.g., CPU performances, memory latency, etc.), thus producing all the different regimes of power consumption of the physical machine.

We chose the NAS Parallel Benchmarks (NPB) benchmark suite [46], that has proven its ability to stress the greatest part of the characteristics of CPU and RAM with a mix of 8 highly targeted benchmarks [160], together with Cachebench [118] and IOzone [85] to stress the bandwidth and the latency of cache hierarchies and I/O hardware by transferring large streams of data. A sample run is showed in Figure 5.4.

No benchmark stresses any kind of network capability: in a virtualized environment, we realized that network can be effectively benchmarked only if some newly discovered and rarely available expedients are in place (multi-queue NICs with tenants direct access to a dedicated queue, no NATting, specialized and modified privileged drivers, etc.). Without these en-

hancements, any benchmark would measure only the hypervisor's overhead in handling virtual networking, instead of the real impact of virtual networking itself [55].

5.3.3 Working regimes identification

The goal of this section is to identify a reasonable number of working regimes over the power traces obtained in Section 5.3.2. We then need to split a mono-dimensional data set into intervals. A simple though meaningful procedure is: (1) estimate the distribution from which the data set has been sampled, (2) find all the local minima of the estimated probability density function and (3) split the data set into intervals using local minima as boundaries. The rationale behind this procedure is that each obtained interval will contain highly "packed" values, thus pointing out the presence of a steady state (i.e., a working regime) of the system that lies inside that interval.

We then resort to a class of algorithms and procedures for monodimensional interval generation: these derive from the specialization of the more general clustering algorithms to the subcase of mono-dimensional data sets. The clustering is then applied on a single feature in the entire data set, i.e., the range of power consumption values. We chose to use a simple though extremely powerful technique based on KDE [136], a non-parametric statistical method to estimate the probability density function of a sampled random variable.

To help the reader understand how this technique fulfils the task of working regimes identification, we: (1) passed the power trace through KDE, obtaining the associated probability density function as in Figure 5.5; (2) identified two local minima at 42W and 57W; (3) split the data set into 3 intervals using the two local minima identified (bottom of Figure 5.5).

5.3.4 Working regimes classification

Once we identified the values of power that delimit each regime, we need to infer this classification directly from the *input features*, i.e., hardware events collected during the experiments. For this purpose, we use *ReliefF* [98], a heuristic, distance measure based, supervised classification algorithm that presents a good trade-off between time complexity and accuracy [38].

Figure 5.6a shows the weights computed by a first round of ReliefF for each input feature. Features names, i.e., hardware events codes, are omitted

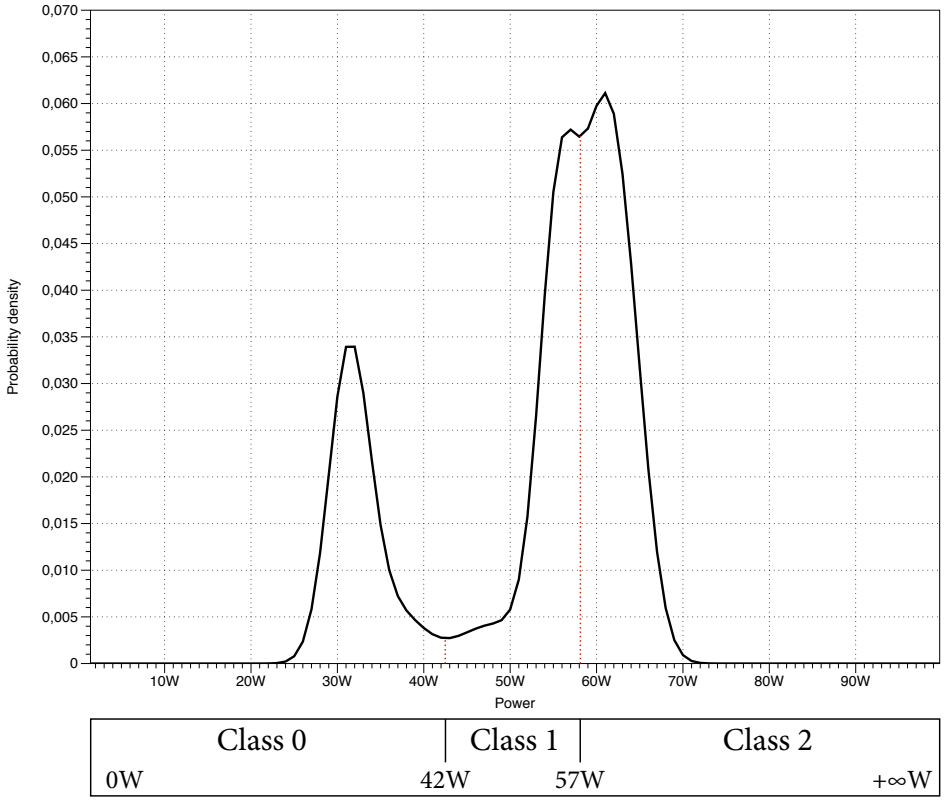
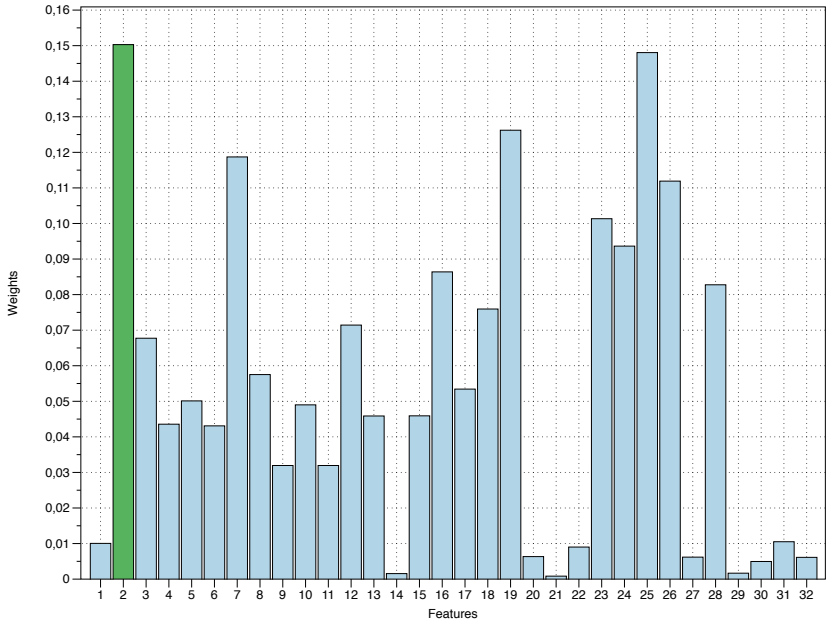
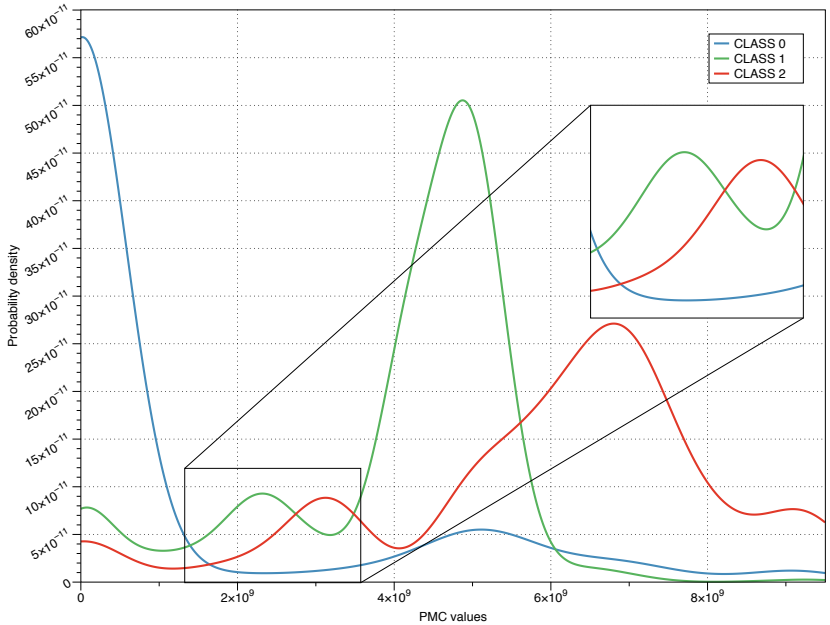


Figure 5.5: *Example of power distribution through KDE analysis; local minima identify classes of power consumption*



(a) *ReliefF results for each input feature; a highest weight indicates a more important feature for classification*



(b) *Probability distribution of the values of Feature 2 for each class, with the range of uncertainty highlighted*

Figure 5.6: *First iteration of the ReliefF algorithm on the current example: Feature 2 (i.e., `INST_RET` hardware event) represents a good candidate feature but it is not enough to perform a good classification per se*

Table 5.2: *Final ruleset for the regimes classifier for the current example*

	INST_RET	MEM_LOAD_ UOPS_RET_L1_HIT
CLASS 0	$[0, 1.235 \times 10^9]$	
CLASS 1	$[3.61 \times 10^9, 5.58 \times 10^9)$	
	$(1.235 \times 10^9, 3.61 \times 10^9)$	$[2.36362 \times 10^8, 5.672 \times 10^8]$
CLASS 2	$[5.58 \times 10^9, +\infty)$	
	$(1.235 \times 10^9, 3.61 \times 10^9)$	$[0, 2.36362 \times 10^8) \cup (5.672 \times 10^8, +\infty)$

to improve readability, as this is meant to be a mere example to help the reader understand the proposed methodology.

Feature 2, that corresponds to the event: `INST_RET`, presents the highest weight, thus being the more important feature for classification. However, this feature alone seems not always sufficient to discriminate among the three classes identified: Figure 5.6b, that shows the probability distribution of the values of this feature for each different class reconstructed using KDE, highlights how there exists at least a range of values for this event for which it is not possible to perform a good classification.

This uncertainty can be solved with another iteration of ReliefF on the remaining features considering just the range of uncertainty: as we remove Feature 2, we have that Feature 25 (i.e., hardware event: `MEM_LOAD_UOPS_RET_L1_HIT`) can help to obtain a good discrimination among the uncertain classes when Feature 2 is not enough to perform an accurate classification.

This process can be repeated multiple times, until enough features have been chosen to guarantee good classification performance. For each chosen feature, we have as outcome a range of PMC values, that can be used to identify each different working regime of the system *a priori*. Table 5.2 shows the outcome for the example discussed in this section.

5.3.5 Power models generation

Once we obtained the ruleset that defines a *classifier* for the working regimes of the system, we can build a power model for each regime, as discussed in Chapter 3.

We start from raw data recorded during the execution of the benchmarks described in Section 5.3.2. These data consist in a time series of power measurements and counters of hardware events related to the last time interval: the former represent the *output feature* f_r, while the latter ones are

the *exogenous input features* f_x of the model. The classifier allows to split the dataset into `batches`, i.e., a group of samples that relates to the same regime. The *training set* for each model is then obtained grouping together batches of the same regime.

At first, raw traces are conditioned: Model Analysis for Resource Consumption (MARC)'s `PHASE1` coherency correction was exploited to correct some data inconsistencies, such as counter overflows; moreover, we translated the classifiers into *feature fusion* rules, labeling each sample with a working regime identifier; then, we configured MARC's `PHASE2A` to generate an ARX(0,1) model for each working regime; finally, MARC's `PHASE3PRE` was exploited to evaluate power models produced by `PHASE2A`, given different initial conditions. As a bottom line, thanks to the many mathematical performance metrics exported by MARC, we had no need to further implement any accessory script for data analysis.

For the case study under analysis, we explored different values of ar and ex lags and we found out that a lag of 1 step for the auto-regressive component and a lag of 0 steps for the exogenous input generally lead to the best modeling performance. We are then implicitly assuming that the power consumption of the system is highly dominated and depends almost entirely on the actual system load.

5.4 Experimental Evaluation

5.4.1 Objectives

The experimental evaluation discussed in this section has three main goals: (1) assess the precision of the power modeling methodology, (2) explore the performance of a model on different hardware platforms and (3) show how the proposed models can be exploited to help a scheduler evaluate a colocation with respect to another one in terms of power efficiency, towards a more effective consolidation on the server farm. The following paragraphs highlight some important details on these goals that has to be kept into consideration through the rest of the chapter.

Model performance

The same methodology can be used to produce two different types of models: generic and specific.

On the one hand, the *generic model* is produced using of a set of micro-benchmarks as discussed in Section 5.3.2. This model is bounded to the

physical machine and can be employed to make predictions on a generic workload that has never been hosted and monitored beforehand.

On the other hand, the *specific model*, makes use of a classifier that has been built ad-hoc for the specific workload. This generally leads to a lower *missclassification* of the working regimes of the system, where misclassification is defined as the fraction of samples that the classifier categorizes in a wrong working regime. Even though we experienced that a high misclassification may not influence the performance of a model, a low misclassification always leads to an improvement of the performance of the model. Of course, the drawback of a specific model is the need for a training dataset to build the specific classifier: we then need to observe the execution of the workload on the physical machine to collect execution traces.

At first, results aim to measure the precision of each generic model built (one for each physical machine considered). Then, we want to measure how a specific power model built for a single tenant can improve the performance of the generic model.

Model portability

As tenant migration is a common practice in a cloud computing infrastructure, *model portability* from one physical machine to another is a desired feature. We then want to measure the error we face when we try to exploit a generic model of a machine to make prediction on power consumption on different machines. This could be the case of new servers that has just been added to the cluster.

Consolidation evaluation

We finally want to show how the power models produced can be used to evaluate the power efficiency of a specific colocation of tenants, to which we will refer to as *colocation configuration*. A colocation configuration defines a certain amount of resources given to each tenant: it may or may not share the same socket with other tenants, or even the same physical core, etc.

Given a certain resource allocation, we observe a certain rate of hardware events accounted to each specific domain: these can be used as input of a power model to estimate at runtime the power consumption $\hat{P}_{single}(i)$ of the same tenant i if it were running in isolation on another physical machine. The same estimation $\hat{P}_{single}(i)$ can be obtained of for each tenant running in the system. Then, the total power consumption that would be required to run the same k tenants on isolation on different machines can

be estimated as follows:

$$\hat{P}_{single} = \sum_{i=1}^{k} \hat{P}_{single}(i) \tag{5.1}$$

If we compare $\hat{P}_{single}$ with the real power consumption $P_{colocation}$ measured on the system with the current colocation configuration c, we can infer a metric to evaluate the improvement on power efficiency as follows:

$$improvement_c = \frac{\hat{P}_{single} - P_{colocation}}{P_{colocation}} \tag{5.2}$$

With $k > 1$, this metric will always be greater than zero, as each $\hat{P}_{single}(i)$ considers the base power of the physical machine while tenants colocation allows to distribute that power overhead on multiple tenants. Then, this metric is interesting when comparing different colocation configurations c, to identify the ones that allow a better power consolidation.

5.4.2 Experimental setup

In order to achieve the aforementioned goals, we need to explore different system configurations to explore how heterogeneity impacts on power consumption in a multi-tenant cloud computing infrastructure. We can define at least 3 levels of heterogeneity: (1) *hardware-level*, i.e., the specific hardware platform analyzed, (2) *workload-level*, i.e., the specific workload type considered, and (3) *colocation-level*, i.e., the number of different workloads concurrently sharing the same resources. As an exhaustive exploration of all the combinations of these levels of heterogeneity may be unfeasible, we now try to set some assumption to cover a reasonable amount of experiments to prove the validity of the methodology proposed in Section 5.3.

Hardware setup

For what concerns hardware heterogeneity, it is quite common that different physical machines take part in the same cluster, as new families of servers are released every year and the replacement of old machines on a server farm is performed gradually. Moreover, server platforms may be configured differently in terms of number of processors, amount of memory available and so on. Thus, our experiments have been conducted on the following 3 physical machines:

A. **WRK** a Dell OptiPlex 990, equipped with one Intel Core i7-2600 @ 3.40GHz and 8GB DDR3 RAM; this represents an old, but still quite common general purpose machine;

B. **SRV1** a Dell PowerEdge T320, equipped with one Intel Xeon CPU E5-1410 @ 2.80GHz and 16GB DDR3 RAM; this represents a recent low-range server machine;

C. **SRV2** a Dell PowerEdge T630, equipped with two Intel Xeon E5-2650 v3 @ 2.3GHz and 128GB RAM DDR4; this represents a recent mid-range server machine;

Experiments have been performed with and without Simultaneous Multi-Threading (SMT) enabled. Intel TurboBoost has been left disabled in all our tests, as it tends to cause unpredictable behaviors [9, 106].

Moreover, we do not want to assume to have the same power measurement sources available over the whole cluster: we may have measurements at different granularities, e.g., rack-level, server-level or even socket-level power measurements, and with a different sampling frequency. Our experimental setup makes use of 2 different sources:

A. **Intel RAPL interface**: a set of Model Specific Registers (MSRs) that Intel provides on its processors (since Sandy Bridge 2nd generation [138]) to get CPU power measurements with a time granularity of 1ms approximately; more specifically, we are interested in the `PKG` MSRs, i.e., the ones related to the whole socket power consumption;

B. **WattsUp power meter** an external power meter [44] that is meant to be placed between the power source and the machine; it logs power consumption every 1s approximately.

Workload setup

Two virtual CPUs are assigned to each domain, that runs a minimal Debian distribution. These vCPUs are pinned onto two physical cores, in order to improve performance stability and to limit scheduling overheads. Each virtual tenant hosts a single workload.

A workload can be classified by its predominant behavior, that may fall in one of the following 3 coarse-grained *workload categories*: (1) CPU-intensive, (2) memory-intensive and (3) I/O-intensive. Even though no real-world application fits exactly one and only one of these categories, we chose some representative benchmarks that are widely adopted in cloud environments and we group them by predominant observed behavior:

A. two compute-intensive algorithms like Support Vector Machines (SVM) and PageRank implemented on Apache Spark, a fast and general engine for big data processing [61], and a stress benchmark for Redis, an in-memory data structure store [134];

B. two representative benchmarks for MySQL [36] and Cassandra [60], to stress a SQL and a NoSQL DataBase Management System (DBMS);

C. a last benchmark on FFmpeg [35], an audio-video processing tool suite.

Colocation setup

Our last experiments involved the colocation of multiple workloads on the same system. For both single socket and dual socket machines, we run an increasing number of tenants up to the maximum number that could be placed without assigning the same physical core to different workloads. We evaluated both the colocation of *homogeneous* tenant, thus running multiple instances of the same workload, as well as of *heterogeneous* tenants, i.e., tenants that present predominant behaviors that fall into different workload categories.

5.4.3 Models and results generation

Given the large amount of experiments that can be conducted, we produced a set of scripts to automate the tests, collect data, build power models and assess their precision. This subsection gives an idea of the tests structure, while scripts have been released open-source and can be downloaded from the same repository linked at the beginning of this chapter. This allows researchers to reproduce the same results and enables performance comparison with works in the fields.

Each script starts setting the system in the most clean and stable state possible. Depending on the testing configuration to be examined within the current experiment, the script launches a set of domains hosting the workloads of interest. In order to deal with workloads' settling periods, we wait for 60 seconds between each workload activation. The same happens on workload deactivations.

5.5 Experimental results

This section discusses all the aspects presented in Section 5.4. Before going through the analysis of the obtained results, it is important to notice that on WRK we explored only a subset of the realistic benchmarks we selected for validating this work. This is due to the limited computational capabilities offered by the WRK machine, which is far from being a representative of a cluster node – i.e., highly distributed environment to which Cassandra, PageRank and SVM belong.

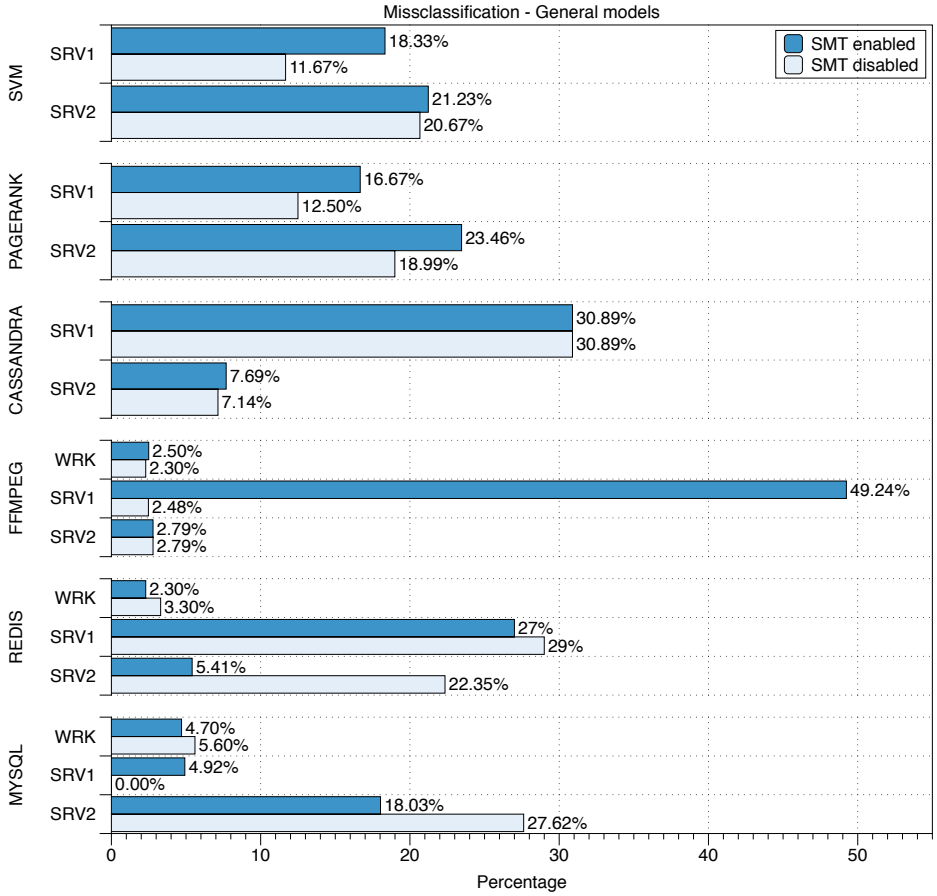


Figure 5.7: *Misclassification results of the general model for different workloads, hardware features and platforms (lower is better)*

5.5.1 Model performance

Figure 5.7 shows an evaluation of the *misclassification* metric for each benchmark and for each hardware platform analyzed, using the *general model*. Misclassification percentage is reported both with SMT enabled and disabled. In the vast majority of the cases, we observe fairly similar results within each benchmark, apart from two outstanding situations.

The first one, FFMPEG on SRV1 with SMT enabled, reports a misclassification rate of almost 50%; this is related to some design choices taken while defining the classifiers for this work. The SRV1 classifier shows its greatest weakness in labeling Class 1 samples. This limitation could be easily mitigated by leveraging another PMC for better tuning the classifier

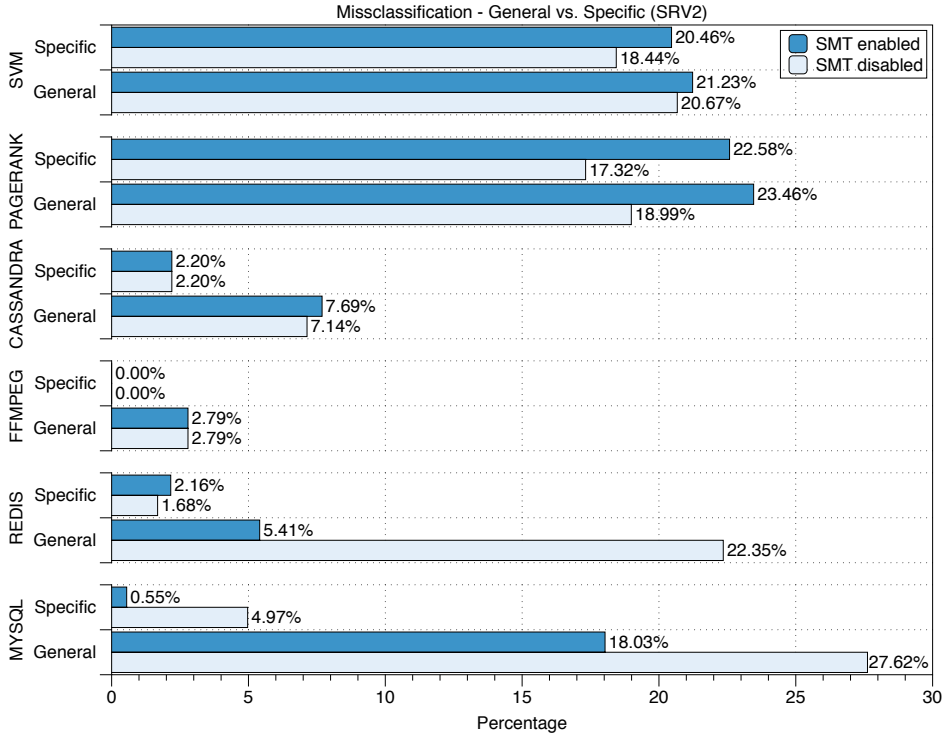


Figure 5.8: *Missclassification results of the specific model compared with the general one for different workloads and hardware features on SRV2 (lower is better)*

rules as explained in Section 5.3.4. However, we deliberately decided to use the same number of PMCs for every classifier for two reasons: (1) to have better comparability among different results and (2) to implement a technique advisable in real-world environments, where the generation of the classifier is the same on all the machines.

The other outlier, MySQL on SRV1 with SMT disabled, clearly shows that even a general classifier is able to perform virtually perfectly in some cases. We suppose that workloads with almost perfect classification are the ones that show better *affinity* with the hardware platform they run on, meaning that they behave *as expected* when analyzing the machines from a workload-agnostic point of view.

Moreover, we want to show how a *specific model* can improve working regimes classification. Figure 5.8 compares misclassification obtained for general and specific models for each benchmark and with both SMT enabled and disabled on SRV2. Similar results have been obtained on all the hardware platforms considered and are not reported here due to space lim-

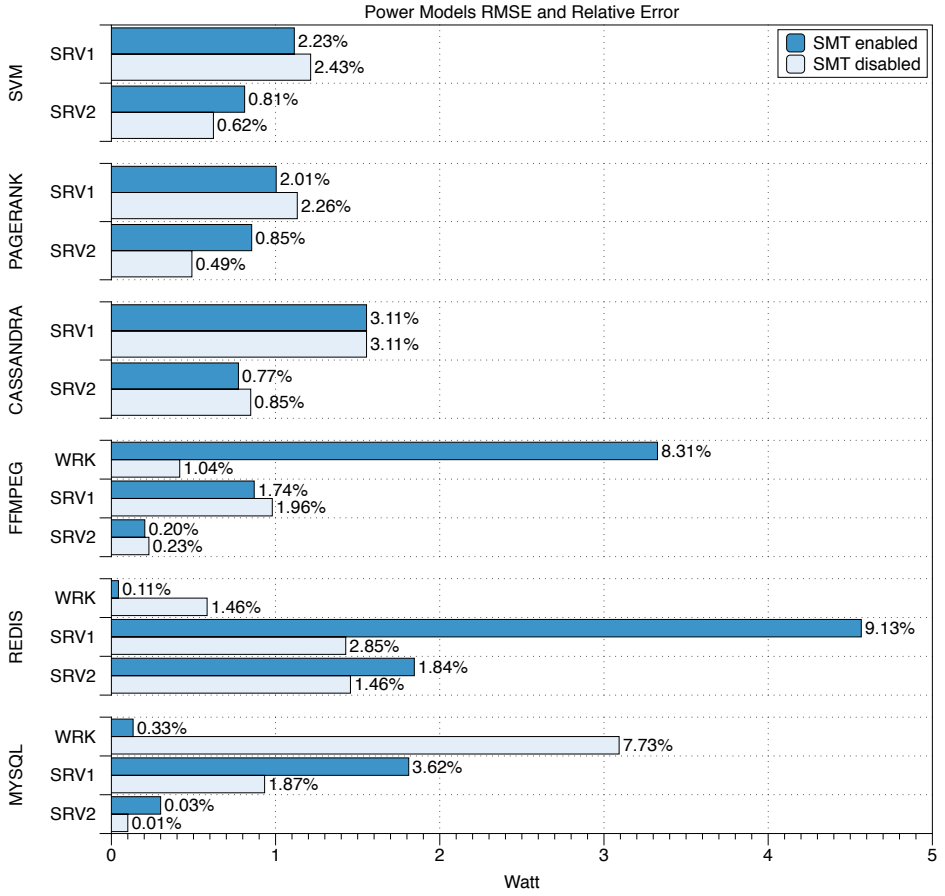


Figure 5.9: *Power models performances, in terms of RMSE and relative error, for different workloads, hardware features and platforms (lower is better)*

itations. As expected, specific models perform better than general ones in all the explored configurations. It is worth noticing that FFMPEG, which showed very poor results with some of the generic models, is almost perfectly classified with a specific one. This is related to the fact that this and other similar workloads usually work on a very stable and definite regime, which might not be identified correctly during a generic classifier training. Using the same methodology but targeting these workloads specifically allows to build extremely good classifiers that can overcome the limitations of generic ones.

Finally, the last results aim to measure the estimation error of the power models built using the proposed methodology. For each benchmark and for each hardware platform considered, Figure 5.9 shows the RMSE evaluated

Table 5.3: *Model portability RMSE (Watts) with MYSQL workload*

Train \Test	WRK	SRV1	SRV2
WRK	0.13	1.15	
SRV1	17.83	1.81	17.15
SRV2		3.84	0.30

Table 5.4: *Model portability RMSE (Watts) with the REDIS workload*

Train \Test	WRK	SRV1	SRV2
WRK	0.58	1.42	
SRV1	0.22	1.42	2.39
SRV2		9.33	1.45

Table 5.5: *Model portability RMSE (Watts) with the FFMPEG workload*

Train \Test	WRK	SRV1	SRV2
WRK	3.32	0.85	
SRV1	3.34	0.86	0.76
SRV2		0.98	0.20

with SMT enabled and disabled, respectively. The length of the bars represents the RMSE (in Watts), while the mean percentage error is reported on top of the bars. The mean percentage error is evaluated with respect to the range of the *dynamic power* of the machine, obtained as peak power minus base power. This error metric has been reported as it is a common and stable metric to compare these results with the ones proposed in the literature. Results show how the proposed methodology guarantees good modeling performance in all the cases, as:

A. RMSE is around 1W on average and under 2W in almost all the cases; only three results present a worse behavior, that is still under the reasonable limit of 5W;

B. relative error is around 2% on average and under 4% in almost all the cases; still, the three worse results present an error that is fairly under 10%.

These results then generally outperform the ones obtained in literature [20, 23, 160], even in the worst cases.

5.5.2 Model portability

Model portability results for MySQL, Redis and FFMPEG are showed in Table 5.3, Table 5.4 and Table 5.5 [3]. Cells contain the RMSE evaluated

[3]Due to space limitation, we are not reporting the Cassandra, SVM and PageRank results, as they cannot be analyzed on the complete set of hardware platforms considered in this work.

using the general models of the platform of the corresponding row on a benchmark executed on the platform specified in the corresponding column.

As expected, the best results are reported on the main diagonals, as these models are built stressing the specific platform, thus being highly dependent on the underlying hardware. However, these results show how it is possible to use the model of a machine to predict the power consumption of a workload on another "similar" machine (in terms of hardware specifications), for which a power model has not been built yet. In this context, two platforms are considered *similar* if, for instance, the processor's microarchitecture is the same (e.g., it is the case of WRK and SRV1, both based on Intel Sandy Bridge 2nd generation microarchitecture) or the processor's model is the same (e.g., both SRV1 and SRV2 are equipped with an Intel Xeon processor, even though they are based on a different microarchitecture). It is then more reasonable that similar machines present a similar power behavior: this is the reason why we are not interested in exploring model portability between WRK and SRV2.

Moreover, a comparison between modeling errors gives a metric to determine energetic similarity of two machines with respect to the specific workload under exam. For example, central rows in Table 5.3, Table 5.4 and Table 5.5 show how SRV2 and WRK present an almost identical behavior when running MySQL while SRV2 is more efficient than WRK when running REDIS. This information can be very useful to balance power consumption in a cluster even when nodes are not identical.

5.5.3 Consolidation evaluation

In this section, we discuss some colocation configurations that present interesting trends in power consumption scalability through consolidation. Using the metric we defined in Section 5.4.1, a scheduler would be able to evaluate which colocation configurations are worth exploring according to its specific power saving policies. As we want to analyze only the results produced on server platforms, we avoid discussing colocation on WRK.

Figure 5.10 shows a colocation analysis on SRV1. Power measurements are provided by the external power meter. Due to hardware limitations, only two workloads can be colocated together. As expected, most of the colocations have an improvement metric around 1: this means that, at least, any colocation would allow to save roughly the power consumption of one dedicated machine running one of the workloads.

Moreover, we can see how there are some colocations that are more power efficient than others, such as `MYSQL+MYSQL` or `SVM+CASSANDRA`.

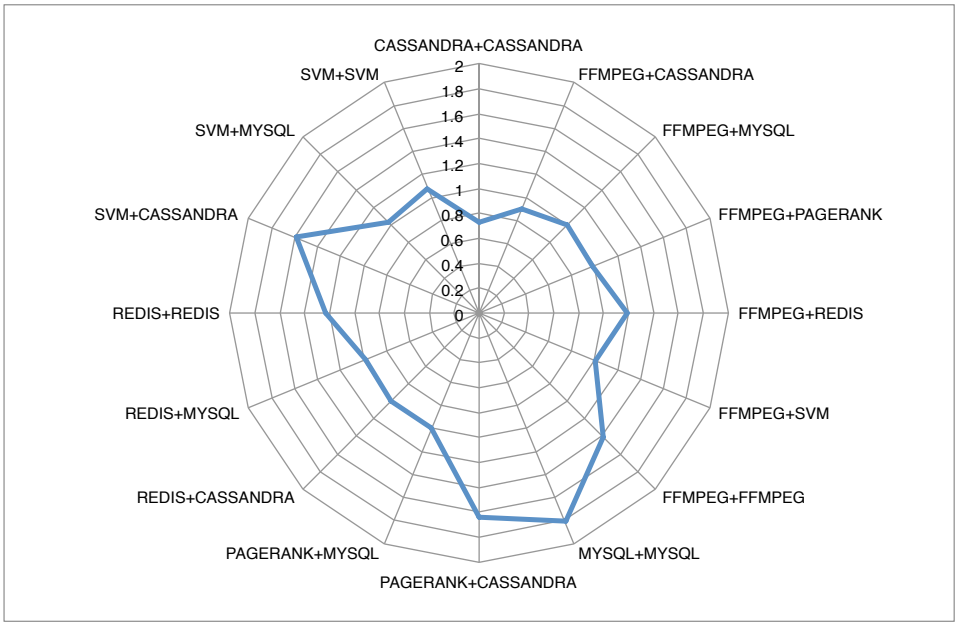


Figure 5.10: *Improvement metrics for different colocations on SRV1 (higher is better)*

A configuration like `SVM+CASSANDRA`, for instance, is efficient because these two workloads have completely orthogonal needs: the former is dominated by CPU performance while the latter mainly stresses the memory; colocating them would allow a scheduler to avoid having an unbalanced resource utilization, splitting the fixed cost of base power among non interfering tasks.

Figure 5.11 shows a similar analysis on SRV2, which can host more workloads even on one single socket. It is worth noticing how the `SVM+CASSANDRA` colocation, that was one of the best achievable on SRV1 (improvement of 1.6), turns out to be one of the worst performing in this case (improvement around 1), if compared to other colocations: a scheduler could then colocate these two workloads if deployed on SRV1 and separate them if deployed on SRV2.

Last results aim to exploit both the socket of SRV2. As the number of colocations possible increases exponentially, we grouped *homogeneous* workloads in Figure 5.12 and *heterogeneous* workloads in Figure 5.13.

Starting from Figure 5.12, we notice how increasing the number of colocated workloads often leads to a greater power efficiency. However, each workload shows its own specific marginal power saving. For example, colocating an increasing number of `CASSANDRA` instances is more efficient

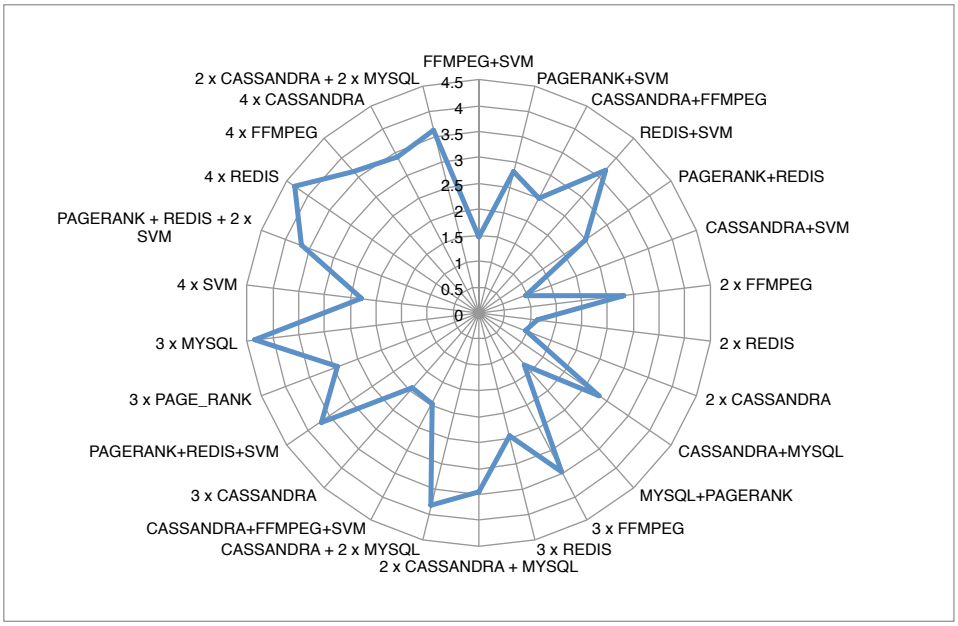


Figure 5.11: *Improvement metrics for different single-socket colocations on SRV2 (higher is better)*

than doing the same with `FFMPEG` instances.

For what concerns heterogeneous colocations in Figure 5.13 [4], we notice how a good balance of homogeneous subsets in an heterogeneous colocation seems a key factor to achieve good power savings. This is the case of `2C+3F+M` and `C+4F+M`: they both are colocations of 6 tenants picked from three workload types; colocating more `FFMPEG` tenants leads the improvement factor to rise sensibly from 6 to 8. However, another 6 tenants colocation like `3F+P+R+S` presents an improvement factor around 5, highlighting again how the choice of the colocated workloads is critical.

Finally, a comparison between Figure 5.12 and Figure 5.13 shows how heterogeneity is fundamental to reach the highest efficiency possible. For instance, a colocation of 6 homogeneous tenants like `6xFFMPEG` achieves a lower improvement with respect to a similar colocation like `C+4F+M` (i.e., 6 with respect to 8).

In all the cases discussed, as well as in all the others that can be explored, a scheduler is not supposed to understand the behavior of the workload: it can just rely on the improvement metric, treating tenants as "black boxes".

[4] In these last results, workloads' names are replaced by their initial letter to improve chart readability.

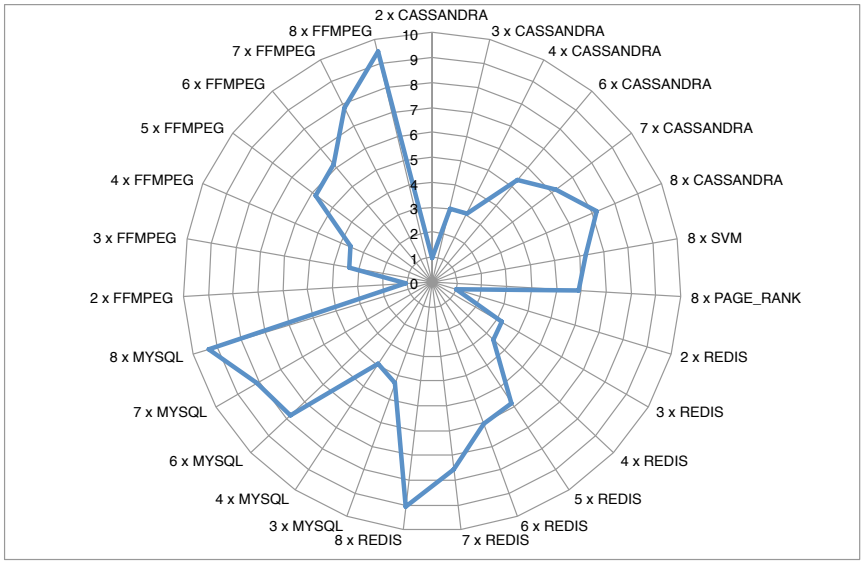


Figure 5.12: *Improvement metrics for different multi-socket homogeneous colocations on SRV2 (higher is better)*

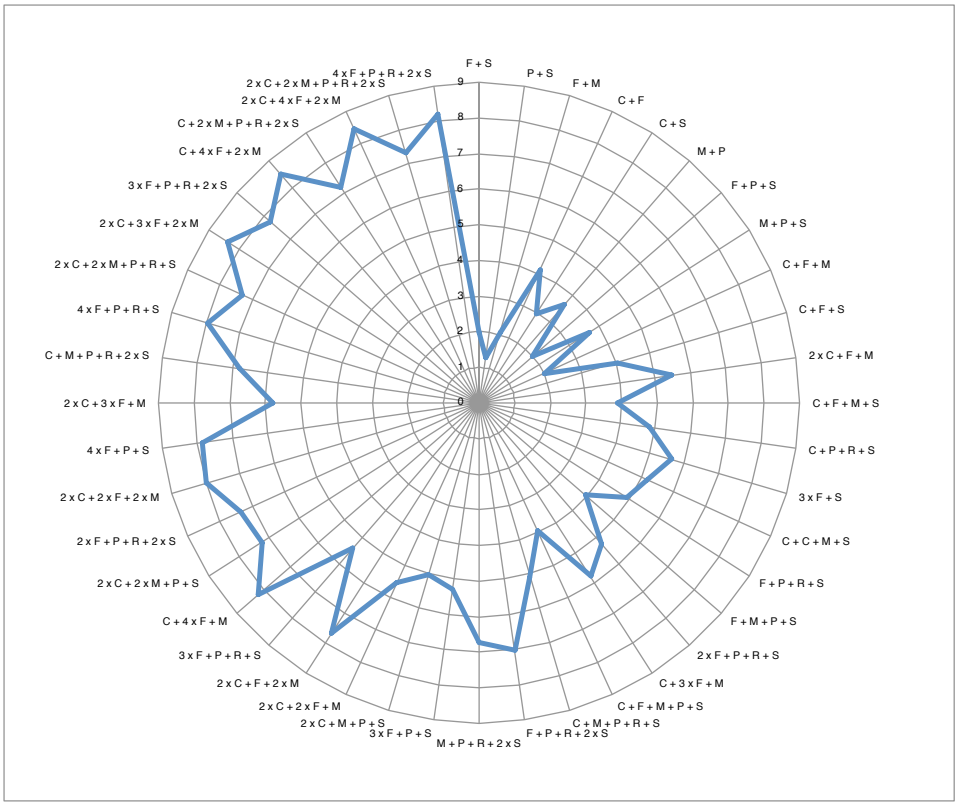


Figure 5.13: *Improvement metrics for different multi-socket heterogeneous colocations on SRV2 (higher is better)*

5.6 Related work

There is a solid state of the art regarding how to build power models for physical machines.

Studies in this field started around 2007 with the influential work by *W. Lloyd Bircher and Lizy K. John* [23]. In this work, the authors' aim was to model the power consumption of a physical machine and not only of the microprocessor, like all the previous studies in the field [19, 24, 86, 101, 104]. Using a subset of the embryonic PMC of the Intel Pentium IV processor, they managed to obtain linear regression models of the energy consumption caused by each subsystem, with less than 10% of relative error. Most important, this early work foresaw the necessity for power accounting over virtual tenants.

Few years later, the rising research community on Cloud Computing started producing interesting methodologies to solve this issue. In 2009, *A. Kansal et al.* proposed *Joulemeter* [90]. The main driver that pushed the authors in developing this power metering tool was that, in traditional systems, the visibility of the OS on the workloads had always been exploited to *"make automated and manual power management decisions"*, but, with the *isolation property* enforced by virtual machines, this visibility got disrupted. The authors pioneered resource usage to energy inference methodology, obtaining models for each hardware subsystem by using only software performance counters offered by the hypervisor, with an absolute error of 5W. However, the main caveat this work presents is the non automatic generation of the power models, undermining the replicability of the study on different machines and architectures.

A more recent and complete study on Virtual Machine (VM) power modelling is *"iMeter: An integrated VM power model based on performance profiling"* by *H. Yang et al.* [160], one of the first works to present an highly systematic approach at any stage of the development. The proposed methodology is composed by the following steps: (1) benchmark selection, where the authors used the *NASA Parallel Benchmark* suite [46, 145] plus *IOzone* [85] and *Cachebench* [118] to stress various workloads of each VM; (2) PMCs trickle-down selection, necessary for reducing the solution space; (3) collection of a measurement baseline, consisting in the physical machine to be stimulated by various VMs; (4) performance counters systematic selection, exploiting PCA and varimax rotation for further model order reduction; (5) modeling, leveraging *support vector regression*; (6) evaluation, where a hierarchical clustering is applied in order to assimilate similar conditions. With this methodology, the authors were able to reach

a relative error of about 5%. In this work, we showed how to improve the modeling performance through working regimes identification, thus raising the bar in the field of power modeling methodologies for multi-tenant server infrastructures.

5.7 Final remarks

This chapter discussed how *XeMPower* and *MARC* have been exploited to build data-driven power consumption models for multi-tenant server infrastructures. Results show a modeling relative error of around 2% on average, and under 4% in almost all the cases and on different workload classes, outperforming previous research in the field. Moreover, model portability across similar architectures has been explored, showing how the proposed models can be also used to evaluate tenants colocation in a multi-tenant infrastructure.

Up to now, this thesis work explicitly focused on power modeling. However, in order to thoroughly explore the topic, workloads' *performances* needs to be included in the loop. The last chapters will then explore how a power-aware system should *plan* future decisions and *execute* the best actions with respect to performance goals and power constraints, i.e., the last two steps of the OLDA control loop introduced in Section 1.2.

CHAPTER 6

Maximizing performance under a power cap: a hybrid hardware-software approach

6.1 Introduction

Previous chapters already discussed how *virtualization* technologies foster hardware *heterogeneity* and software *multi-tenancy*, and the subsequent implications that these have on *power consumption*.

This chapter discusses the need to control and limit the power consumption of a system, i.e., the need to set a *power cap*, and its implications on the guest applications. This need can be obvious for battery-powered devices, that have to deal with limited energy budgets, as their batteries need to be small and lightweight. However, the same need is present also in those contexts that do not involve batteries, to decrease operating costs and to improve system reliability. For instance, even modern data centers need to deal with power caps: given the huge density of servers, the power grid may not be able to supply enough energy to run all of them at their peak performance.

To face the need for power capping, Intel introduced the RAPL interface since its second generation of Sandy Bridge processors [39]: this interface

enforces a strong and precise limit on the power consumption of a processor, i.e., the component that contributes the most on the *dynamic* power consumption of a common workstation [159]. RAPL uses Dynamic Voltage and Frequency Scaling (DVFS) techniques to guarantee the desired power cap but is not aware of the impacts that these have on the performances of the hosted applications. Of course, these performances need to be maximized even when a power cap is enforced: we want to find the most *power efficient* hardware configuration under a certain power cap, thus maximizing the performance-per-watt ratio. In order to accomplish our goal, a uniform metric of performance has to be defined, as well as a smart orchestration policy to guarantee the stability of the system as soon as its runtime conditions change.

In this chapter, I propose *XeMPUPiL*, a hybrid hardware and software power capping orchestrator for the Xen hypervisor, based on the PUPiL control loop [163], that aims at maximizing the performance of a workload under a power cap. The main contributions are the following:

A. I propose a *observe* phase that takes into account a generic performance metric for all the hosted tenants, avoiding any instrumentation of the workloads;

B. I improved the decision phase of PUPiL, to deal with the resources available in a multi-tenant virtualized environment;

C. I implemented a new *Actuation* phase, to support all the *knobs* that Xen provides to control the resources assigned to each tenant.

The chapter is organized as follows: Section 6.2 discusses some related work, while Section 6.3 presents the proposed approach and some implementation details; preliminary results are detailed in Section 6.4, finally drawing some conclusions in Section 6.5.

6.2 Related work

Several works in the literature propose different approaches to both performance maximization under a power cap and power consumption minimization under performance constraints. For instance, some of them exploit DVFS techniques and try to pack together similar threads [34], while others try to minimize the times the cores go into idle states, in order to save the power spent in going from an idle state back to an active one [94]. Most of these works aims at reducing costs in data centers [81, 112, 142] or to increase battery life in power-constrained devices [58, 95, 116], while

the main focus of this chapter is performance maximization under a strict power cap.

A remarkable work with the same goal is PUPiL, a framework that aims to minimize and to maximize respectively the concept of timeliness and efficiency: *timeliness* is intended as the ability of the system in enforcing a new cap, while *efficiency* is meant as the performance delivered by the applications under a fixed power cap [163]. In order to achieve these goals, PUPiL exploits both hardware (i.e., the Intel RAPL interface) and software (i.e., resource partitioning and allocation) techniques.

Even though the approach proposed by PUPiL is effective, we can identify two non-negligible limitations of the proposed solution: first, the applications running on the system need to be instrumented with the *Heartbeat framework* [77, 78], in order to provide a uniform metric of throughput to the decision phase; second, the tool is meant to work with applications running bare-metal on Linux. Both these conditions might not be met in the context of a multi-tenant virtualized environment, in which a virtualization layer allows the execution of multiple workloads and ensures isolation to each of them. This is the case of the *Xen hypervisor*, that runs directly as an abstraction layer between the hardware and the hosted virtual machines, called *domains* in the Xen terminology. In this context, the high isolation of each tenant, seen as a *black box*, makes any instrumentation of the code of the hosted applications not feasible in a real production environment.

In the following sections, I want to extend the current implementation of PUPiL[1] to make it work in a virtualized environment based on the Xen hypervisor, without requiring any instrumentation of the guest workloads.

6.3 System design and implementation

XeMPUPiL is a hybrid hardware and software power capping orchestrator for the Xen hypervisor. It is *hybrid* as it makes use of the RAPL *hardware* interface to set a strict limit on the processor's power consumption, while a *software*-level OLDA loop structure performs an exploration of the available resource allocations, to find the most power efficient one for the running workload. Of course, the innovation does not lie in the exploitation of the well-known loop structure, but in the adoption of an hybrid power capping approach in a virtualized environment.

An overview of the system is presented in Figure 6.1: in this chapter, we do not distinguish between Learn and Decide, as the focus here is not in the former but in the latter. Each different phase of the loop needs to interact

[1] All source code, scripts, inputs, and patches are available at: https://github.com/PUPiL2015/PUPIL.git

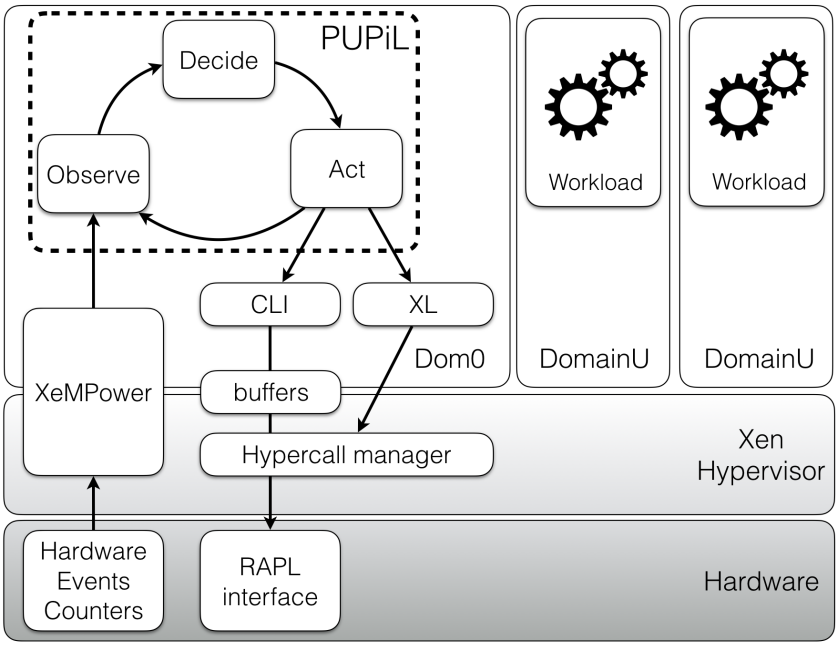


Figure 6.1: *Overview of the proposed approach*

with different tools throughout all the layers of the stack: some tools are available in Dom0, while other APIs are provided by specific hypercalls to the Xen hypervisor, that allows *XeMPUPiL* to set the domains configurations and guarantees a controlled access to the underlying hardware.

In more detail, a brief description to the high-level flow is here given:

- *XeMPUPiL observes* the power consumption of the system and a set of hardware events of interest for each running domain;
- the traced events are then used as metrics of performance, in order to *decide* which hardware configuration is the most power efficient for the current workload;
- finally, the *actuation* phase sets the system to the best configuration found, to maximize the performance under the desired power cap enforced through the RAPL interface.

In this section, we present the design and the implementation of the three phases, describing the limitations faced while working in a virtualized environment.

6.3.1 Observe

This first phase is in charge of monitoring the system and the hosted domains, gathering all the information needed by the subsequent *decide* phase.

As stated in the previous sections, we need to choose a uniform metric of performance without any instrumentation of the guest workloads: each domain remains a black box to the hypervisor, as well as by the other domains (e.g., Dom0 itself). We decided to use hardware event counters as low level metrics of performance, exploiting the Intel PMU to monitor the number of Instruction Retired (IR) accounted to each domain in a certain time window. Among all the available hardware events that can be monitored, we chose to count the IR events on purpose, because these give an insight on how many micro-instructions were completely executed (i.e., that *successfully* reached the end of the pipeline) between two samples of the counter, thus representing a reasonable indicator of performance, as the same manufacturer suggests in [1].

In order to monitor these hardware events, we exploited the same *XeMPower* tool described in Chapter 4, configured to provide *XeMPUPiL* the amount of IR counted for each running domain over the last second: more details on how we use this rate are provided in the following section.

6.3.2 Decide

The *decision* phase is similar to the one implemented in PUPiL. The major changes are in how we evaluate the metrics gathered in the previous phase and in how we assign the physical resources to each virtual domain.

The evaluation criterion is based on the average IR rate measured over a certain time window: this allows the workload to adapt to the actual configuration in that time window before a new decision is taken. The comparison of the two IR rates highlights which one makes the workload perform better, thus discarding the worse one.

Once the configuration has been chosen, the second part of the decision phase begins: it concerns the allocation of resources to each domain. I chose to work at a *core-level* granularity: on the one hand, each domain owns a set virtual CPUs (vCPUs), while, on the other hand, we have a set of physical CPUs (pCPU) present on the machine. Each vCPU can be mapped on a pCPU for a certain amount of time, while it may happen that multiple vCPUs can be mapped on the same pCPU.

We wanted our allocation policy to be as fair as possible, covering the whole set of pCPUs if possible; given a workload with M virtual resources

and an assignment of N physical resources, to each $pCPU_i$ we assign:

$$vCPUs(i) = \left\lceil \frac{M - \sum_{j=0}^{i} vCPUs(j)}{N - i} \right\rceil \tag{6.1}$$

where i is an integer between 0 and $N - 1$, i.e., it spans over the set of pCPUs.

6.3.3 Act

The *act* phase essentially consists in: (1) setting the desired power cap and (2) actuating the selected resource configuration.

On the one hand, I decided to implement the same hardware technique proposed by PUPiL to set the power cap, i.e., exploiting the Intel RAPL interface. This provides a fast and strict response to power oscillations, harshly cutting the frequency and the voltage of the whole CPU socket, ignoring the performance of the applications actually running on the system.

On the other hand, we had to support the *knobs* made available by the hypervisor to assign resources to each domain. This second step allows a fine tuning of the resources to improve domains' performance, but it is of course slower than the hardware actuation in responding to power variations.

This is the reason why we use both the approaches to provide a fast response, still trying to find the best resource allocation to maximize the performance of each domain under the power cap.

Hardware power cap

A bare metal operating system can easily access the RAPL interface to set a power cap on the system by writing data into the right Model Specific Register (MSR) of the processor. The two registers of interest to our purposes are `MSR_RAPL_POWER_UNIT` and `MSR_PKG_RAPL_POWER_LIMIT`: the former contains processor-specific time, energy and power units, used to scale each value read or written on the RAPL MSR, in order to obtain a valid power or energy measure; the latter can be written to set a limit on the power consumption of the whole CPU socket.

In a virtualized environment, these registers are not directly accessible by the virtual domains, even from the privileged tenant Dom0. However,

this limitation can be overcome by invoking custom hypercalls that can directly access the underlying hardware. To the best of my knowledge, the Xen hypervisor does not natively support specific hypercalls to interact with the RAPL interface: as a consequence, I implemented our custom hypercalls to this purpose. In order to be generic enough, I implemented two hypercalls: "`xempower_rdmsr`" and "`xempower_wrmsr`". The first one allows to read, while the second one allows to write a specified MSR from Dom0.

Each hypercall needs to be declared inside the kernel of the hypervisor, that runs bare metal on the hardware. The kernel keeps track of the list of hypercalls available and the input parameters they accept. For each of them, a callback function has to be declared and implemented to be accessible by the kernel at runtime: our implementation makes use of two Xen build-in functions to safely read and write MSR registers, i.e., `wrms_safe` and `rdmsr_safe`; these raise exceptions if something goes wrong in accessing the registers, avoiding errors and faults to undermine the kernel stability.

We then implemented our own CLI tools to access these hypercalls from Dom0: `xempower_RaplSetPower` to set and `xempower_RaplPower Monitor` to read the power consumption of the socket. Arguments (e.g., the desired value of power cap and the power consumption measured) are passed through the whole stack using a set of buffers that allow a fast and safe communication between different hierarchical protection domains [91] (i.e. ring0 for Xen and ring3 for Dom0). The CLI tools are in charge of performing some checks on the input parameters, as well as of instantiating and invoking the Xen command interface to launch the hypercalls.

Software resource management

The current implementation of *XeMPUPiL* exploits two tools provided by the Xen hypervisor to tune the performance and assign resources to domains.

The first one is the *cpupool* tool: this is part of the Xen *xl* CLI and allows to cluster the physical CPUs in different pools. Once a pool is declared, it is possible to create a domain that uses that pool: a new scheduler is instantiated in order to manage the pool. It will then schedule the domain's vCPUs only on the pCPUs that are part of that cluster. Our approach exploits this tool to assign more pCPUs to a domain at runtime: as a new resource allocation is chosen by the *decide* phase, we increase or decrease the number of pCPUs in the pool and pin the domain's vCPUs to these, to increase workload stability. The domain still has the same amount of virtual resources,

that *XeMPUPiL* distributed over the maximum number of physical ones available, potentially causing more vCPUs to be time-multiplexed on the same core.

The second tool supported is *xenpm*: this allows to set a maximum and minimum frequency for each pCPU. After a first evaluation, we decide to leave the actuation of the core frequencies out of the *decision* phase, as it may interfere with the actuation made by RAPL.

6.4 Experimental results

The goals of our experiments are twofold: (1) we want to show how the metric of performance we propose in this chapter behaves when subject to a power limit and (2) that *XeMPUPiL* is able to maximize that metric given a certain power limit.

Tests have been performed on a system equipped with a 2.8-GHz quad-core Intel Xeon E5-1410 processor (4 hardware threads, TurboBoost and HyperThreading disabled) with 32GB RAM. The system runs the Xen hypervisor version 4.4, with a *paravirtualized* instance of Ubuntu 14.04 as Dom0, pinned on the first core and with 4GB of RAM.

We set up three distinct *paravirtualized* domains, each of those running one of the following four different microbenchmarks, each one representing a different computational class, able to stress the performance of the system:

A. *NPB3.3 Embarrassingly Parallel (EP)*, a CPU-bound benchmark;

B. *IOzone*, an IO-bound benchmark;

C. *cachebench*, a memory-bound benchmark;

D. *NPB3.3 Block Tri-Diagonal solver (BT)*, a mixed-class benchmark.

EP generates pairs of Gaussian random deviates: this is quite typical of many Monte Carlo simulation applications [46]. IOzone is a filesystem benchmark tool, generating and measuring a variety of different file operations [85]. Cachebench is designed to test memory and cache bandwidth performance [3]. BT is a pseudo application, more specifically a Block Tri-diagonal solver [46]. Experiments have been repeated multiple times, to improve the accuracy of the results.

Our first set of experiments aims to show how the number of IR over a certain time window decreases as the power cap becomes stricter. We run each benchmark in a domain with three virtual CPUs assigned and pinned

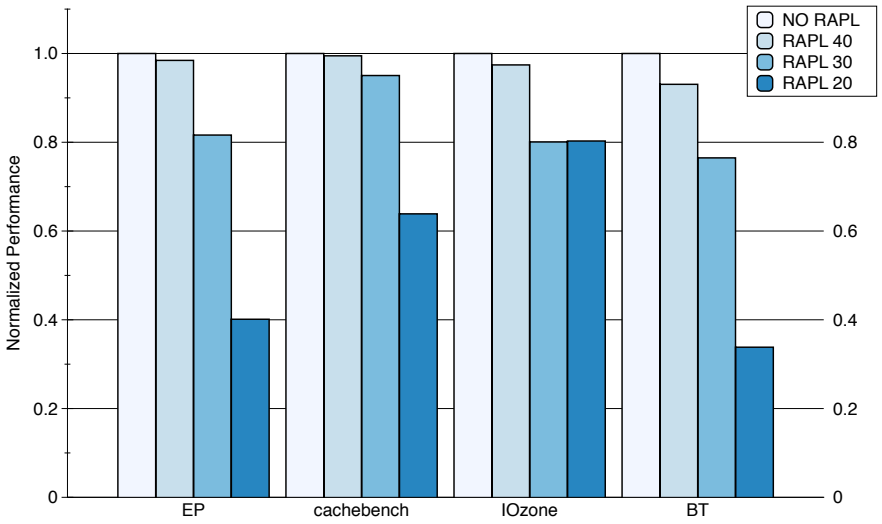


Figure 6.2: *Benchmarks' performance normalized with respect to NO RAPL, under different power caps*

on the three available physical CPUs, to avoid interferences and to maximize resource utilization. We repeated the tests in four different scenarios, namely:

A. `NO RAPL`, as no power limit was set;

B. `RAPL 40`, as a 40W power limit is set (using RAPL);

C. `RAPL 30`, as a 30W power limit is set (using RAPL);

D. `RAPL 20`, as a 20W power limit is set (using RAPL);

We chose 40W and 20W as the maximum and minimum power caps as we observed that, in an idle state, the entire socket consumes around 17W, while the maximum power consumption we reached was around 43W. The comparison between the performance of each single benchmark under different power caps is shown in Figure 6.2, where the Y-axis reports the performance expressed as the average IR over a time window of 5 seconds. All the results have been normalized with respect to the performance obtained under the `NO RAPL` condition: as expected, the chosen metric is a reasonable indicator of the performance of the application and decreases with a stricter power cap. More in details, with CPU-bound benchmarks (i.e., EP and BT) the difference are greater than in benchmarks where the bottleneck are IO and memory accesses: in these cases, the performance degradation is less significant between different power caps.

The second set of experiments is meant to achieve our second goal: we want to compare the performance of the workloads when *XeMPUPiL* performs its resource allocation in the same scenarios described above, i.e., under a power cap of 40W, 30W and 20W respectively. Results, normalized with respect to the ones obtained in the NO RAPL configuration, are shown in Figures 6.3a-c.

XeMPUPiL is able to achieve higher performance under the same power cap in all the scenario and for all the benchmarks: this is due to the decision of assigning in a smart way all the possible domain's vCPUs on fewer pCPUs than the available ones. However an exception to this trend is represented by the EP benchmark, where in any case the performance gets better for the same benchmark with a cap of 20W, as the framework redistributes the virtual resources over just two physical cores, thus obtaining a configuration that is more power efficient.

As mentioned before it is interesting to note how the performance achieved in case of the IO-bound, the memory-bound and the mixed benchmark are even better than the ones achieved by the `NO RAPL` experiment: for IOzone and cachebench, *XeMPUPiL* converged to a configuration with just one core assigned to the domain, while two cores have been assigned to the BT benchmark. These assignments are more power efficient, as they reduce memory and IO contention for non strictly CPU-bound workloads.

6.5 Final remarks

This chapter presented some preliminary results and opportunities towards a performance-aware power capping orchestrator for the Xen hypervisor. The proposed approach is a hybrid hardware-software power capping solution, based on the PUPiL control loop, that aims at maximizing the performance of a workload under a power cap.

At the moment, *XeMPUPiL* has been validated with only one guest application running at a time: this condition may not be very common in a real production environment, where multiple guests may be hosted on the same node, each one with different performance requirements. In order to tackle these issues, a smart resource manager must be put in place to deal with resource allocation, power constraints and performance requirements, as discussed in the next chapter.

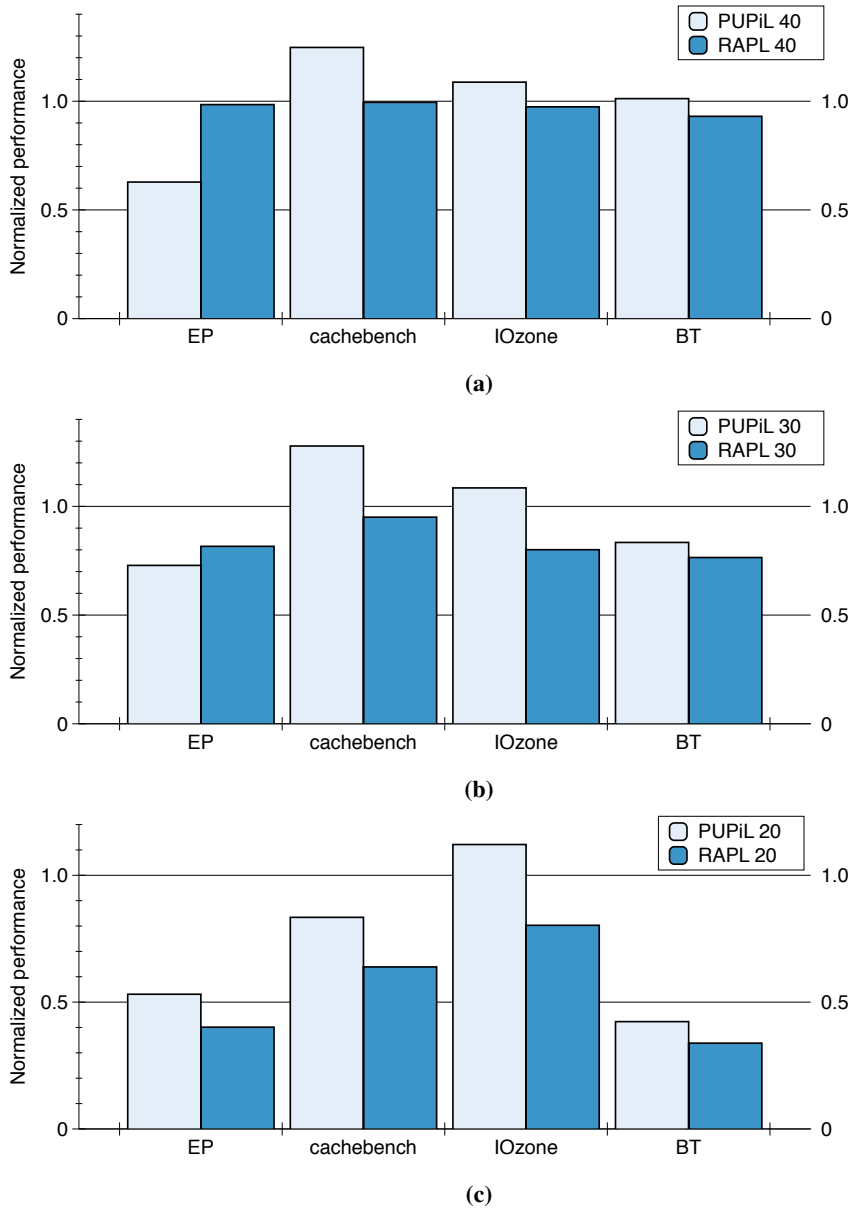


Figure 6.3: *Benchmarks performance normalized with respect to NO RAPL under a cap of 40W (a), 30W (b) and 20W (c), imposed by RAPL and by* XeMPUPiL

CHAPTER 7

Moving forward: containerization, challenges and opportunities

7.1 Introduction

The need to control and limit the power consumption of a system has already been introduced in Chapter 6, as well as its implications on the *performances* of the guest applications and related work in the field. This chapter introduces priority *policies* and QoS *requirements*, exploring a new approach to software multi-tenancy: *containerization*.

Docker [47] and *containerization* techniques are becoming a remarkable trend towards a simpler deployment and migration of applications in a multi-tenant and heterogeneous ecosystem: this then helps to overcome issues related to applications *isolation* and *portability*, but still does not fulfill the lack of a smart resource manager, able to deal with resource allocation, power constraints and performance requirements.

This chapter presents *DockerCap*, a power-aware orchestrator for multi-tenant containerized systems. DockerCap is able to satisfy a constraint on the maximum power consumed by the node, i.e., a *power cap*; moreover, it supports the definition of different resource allocation policies to guarantee

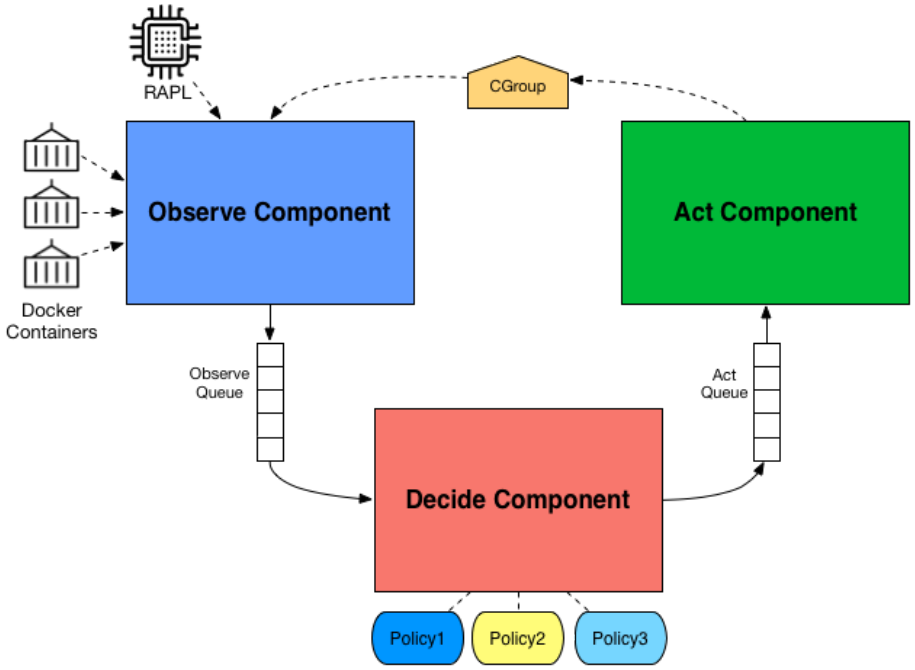


Figure 7.1: *Architecture of DockerCap*

the performances of the containers while meeting the desired power cap. Results are compared with the state of the art solution provided by Intel RAPL interface. Moreover, DockerCap is able to control the performance of the containers and even guarantee Service Level Agreements (SLA) constraints: this is something that a completely hardware solution like the one implemented by RAPL is not able to handle.

This chapter is organized as follows: Section 7.2 presents DockerCap, providing an high level overview of the components, while Section 7.3 discusses the details of the implementation of the system; then, Section 7.4 presents the experimental setup, discussing the results obtained, while Section 7.5 concludes and describes future work in the field.

7.2 Proposed methodology

This section gives an overview of the proposed methodology, along with a high level description of the control steps performed by DockerCap to enforce performance requirements and the SLA of each container.

The architecture of DockerCap is based on the ODA loop structure widely described throughout the whole thesis: a bird's-eye view is represented in

Figure 7.1. It is interesting to note how the *Learning* phase here is implicit: it is performed only once in the current prototype, as discussed in Section 7.2.1: since that phase has been widely discussed in previous chapters, no additional discussion will be held in this chapter.

In this context, the observe phase continuously monitors the power consumption of the system, retrieving this information from the *power source* available. The choice of the power source is crucial in the control loop, since its precision and sampling frequency influence the performance of the controller. Then, the decide phase produces the new allocation of resources that will be assigned to each container. The whole decision process is divided in two distinct steps: the choice of the allocation of resources that meets the power cap (i.e., *resource control* step) and the partitioning of the resources assigned to each container (i.e., *resource partitioning* step). Finally, in the act phase, DockerCap performs the actual resources allocation, using the information obtained from the decide phase; of course, the actual implementation strongly depends on the nature of the resource under control. Once the actuation has been performed, DockerCap waits for a reasonably small amount of time (tunable, depending on the timing constants of the system under control) to allow the system to reach a stable state before the next iteration of the ODA loop.

The way in which resources are partitioned across the running containers depends only on the *decision policy* adopted. In the following sections, the adoption of three different policies is analyzed, comparing them with the performance obtained using RAPL, the hardware power capping solution provided by Intel on recent processors.

7.2.1 Resource control step

The main goal of this step is to find the right amount of resources that will not exceed the power cap, still guaranteeing the highest throughput possible to all the containers. Therefore, we introduced a *PI Controller* to control the resource assignment, as in Figure 7.2.

In order to work properly, the controller needs to know the *power-resource* model of the physical machine, i.e., the relation between its power consumption and the amount of allocated resources in a given time window.

This relation can be modeled using an ARX model, as proposed in other works in the field [79,144,154]: here is where the *MARC* modeling methodology comes into play, as extensively discussed in Chapter 5, to estimate the parameters of (7.1), where $r(k)$ represents the amount of resources allocated at time k, while $p(k)$ and $p(k+1)$ represent the power consumption

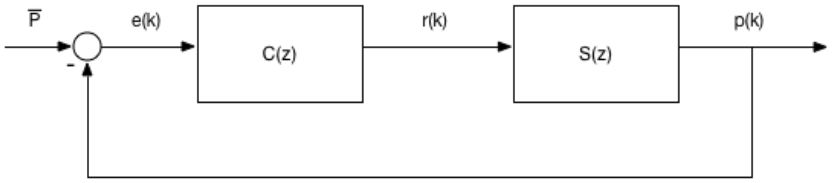


Figure 7.2: *Schema of the PI controller adopted*

at time k and $k+1$, respectively:

$$p(k+1) = a \cdot p(k) + b \cdot r(k) \tag{7.1}$$

In this work, the controlled resource is the *CPU quota* assigned to each container. Nevertheless, the same approach is general enough to deal with multiple resources, if we consider $r(k)$ as a vector and we treat the problem as a MISO system.

We obtained the model parameters through *MARC*, using a cpu-bound benchmark to stress the processor of the machine. In the current prototype, the model is not updated during the lifetime of the system; however, continuous learning can be easily integrated, given the "as-a-service" nature of the *MARC* infrastructure.

Once we have an estimation for the model's parameters, we can apply the Z-transform [109] to equation (7.1) and obtain the equation (7.2) in the frequency domain:

$$z \cdot P(z) = a \cdot P(z) + b \cdot R(z) \tag{7.2}$$

From this equation, we derive the system transfer function $S(z)$:

$$S(z) = \frac{P(z)}{R(z)} = \frac{b}{z-a} \tag{7.3}$$

The PI Controller $C(z)$ has then been introduced in a feedback loop with $S(z)$, as shown in Figure 7.2, in order to control the amount of resources needed to meet the power cap. The transfer function of the feedback loop $L(z)$ is then obtained as follows:

$$L(z) = \frac{C(z) \cdot S(z)}{1 + C(z) \cdot S(z)} \triangleq \frac{1-p}{z-p}, p \in (0,1) \tag{7.4}$$

I used a first-order transfer function with a single pole in p. As in equation (7.4), the value of p should be chosen in the interval $(0,1)$, to ensure the asymptotic stability of the system.

From equations (7.3) and (7.4), we obtain the transfer function of the controller $C(z)$, as follows:

$$C(z) = \frac{(1-p)\cdot(z-a)}{b\cdot(z-1)} \triangleq \frac{R(z)}{\mathcal{E}(z)} \tag{7.5}$$

The input of the controller is the error: $e(k) = \bar{p} - p(k)$, with $E(z)$ still in the frequency domain and $\bar{p}$ as the reference power cap.

Finally, applying the inverse Z-transform and a time shift, we obtain the equation:

$$r(k) = r(k-1) + \frac{1-p}{b}\cdot(e(k) - a\cdot e(k-1)) \tag{7.6}$$

The controller will use it to estimate the value of $r(k)$, given the current error $e(k)$.

7.2.2 Resource partitioning step

Once the power constraint is met, we can partition the amount of CPU quota Q across each distinct container. In order to do so, I developed three *policies*: these policies take into account different aspects of the containers, with different impacts on performance. Here I propose an high level overview of those policies, while Section 7.4 analyzes their impact.

The *Fair resource partitioning* policy assigns the same amount of CPU quota to all the containers, as in equation (7.7), where C is the set of all the containers:

$$q_c = \frac{Q}{|C|} \quad \forall c \in C \tag{7.7}$$

The *Priority-aware resource partitioning* policy splits the CPU quota Q based on a priority assigned to each container, as in equation (7.8), where w_c is the weight associated to the priority of the container:

$$q_c = Q \cdot \frac{w_c}{\sum_{\forall i \in C} w_i} \quad \forall c \in C \tag{7.8}$$

The *Throughput-aware resource partitioning* policy partitions the CPU quota considering the requirements of each container in terms of both its SLA and priority. We define the SLA of a container in terms of *completion time* of the hosted workload; in this context, we assume that the more resources are given to a container, the faster it will be to complete its tasks, with a fixed input size. In order to understand the relation between completion time and resources, we built a *resource-time* model offline: again,

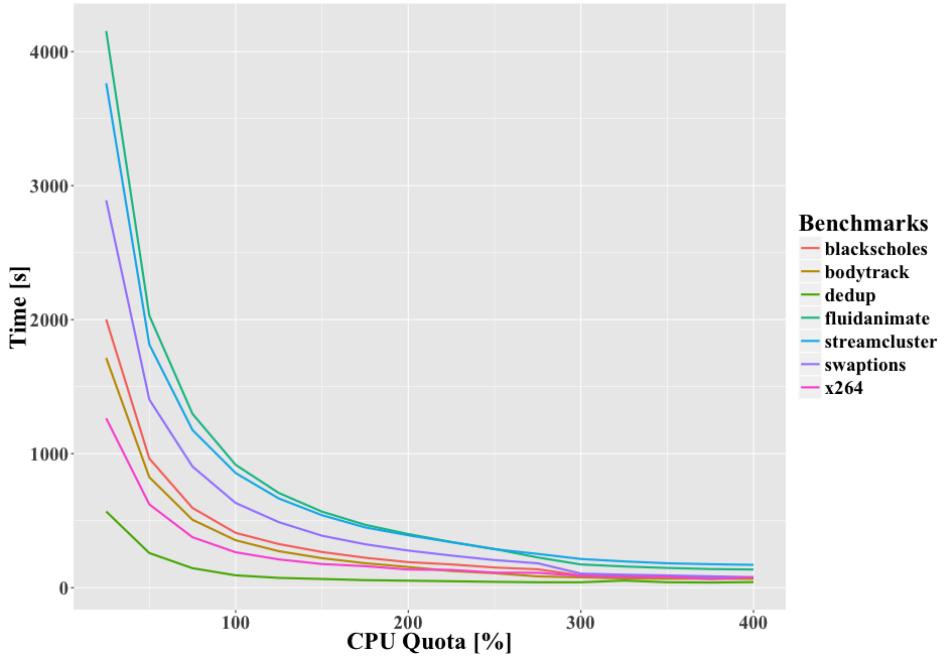


Figure 7.3: *Relationship between CPU quota and completion time for different PARSEC benchmarks*

the need for a learning phase is fundamental to tackle the complexity of this use case. This information can then be used at runtime, to predict the completion time of a container, with a certain resource allocation.

Figure 7.3 shows the actual relationship between the CPU quota and the completion time of the most common workload of the PARSEC [22] benchmark suite. This relationship is well explained by equation (7.9), where t_c is the time to complete the workload of the container c, q_c is the CPU quota assigned to the container c and K_c is the constant we obtained with a regression on the tests performed, with a fixed input size. To obtain the data to produce those models, we perform multiple run of the benchmarks in distinct cpu quota configuration.

$$t_c \cdot q_c \approx K_c \quad \forall c \in C \tag{7.9}$$

All the models obtained from the workloads of the PARSEC benchmark suite have minimum R^2 coefficient [49] of 97.54%, thus showing how this model fits well the real behavior of the workload.

The average throughput T_c of each container can then be expressed as in equation (7.10), where I_c is the input size of the container c.

$$T_c = \frac{I_c}{t_c} \approx \frac{I_c \cdot q_c}{K_c} \quad \forall c \in C \tag{7.10}$$

The SLA requirement can be finally expressed as a target throughput: T_{sla_c}. Thus, from equation (7.11), we can find the minimum quota to be assigned to meet the specific constraint:

$$q_c = \frac{Tsla_c \cdot K_c}{I_c} \quad \forall c \in C \tag{7.11}$$

This policy will then allocate the minimum amount of resources that satisfies the SLA of each container, ordered by priority.

7.3 Implementation

DockerCap is modular by design: the distinct phases of the Observe-Decide-Act (ODA) loop have been decoupled and implemented, fully available open source[1], in three different components: the *Observe Component*, the *Decide Component* and the *Act Component*. This section goes through some implementation details for each one of them.

7.3.1 Observe

We developed two distinct interfaces to support measurements from the Intel *RAPL* interface [39] and from an external *WattsUp power meter* [44]. Each power source supports different logging frequencies: for instance, RAPL provides measurements every 1ms approximately, while the WattsUp power meter logs power consumption every 1s approximately. Whenever a sample is read from the source, it is added into the ObserveQueue, as in Figure 7.1: this allows to fully decoupling the observe and the decide components.

7.3.2 Decide

The *Decide Component* takes the samples from the *Observe Component* as input and provides the resource allocation for the *Act Component*, if an actuation is needed to guarantee the power cap. When a power sample in the queue is greater than the power cap (plus a tuneable level of tolerance), the component starts the decision phase. The constraints on the resources are relaxed as soon as the power consumption goes below the power reference. The decision is computed each time the *Decide Component* fetches

[1] All the source code of DockerCap can be found here: https://bitbucket.org/necst/dockercap-euc

a power sample from the queue, which depends on the frequency at which the *Observe Component* provides the samples.

As discussed in Section 7.2, the decision process is divided in two steps, the *resource control* and the *resource partitioning*: each of them is decoupled from the other, to guarantee the modularity of the system. The resource control phase consists in a PI controller that implements the computations discussed in Section 7.2.1. The actual implementation of DockerCap supports only the CPU quota of the single Docker container, i.e., the amount of time that the processes inside a container can use during one scheduling period of the CPU. The resource partitioning phase, instead, implements three distinct policies, as follows.

Fair resource partitioning

This policy is a trivial implementation of the Equation (7.7); a high level description of the algorithm is provided in Algorithm 2.

Algorithm 2 Fair resource partitioning

```
1: procedure PARTITION(Q, containers)
2:     count ← length(containers)
3:     for c ∈ containers do
4:         quota_c ← Q / count
5:     end for
6:     return quota
7: end procedure
```

Priority-aware resource partitioning

As already introduced, this policy relies on information about the priorities of each container, that can be stored in a configuration file. For what concerns model parameters, these have been estimated once, but the system can be easily extended to continuously learn them as the environment evolves.

The current implementation of DockerCap supports three categories of priorities: *high*, *low* and *best effort*, each one associated to a specific weight w_c. When DockerCap detects a new running container, it tries to determine its priority. If it is known, the system assigns the respective priority weight (high or low), otherwise the system will assign the lowest priority (best effort) to it. A high level description of the algorithm is provided in Algorithm 3.

Algorithm 3 Priority-aware resource partitioning

1: **procedure** PARTITION($Q, containers$)
2: $weights \leftarrow priority\_weights(containers)$
3: **for** $c \in containers$ **do**
4: $quota_c \leftarrow Q \cdot \frac{weight_c}{\sum_{\forall i \in C} w_i}$
5: **end for**
6: **return** $quota$
7: **end procedure**

Throughput-aware resource partitioning

This last policy combines the priority associated to a container with estimations about its throughput. Each container has a corresponding constraint SLA_c, that represents the SLA of the specific container, and a parameter K_c that represents the parameter of the time-quota model trained in Section 7.2.2.

A high level description of the algorithm is provided in Algorithm 4. It starts assigning a minimum value of quota to all the running containers, regardless their priority. Then, from the higher to the lower priority, it tries to assign the amount of CPU quota needed to satisfy the container's SLA. If we have not enough resources, it performs a fair resource allocation only on those containers with the highest priority and then terminates. Otherwise, it repeats the same procedure with the lower priority containers.

Then, once both high and low priority containers are covered, it performs a fair resource allocation on the best effort containers. At the end of the algorithm, it performs a leftover assignment to assign all the remaining CPU quota to all the containers, following the priority order again. This final step is performed as there may be some categories not covered yet if the algorithm does not assign all the available quota.

7.3.3 Act

The *Act Component* performs the actuation of the decisions taken by the *Decide Component*. As for the *Observe Component*, it supports any generic resource that provides an abstraction to its specific implementation through a shared interface.

We developed an interface to perform the updates of the CPU quota of the single container through *CGroups* [113]. CGroups are an interface provided by the Linux kernel to manage specific resources of *tasks* (i.e. processes) through a hierarchical structure of directories. Each *subsystem* (i.e resource manager) is represented by a hierarchy and each task is part of

Algorithm 4 Throughput-aware resource partitioning

```
1:  procedure PARTITION(Q, containers)
2:      quota ← baseline_assignment(containers)
3:      for c ∈ high_priority do
4:          quota_c ← min_quota(SLA_c, K_c)
5:      end for
6:      if Σ_{∀i∈C} quota_i > Q then
7:          for c ∈ high_priority do
8:              quota_c ← Fair_partition(Q)
9:          end for
10:         return quota
11:     end if
12:     for c ∈ low_priority do
13:         quota_c ← min_quota(SLA_c, K_c)
14:     end for
15:     if Σ_{∀i∈C} quota_i > Q then
16:         for c ∈ low_priority do
17:             quota_c ← Fair_partition(Q)
18:         end for
19:         return quota
20:     end if
21:     for c ∈ best_effort do
22:         quota_c ← Fair_partition(Q)
23:     end for
24:     quota ← assign_leftover(Q)
25:     return quota
26: end procedure
```

a single cgroup inside a subsystem; all the tasks in the same cgroup share the same property. Moreover, at one time there may be multiple active hierarchies in the system.

Docker exploits CGroups to manage its containers: for each subsystem, Docker creates a cgroup that contains all the processes of the container, hence all the tasks inside that cgroup have the same constraints on resources. The *Act Component* automatically detects the cgroup of the container by its runtime id and performs the actuation on the resource by writing on the proper files of the subsystem. File access is managed through an interface that wraps the file writing as a simple resource update.

7.4 Experimental results

In this Section we compare the performance of DockerCap along with the three control policies, analyzing:

A. its precision with respect to different desired power caps;

B. its ability to partition the resources with a priority scheme and to guarantee a SLA on the performance of a container.

We also compare our solution with respect to the performance of RAPL, the current state of the art solution for power capping. We perform our tests on a server equipped with a 2.8-GHz quad-core Intel Xeon E5-1410 processor (4 hardware threads) and 32GB of RAM. The machine runs an Ubuntu 14.04 with the Linux kernel 3.19.0-42 and Docker 1.9.0.

In order to compare all the proposed policies with RAPL, we chose three distinct benchmark from the PARSEC benchmark suite [22], each one representing a reasonable workload for the cloud computing context [28, 166]:

A. *dedup*, a compression benchmark;

B. *x264*, a video streams encoding benchmark;

C. *fluidanimate*, it simulates an incompressible fluid for interactive animation purpose by solving the Navier-Stokes equation [129].

We set up a container for every benchmark, assigning a priority and a constraint to each of them. Moreover, we choose as sampling time in the *Observe Component* $1s$.

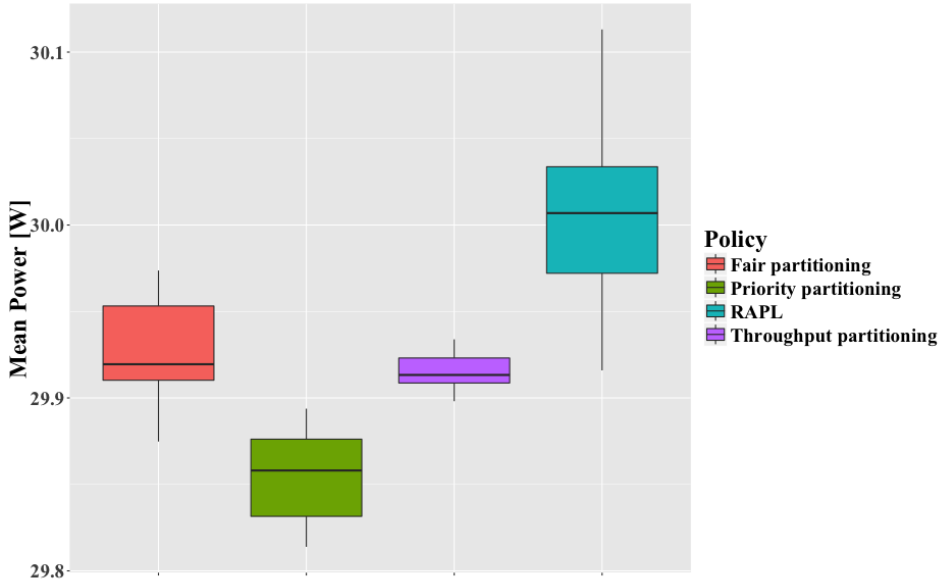


Figure 7.4: *Tukey boxplot of the mean power consumption under a power cap of 30W*

7.4.1 Power capping precision

Here we want to analyze how the controlled system behaves when DockerCap enforces a power cap, expressed in terms of mean power consumption. We performed multiple distinct runs of the group of workloads for each controlling policy of DockerCap and RAPL, our performance reference. From each run, we got a time series of power data, that we used to compute the mean power consumption for each run. Figure 7.4 shows the power consumption distribution of each policy, using a box plot for each distinct policy.

As expected, the results show that we are able to achieve an average power consumption very close to the desired power cap, even if our software approach is not as fast as the hardware approach implemented by Intel RAPL.

7.4.2 Impact on the benchmarks performance

In this Section, we want to analyze the performance of each benchmark under a power cap. As in the previous analysis, we compared our performance with the state of the art solution RAPL. We performed multiple runs of each controlling policies, including RAPL: for each run, we acquired the Time to completion (TTC) of each container, given the same fixed input. We

adopted the TTC as a performance metric because we want to have a metric that gives us a direct view on the performance of the workload without the need of instrumenting our containers to extract domain-specific performance metrics. As each workload processes a specific fixed-size input (i.e. *native* [22]), it is trivial to obtain the throughput of the workload through equation (7.10). Moreover, with TTC, we have a unified performance metric for all the different types of workload, thus making the performances of the containers comparable between distinct workload.

We now want to show how our software-level power capping orchestrator is effective in controlling the performances of the containers under a power cap.

Figure 7.5 compares the performance of the Fair policy with respect to RAPL, as both ignore the actual workloads running on the machine. The results show that our Fair approach performs better than RAPL under lower power caps for each distinct benchmark, considering our processor with TDP of 80W. Instead, we have results that are comparable with a high power cap and two out of three benchmarks perform worse.

Then, we want to explore the performance of the other two proposed policies under different power caps, as in Figure 7.6. For what concerns the Priority partitioning, we assigned a high priority to *fluidanimate* (i.e., the one with the longest TTC), while low and best effort priorities are assigned to *x264* and *dedup*, respectively. The results show how it is possible to find the right mix of quotas to obtain a distinct performance pattern of the workloads with respect to the Fair partitioning, still guaranteeing the same power cap.

Finally, the same Figure 7.6 allows to compare the Fair partitioning policy with the Throughput one. We used the same priorities of the Priority partitioning policy, setting a constraint of 400s on the TTC of *fluidanimate*. Results show how, under a cap of 40W and 30W, all the solutions stay under the time constraint, while under a stricter power cap of 20W, the Throughput partitioning policy penalizes the other two containers to permit our high priority workload to satisfy its time constraints.

7.5 Final remarks

This chapter concludes our journey towards power-awareness, exploring how a power-aware system can *plan* future decisions and *execute* the best actions with respect to performance goals and power constraints, thanks to the OLDA control loop introduced in Section 1.2. Moreover, it explores opportunities provided by *containerization*, i.e., a different approach to multi-

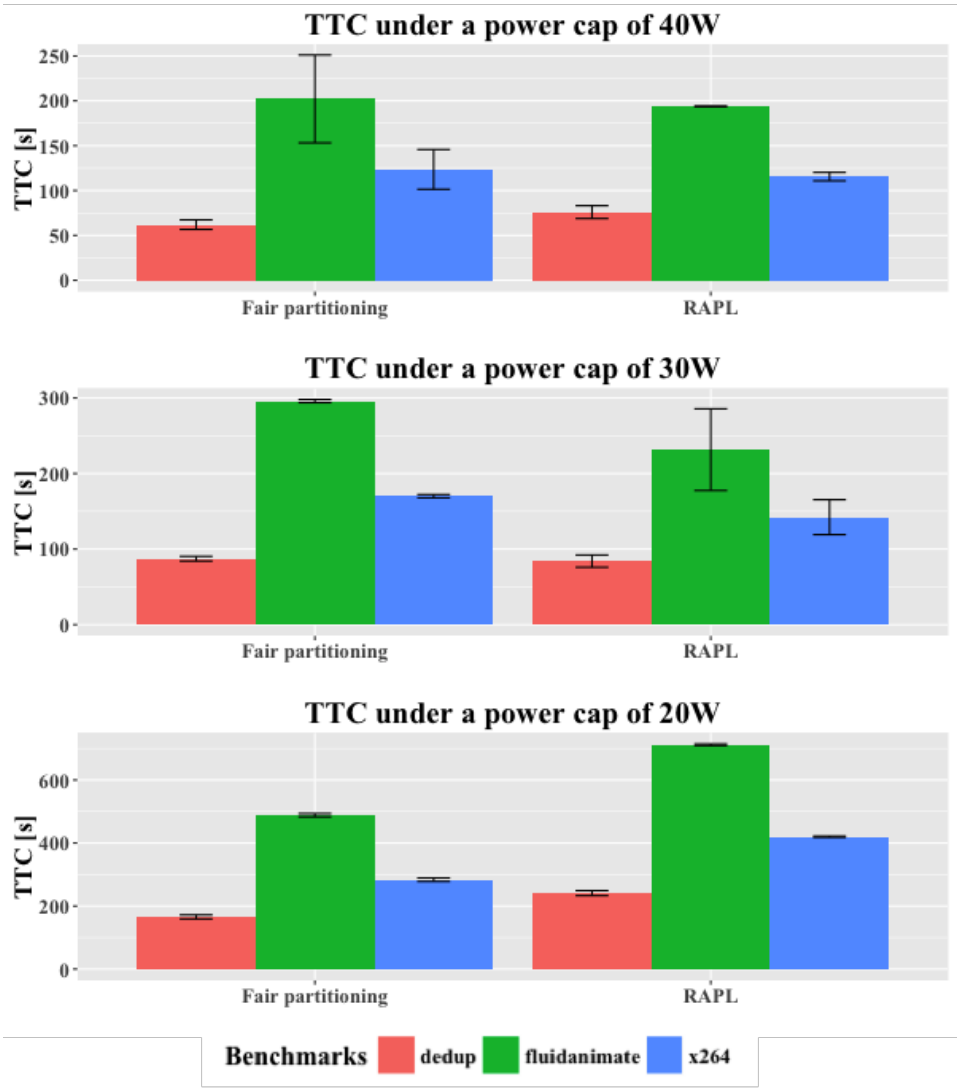


Figure 7.5: *Comparison of the TTCs obtained using RAPL and the Fair partitioning policy*

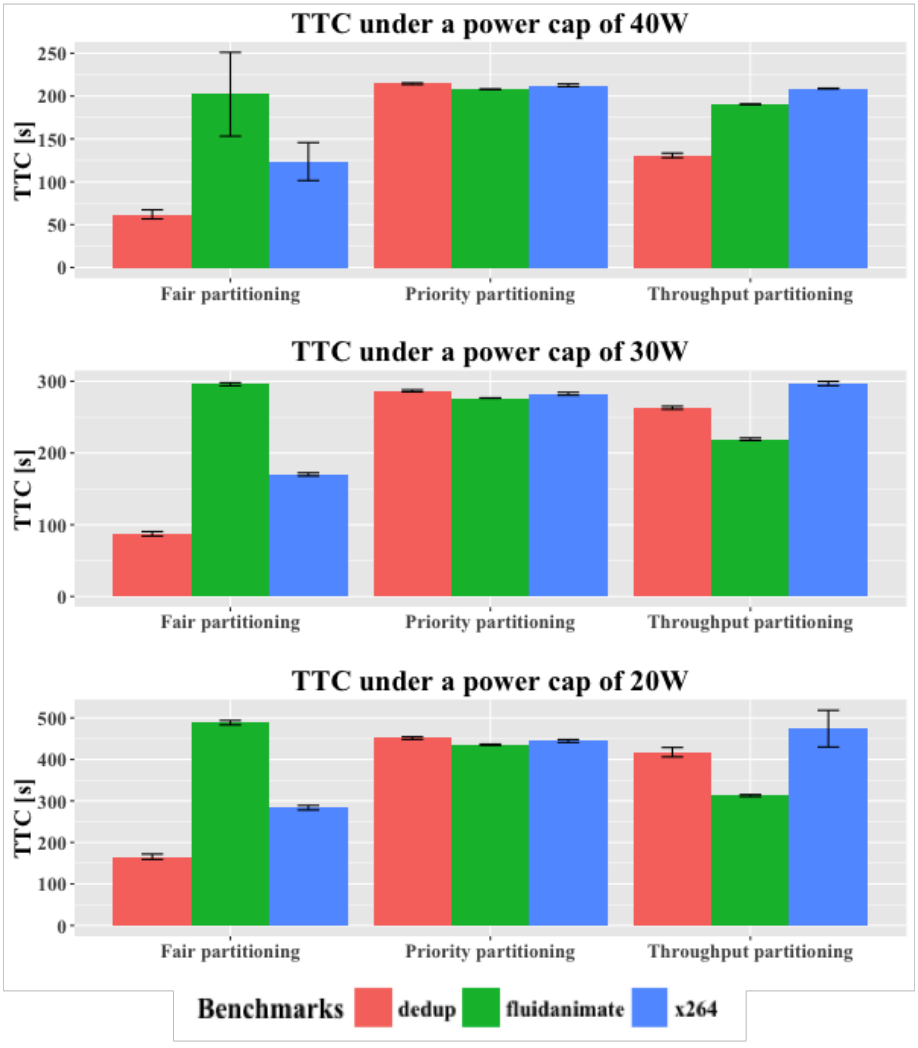


Figure 7.6: *Comparison of the TTCs obtained using the three proposed policies. Priority and Throughput policies explicitly target an improvement on the TTCs of the fluidanimate benchmark*

tenancy.

The current implementation still suffers of oscillations around the power cap. Those variations lead to peaks in the power consumption of the system, making our software-level power capping solution less stable than the RAPL one. The main reason of this issue could be due to the offline estimation of the parameter of the model: a possible solution to this issue could be the adoption of techniques that learn the parameters of the model at runtime, like Recursive Least Squares (RLS) [72] or a Kalman Filter [88]. Furthermore, the current solution only involves the PARSEC benchmark suite [22], characterized by multi-threaded workloads that scale well with the available cores. This may not be true for other types of workloads, thus leaving room for future work in the field of power-awareness and QoS-awareness for Docker container orchestration.

CHAPTER 8

Conclusion and future work

This chapter concludes our journey towards power-awareness for multi-tenant systems, from mobile devices to data centers. These two scenarios have been widely discussed throughout this thesis work, addressing the following questions:

A. *how much power is a system going to consume, given certain working conditions?*

B. *is it possible to control a system to consume less power, still satisfying its functional requirements?*

For what concerns mobile devices, the first question has been addressed in Chapter 2, presenting the novel contributions developed during my first year of Ph.D. on the *MPower* project. Then, Chapter 3 discussed how it is possible to generalize the same concepts towards a comprehensive and general methodology for power consumption modeling, codename *MARC*: this has been validated in Chapter 5 to answer the first question in the context of multi-tenant virtualized infrastructures, enabled by *XeMPower*, the lightweight monitoring tool for the Xen hypervisor presented in Chapter 4.

The experience developed on power consumption models for server infrastructures led me to the design of a *power-aware* and *QoS-aware* orches-

trator for multi-tenant systems, thus tackling the second question in the last two chapters. The same *XeMPower* tool represented the starting point in the development of *XeMPUPiL*, a performance-aware power capping orchestrator for Xen that aims at maximizing the performance of a workload under a power cap, as discussed in Chapter 6. Finally, I brought the same concepts into a different approach to multi-tenancy, i.e., *containerization*: Chapter 7 presented *DockerCap*, moving the first step towards power-awareness and QoS-awareness for Docker container orchestration.

Current limitations and future work of each step of this journey have already been discussed at the end of every chapter, in each *"Final remarks"* section. In a more broader and comprehensive vision, future directions revolve around the development of hybrid orchestration approaches: the idea is to combine hardware-level (e.g., the Intel RAPL interface) and OS-level (e.g., resource allocation) power management techniques with application-level *load-shedding* policies. This will allow to embrace performance bounds and achieve graceful services degradation in favor of other system-level requirements: this research is left for the next journey, towards a complete *collaborative* and *power-aware* multi-tenant system.

Bibliography

[1] Clockticks per instructions retired (cpi). https://software.intel.com/en-us/node/544403. Accessed: 2016-06-01.

[2] The embedded and automotive team within the xen project. https://www.xenproject.org/developers/teams/embedded-and-automotive.html. Accessed: 2016-09-17.

[3] Openbenchmarking.org. https://openbenchmarking.org/test/pts/cachebench. Accessed: 2016-06-01.

[4] Tuning Xen for performance. http://wiki.xenproject.org/wiki/Tuning_Xen_for_Performance. Accessed: 2015-11-19.

[5] The unofficial linux perf events web-page. http://web.eece.maine.edu/~vweaver/projects/perf_events/. Accessed: 2015-11-13.

[6] *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume B. 2015. 19-2.

[7] Hisahisabanbi App. Battery chart. https://play.google.com/store/apps/details?id=net.hisahisabanbi.btchart, december 2013.

[8] 3C Portal. Battery monitor widget. https://play.google.com/store/apps/details?id=ccc71.bmw, december 2013.

[9] Bilge Acun, Phil Miller, and Laxmikant V. Kale. Variation among processors under turbo boost in hpc systems. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 6:1–6:12, New York, NY, USA, 2016. ACM.

[10] Ishtiaq Ali and Natarajan Meghanathan. Virtual machines and networks-installation, performance study, advantages and virtualization options. *arXiv preprint arXiv:1105.0061*, 2011.

[11] Ethem Alpaydin. *Introduction to machine learning*. The MIT Press, Cambridge, MA, 2004.

[12] Aijun An, Christine Chan, Ning Shan, Nick Cercone, and Wojciech Ziarko. Applying knowledge discovery to predict water-supply consumption. *IEEE Expert*, 12(4):72–78, 1997.

[13] Vlasia Anagnostopoulou, Susmit Biswas, Heba Saadeldeen, Ricardo Bianchini, Tao Yang, Diana Franklin, and Frederic T Chong. Power-aware resource allocation for cpu-and memory-intense internet services. In *Energy Efficient Data Centers*, pages 69–80. Springer, 2012.

Bibliography

[14] Manish Anand, Edmund B. Nightingale, and Jason Flinn. Ghosts in the machine: interfaces for better power management. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, MobiSys '04, pages 23–35, New York, NY, USA, 2004. ACM.

[15] Apache. Akka framework, 2016.

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

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[18] Davide B Bartolini, Filippo Sironi, Donatella Sciuto, and Marco D Santambrogio. Automated fine-grained cpu provisioning for virtual machines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):27, 2014.

[19] Frank Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 37–42. ACM, 2000.

[20] Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, Albert Zomaya, et al. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in computers*, 82(2):47–111, 2011.

[21] Andreas Bergen, Nina Taherimakhsousi, and Hausi A Müller. Adaptive management of energy consumption using adaptive runtime models. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 120–126. IEEE Press, 2015.

[22] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[23] W Lloyd Bircher and Lizy K John. Complete system power estimation: A trickle-down approach based on performance events. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 158–168. IEEE, 2007.

[24] William Lloyd Bircher, Madhavi Valluri, Jason Law, and Lizy K John. Runtime identification of microprocessor energy saving opportunities. In *ISLPED'05. Proceedings of the 2005 International Symposium on Low Power Electronics and Design, 2005.*, pages 275–280. IEEE, 2005.

[25] Christopher M Bishop et al. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[26] Sergio Bittanti. *Teoria della Predizione e del Filtraggio*. Pitagora, 2002.

[27] A Bonetto, M Ferroni, D Matteo, AA Nacci, M Mazzucchelli, D Sciuto, and MD Santambrogio. Mpower: towards an adaptive power management system for mobile devices. In *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, pages 318–325. IEEE, 2012.

[28] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[29] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, August 2000.

[30] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[31] A. Charlesworth. The ascent of smartphone. *Engineering & Technology*, 4(3):32–33, 2009.

[32] Guanling Chen, David Kotz, et al. A survey of context-aware mobile computing research. Technical report, Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, 2000.

[33] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2007.

[34] Ryan Cochran, Can Hankendi, Ayse K Coskun, and Sherief Reda. Pack & cap: adaptive dvfs and thread packing under power caps. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pages 175–185. ACM, 2011.

[35] FFmpeg Community. Ffmpeg. `https://trac.ffmpeg.org/wiki/Null`. Accessed: 2016-08-09.

[36] Oracle Corporation. Mysql benchmark tool. `https://dev.mysql.com/downloads/benchmarks.html`. Accessed: 2016-08-09.

[37] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[38] Manoranjan Dash and Huan Liu. Feature selection for classification. *Intelligent data analysis*, 1(3):131–156, 1997.

[39] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194. IEEE, 2010.

[40] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *18th ACM Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–88, 2013.

[41] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *19th ACM Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 127–144, 2014.

[42] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F Wenisch, and Ricardo Bianchini. Multiscale: memory system dvfs with multiple memory controllers. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 297–302. ACM, 2012.

[43] Qingyuan Deng, David Meisner, Arup Bhattacharjee, Thomas F Wenisch, and Ricardo Bianchini. Coscale: Coordinating cpu and memory system dvfs in server systems. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 143–154. IEEE, 2012.

[44] Electronic Educational Devices. Watts up? plug load meters. `https://www.wattsupmeters.com/secure/products.php?pn=0&wai=0&more=2`. Accessed: 2016-08-07.

[45] Gaurav Dhiman, Kresimir Mihic, and Tajana Rosing. A system for online power prediction in virtualized environments using gaussian mixture models. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 807–812. IEEE, 2010.

Bibliography

[46] NASA Advanced Supercomputing Division. Nas parallel benchmarks (npb). http://www.nas.nasa.gov/publications/npb.html, 2016.

[47] Docker. Docker - build, ship, and run any app, anywhere, 2013.

[48] Mian Dong and Lin Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 335–348, New York, NY, USA, 2011. ACM.

[49] Norman R Draper and Harry Smith. *Applied regression analysis*. John Wiley & Sons, 2014.

[50] Naeem Esfahani, Eric Yuan, Kyle R Canavera, and Sam Malek. Inferring software component interaction dependencies for adaptation support. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(4):26, 2016.

[51] Giulio Eulisse and Lassi Tuura. Igprof profiling tool. 2005.

[52] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. Diversity in smartphone usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 179–194. ACM, 2010.

[53] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 13–23. ACM, 2007.

[54] Funmilade Faniyi, Peter R Lewis, Rami Bahsoon, and Xin Yao. Architecting self-aware software systems. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 91–94. IEEE, 2014.

[55] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[56] Wes Felter, Karthick Rajamani, Tom Keller, and Cosmin Rusu. A performance-conserving approach for reducing peak power consumption in server systems. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 293–302. ACM, 2005.

[57] M. Ferroni, A. Cazzola, F. Trovo, D. Sciuto, and M.D. Santambrogio. On power and energy consumption modeling for smart mobile devices. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 273–280, Aug 2014.

[58] Matteo Ferroni, Andrea Cazzola, Domenico Matteo, Alessandro Antonio Nacci, Donatella Sciuto, and Marco Domenico Santambrogio. Mpower: gain back your android battery life! In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, pages 171–174. ACM, 2013.

[59] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[60] Apache Foundation. Apache cassandra. http://cassandra.apache.org/. Accessed: 2016-08-09.

[61] Apache Foundation. Benchmark suite for apache spark. https://github.com/SparkTC/spark-bench. Accessed: 2016-08-09.

[62] Jesús García-galán, Liliana Pasquale, Pablo Trinidad, and Antonio Ruiz-Cortés. User-centric adaptation analysis of multi-tenant services. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(4):24, 2016.

[63] Gartner. Gartner says worldwide smartphone sales grew 3.9 percent in first quarter of 2016. http://www.gartner.com/newsroom/id/3323017.

[64] Google. Android 5.0, lollipop. https://www.android.com/intl/it_it/versions/lollipop-5-0/.

[65] Google. Power profiles for android, december 2013.

[66] Google Play Store. Battery notify,prediction lite. https://play.google.com/store/apps/details?id=com.fred.BatteryPredictionLite, december 2013.

[67] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. 17(6):120–126, 1982.

[68] Diwaker Gupta, Rob Gardner, and Ludmila Cherkasova. Xenmon: QoS monitoring and performance profiling tool. *Hewlett-Packard Labs, Tech. Rep. HPL-2005-187*, 2005.

[69] Selim Gurun and Chandra Krintz. A run-time, feedback-based energy estimation model for embedded devices. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, pages 28–33, New York, NY, USA, 2006. ACM.

[70] Hyung Kil Ham and Young Bom Park. Mobile application compatibility test system design for android fragmentation. In *Software Engineering, Business Continuity, and Education*, pages 314–320. Springer, 2011.

[71] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 83–92. IEEE, 2012.

[72] Monson H Hayes. *Statistical digital signal processing and modeling*. John Wiley & Sons, 2009.

[73] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, pages 11–16. ACM, 2008.

[74] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback control of computing systems*. John Wiley & Sons, 2004.

[75] Jorg Henkel, Heba Khdr, Santiago Pagani, and Muhammad Shafique. New trends in dark silicon. In *52nd ACM/EDAC/IEEE Design Automation Conference (DAC),*, pages 1–6, 2015.

[76] Nikolas Roman Herbst, Samuel Kounev, Andreas Weber, and Henning Groenda. Bungee: an elasticity benchmark for self-adaptive iaas cloud environments. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 46–56. IEEE Press, 2015.

[77] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats: A generic interface for expressing performance goals and progress in self-tuning systems. In *4th Workshop on Statistical and Machine learning approaches to ARchitecture and compilaTion (SMART)*, 2010.

[78] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats for software performance and health. Technical report, August 2009.

[79] Henry Hoffmann and Martina Maggio. Pcp: A generalized approach to optimizing performance under power constraints through resource management. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 241–247, 2014.

[80] Henry Hoffmann, Martina Maggio, Marco D Santambrogio, Alberto Leva, and Anant Agarwal. Seec: A framework for self-aware computing. 2010.

[81] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. Dynamic voltage scaling in multitier web servers with end-to-end delay control. In *Computers, IEEE Transactions*. IEEE, 2007.

Bibliography

[82] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1):489–501, 2006.

[83] Monsoon Solutions Inc. Monsoon power monitor. `https://www.msoon.com/LabEquipment/PowerMonitor/`, 2015.05.29.

[84] Intel. Flexible, low power microservers for lightweight scale-out workloads. Technical report, White paper, Intel Corporation, 2013.

[85] IOzone.org. Iozone filesystem benchmark. `http://www.iozone.org`, 2007.

[86] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93. IEEE Computer Society, 2003.

[87] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004.

[88] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[89] Joon-Myung Kang, Sin seok Seo, and James Won-Ki Hong. Personalized battery lifetime prediction for mobile devices based on usage patterns. *Journal of Computing Science and Engineering*, 5(4):338–345, 2011.

[90] Aman Kansal, Feng Zhao, Jie Liu, Nupur Kothari, and Arka A Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 39–50. ACM, 2010.

[91] Paul Karger and Andrew Herbert. An augmented capability architecture to support lattice security and traceability of access. In *IEEE Symposium on Security and Privacy*, 1984.

[92] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[93] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[94] David H. K. Kim, Connor Imes, and Henry Hoffmann. Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[95] M. Kim, M. O. Stehr, C. Talcott, N. Dutt, and N. Venkatasubramanian. xtune: A formal methodology for crosslayer tuning of mobile embedded systems. In *ACM Trans. Embed. Comput. Syst. 11.4*. ACM, 2013.

[96] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv: optimizing the operating system for virtual machines. In *USENIX Annual Technical Conference*, pages 61–72, 2014.

[97] Mikkel Baun Kjærgaard and Henrik Blunck. Unsupervised Power Profiling for Mobile Devices. In *Proceedings of the 8th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 2011)*. Springer, 2011.

[98] Igor Kononenko. Estimating attributes: analysis and extensions of relief. In *Machine Learning: ECML-94*, pages 171–182. Springer, 1994.

[99] Rakesh Kumar and Shilpi Charu. Comparison between cloud computing, grid computing, cluster computing and virtualization.

[100] Charles L Lawson and Richard J Hanson. *Solving least squares problems*, volume 15. SIAM, 1995.

[101] K-J Lee and Kevin Skadron. Using performance counters for runtime temperature sensing in high-performance processors. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 8–pp. IEEE, 2005.

[102] John Levon and Philippe Elie. Oprofile: A system profiler for Linux, 2004.

[103] Chao Li, Rui Wang, Depei Qian, and Tao Li. Managing server clusters on renewable energy mix. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 11(1):1, 2016.

[104] Tao Li and Lizy Kurian John. Run-time modeling and estimation of operating system power consumption. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 160–171. ACM, 2003.

[105] Lennart Ljung. *System identification*. Wiley Online Library, 1999.

[106] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[107] Pattie Maes. Modeling adaptive autonomous agents. *Artificial life*, 1(1_2):135–162, 1993.

[108] P. C. Mahalanobis. On the generalised distance in statistics. In *Proceedings National Institute of Science, India*, volume 2, pages 49–55, April 1936.

[109] Pierre Simon marquis de Laplace. *Théorie analytique des probabilités*. V. Courcier, 1820.

[110] John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta. Evaluating the effectiveness of model-based power characterization. *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, pages 12–12, 2011.

[111] Reto Meier. *Professional Android 4 Application Development*. Wrox, 3rd edition, May 2012.

[112] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. Power management of online data-intensive services. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 319–330. IEEE, 2011.

[113] Paul Menage, P Jackson, and C Lameter. Cgroups. *Available on-line at: http://www. mjmwired. net/kernel/Documentation/cgroups. txt*, 2008.

[114] Aravind Menon, Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *1st ACM/USENIX Int'l Conference on Virtual Execution Environments*, pages 13–23, 2005.

[115] Mark F Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for high-performance computing. *ACM SIGOPS Operating Systems Review*, 40(2):8–11, 2006.

[116] S. Mohapatra, R. Cornea, H. Oh, K. Lee, M. Kim, N. Dutt, R. Gupta, A. Nicolau, S. Shukla, and N. Venkatasubramanian. A cross-layer approach for power performance optimization in distributed mobile systems. In *International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2005.

[117] Douglas C Montgomery and George C Runger. *Applied statistics and probability for engineers*. Wiley. com, 2010.

[118] Philip J. Mucci. Cachebench. `http://www.weblearn.hs-bremen.de/risse/RST/WS06/x86_SUN/Sourcen/LLCBench/www/cachebench.html`, 2007.

[119] P Murukutla. Single sign on for cloud. *Computing Sciences (ICCS)*, pages 176 – 179, 2012.

[120] AA Nacci, Francesco Trovò, Federico Maggi, Matteo Ferroni, Andrea Cazzola, Donatella Sciuto, and Marco D Santambrogio. Adaptive and flexible smartphone power modeling. *Mobile Networks and Applications*, 18(5):600–609, 2013.

Bibliography

[121] Ripal Nathuji and Karsten Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 265–278. ACM, 2007.

[122] NECST-Laboratory. Mpower sense library source code, december 2013.

[123] NECST-Laboratory. Mpower technical reports, december 2013.

[124] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[125] Ruslan Nikolaev and Godmar Back. Perfctr-Xen: a framework for performance counter virtualization. 46(7):15–26, 2011.

[126] Ali Yadavar Nikravesh, Samuel A Ajila, and Chung-Horng Lung. Towards an autonomic auto-scaling prediction system for cloud resource provisioning. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2015 IEEE/ACM 10th International Symposium on*, pages 35–45. IEEE, 2015.

[127] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[128] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 153–168, New York, NY, USA, 2011. ACM.

[129] D. Pnueli and C. Gutfinger. *Fluid Mechanics*. Cambridge University Press, 1992.

[130] Jon Pretty. Rapture, 2014.

[131] Qualcomm. Snapdragon batteryguru. `https://play.google.com/store/apps/details?id=com.xiam.snapdragon.app`, december 2013.

[132] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No power struggles: Coordinated multi-level power management for the data center. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 48–59. ACM, 2008.

[133] Krishna K Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: fine-grained power management for multi-core systems. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 302–313. ACM, 2009.

[134] Redislab. Redis, 2009.

[135] Andrew Colin Rice and Simon Hay. Decomposing power measurements for mobile devices. In *PerCom*, pages 70–78. IEEE Computer Society, 2010.

[136] Murray Rosenblatt et al. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, 27(3):832–837, 1956.

[137] Daniel Rossier. EmbeddedXen: A revisited architecture of the Xen hypervisor to support ARM-based embedded virtualization. *White paper, Switzerland*, 2012.

[138] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan, and Eliezer Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, (2):20–27, 2012.

[139] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14, 2009.

[140] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[141] Amir Ali Semnanian, Jeffrey Pham, Burkhard Englert, and Xiaolong Wu. Virtualization technology and its impact on computer hardware architecture. In *Eighth Int'l Conference on Information Technology: New Generations (ITNG)*, pages 719–724. IEEE, 2011.

[142] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power containers: An os facility for finegrained power and energy management on multicore servers. In *IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*. IEEE, 2015.

[143] Alex Shye, Benjamin Scholbrock, and Gokhan Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 168–178, New York, NY, USA, 2009. ACM.

[144] Filippo Sironi, Martina Maggio, Riccardo Cattaneo, Giovanni F Del Nero, Donatella Sciuto, and Marco D Santambrogio. Thermos: System support for dynamic thermal management of chip multi-processors. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 41–50. IEEE, 2013.

[145] Ibrahim Takouna, Wesam Dawoud, and Christoph Meinel. Accurate mutlicore processor power models for power-aware resource management. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 419–426. IEEE, 2011.

[146] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*, volume 2. Prentice hall Englewood Cliffs, 2002.

[147] Andrey Nikolayevich Tikhonov. On the stability of inverse problems. In *Dokl. Akad. Nauk SSSR*, volume 39, pages 195–198, 1943.

[148] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library OSes for multi-process applications. In *9th European Conference on Computer Systems*, 2014.

[149] Geoffrey KF Tso and Kelvin KW Yau. Predicting electricity energy consumption: A comparison of regression analysis, decision tree and neural networks. *Energy*, 32(9):1761–1768, 2007.

[150] Narseo Vallina-Rodriguez and Jon Crowcroft. Energy management techniques in modern mobile handsets. *Communications Surveys & Tutorials, IEEE*, 15(1):179–198, 2013.

[151] Narseo Vallina-Rodriguez, Pan Hui, Jon Crowcroft, and Andrew Rice. Exhausting battery statistics: understanding the energy demands on mobile handsets. In *Proceedings of the second ACM SIGCOMM workshop on Networking, systems, and applications on mobile handhelds*, MobiHeld '10, pages 9–14, New York, NY, USA, 2010. ACM.

[152] Jóakim von Kistowski, Nikolas Herbst, Daniel Zoller, Samuel Kounev, and Andreas Hotho. Modeling and extracting load intensity profiles. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2015 IEEE/ACM 10th International Symposium on*, pages 109–119. IEEE, 2015.

[153] Micha Vor Dem Berge, Georges Da Costa, Mateusz Jarus, Ariel Oleksiak, Wojciech Piatek, and Eugen Volk. Modeling data center building blocks for energy-efficiency and thermal simulations. In *Energy-Efficient Data Centers*, pages 66–82. Springer, 2014.

[154] Xiaorui Wang, Ming Chen, and Xing Fu. Mimo power control for high-density servers in an enclosure. *Parallel and Distributed Systems, IEEE Transactions on*, 21(10):1412–1426, 2010.

[155] Ye Wen, Rich Wolski, and Chandra Krintz. Online prediction of battery lifetime for embedded and mobile devices. In *IN PROCEEDINGS OF PACS*, page 2004, 2003.

Bibliography

[156] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *2011 IEEE Int'l Conference on Embedded Software*, pages 39–48, 2011.

[157] Yu Xiao, Rijubrata Bhaumik, Zhirong Yang, Matti Siekkinen, Petri Savolainen, and Antti Yla-Jaaski. A system-level model for runtime power estimation on mobile devices. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, pages 27–34, Washington, DC, USA, 2010. IEEE Computer Society.

[158] Xia Xie, Haiou Jiang, Hai Jin, Wenzhi Cao, Pingpeng Yuan, and Laurence Tianruo Yang. Metis: a profiling toolkit based on the virtualization of hardware performance counters. *Human-centric Computing and Information Sciences*, 2(1):1–15, 2012.

[159] C. Xu, Z. Zhao, H. Wang, and J. Liu. On the interplay between network traffic and energy consumption in virtualized environment: An empirical study. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 392–399, June 2014.

[160] Hailong Yang, Qi Zhao, Zhongzhi Luan, and Depei Qian. imeter: An integrated vm power model based on performance profiling. *Future Generation Computer Systems*, 36:267–286, 2014.

[161] F. Zappa. *Elettronica. Semiconduttori, diodi e transistori, amplificatori, convertitori DAC e ADC*. Esculapio, 2008.

[162] Yan Zhai, Xiao Zhang, Stephane Eranian, Lingjia Tang, and Jason Mars. Happy: Hyperthread-aware power profiling dynamically. In *USENIX Annual Technical Conference*, pages 211–217, 2014.

[163] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[164] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.

[165] Xia Zhao, Yao Guo, Qing Feng, and Xiangqun Chen. A system context-aware approach for battery lifetime prediction in smart phones. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 641–646, New York, NY, USA, 2011. ACM.

[166] Jiang Zhu, Douglas S Chan, Mythili Suryanarayana Prabhu, Prem Natarajan, Hao Hu, and Flavio Bonomi. Improving web sites performance using edge servers in fog computing architecture. In *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, pages 320–323. IEEE, 2013.

[167] Zhenyun Zhuang, Kyu-Han Kim, and Jatinder Pal Singh. Improving energy efficiency of location sensing on smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 315–330. ACM, 2010.

[168] Parisa Zoghi, Mark Shtern, Marin Litoiu, and Hamoun Ghanbari. Designing adaptive applications deployed on cloud environments. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 10(4):25, 2016.