Multi-Dimensional Analysis of Software Power Consumptions in Multi-Core Architectures

Maxime Colmant

Université de Lille 1 SCIENCES ET TECHNOLOGIES

CRIStAL Centre de Recherche en Informatique, Signal et Automatique de Lille

ADEME Agence de l'Environnement et de la Maîtrise de l'Energie

Inria INVENTEURS DU MONDE NUMÉRIQUE

Multi-Dimensional Analysis of Software Power Consumptions in Multi-Core Architectures

THÈSE

pour l'obtention du

DOCTORAT Informatique

École doctorale Sciences Pour l'Ingénieur (Lille)

Écrite par

Maxime Colmant

présentée et soutenue publiquement le 24 Novembre 2016

devant le jury composé de :

Mr.	**Olivier**	**Barais**	**Université Rennes 1**	**Rapporteur**
Mr.	**Rüdiger**	**Kapitza**	**Université Braunschweig**	**Rapporteur**
Mr.	**Giuseppe**	**Lipari**	**Université Lille 1**	**Président**
Mme.	**Anne-Cécile**	**Orgerie**	**CNRS**	**Examinatrice**
Mr.	**Romain**	**Rouvoy**	**Université Lille 1**	**Directeur**
Mr.	**Lionel**	**Seinturier**	**Université Lille 1**	**Directeur**
Mr.	**Alain**	**Anglade**	**ADEME**	**Invité**

Acknowledgements

First of all, I would like to really thank my supervisors, Romain and Lionel. More particularly Romain without whom I would never have started a PhD after my master degree. Thank you for taking me under your wing and taking my wish into consideration. Thank you again for your constant support, your everyday precious advices, all our interesting discussions over a coffee and, for being patient when I missed (a few) deadlines. Even if I still have a long way to go, I am now a young researcher thanks to you.

Next, I would like to thank the members of my thesis committee. Rüdiger Kapitza and Olivier Barais, thanks for having accepted to review my manuscript and for the valuable comments you gave me. I really appreciated your feedbacks. Moreover, I would like to thank Giuseppe Lipari and Anne-Cécile Orgerie to have accepted to be part of the committee, I am very honored.

I also would like to really thank all the members of the Spirals team. It was a great pleasure to share ideas and discussions with you. Special dedications to the other PhD students who have started their thesis at the same time as me: Geoffrey, María, and Bo. We started together, now we end together! Thanks all for your support, especially the last few months where we supported each other during the tedious process of the thesis writing. Special thanks to Geoffrey that I know for more than 5 years now, I really appreciate your hospitality and your time when I came to Montréal. We shared a lot of “last-minute” student projects together. Even if it is was hard sometimes, we did it and we can be proud to be where we are today. I hope we will continue to share beers together, you are the master and I still have a lot of “homework” to do before to surpass you. Another special thanks to María, it was a great pleasure to meet you and to share these three years with you. I would like to apologize for the bad advice on the activity tracker you bought, I hope the bugs will be fixed soon! I will not forget all members of the team with whom we share a lot of interesting discussions (Loïc, Julien, Antoine, Christophe (x2), Fawaz, Clément, Nicolas, Matis ...). Sorry for those I forgot :-) For all, I wish you all the best for the future.

Thanks to the French Environment & Energy Management Agency (ADEME) and the University of Lille 1 for financially supporting this thesis, and to Alain Anglade who followed and advised me during these three years.

Finally, I would like to thank my family. Lot of thanks to my parents, Myriam and Jean, to my sister Laetitia, to my brother-in-law Olivier, my fresh air breathes, my nephew Nathan and my goddaughter Romane. Thanks for believing me during these three years, I would not be who I am without you. Many thanks to my partner’s family, René for the swimming pool sessions and discussions, Corinne for her delicious dishes, Coralie, Anthony, Clarisse, Sébastien, Mickaël for all the good times we shared together. Thanks to my friends, Angélique, Dona, Nicolas, and Alexis, who encouraged me when I took the decision to start a PhD.

Last but not least, I would like to say a special thank to my beloved girlfriend, Mélanie. Thanks for supporting me, believing in me, and to loving me for more than 2 years now. I really appreciate all the efforts you made, and I am very grateful for that. You are the person who makes me smile every day, who inspiring me.

Abstract

Energy represents one of the largest cost factors when operating data centers, largely due to the consumption of air conditioning, network infrastructure, and host machines. Energy-efficient computing is becoming increasingly important. Among the reasons, one can mention the massive consumption of large data centers that consume as much as 180,000 homes. This trend, combined with environmental concerns, makes energy efficiency a prime technological and societal challenge. Currently, widely used power distribution units (PDUs) are often shared amongst nodes to deliver aggregated power consumption reports, in the range of hours and minutes. However, in order to improve the energy efficiency of software systems, we need to support process-level power estimation in real-time, which goes beyond the capacity of a PDU. In particular, the CPU is considered by the research community as the major power consumer within a node and draws attention while trying to model the system power consumption. Software power estimation of CPUs is therefore a central concern for energy efficiency and resource management in data centers. It is hence required to provide critical indicators for driving scheduling policies or power capping heuristics.

Over the last few years, a dozen of *ad hoc* power models have been proposed to cope with the wide diversity and the growing complexity of modern CPU architectures. However most of these CPU power models rely on a thorough expertise of the targeted architectures, thus leading to the design of hardware-specific solutions that can hardly be ported beyond the initial settings. We therefore promulgate in this thesis, PowerAPI, a middleware toolkit that implements learning techniques that automatically learn the power models of a CPU, independently of the features and the complexity it exhibits. The aforementioned learning techniques are not limited to CPUs and we therefore extend these techniques for other hardware components, such as SSD disks.

PowerAPI is built as a modular solution to assemble software-defined power meters "à la carte", thus fostering the wide adoption of power models. These solutions are customizable and can deliver power estimation reports at various frequencies upon user requirements. Furthermore, PowerAPI is designed to monitor concurrent applications on modern architectures, thus allowing accurate and efficient energy analysis.

With the emergence of cloud computing, the research community already proposed several solutions to monitor the power consumption of virtualized environments. State-of-the-art solutions fail to propose a required per-process power estimation inside VMs and rather propose to only consider the overall power estimation of VMs (as black boxes). Yet, VM-based systems are nowadays commonly used to host multiple applications for cost savings and better use of energy by sharing common resources and assets. We therefore

propose a middleware toolkit, BitWatts, built on top of PowerAPI that paves the way to a better monitoring and provides real-time and accurate power estimation of software processes running at any level of virtualization. In addition, we propose WattsKit to better understand how the power consumption of processes can be distributed across the nodes of a cluster.

Even if software-defined power meters are the first step to better understand and optimize the software power consumption, a finer level of estimation may be required to further evaluate the effectiveness (or ineffectiveness) of the software optimizations. One may want, for example, to compare 2 versions of a same software and thus observe the energy leaks or improvements brought by the modifications. To address these shortcomings, we develop an approach, codEnergy, that leverages the use of dynamic and interactive reports to better help the developers to analyze the energy distribution of methods and to easily detect deterioration and/or improvements.

All the aforementioned approaches and tools are deeply assessed in this thesis, thus demonstrating the usefulness of PowerAPI to better understand the software power consumption on modern architectures.

Résumé

L'Énergie représente l'un des principaux postes de dépense pour un centre de données, et est principalement liée à l'air climatisé, à l'infrastructure complexe du réseau sous-jacent et au grand nombre de machines utilisées. L'Informatique “verte” est dorénavant un enjeu majeur. Parmi les raisons principales, nous pouvons mentionner les centres de données qui consomment autant que 180 000 foyers en électricité. Associé aux préoccupations énergétiques, cet enjeu représente un challenge technologique et sociétal de premier ordre. Des wattmètres sont actuellement utilisés et partagés pour récupérer un ensemble agrégé de rapports énergétiques sur plusieurs heures ou minutes. Pour améliorer l'efficacité énergétique des logiciels, nous devons donc dépasser ces limitations et proposer des estimations plus fines, c'est-à-dire au niveau processus. Plus particulièrement, la communauté scientifique considère le CPU comme étant le composant le plus énergivore et est donc principalement considéré lors de la modélisation énergétique d'un système. L'estimation énergétique du processeur au niveau logiciel représente donc un enjeu majeur pour améliorer l'efficience énergétique et l'allocation des ressources des centres de données. Il est donc crucial de proposer des indicateurs critiques pour permettre l'application de nouvelles politiques d'ordonnancement ou de limitation énergétique.

Durant les dernières années, des dizaines de modèles de consommation ont été proposées pour prendre en compte la grande diversité et la complexité grandissante des architectures récentes de CPU. Cependant, la plupart de ces modèles se basent sur de profondes connaissances des architectures sous-jacentes, conduisant ainsi à la création de solutions spécifiques non évolutives. Nous proposons donc dans cette thèse, PowerAPI, un intergiciel qui implémente des techniques d'apprentissage permettant d'inférer les modèles de consommation d'un processeur, indépendamment de ses technologies et de sa complexité interne. Ces techniques ne sont cependant pas limitées au CPU et nous avons d'ores et déjà démontré leur applicabilité sur d'autres composants, comme les disques SSDs.

PowerAPI est une solution modulaire permettant d'assembler des wattmètres logiciels à la carte, favorisant ainsi l'adoption des modèles de puissance. Ces solutions sont configurables et peuvent délivrer des estimations énergétiques à des fréquences variées, répondant ainsi au mieux aux besoins des utilisateurs. De plus, PowerAPI a été conçu pour suivre énergétiquement des applications concurrentiels s'exécutant sur des architectures modernes, permettant ainsi des analyses énergétiques efficaces et précises.

Avec l'émergence de l'Informatique dématérialisée, la communauté scientifique propose déjà plusieurs solutions pour permettre de suivre la consommation énergétique au sein d'environnement virtualisé. Ces solutions échouent cependant quant à estimer la consommation à grain fin, c'est-à-dire au niveau applicatif, et se limitent à la consom-

mation globale de ces environnements. Cependant, ce type d'environnement est déjà couramment utilisé pour héberger plusieurs applications, minimisant ainsi les coûts et les ressources utilisées. Nous proposons pour cela, BitWatts, développé en complément de PowerAPI, pour permettre un meilleur suivi énergétique en estimant en temps réel la consommation des applications s'exécutant à différents niveaux de virtualisation. Nous proposons également une autre extension, WattsKit, permettant de mieux comprendre la consommation énergétique de services distribués au sein d'un ensemble de machines.

Les wattmètres logiciels sont la première étape pour mieux comprendre et optimiser la consommation des logiciels. Cependant, il est parfois nécessaire d'avoir un niveau encore plus fin pour mieux comprendre ce qu'il se passe au sein du logiciel. Par exemple, on pourrait vouloir comparer différentes versions d'un même logiciel pour mettre en évidence des fuites (ou optimisations) énergétiques induites par certaines modifications. Nous avons développé codEnergy pour pallier à ce problème. codEnergy favorise l'utilisation de rapports interactifs et dynamiques pour aider les développeurs à mieux analyser la répartition énergétique entre les différents composants d'un logiciel et ainsi détecter plus rapidement les améliorations et/ou détériorations énergétiques.

Toutes ces approches et outils susmentionnés ont été validés durant cette thèse, démontrant ainsi l'utilité de PowerAPI pour mieux analyser et comprendre les consommations logicielles sur les architectures modernes.

Table of Contents

Acronyms

ACPS	Active Cycles Per Second
BLKIO	Block IO Controller
CMOS	Complementary Metal-Oxide Semiconductor
CS	C-States
DVFS	Dynamic Voltage/Frequency Scaling
DWARF	Debugging With Attributed Record Formats
HPC	Hardware Performance Counters
HT	Hyper-Threading
ICT	Information and Communications Technology
IPC	Instruction Per Ccyle
IaaS	Infrastructure-as-a-Service
JIT	Just-In-Time
JVM	Java Virtual Machine
MSD	Mean Square Deviation
MPI	Message Passing Interface
NPB	NAS Parallel Benchmark
NRMSD	Normalized Root Mean Square Deviation
PaaS	Platform-as-a-Service
PDU	Power Distribution Unit
PMU	Performance Monitoring Unit
RAPL	Running Average Power Limit
RHEL	Red Hat Enterprise Linux

RMS	Recognition, Mining, and Synthesis
RTT	Round Trip Time
SMT	Simultaneous Multi-Threading
SaaS	Software-as-a-Service
SS	SpeedStep
TB	TurboBoost
TDP	Thermal Design Power
TLP	Thread-Level Parallelism
TSC	TimeStamp Counter
YCSB	Yahoo! Cloud Serving Benchmark

List of Figures

List of Tables

List of Snippets

Chapter 1

Introduction

Table of Contents

Energy-efficient computing is becoming increasingly important. Among the reasons, one can mention the massive power consumption of large data centers, estimated to account for about 2% of global greenhouse gas and some of which consume as much as 180,000 homes [The08; Coo12]. This trend, combined with environmental concerns, makes energy efficiency a prime technological and societal challenge.

Researchers and operators have been proposing solutions to increase energy efficiency at all levels, from application to runtime and to hardware. As surveyed by Surgerie *et al.* [ODL14], examples include methods for energy-based task scheduling, energy-efficient software, dynamic frequency and voltage scaling, and energy-aware workload consolidation using virtualization.

While trying to improve the intrinsic power consumption of runtime applications, one needs to have a powerful and lightweight energy monitoring solution. Such solution has to fulfill all system and user requirements, and must help developers and designers to build energy efficient software. To limit hardware investments, such solutions are often based on the design of power models. Since decades, researchers have already been proposed a dozen of *ad hoc* power models that fit a specific type of hardware components. However, every time a new architecture is released, one has to propose a new power model that fits its requirements. So far, the research community mainly paid a careful attention to the CPU hardware component that contributes for the most part of the overall power consumption of a system [ERK06; NRS15].

We introduce in this thesis different automatic approaches to learn automatically the CPU power models, regardless of their underlying architectures. Beyond CPU, one of these techniques has been reused to learn SSD power models and can therefore be extended to learn any kind of hardware power models. Beyond the learning techniques, we offer an open-testbed to foster the research on green computing and to offer a tool for building software-defined power meters “à la carte”. This solution can be then used to support the design of energy-aware scheduling heuristics in homogeneous systems [Bam+13;

Bel00; Mog+13; Ras15], or in heterogeneous data centers [KOS16], to serve the energy-proportional computing [BH07; Kri+10; Mei+11; Pre+15] or to evaluate the effectiveness of optimizations applied on binaries [Sch+14]. It also targets system administrators and software developers alike in monitoring and better understanding the power consumption of their software assets [NRS14; NRS15; Ste13].

On top of these approaches, we developed a toolkit, named PowerAPI, for assembling *software-defined* power meters upon needs. We define a *software-defined* power meter as a software solution that can achieve runtime power measurements or estimation at different granularities and frequencies.

Different scenarios are then described to demonstrate the effectiveness of PowerAPI to build accurate software power meters that can next be used for energy monitoring or optimizations.

Among the effervescence of virtualized environments, we propose an extension of PowerAPI, called BitWatts, to virtually hide all the levels of virtualization and to allow per process energy monitoring inside VMs. As services can now be distributed among several nodes which are not necessarily homogeneous, we define WattsKit to monitor and analyze the power consumption of a distributed systems spanning several nodes and thus proposing a finer granularity than observing the overall power consumption of nodes.

To better analyze the energy consumption of software assets, one may need to go deeper and get insights about how the energy is distributed among software assets. We describe in this purpose codEnergy as a scalable solution to build an interactive cartography of the software energy distribution among source-code and thus allowing developers to focus their efforts on energy hotspots.

This chapter is organized as follows. Section 1.1 introduces the problem statements extracted from the state-of-the-art. Section 1.2 exposes the goals of this thesis and shows how our contributions bring new solutions to face these problems. Section 1.3 describes all contributions detailed in this manuscript. Section 1.4 reports on all the papers and articles (published, under evaluation or to be published) contributed as part of this thesis. And finally, Section 1.5 summarizes the content of this thesis.

1.1 Problem Statement

Software power estimation of CPUs is a central concern for energy efficiency and resource management in data centers. Over the last decade, a dozen of *ad hoc* power models have been especially designed to cope with the wide diversity and the growing complexity of modern CPU architectures. However, most of CPU power models rely on a thorough expertise of the targeted architectures, thus leading to the design of hardware-specific solutions that can hardly be ported beyond the initial settings. More specifically, the state-of-the-art in this domain faces several key limitations, such as a simplified CPU architecture [Bir+05], the deprecation of the CPU model [IM03], the unavailability of the selected metrics [Zha+14], the handcrafted power models [LJ03], the unavailability of the benchmarks [Zha+14], and the limited diversity of tested workloads [Ber+10] to name a few. These limitations prevent from a wider adoption and deployment of power models, thus limiting the power monitoring of software.

Power monitoring is usually achieved by hardware measurement equipments, such as power meters or specialized integrated circuits. This solution is obviously not suitable at a larger scale, thus requiring costly hardware investments. Furthermore, this kind of power monitoring only provides power measurements at a coarse-grained level—*i.e.*, at machine-level. However, one needs to have a finer level for better optimizing the energy of the whole system by targeting the applications that are running on it. Consequently, several solutions have been proposed over the years to estimate the power consumption at software-level [FS99; DRS09; Int15b], and even at code-level [CV16; INB16; NRS15]. Nevertheless, most of existing solutions require hardware investments [Bed+10; Ge+10; LPD13; Ras+15], are not suitable while targeting concurrent applications [Ge+10; Ras+15] or lack of modularity [CV16; IM03; INB16; FS99; LBL07; NRS15; NIB16; Int15b].

Power estimation of software processes provide critical indicators to drive scheduling or power capping heuristics. State-of-the-art solutions are facing several key limitations and are often limited to specific hardware components and are not suitable while monitoring concurrent applications.

While considering virtualized environments, we can find solutions that propose coarse-grained power estimation in virtualized environments, typically treating *virtual machines* (VMs) as black boxes. Yet, VM-based systems are nowadays commonly used to host multiple applications for cost savings and better use of energy by sharing common resources and assets.

The design of energy-efficient distributed systems is a challenging task, which requires software engineers to consider all layers of a system, from hardware to software. In particular, monitoring and analyzing the power consumption of a distributed system spanning several—potentially heterogeneous—nodes becomes particularly tedious when aiming at a finer granularity than observing the power consumption of hosting nodes. The state-of-the-art fails to deliver adaptive solutions to offer this perspective and to cope with the diversity of architectures.

New kinds of diagnosis tools are required to better analyze the energy consumed by the software assets. Such level of monitoring requires to retrieve critical runtime informations about the software's call graph while running. These tools need to minimize the instrumentation of the target application in order to be as lightweight as possible and minimize the overhead and disruption.

1.2 Thesis Goals

The first goal of this thesis is to propose automatic learning approaches for hardware power models that do not rely on an *a priori* knowledge of the underlying architecture. With these approaches, everyone can build power models with minimal hardware investments. Once learned, power models can be coupled together with software-defined power meters for learning more about the software energy usages.

Next, we want to propose an efficient toolkit to build these software-defined power meters upon needs. We therefore believe of a solution that can bring the required modularity to end-users for fulfilling all their requirements. With such tool, one can imagine several use cases to better analyze the power consumed by concurrent applications or to find out which parameters can impact the software power consumption.

This thesis aims also to answer several new challenges raised by the diversity of dimensions that can be exploited by modern software systems. In particular, we go beyond the state-of-the-art by proposing an innovative way for better handling processes that run inside VMs and thus do not see them as black-boxes anymore. Moreover, we also propose an approach to allow the energy monitoring of distributed services, where no solutions exist for this problem. We finally describe a new approach for creating a wider energy cartography of the software source-code and thus helping developers to build energy efficient software.

1.3 Contributions

This thesis first proposes an open testbed to leverage the research on power models. More specifically, 3 techniques that accurately learn the power models of CPU and SSD hardware components are described. Our CPU learning approaches have been assessed on the main CPU manufacturers—*i.e.*, Intel, AMD, ARM—and exhibits an average accuracy greater than 95% at runtime when compared to power measurements. We extend one of the CPU learning approach to SSD component and thus demonstrating the generalization of our techniques. This extended technique for SSDs exhibit an average accuracy of 99%. We believe that these approaches can therefore be extended to other power consuming components, such as GPU [JSM12], in order to incrementally learn their power model and thus provide a wider cartography of the power consumption of a software system.

These learning approaches described in this manuscript are combined to build accurate component power models. They can furthermore be used to produce accurate per-process power estimation upon requirements. We therefore propose PowerAPI, a middleware toolkit for building software-defined power meters "à la carte". A software-defined power meter is defined as a software solution that allows to propose several power estimation (or measurements) granularities at different frequencies. The main purpose of PowerAPI is to be as modular as possible in order to fit all user requirements. PowerAPI can be thus used on top of various power models to build software-defined power meters. Several scenarios are presented in this thesis, thus demonstrating the modularity and the usability of PowerAPI to monitor multiple concurrent processes at runtime in various situations.

Based on these power models, we propose a fined-grained monitoring middleware that provides real-time and accurate power estimation of software processes running at any virtualized level. Our middleware implementation, BitWatts, is built on top of PowerAPI, and uses high-throughput communication channels to spread the power consumption across the VM levels and between machines. This non-invasive monitoring solution therefore paves the way for scalable energy accounting that takes into account the dynamic nature of virtualized environments.

To better understand how the power consumption of the system's processes is distributed across nodes at runtime, we define a new specific software-defined power meter, WattsKit. Beyond the demonstrated capability of covering a wide diversity of nodes with high accuracy, we show the benefits of adopting software-defined power meters to analyze the power consumption of legacy complex distributed systems.

We finally propose a novel approach, codEnergy, for building fined-grained interactive energy reports about the software energy usages. This approach paves the way for better analyzing the software power consumption and thus allowing to observe energy improvements or decreases during the development process.

1.4 Publications

Several works presented in this manuscript are under evaluation, not yet published or about to be submitted while writing this manuscript. We present below all published, submitted and upcoming papers related to the work presented through this manuscript.

Conferences

[Col+15b] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. "Process-level Power Estimation in VM-based Systems". In: *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. 2015.

[CRS14] M. Colmant, R. Rouvoy, and L. Seinturier. "Improving the Energy Efficiency of Software Systems for Multi-Core Architectures". In: *Middleware 2014 Doctoral Symposium*. 2014.

[CRS15a] M. Colmant, R. Rouvoy, and L. Seinturier. "Estimation de la consommation des systèmes logiciels sur des architectures multi-coeurs". In: *Conférence d'informatique en Parallélisme, Architecture et Système (Compas)*. 2015.

[Hav+ar] A. Havet, V. Schiavoni, P. Felber, M. Colmant, R. Rouvoy, and C. Fetzer. "GENPACK: A Generational Scheduler for Cloud Data Centers". In: *IEEE International Conference on Cloud Engineering (IC2E)*. 2017. (To appear).

Posters

[Col+14] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. *BitWatts: A Process-level Power Monitoring Middleware*. Middleware- Poster session. 2014.

[Col+15a] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. *Process-level Power Estimation in VM-based Systems*. European Conference on Computer Systems (EuroSys) - Poster session. 2015.

[CRS15b] M. Colmant, R. Rouvoy, and L. Seinturier. *Process-level Power Estimation in Multi-core Architectures*. Conférence d'informatique en Parallélisme, Architecture et Système (Compas) - Poster session. 2015.

[Kur+14] M. Kurpicz, M. Colmant, L. Huertas, A. Sobe, P. Felber, and R. Rouvoy. *How energy-efficient is your cloud app?* Conférence d'informatique en Parallélisme, Architecture et Système (Compas) - Poster session. 2014.

Under Evaluation

Journals

[Col+16] M. Colmant, R. Rouvoy, M. Kurpicz, P. Felber, A. Sobe, and L. Seinturier. "The Next 700 CPU Power Models". In: *ACM Trans. Model. Perform. Eval. Comput. Syst. (ACM TOMPECS)* (2016).

Future Submissions

Conferences

[Col+17] M. Colmant, R. Rouvoy, M. Kurpicz, P. Felber, A. Sobe, and L. Seinturier. "WattsKit: Software-Defined Power Monitoring of Distributed Systems". In: *To be chosen*. 2017.

[CRS17] M. Colmant, R. Rouvoy, and L. Seinturier. "codEnergy: an Approach For Leveraging Source-Code Level Energy Analysis". In: *To be chosen*. 2017.

1.5 Outline

The remainder of the document is organized as follows. We first present the research background that motivates this thesis in Part I. Chapter 2 introduces all recent research approaches for learning power models, while Chapter 3 present different tools that can estimate or measure the power consumption at different granularities.

Secondly, Part II describes our contributions to address the state-of-the-art limitations. In Chapter 4, we propose 2 learning approaches for modern CPU power models, and 1 for SSD power models. All the described approaches aim to build efficient and non-invasive runtime power models that can be used at runtime to estimate the power consumption at software-level in real-time. In Chapter 5, we describe our middleware toolkit, PowerAPI, that builds on top of the state-of-the-art solutions and is as modular as possible. The learned power models can be used together with PowerAPI to compute runtime power estimation of concurrent processes.

Next, Part III validates the contributions of this thesis. Chapter 6 assesses the learning approaches detailed in Part II, and demonstrates the applicability of PowerAPI in various monitoring scenarios. We next propose to go beyond the state-of-the-art in Chapter 7 and Chapter 8. Chapter 7 proposes to extend PowerAPI for estimating the power consumption of applications that run inside virtualized environments and thus seeing VMs as white boxes—*i.e.*, virtualization becomes transparent. In addition, we describe another extension for building a finer view of the power consumed by distributed services that run inside a cluster of heterogeneous nodes. Chapter 8 reports on our proposal for better understanding the power consumption drawn by the software and therefore proposes an energy diagnosis tool for building interactive energy source-code reports.

We finally conclude and present new research perspectives in Chapter 9.

Part I

State-of-the-Art

Chapter 2

Learning Power Models

Table of Contents

As current multi-core platforms do not provide fine-grained power measurement capabilities, McCullough *et al.* [McC+11] argue that power models are the first step towards enabling dynamic power management for power proportionality at all levels of a system. Power modeling often consider learning techniques—for example based on sampling [Ber+12]—that assume the proportionality of the system events to power consumption. Measurements of a hardware power meter are gathered and subsequently used, together with a set of normalized estimated values, in various regression models.

The remainder of this chapter is organized as follows. Section 2.1 discusses the state-of-the-art of CPU power models, while Section 2.2 describes the VM power models and Section 2.3 the disk power models.

2.1 CPU Power Models

Along the last decade, the design of CPU power models has been regularly considered by the research community [Bel00; Col+15b; Kan+10; McC+11; VWT13]. Currently, the closest approach to hardware-based monitoring is RAPL, introduced with the Intel "Sandy Bridge" architecture to report on the power consumption of the entire CPU package. As this feature is not available on other architectures and is not always accurate [Col+15b], CPU power models are generally designed based on a wider diversity of raw metrics.

Standard operating system metrics (CPU, memory, disk, or network), directly computed by the kernel, tend to exhibit a large error rate due to their lack of precision [Kan+10; VWT13]. Contrary to usage statistics, hardware performance counters (HPC) can be obtained from the processor (*e.g.*, number of retired instructions, cache misses, non-halted cycles). Modern processors provide a variable number of HPC, depending on architectures and generations. As shown by Bellosa [Bel00] and Bircher [BJ07], some HPC are highly correlated with the processor power consumption whereas the authors in [RRK08] conclude that several performance counters are not useful as they are not directly correlated with dynamic power. Nevertheless, this correlation depends on the processor architecture and

Table 2.1: Summary of existing CPU power models.

Author(s)		**Processor(s)**	**Feature(s)**	**Regression(s)**	**Benchmarks**	**Error(s)**
A.Aroca *et al.*	[Arj+14]	Xeon W3530	HW sensors	polynomial, multiple linear	*eval.:* Hadoop App.	<7% of total energy
Bertran *et al.*	[Ber+10]	Core 2 Duo	14 HPCs regrouped by component	multiple linear by component	*sampl.:* μ-benchs *eval.:* SPEC CPU 06	5%
Bircher *et al.*	[Bir+05]	Pentium 4	μ-ops trace-cache, micro-code ROM	multiple linear	*sampl.:* SPEC CPU 00 *eval.:* SPEC CPU 00	2.5%
Contreras *et al.*	[CM05]	XScale PXA255	5 HPCs	multiple linear	*eval.:* SPEC CPU 00, Java CDC/CLDC	4%
Dolz *et al.*	[Dol+15]	Xeon E3-1275	3 HPCs HW sensors	linear	*sampl.:* linpack, stream, iperf, IOR *eval.:* Quantum Espresso	3 W 70 W max.
Economou *et al.*	[ERK06]	Turion, Itanium 2	HW sensors	multiple linear	*sampl.:* Gamut *eval.:* SPECs, Matrix, Stream	5%
Isci *et al.*	[IM03]	Pentium 4	15 HPCs	multiple linear	*eval.:* μ-benchs, AbiWord, Mozilla, Gnumeric	3 W
Li *et al.*	[LJ03]	8-way issue superscalar	IPC	linear	*sampl./eval.:* DB, email, SPEC JVM 98, SPEC INT 95	1% off. 6% run.
Rivoire *et al.*	[RRK08]	Core 2 Duo & Xeon, Itanium 2, Turion	HW sensors HPCs	multiple linear	*sampl.:* calibration suite *eval.:* SPECs, stream, Nsort	<5%
Yang *et al.*	[Yan+14]	Xeon E5620 & E7530	7 components 91 preselected	support vector	*sampl.:* NPB, IOzone, CacheBench *eval.:* SPEC CPU 06, IOzone	4.7%
Zamani *et al.*	[ZA12]	Opteron	3 to 5 HPCs	ARMAX	*sampl./eval.:* BT.C, CG.C, LU.C, SP.C	0.1% – 0.5% offline
Zhai *et al.*	[Zha+14]	Sandy Bridge	non-halted cycles	linear	*eval.:* Google, SPEC CPU 06	7.5%

the CPU power model computed using some HPCs may not be ported to different settings and architectures. Furthermore, the number of HPC that can be monitored simultaneously is limited and depends on the underlying architecture [Int15a], which also limits the ability to port a CPU power model on a different architecture. Therefore, finding an approach to select the relevant HPC represents a tedious task, regardless of the CPU architecture.

Power modeling often builds on these raw metrics to apply learning techniques—for example based on sampling [Ber+10]—to correlate the metrics with hardware power measurements using various regression models, which are so far mostly linear [McC+11].

A.Aroca *et al.* propose to model several hardware components (CPU, disk, network). It is the closest approach of our empirical learning method describes in Section 4.1.1. They use the `lookbusy` tool to generate a CPU load for each available frequency and a fixed number of active cores. They capture active cycles per second (ACPS) and raw power measurements while loading the CPU. A polynomial regression is used for proposing a power model per combination of frequency and number of active cores. They validate their power models on a single processor (Intel Xeon W3520) by using a map-reduce Hadoop application. During the validation, the authors have not been able to correctly capture the input parameter of their power model—*i.e.*, the overall CPU load—and they use an estimation instead. The resulting "tuned" power model with all components together exhibits an error rate of 7% compared to total amount of energy consumed.

Bertran *et al.* [Ber+10] model the power consumption of an Intel Core2 Duo by selecting 14 HPCs based on an *a priori* knowledge of the underlying architecture. To compute their model, the authors inject both selected HPCs and power measurements inside a multivariate linear regression. A modified version of `perfmon2`[1] is used to collect the raw HPC values. In particular, the authors developed 97 specific micro-benchmarks to stress each component identified in isolation. These benchmarks are written in C and assembly, and cannot be generalized to other architectures. They assess their solution with the SPEC CPU 2006 benchmark suite, reporting an error rate of 5% on a multi-core architecture.

Bircher *et al.* [Bir+05] propose a power model for an Intel Pentium 4 processor. They provide a first model that uses the number of fetched μ-operations per cycle, reporting on an average error rate of 2.6%. As this model was performing better for benchmarks inducing integer operations, the authors refine their model by using the definition of a floating point operation. As a consequence, their second power model builds on 2 HPC: the μ-operations delivered by the trace cache and the μ-operations delivered by the μ-code ROM. This model is assessed using the SPEC CPU 2000 benchmark suite, which is split in 10 groups. One benchmark is selected per group to train the model and the remaining ones are used to assess their estimation. Overall, the resulting CPU power model reports on an average error of 2.5%.

In [CM05], Contreras *et al.* propose a multivariate linear CPU power model for the Intel XScale PXA255 processor. They additionally consider different CPU frequencies on this processor to build a more accurate power model. They carefully selected the HPCs with the best correlation while avoiding redundancy, resulting in the selection of only 5 HPCs. In their paper, they also consider the power drawn by the main memory using 2 HPCs already used in the CPU power model. However, given that this processor can only monitor 2 events concurrently, they cannot implement an efficient and usable runtime power estimation. They test their solution on SPEC CPU 2000, Java CDC, and Java

[1] `http://perfmon2.sourceforge.net`

CLDC, and they report an average error rate of 4% compared to the measured average power consumption.

The authors in [Dol+15] propose an approach to build linear power models for hardware components (CPU, memory, network, disk) by applying a per component analysis. Their technique uses 4 benchmarks during the training phase and collect various metrics gathered from hardware performance counters, OS statistics, or sensors. They build a correlation matrix on all gathered metrics (including power measurements) and then apply a clustering algorithm on top of it. The power models presented in this article are manually extracted from these groups. Without considering the power models which include directly power measurements, the best one exhibits an absolute error of 3 W on average with a maximum absolute error of 70 W.

Economou *et al.* [ERK06] model the power consumption of 2 servers (Turion, Itanium) as a multiple linear regression that uses various utilization metrics as input parameters. The authors use the CPU utilization, the off-chip memory access count, the hard-disk I/O rate, and the network I/O rate. The input parameters are learned by using Gamut that emulates applications with varying levels of CPU, memory, hard disk, and network utilization. In order to retrieve raw power measurements, the authors uses board-level modifications and 4 "power planes" (extracted from the paper), which is heavy for end-users and represents a major hardware investment. On average, their power models exhibit an error rate less than 5% (varying between 0% and 15% in all cases) while using SPEC benchmarks, matrix and stream.

Isci and Martonosi [IM03] use an alternative approach to estimate the power consumption of an Intel Pentium 4 processor. They isolate 22 processor subunits with the help of designed micro-benchmarks and live power measurements. For each subunit, they use simple linear heuristics, which can include one or more HPC. For the others (trace cache, allocation, rename...), they use a specific piecewise linear approach. They selected 15 different HPC to model all subunits, some of them are reconfigured or rotated when needed. At the end, they express the CPU power consumption as the sum of all subunits. They train their model on designed micro-benchmarks, SPEC CPU 2000 and some desktop tools (AbiWord, Mozilla, Gnumeric) and they report on an average error of 3 W.

Li and John [LJ03] rely on per OS-routines power estimation to characterize at best the power drawn by a system. They simulate an 8-way issue, out-of-order superscalar processor with function unit latency. The authors use 21 applications, including SPEC JVM 98 and SPEC INT 95. During their experiments, they identify instruction per ccyle (IPC) to be very relevant to model the power drawn by the OS routines invoked by the benchmarks. The resulting CPU power model exhibits an average error of up to 6% in runtime testing conditions.

Rivoire *et al.* [RRK08] propose an approach to generate a family of high-level power models by using a common infrastructure. In order to choose the best input metrics for their power models, they compare 5 types of power models that vary on their input metrics and complexity. The first 4 power models are defined in the literature and use basic OS metrics (CPU utilization, disk utilization) [FWB07; Hea+05; RRK08]. They propose the fifth power model that uses HPC in addition to CPU and disk utilizations. The last proposed power model exhibits a mean absolute error less than 4% over 4 families of processors (Core2 Duo, Xeon, Itanium, Turion) by using SPECfp, SPECint, SPECjbb, stream, and Nsort. The authors do not detail the underlying architectures of the testbed CPU, making thus a fair comparison difficult.

iMeter [Yan+14] covers not only CPU, but also memory and I/O. To get a practical model, the authors need to select the proper number of counters. After benchmarking the

VMs under different loads, they empirically extract 91 out of 400 HPCs. In a second step, a principal components analysis is applied to identify a statistical correlation between the power consumption and the performance counters. With this method, highly correlated values are clustered into a smaller set of principal components that are not correlated anymore. The selection of the principal components depends on the cumulative contribution to the variance of the original counters, which should reach at least 85%. The final model is derived by the usage of support vector regression and 3 manually selected events per principal component [VGS97] and reports an average error of 4.7%.

In [ZA12], Zamani *et al.* study the relationship between HPC and power consumption. They use 4 applications from NAS parallel benchmarks (BT.C, CG.C, LU.C, SP.C) running on 8 threads in a 2 quad-core AMD Opteron processors. Given the limitation of events that they can open simultaneously, the authors first show that the measurement variability other different executions is not significantly large, enabling different runs for sampling all events. This article proposes a deep analysis for HPC selection (single or multiple). The authors demonstrate that a collinearity can exist between events and then propose a novel method to find the best combination of HPC with good performance. They use the ARMAX technique to build their power models. They evaluate their solution by producing a model per application and exhibit a mean absolute error in signal between 0.1%-0.5% for offline analysis.

HaPPy [Zha+14] introduces a hyperthread-aware power model that uses only the *non-halted cycles* event. The authors distinguish different cases where either single or both hardware threads of a core are in use. This power model is linear and contains a ratio computed according to their observations. They demonstrate that when both threads of a core are activated, they share a small part of non-halted cycles. They extend the `perf`[2] tool to access to RAPL. Their model is tested on a Intel “Sandy Bridge” server with private benchmarks provided by Google, that cannot be reused, and 10 benchmarks taken from SPEC CPU 2006. To assess their power estimation, they used the RAPL interface reachable on this server. Compared to RAPL, they manage to have an average error rate of 7.5%, and a worst case error rate of 9.4%. Nevertheless, as demonstrated in [Col+15b], these error rates can be exceeded in scenarios where only single cores of a CPU are monitored.

Summary

As a summary of the current state of practice, the existing CPU power models found in the literature cannot be reproduced because *i).* the details of the selected HPC events are not provided [WCS11] or sufficiently documented [Zha+14], *ii).* they are tailored to a specific processor architecture (including a limited set of power-aware features) [LPF10], or *iii).* they are built on private workloads that cannot be reused to assess alternative power models [Zha+14]. The state-of-the-art is resumed in Table 2.1.

The main contributions of this thesis for overcoming these limitations are: *i).* proposing an in-depth description of our contributions, *ii).* defining architecture-agnostic and automatic learning approaches, *iii).* assessing our approaches on a large set of CPUs that exhibit different architectures and features and, *iiii).* to use real and public workloads. All our contributions are described in Chapter 4.

[2] `https://perf.wiki.kernel.org`

2.2 VM Power Models

In data centers, the efficiency of VM consolidation, power dependent cost modeling, and power provisioning are highly dependent on accurate power models [VAN08]. Such models are particularly needed because it is not possible to attach a power meter to a virtual machine [Kri+11]. In general, VMs can be monitored as black-box systems for coarse-grained scheduling decisions. If we want to be able to do fine-grained scheduling decisions—*i.e.*, with heterogeneous hardware—we need to be able to consider finer-grained estimation at sub-system level and might even need to step inside the VM.

So far, fine-grained power estimation of VMs require profiling each application separately. One example is WattApp [KVN10], which relies on application throughput instead of performance counters as a basis for the power model. The developers of pMapper [VAN08] argue that application power estimation is not feasible and instead perform resource mapping using a centralized step-wise decision algorithm.

To generalize power estimation, some systems like Joulemeter [Kan+10] assume that each VM only hosts a single application and thus treat VMs as black boxes. In a multi-VM system, they try to compute the resource usage of each VM in isolation and feed the resulting values in a power model.

Bertran *et al.* [Ber+12] use a sampling phase to gather data related to HPCs and compute energy models from these samples. With the gathered energy models, it is possible to predict the power consumption of a process, and therefore apply it to estimate the power consumption of the entire VM. Their work does neither consider modern CPU features.

Another example is given by Bohra *et al.* in [BC10], where the authors propose a tool named VMeter that estimates the consumption of all active VMs on a system. A linear model is used to compute the VMs power consumption with the help of available statistics (processor utilization and I/O accesses) from each physical node. The total power consumption is subsequently computed by summing the VMs consumption with the power consumed by the infrastructure.

Janacek *et al.* [Jan+12] use a linear power model to compute the server consumption with *postmortem* analysis. The computed power consumption is then mapped to VMs depending on their load. This technique is not effective when runtime information is required.

In Stoess *et al.* [SLB07] the authors argue that, in virtualized environments, energy monitoring has to be integrated within the VM as well as the hypervisor. To that end, they use the L4 micro-kernel as hypervisor and adapt a guest OS to run on L4. They assume that each device driver is able to expose the power consumption of the corresponding device as well as an energy-aware guest operating system and is limited to integer applications. For application level power monitoring, the VM connects to the hypervisor and maps virtualized performance counters to the hardware counters.

Summary

State-of-the-art solutions provide no or limited support for fine-grained monitoring of applications running within a VM. The few existing approaches either consider the VM as a black-box running a single application, or require extensions to the hypervisor or

[3] VM as a Black-Box

[4] VM as a White-Box

Table 2.2: Summary of existing VM power models.

Author(s)		Processor(s)	VMaaBB[3]	VMaaWB[4]	Error(s)
Bertran *et al.*	[Ber+12]	Core2 Duo T9300	✓✓✓	✗	< 5%
Bohra *et al.*	[BC10]	Opteron	✓✓	✗	6% – 7%
Janacek *et al.*	[Jan+12]	2 × Intel Xeon X5560 2 × Intel Xeon X5550	✓✓	✗	✗
Kansal *et al.*	[Kan+10]	Nehalem L5520	✓✓	✗	0.4 W – 2.4 W
Koller *et al.*	[KVN10]	2 × Xeon, Core2 Duo	✓✓	✗	5%
Stoess *et al.*	[SLB07]	Pentium D830	✓✓	✓	✗
Verma *et al.*	[VAN08]	Simulator	✓	✗	✗

to the host and guest operating systems for being operational. All these approaches are summarized in Table 2.2.

For answering these challenges, we propose in Chapter 7, a middleware solution, BitWatts, that considers the VM as a white-box and thus allowing to estimate the power consumption of software processes in virtualized environments. Moreover, our middleware framework can work in a distributed setup (multi-tenant environment) and can go through all the levels of virtualization—*i.e.*, can estimate the power consumption of a process that runs inside a VM of another VM, etc.).

2.3 Disk Power Models

In [Nou14], the authors propose an *Hard Disk Drive* (HDD) power model that uses I/O statistics as input parameters. They particularly use the number of bytes read/written on the disk, and multiply these raw informations by the power consumed per operation. The end-users have thus to retrieve the power consumption for the read and write operations in the documentation, sometimes hard to find. The authors do not propose a proper learning approach that can be easily reused for learning new disk power model.

In [ERK06], Economou *et al.* propose to model different hardware components for representing the overall power consumption of a server. For the hard drive component, they propose to model its power consumption by using the I/O rate as input metric that can be easily retrieved from the OS. Unfortunately, they do not evaluate the efficiency of each power model separately. As in [Nou14], there is a not a proper evaluation of the hard drive power estimation.

The authors in [Kan+10] also use the number of bytes read and written and they inject these parameters inside a multivariate linear regression. Similar approaches are described in [BC10; SLB07].

All the above power models are therefore limited to HDDs while SSDs are widely used nowadays.

Summary

As described above, all the power models proposed by the state-of-the-art mainly uses basic metrics that can be retrieved from the OS. The bandwidth, the number of bytes read and written can be easily extracted from the state-of-the-art as the most used metrics while modeling the HDD power consumption.

However, the few existing approaches extracted from the state-of-the-art are limited because *i)*. they only consider HDDs whereas SSDs are now widely spread among servers [ERK06; Nou14] and, *ii)*. they do not allow a fine-grained level of power estimation [BC10; Kan+10; SLB07]. To stretch these limitations, we therefore propose to extend our automatic learning approaches to SSD components and thus confirming the flexibility of our approaches for new hardware components. We detailed this extension for learning SSD power models in Chapter 4.

Chapter 3

Power Measurement Granularities

Table of Contents

For decades, a lot of efforts have been spent by the research community to propose power measurement solutions. This chapter reviews the state-of-the-art solutions that provide such capabilities. Different levels of power estimation are proposed by these solutions. The hardware-level granularity is the most commonly proposed while browsing through the existing solutions. However, such a coarse-grained granularity prevents from achieving a detailed energy-driven analysis of software systems. To overcome these limitations, the community brought a finer scale of power estimation, at software-level. Such a deeper level of power estimation allows stakeholders to get better insights about how the power is consumed by software components and can therefore be used for scheduling or capping heuristics. An even thinner level of power estimation is proposed by tools that support to monitor the energy distribution at source-code level. This level introduces another level of energy analysis and proposes to easily compare the energy impact of design choices or optimizations.

The rest of this chapter study the state-of-the-art in this area and is organized as follows. Section 3.1 presents the solutions that can achieve runtime power estimation at level of hardware components, while Section 3.2 describes the software-based approaches and Section 3.3 show solutions that can go further and can provide deep analysis of the energy consumed by source-code.

3.1 Hardware-Level Granularity

PowerMon2 [Bed+10] uses an external power monitoring board that needs to be inserted between a system's power supply and a motherboard. This board therefore allows to retrieve the power consumption per connected hardware component and can physically be integrated into a target system by fitting into a 3.5" drive bay. PowerMon2 allows to measure up to 8 individual DC rails, thus allowing to attach several rails to the motherboard in addition to hardware components (GPUs, disks, etc.). It can read and report power measurements of hardware components at a rate up to 3 KHz to the user through an USB interface. All schematics and source-code are freely available online, but this solution requires some rather expensive investments (up to $150).

PowerInsight [LPD13] follows the same principle as PowerMon2 and is built on top of another external board (BeagleBone[1]) that uses an ARM Cortex processor. This external board can be connected up to 15 components and is used to acquire power measurements from custom power sensing boards connected to it. It was first designed to work within a cluster and it is therefore required to install and configure one board per available node. Each board is then connected through Ethernet and can send the acquired data to a master node. The master node is finally responsible for aggregating data for *postmortem* analysis. PowerInsight can provide user-space samples at a rate up to 1 KHz but they only validate their approach while using 1 HZ.

RAPL [Rot+12] offers specific HPCs to retrieve the power consumption of CPU power packages since the “Sandy Bridge” architecture. Intel divides the system into domains (PP0, PP1, PKG, DRAM) to retrieve various power informations according to the requested context. The first domain PP0 represents the core activity of the processor (cores + L1 + L2 + L3), the PP1 domain the uncore activities (LLC, integrated graphic cards, etc.), and PKG represents the sum of PP0 and PP1. The last domain DRAM only exhibits the RAM power consumption. The RAPL tool can be thus easily used in recent Intel architectures as it does not require any hardware modification. Moreover, this tool can also be used as a power capping solution for limiting the CPU power consumption. However, it is limited to specific processor generations and further limited to Intel processors.

Icsi *et al.* [IM03] describe an approach for learning CPU power models based on predefined 15 HPC for 22 selected processor subunits. In addition, they propose a live CPU power monitoring solution that implies different modules. First, a reader runs inside the system under test for collecting values for the selected HPC. Once collected, the values are sent via the network to a logger machine. This logger uses together the power model and the extracted values for producing live power estimation of the 22 processor subunits. With this approach, the authors show runtime power estimations for one concurrent running application that can be divided per involved subunit.

Built on top of their CPU usage based power model, Lien *et al.* [LBL07] propose a window-based GUI to monitor in real-time the overall power consumption of Windows streaming-media servers through a time period.

3.2 Process-Level Granularity

The PowerPack [Ge+10] framework can monitor all hardware components separately. To retrieve power measurements, a precision sensing resistor is attached to each DC power line, thus allowing to measure the voltage differences with a power meter. This approach is not limited to a single physical power meter but can use several, such as NI data acquisition board, Watt’s Up Pro, or ACPI interface. Power measurements are simultaneously collected on all power lines to be representative of all hardware components. The data retrieved are then recorded and used in *postmortem* analysis. The authors consider their approach as being able to provide per-process power estimation but they only consider one concurrent application during their validation. They also mention that it can target a cluster but only one node can be monitored at a time. The authors propose then a solution by replaying the same workload m times, where m represents the number of nodes. It is therefore not

[1] `http://beagleboard.org`

a suitable solution in practice. Furthermore, the different components used for acquiring or analyzing data are expensive.

WattProf [Ras+15] supports the profiling of *High-Performance Computing* applications. As PowerMon2, PowerInsight and PowerPack, this solution uses an author-defined external board as a cornerstone of their solution. This board is fully configurable and can collect raw power measurements that come from various connected hardware components (CPU, disk, memory, etc.) through external sensors attached to power lines. The board is able to be connected up to 128 sensors that can be sampled at up to 12 KHz. The data can be thus be retrieved via Ethernet interface, or can be buffered inside the board for later analysis. As in [Ge+10], the authors argue that this solution is able to perform per-process power estimation, but they only validates their approach while running a single application.

WattWatcher [LeB+15] is the tool that can characterize workload energy power consumption. The authors use several calibration phases for computing a power model that fits a modern CPU architecture. This power model uses a lot of predefined set of HPCs as input parameters. As the authors use a special power model generator that can target any CPU architecture, they have to describe entirely the underlying CPU architecture via a configuration file. This file contains all mappings required to match specific HPCs from the underlying CPU with the ones used by the generator and thus requiring a deep knowledge of the underlying architecture. To limit the overhead, the generator is located on another machine and thus requiring at least 2 machines. An efficient network connexion is required to send data to the generator and to monitor the power estimation in real-time.

Jolinar [NIB16] use predefined power models built on top of hardware characteristics, such as the CPU's thermal design power (TDP), the maximum rate of read/write disk operations, and the power consumed by each operation, etc. These values have to be found inside OEM documentations. This process can therefore be tedious for end-users. The authors explain that the CPU power model may has a lower accuracy on modern architectures because of the simplicity of the input parameters (already demonstrated by the state-of-the-art). Jolinar launches the monitored application in background and the users can use the application while the monitoring solution keeps collecting data. However, only one application can be monitored by Jolinar and only the applications which are composed of a main process can be considered. Once the execution of the application completes, the tool reports on the overall energy estimation and the energy distribution between CPU, disk, and memory components.

3.3 Code-Level Granularity

The PowerPack solution [Ge+10] previously presented is able to propose per-process power estimation for various hardware components. In addition, the authors extend their approach to be able to correlate power consumption to source-code. The authors then propose a *power meter profile* API that contains several subroutines. These routines will be next used in the target application to link a specific region of code with the power consumption. This solution is therefore very intrusive and force to manually edit the target application. Moreover, this paper lacks of an impact analysis of such an instrumentation.

In addition to the per-process power monitoring proposed by WattProf [Ras+15], the authors also define an API to instrument the source-code. Several *power monitoring tasks* can be used to perform independent energy analysis. As the main goal of WattProf is to monitor HPC applications, the authors propose first an extension of a classical profiling tool used in this domain for adding energy informations inside the generated reports. This solution also remains very intrusive and the cost of such instrumentation is neither discussed.

Islam *et al.* [INB16] describe an approach that uses an Observation-based Slicing [Bin+14] technique, for detecting all the features—*i.e.*, parts of the code really used at runtime—needed by the program to behave normally and to produce the correct output. Once a feature detected, the tool is able to isolate it and to produce an executable version that contains only this feature. Each new created program is then injected inside Jolinar for producing its energy report. If one wants to analyze n features of a software, it requires to launch Jolinar with the featured program n times, thus requiring a lot of time to perform a comprehensive analysis.

Jalen [NRS15] is able to monitor at fined-grained level—*i.e.*, at the level of methods—the software power consumption. The authors made the design choice to only target Java programs because of the JVM facilities to instrument code and to get important runtime informations. 2 versions of Jalen have been implemented, one that instruments methods, and another one that uses statistical sampling to collect informations at a defined rate. The authors demonstrate that the instrumented version has a non-negligible intrinsic overhead and therefore push forward the statistical method as the lightweight solution. Their statistical sampling approach consists of 2 phases. The first one runs the application and collects the per component power consumption over its execution. Once the profiling is completed, the second phase captures the JVM's current stack trace with the CPU time of threads each 10 milliseconds and next computes statistics about the number of times each method was captured as the top of the stack. These statistics are then used to proportionally distribute the energy consumption across methods.

enDebug [CV16] is a tool that uses predefined power models to compute the energy distribution of software functions. To do so, the implemented approach instruments each of the function calls and records the selected HPC values at the caller and callee sites. The same technique is applied on function returns. Once the execution of the program completes, the tool accumulates all function fragments (due to multiple calls or returns) for producing energy report per individual function. The authors do not propose a time-series view of the software energy distribution but nonetheless it can be useful for analyzing energy phases. The impact of their solution on the target application is not evaluated in this paper. However, Noureddine *et al.* [NRS15] demonstrate the non-negligible cost of such instrumentation. In addition to this tool, the authors propose an automated recommendation system for energy optimization based on genetic programming and show energy improvements on a large set of programs.

Summary

As a summary of the state-of-the-art, mostly of the proposed solutions are not easily customizable to meet the user requirements. Most of the existing solutions are limited to hardware components and require hardware investments to be fully operational [Bed+10; SLB07]. This kind of tools cannot be then used while targeting fine-grained power estimations. To overcome these limitations, several tools propose to target software power consumption, but a few of them implies costly investments and/or are not usable in practice [Ge+10; LeB+15; Ras+15]. Few existing approaches go further and propose to analyze the energy distribution of source-code. However, only a few approaches discuss the overhead of their approach [NRS15] and a lot of research efforts remain to do. Finally,

Table 3.1: Summary of power monitoring solutions.

Author(s)		**Hadware-level**	**Process-level**	**Code-level**	**Pricey**	**Modular**
Doe[2]	[IM03]	✓	✓	✗	✓	✗
Doe[2]	[INB16]	✓	✗	✓	✗	✗
Doe[2]	[LBL07]	✓	✗	✗	✓	✗
enDebug	[CV16]	✗	✗	✓✓	✗	✗
Jalen	[NRS15]	✓	✗	✓✓	✗	✗
Jolinar	[NIB16]	✓	✓	✗	✗	✗
PowerInsight	[LPD13]	✓✓	✗	✗	✓✓	✓
PowerMon2	[Bed+10]	✓✓	✗	✗	✓✓	✓
PowerPack	[Ge+10]	✓✓	✓	✓	✓✓	✓
PowerScope	[FS99]	✗	✓	✓	✓	✗
PowerTOP	[Int15b]	✓	✓	✗	✗	✗
pTop	[DRS09]	✓	✓✓	✗	✗	✗
RAPL	[Rot+12]	✓✓	✗	✗	✗	✗
SPAN	[WCS11]	✓	✓	✓	✗	✗
WattProf	[Ras+15]	✓✓	✓	✗	✓✓	✓
WattWatcher	[LeB+15]	✓	✓	✗	✓✓	✗

based on this fine-grained analysis, some papers already propose techniques to improve the code energy footprint [CV16; NR15]. All the approaches and solutions described in this manuscript are resumed in Table 3.1.

To address these limitations, we rather propose a middleware toolkit, PowerAPI, for building specific software-defined power meters "à la carte". PowerAPI can work in all aforementioned granularities, therefore promoting its use for energy monitoring in large scale. We detailed PowerAPI in Chapter 5, and we build different power meters for all above granularities (cf. Chapter 6, Chapter 8), plus a new granularity for distributed services (cf. Chapter 7).

[2] Doe: name used when the solution does not have a name.

Part II

Contributions

Chapter 4

Learning Power Models Automatically

Table of Contents

Energy represents one of the largest cost factors when operating data centers, largely due to the consumption of air conditioning, network infrastructure, and host machines [ODL14]. Currently, widely used PDUs are often shared amongst nodes to deliver aggregated power consumption reports, in the range of hours or minutes. However, in order to improve the energy efficiency of software systems, we need to support process-level power estimation in real-time which goes beyond the capacity of PDUs [Tan+15]. In particular, the CPU is considered as the major power consumer [ERK06] within a node and requires to be closely and accurately monitored. Software power estimation of hardware components are a central concern for energy efficiency and resource management in data centers. Over the last decade, a dozen of ad-hoc power models have been proposed to cope with the wide diversity and the growing complexity of modern hardware architectures [Bel00; Ber+10; Bir+05; Col+15b; Dol+15; IM03; Kan+10; LJ03; LPF10; McC+11; VWT13; Yan+14; ZA12; Zha+14]. However, most of the proposed power models rely on a deep expertise of targeted architectures, thus leading to the design of hardware-specific solutions that hardly can be ported to different settings [Ber+10; IM03; Zha+14]. Developing power models that can accurately cover a large set of power consumer components is a complex task and is challenging. Rather than proposing yet other power models, we therefore introduce different approaches to automatically learn power models of several power consuming components by exploring theirs power consumption spaces.

The remainder of this chapter is organized as follows. Section 4.1 proposes 2 approaches for automatically learning CPU power models, the empirical approach is detailed in Section 4.1.1, while the architecture-agnostic approach in Section 4.1.2. We finally explain the need to model the SSD power consumption in Section 4.2 and describe our empirical learning approach for SSD power models in Section 4.2.1.

4.1 Learning CPU Power Models

In a typical server, the major power consumer is the CPU, covering at least one third of the overall power consumption [ODL14]. Hence, like other studies [Ber+12; Jan+12; LPF10; Zha+14], our power model focuses on processor consumption and accurately monitors applications that are CPU—and memory—intensive. To limit the disk impacts, we therefore select the workloads used for learning the power models in such a way that the additional power possibly consumed by the disk is negligible. Studies in data centers [NRS15; ODL14] show that the network I/O (in the case of Ethernet) is not impacting the power consumption as the difference between idle and fully utilized links is very low.

As the density of transistors has grown steadily following Moore's law, modern process have become extremely powerful yet complex computational units (see Figure 4.1). To control energy consumption, CPUs rely on frequency scaling and power saving modes to adjust their performance upon to the computation requirements. In particular, the multi-core processors designed by Intel integrate the following features:

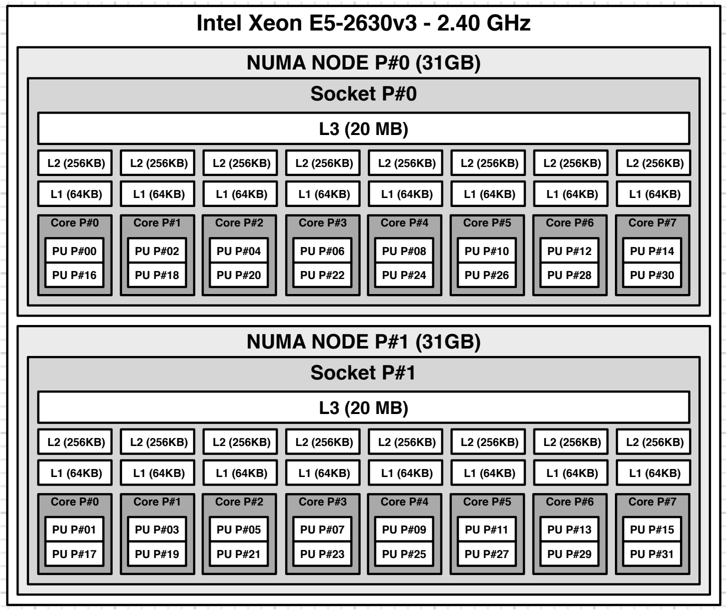
Hyper-Threading (HT) is used on some processor generations (*e.g.*, Pentium IV, Xeon) to separate each core in 2 threads. The technology is based on the simultaneous multi-threading (SMT) principle, that allows the processor to seamlessly support thread-level parallelism (TLP) in hardware and share more effectively the available resources. Performance gains strongly depend on software parallelism, and for a single-threaded application it may be more effective to actually disable this technology;

SpeedStep (SS) is Intel's implementation of dynamic voltage/frequency scaling (DVFS), allowing a processor to adjust its clock speed and run at different frequencies or voltages upon need. The OS can increase the frequency to quickly execute operations or reduce it to minimize dissipated power when the processor is under-utilized. The characteristic switching power dissipated by a processor is given by the well-known complementary metal-oxide semiconductor (CMOS) formula $P = C \cdot f \cdot V^2$, where C is the capacitance, f the frequency, and V the voltage. Thus, while decreasing the frequency or voltage, less power is required and vice-versa;

TurboBoost (TB) can dynamically increase the processor frequency beyond the maximum bound, which can be greater than the TDP, for a limited period of time. It therefore allows the processor cores to execute more instructions by running faster. TB is however only activated when some specific conditions are met, notably related to the number of active cores and the current CPU temperature. It also depends on the OS, that may request to trigger it when some applications require additional performance;

C-States (CS) were introduced to save energy and allow the CPU to use low-power modes. The idea is to lower the clock speed, turn off some of the processor units, and thus reduce the overall power consumed. More units are shut down, more the power savings are high. Different types of CS are available: core (individual hardware core's view, C-state), processor (global hardware core's view, PC-state), or logical (logical core's view, CC-state). They are numbered starting at 0, which corresponds to the most operational mode (100%). Higher the index, deeper the sleep state.

Figure 4.1: Example of Modern CPU architecture.



While the state-of-the-art CPU power models demonstrate that achieving accurate power estimation is possible, they are barely generalizable to processors or applications which were not part of the original study. Therefore, we rather propose a tooled approach capable of learning the specifics of a processor and building the fitting CPU power model. Hence, rather than proposing yet other hand-crafted power models, our approach intends to cover evolving architectures and aims at delivering a solution that will be able to deal with current and future generation of CPU architectures.

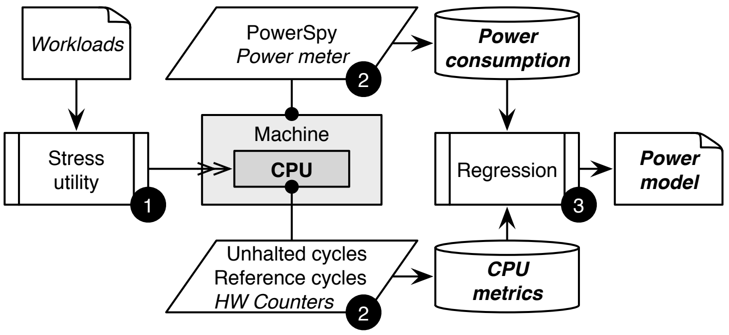
As reported by [Kan+10], the CPU load does not accurately reflect the diversity of CPU activities. In particular, to faithfully capture the power model of a CPU, the types of task executed by the CPU have to be clearly identified. We therefore decided to base our power models on HPC to collect raw, yet accurate, metrics reflecting the types of operation that are truly executed by the CPU. Nevertheless, the number and the nature of HPC events provided by the CPU strongly vary according to the processor type.

More specially, a CPU can expose several performance monitoring units (PMUs) depending on its architecture and model. For example, 2 PMUs are detected on an Intel Xeon W3520: nehalem and nehalem uncore, each providing two types of HPCs that cover either *fixed* or *generic* HPC events. A fixed HPC event can only be used for one predefined event, usually cycles, bus cycles, or instructions retired, while a generic one can monitor any event. If there are more events monitored than available counters for a PMU, the kernel applies multiplexing to alter the frequency and to provide a fair access to each HPC event. When multiplexing is triggered, the events cannot be accurately monitored anymore and an approximation is returned instead. The PMUs of the testbed CPU are described in Appendix A

4.1.1 Empirical Approach

Learning the CPU power model of a multi-core processor requires the definition of a workload that carefully stresses the various features it supports. Thereby, it is important to isolate the noise induced by other hardware components to properly capture the power consumption of the CPU under study.

Figure 4.2: Empirical CPU power modeling approach.



In the following paragraphs, each step of our empirical approach for learning CPU power models (cf. Figure 4.2) is detailed below.

Input workload. We choose the `stress`[1] utility, available on most UNIX systems, to perform specific workload scenarios. This tool allows us to incrementally stress different hardware components, such as the CPU or the memory.

Using the options provided by the `stress` utility, we generate different workloads. First, we stress the processor core by core under full load in order to capture its maximum frequency and to observe the effect of the SMT feature on the power consumption. Then, we dynamically change the CPU load to characterize the effects of the DVFS feature. This designed workload is applied for each frequency available on the CPU using `cpufreq-utils`. Finally, by stressing an increasing number of cores, we are able to identify the dynamic frequencies involved when the CPU is boosted.

Acquisition of raw metrics. To learn the power model, we need to collect run-time metrics that faithfully capture the specificity of a large set of CPU workloads. As previously mentioned, the CPU load does not correctly reflect the variety of tasks executed and we therefore decide to base the approach on HPC to collect low-level and accurate metrics reflecting trustfully the operation types ran by the CPU. Especially, we use the `libpfm4`[2] library for accessing counters available on modern CPU architectures, regardless of the OS. The counters used to estimate the power consumption of processors have to be carefully selected according to two criteria: *i).* their availability on a large family of architectures and, *ii).* the overhead induced by their exploitation.

The purpose of this approach is to build the most efficient model in order to keep the lowest overhead during run-time. We consequently choose as in [LPF10; WCS11; Zha+14] the unhalted-cycles (uc)[3] and reference-cycles (rc)[4] counters to accurately characterize the power model of multi-core architectures. While the first counter represents the number of cycles truly executed and thus the activity of cores, the second one represents the number of clock ticks at the frequency of the *time-stamp counter* (TSC); it is, therefore, very useful to approximate the core frequency, even when the turbo mode is triggered.

[1] `http://linux.die.net/man/1/stress`

[2] `http://perfmon2.sourceforge.net`

[3] CPU_CLK_UNHALTED:THREAD_P, event=0x003c

[4] CPU_CLK_UNHALTED:REF_P, event=0x013c

The average frequency (f) is computed by dividing the number of unhalted-cycles by the number of reference-cycles and then multiplying it by the timestamp counter (TSC) frequency (f_{TSC}) ($f = (uc/rc) * f_{TSC}$). f is then used to build one power model per detected frequency and to choose at runtime which one to use.

To monitor the power consumption during the learning phase, we consider a power meter that reports on the consumption of the whole machine as "ground truth". More precisely, we use the PowerSpy[5] Bluetooth power meter. Depending on the country, the PowerSpy power meter samples the raw power consumption between 45 Hz and 65 Hz. To improve the accuracy of the power model, we run the identified workload several times for reducing the variance introduced by the physical measures.

Power model inference by regression. The hardware performance counters and power information collected during the execution of the workloads are then correlated using a polynomial regression for connecting the evolution of the power consumption with the number of unhalted-cycles. A power model is built for each processor frequency and represents the power consumption of a single core, SMT feature included [Zha+14], and we assume that the power consumption grows linearly with the number of active cores on homogeneous architectures.

In practice, the power model we obtained for a machine (*host*) running at a given frequency (f) for a short period of time can be represented by the equation:

$$P_{host}(f) = P_{idle}(f) + \sum_{pid \in PIDs} P_{cpu}(f, uc^1_{pid}...uc^N_{pid}). \tag{4.1}$$

where $P_{idle}(f)$ corresponds to the static power consumption—*i.e.*, the idle power—of the *host* for the frequency f that we inferred from the regression step, and $uc^1_{pid}...uc^N_{pid}$ is a vector of unhalted-cycles collected at runtime per active process identifier *pid* and per core $1..N$. The power consumption of the CPU, P_{cpu}, is defined as the sum of the power consumption per frequency, P_f, for each core n:

$$P_{cpu}(f, uc^1_{pid}...uc^N_{pid}) = \sum_{n=1}^{N} P_f(uc^n_{pid}). \tag{4.2}$$

We finally obtain a power model per frequency, including boost specific frequencies. One of the resulting formula is described below for a boost frequency on a Xeon W3520 processor (cf. Appendix A) for the 2.90 GHz frequency:

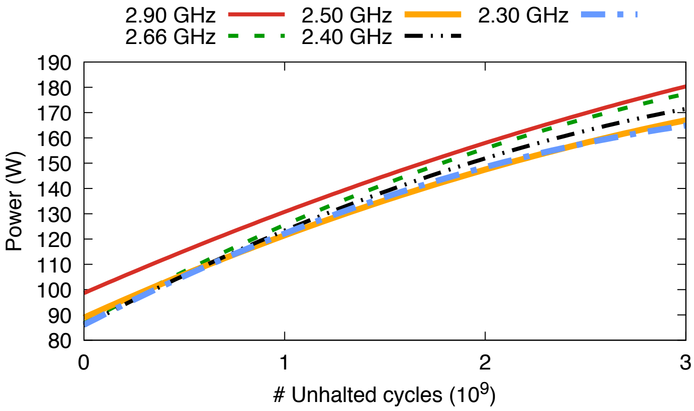
$$P_{2.90}(uc_{pid}) = \frac{8.64 \cdot uc_{pid}}{10^9} - \frac{6.10 \cdot uc^2_{pid}}{10^{18}}. \tag{4.3}$$

The resulting formula is a polynomial of degree 2 (depicted in Figure 4.3), which conforms to the results published in the literature and the impacts of the SMT feature on the power consumption [Zha+14].

Figure 4.3 plots the power estimation according to the number of unhalted-cycles for each power model inferred per frequency on a Xeon processor (cf. Appendix A). For the sake of clarity, only the frequencies above 2.30 GHz are reported. The idle power consumption ($P_{idle}(f)$ when $x = 0$) is computed during the polynomial regression and is clearly impacted by the current processor frequency. One can observe that the 2.50 GHz line is above the 2.40 GHz one, which is mainly due to the inaccuracy of `cpufreq-utils`: it keep tracks of the average frequency and might not report exact values at any given time, notably ignoring the boost frequencies.

[5] http://www.alciom.com/en/products/powerspy2.html

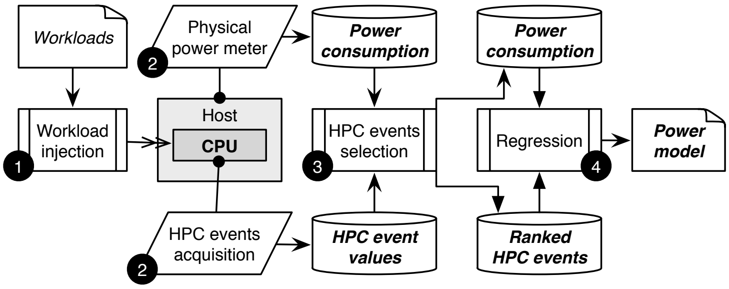
Figure 4.3: Power models for the highest frequencies on an Intel Xeon W3520 processor.



4.1.2 Architecture-Agnostic Approach

The learning phase we propose in Figure 4.4 analyses the power consumption and triggered HPC events of the target CPU, in order to identify the key events that impact the power consumption. The combination of these events is then used to learn automatically the CPU power model.

Figure 4.4: Architecture-agnostic CPU power modeling approach.



Our goal is to automatically classify the HPC events in order to identify those which are best characterizing the CPU activity and are correlated with its power consumption. Each step of our architecture-agnostic approach for learning CPU power models is depicted in Figure 4.4 and described below.

Input workload injection. For exploring the activity of a CPU, we consider a set of representative applications covering the features provided by a CPU. In particular, to promote the reproducibility of our results, we favor freely available and widely used benchmark suites, such as PARSEC [BL09]. However, this choice does not prevent us from including additional benchmark suites or any sample workloads. All workloads are then launched several times in isolation for reducing the noise that can be experienced during the learning phase.

Acquisition of raw HPC counters. Unfortunately, the CPU cannot monitor hundreds of HPC events simultaneously [Int15a]. Thus, we have to split the list of available events into subsets of events to avoid multiplexing that might cause inaccuracies. Resulting of our approach, we dynamically learn the number of events that we can monitor together for getting accurate raw measurements. Table A.1 shows the number of events that we can read in parallel in several CPU architectures. For a given CPU, the number of workload executions w to be considered is therefore defined as:

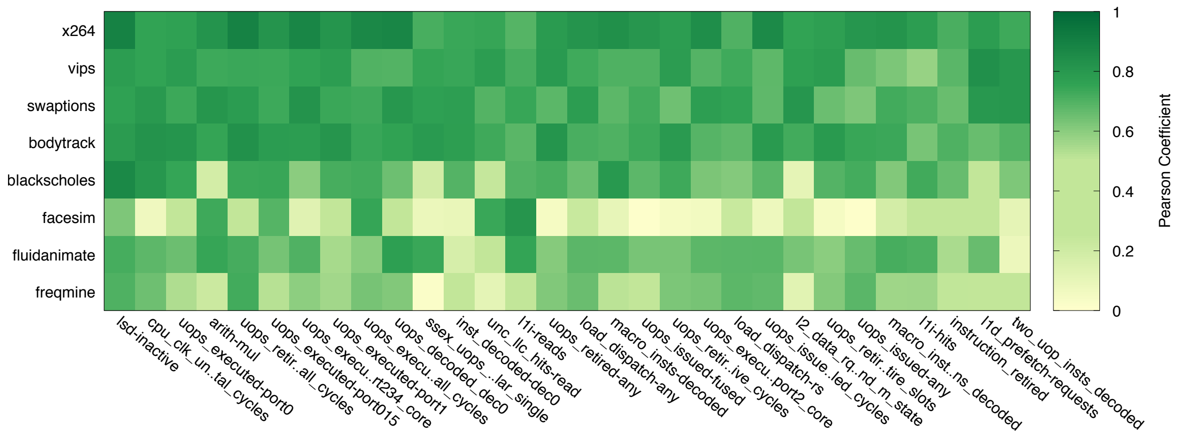
$$w = \left\lceil \sum_{p \in \mathrm{PMU}_s} \frac{|E_p|}{|C_p|} \right\rceil \times |W| \times i \tag{4.4}$$

where E is the set of events made available by the processor for a given PMU, C is the set of generic counters available for a PMU, W is the set of input workloads, and i is the number of sampling iterations to execute.

Combining HPC events and sample applications may quickly lead to the comparison of thousands of candidate metrics. Hence, a filtering step is required to guarantee an acceptable duration for the learning phase. Our approach proposes an automated way to focus on the most relevant events. In the first step, each workload is only executed for a few seconds while collecting values from HPC events and from a power meter. We then select relevant HPC events by applying the Pearson correlation coefficient [CM05; ZA12]. We compute the Pearson correlation coefficient $r_{e,p}$ for each workload between the n values reported by each monitored HPC event e and the collected power consumption p:

$$r_{e,p} = \frac{\sum_{i=1}^{n} (e_i - \overline{e})\,(p_i - \overline{p})}{\sqrt{\sum_{i=1}^{n} (e_i - \overline{e})^2}\,\sqrt{\sum_{i=1}^{n} (p_i - \overline{p})^2}} \tag{4.5}$$

Figure 4.5: Pearson coefficients of the Top-30 correlated events for the PARSEC benchmarks on an Intel Xeon W3520.



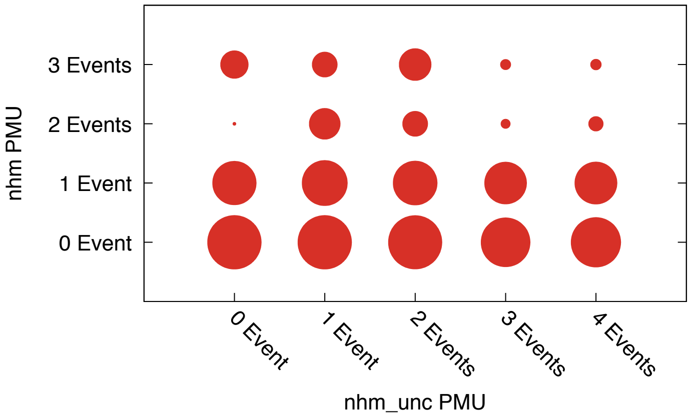
Selection of relevant HPC events. As next step, we eliminate the HPC events that have a median correlation coefficient ($\tilde{r}$) below a given threshold. In particular, we consider that any coefficient below 0.5 clearly indicates a lack of correlation between the considered event (e) and the associated power consumption (p). With this step, we quickly filter out hundreds of uncorrelated—and therefore irrelevant—events, resulting for instance in 253 left out of 514 events on an Intel Xeon W3520 (cf. Appendix A). The reduced set of HPC events is then used to relaunch all the workloads, but this time with default runtime. At the end of the full execution, we rank the remaining HPC events for all the workloads based on their newly calculated median correlation with the power consumption, as depicted in Figure 4.5.

The distribution of Pearson coefficients for the 30 best events varies for each of the workloads W taken from the PARSEC benchmark suite on the Intel Xeon W3520 processor. One can clearly distinguish the benchmarks that simulate all selected HPC events (*e.g.*, `x264`, `vips`) from the ones whose power consumptions match only specific events (*e.g.*, `freqmine`, `fluidanimate`). Deriving a CPU power model that is capable of covering all kinds of workloads accurately is consequently a challenging task.

Power model inference We finally apply a regression analysis to derive the CPU power model from the previously selected HPC events. In particular, we use the robust ridge regression [LPF10; RL87], which belongs to the family of multivariate linear regression. Our approach being fully configurable, the aforementioned linear regression can be thus chosen upon needs. Our choice was guided by the need to easily eliminate outliers. It has been furthermore validated by the experiments and results presented around power models in this manuscript.

The computation of the multiple linear regression should balance the gain in terms of estimation error with the cost of including an additional event into the CPU power model. To design the CPU power model as accurately as possible, we consider a subset R_n of n benchmarks ($\forall n < |W|, R_n \subseteq W$), composed from those exhibiting the lowest median Pearson coefficients, as input for our regression process. From R_n, we compute a CPU power model for each combination of HPC events, by taking into account the limited number of event that can be monitored in parallel. For each training set R_n, from all the computed power models, we only keep the one with the smallest regression error. Finally, we compare the CPU power model obtained for each R_n and we pick the one that minimizes the absolute error between the regression and the remaining benchmarks, not included in the training set ($E_n = W \setminus R_n$).

Figure 4.6: Average error per combination of events for R_3 on an Intel Xeon W3520 processor.



As an illustration, Figure 4.6 shows the distribution of the average error per CPU power model built for R_3 (`freqmine`, `fluidanimate`, and `facesim`) depending on the number of HPC events included in resulting power model. A larger circle means a larger error. One can clearly see that a CPU power model that combines a high number HPC events may exhibit a larger than one that uses a lower number of events.

An an example, in the Intel Xeon W3520 processor, the CPU power model composed of 2 events taken from the PMU nhm (see Appendix A) emerges from this analysis and reports an average error of 1.35%, 1.60 W respectively.

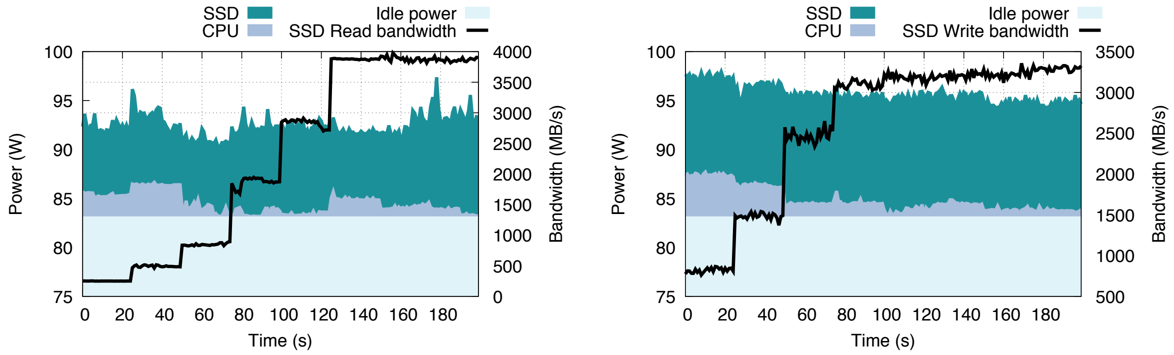
All the above steps allow us to compute an effective CPU power model. As a matter of comparison, this approach takes 34 hours approximatively on an Intel i3 2120 processor (373 events available) while it takes 16 hours approximatively on an ARM Cortex A15 processor (67 events available).

4.2 Learning SSD Power Models

Meisner et al. [MGW09] have been demonstrating that the dynamic power consumption drawn by I/O disk operations on typical servers are non-negligible, and thus have to be considered while modeling. As reported by Krevat et al. [KTG11], 2 rotational hard drives of the same provider and model may exhibit differences in their bandwidths, thus leading to impact the power consumption. Moreover, the data location is very important while reading or writing data into a rotational hard drive. Indeed, a rotational hard drive is composed of 1 or several platters, each of them having at least 1 head to handle data. Each action may therefore implies several mechanical movements and impacts the disk power consumption, making thus the reproducibility of experiments and the accuracy of the underlying power models almost bad.

Given the increasing spreading use of SSD hard drives over the last few years, we therefore choose to focus on the study of their power consumption. This decision have also been strengthen by the reproducibility of the experiments we target.

Figure 4.7: Comparison of power consumptions between CPU and SSD by varying the throughput with the `fio` tool.



(a) SSD read operations.

(b) SSD write operations.

To demonstrate that the SSD component is a power consuming component, we first made an experiment for comparing CPU and SSD power consumptions while performing reading and writing operations on the Intel Xeon E5-2630 server (cf. Appendix A) with 1 SSD of 372GB (Intel SSDSC2BX40). The PowerSpy Bluetooth power meter was also used to retrieve the overall power consumption of the server, and PowerAPI, to estimate the CPU power consumption among the execution.

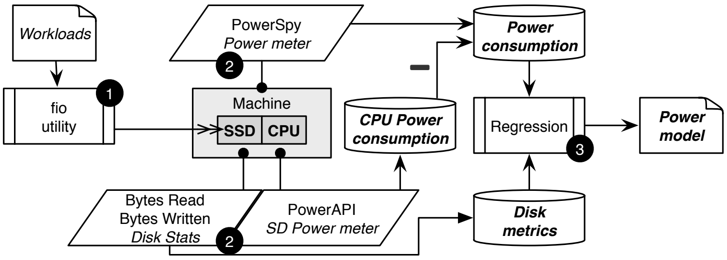
Figure 4.7a depicts power consumptions while performing read operations, and Figure 4.7b, writing operations. As it can be observed, the SSD component can have an important impact on the power consumption and may represent nearly 10 W for a write operation and nearly 7 W for a read operation, exhibiting bandwidths of 775 MB/s and 251 MB/s, respectively (reached with a block size of 4Kb). It also assesses that the CPU still consumes power consumption even if only disk operations are performed. This can be explained because all the I/O requests have to go through the CPU before being redirected to the SSD component itself. One can also mention that the power consumption stops to increase above a certain threshold—*i.e.*, from 8 MB/s for the reading, and from 1 MB/s for the writing. We therefore conclude that the relationship between these operations and the power consumption is not linearly proportional and cannot be expressed as $P_{SSD} = a \times Bytes_{read} + b \times Bytes_{written}$, where a and b are constants, contrary to the conclusions made by Noureddine et al. [Nou14].

Based on these observations, we accordingly propose an empirical and adaptive approach for learning automatically SSD power models.

4.2.1 Empirical Approach

To learn the SSD power models, one has to carefully select a workload that can perform and synchronize the various disk operations to correctly capture the power consumption.

Figure 4.8: Empirical SSD power modeling approach.



We describe each step depicted in Figure 4.8 in the following paragraphs.

Input workload. We choose the `fio`[6] tool, an open-source software, to perform specific disk operations. This tool allows to create specific I/O operations on the disk as defined by the user. Its main strengths come from its community, its flexibility, and its ability to work well on classical or SSD hard drives. Given the options provided by `fio`, we therefore choose to synchronize the operations made on the SSD disk and to invalidate the cache before each operation for limiting the buffering and cache effects on the collected data. We designed 2 workload scenarios: the first one represents the reading operation, while the other describes the writing operation. During each scenario, we incrementally change the bandwidth to detect and to reach automatically the maximum bandwidth.

Acquisition of raw metrics. Different metrics are collected along the learning phase. We first collect the number of bytes read and written at runtime to compute the bandwidths that represent the SSD activities. Then, we use PowerAPI for estimating the CPU power consumption along the execution. Indeed, as shown in Figure 4.7, each disk operation

[6] `https://github.com/axboe/fio`

impacts the CPU and it is thus reflected in its power consumption. It is therefore important to track the CPU power consumption while collecting raw power measurements to trustfully represent the real SSD power consumption. In addition, we use the PowerSpy power meter for reporting the power consumption of the whole machine as “ground truth”. To improve the accuracy of the power model, the scenarios described above are running several times for limiting the variance in the metrics collected.

Power model by regression. As SSD disk operations lead to CPU activities, we therefore subtract the CPU power consumption from the data retrieving by the PowerSpy power meter. The bandwidths and raw power measurements collected along the workload execution are then injected into a regression for connecting the evolution of the power consumption while reading or writing on the SSD disk. A power model is built per disk activity as the reading and writing activities do not have the same power consumptions.

In practice, we extend the formula described in Section 4.1.1 to represent the power consumption of an host, P_{host}, as:

$$P_{host} = P_{idle} + P_{CPU} + \left(\sum_{pid \in PIDs} P_{SSD_r} \times (br_{pid}/br_{SSD}) + P_{SSD_w} \times (bw_{pid}/bw_{SSD}) \right) \quad (4.6)$$

where P_{idle} represents the intrinsic power consumption of the host machine—the idle power, P_{CPU} is the CPU power consumption estimated by PowerAPI while doing I/O operations, P_{SSD_r} and P_{SSD_w} define the overall SSD power consumptions while reading and writing on disk respectively, bw_{pid} and br_{pid} are 2 vectors that contain the effective bandwidths at runtime per running process pid, br_{SSD} and bw_{SSD} also are 2 vectors that contain the overall bandwidths at runtime of the SSD read and write operations.

As can be seen in Figure 4.7, the smaller bandwidth to read or write on a SSD disk implies the greater CPU power consumption and can lead to an higher variance in data collected. We consequently choose to represent each SSD disk operation as piece-wise linear functions to model the relationship between the power consumption and the bandwidths that have not been reached during the learning phase.

As an example, the SSD power models of the Intel Xeon E5-2630 server are defined as follows:

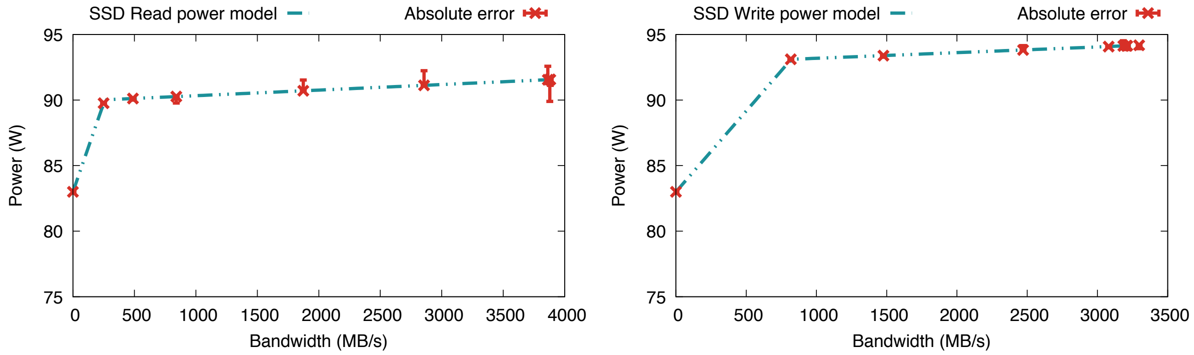
$$P_{SSD_r} = \begin{cases} 0.03 \cdot br_{SSD} & \text{if } 0 \leq br_{SSD} \leq 250 \text{ MB/s} \\ 4.25\text{e}{-04} \cdot br_{SSD} + 6.91 & \text{if } br_{SSD} > 250 \text{ MB/s} \end{cases} \quad (4.7)$$

$$P_{SSD_w} = \begin{cases} 0.01 \cdot bw_{SSD} & \text{if } 0 \leq bw_{SSD} \leq 818 \text{ MB/s} \\ 4.32\text{e}{-04} \cdot bw_{SSD} + 9.75 & \text{if } bw_{SSD} > 818 \text{ MB/s} \end{cases} \quad (4.8)$$

Equation 4.7 is depicted in Figure 4.9a, while Equation 4.8 in Figure 4.9b. The SSD read and write power models exhibit relative errors of 3.61% (0.18 W) and 0.77% (0.09 W), respectively. The piece-wise linear regression is thus well-suited for learning SSD power models. To further validate the regression technique used, we compute the Normalized Root Mean Square Deviation (NRMSD). The NRMSD is defined as $\frac{\sqrt{\text{MSD}}}{\overline{y}}$, where the Mean Square Deviation (MSD) allows to quantify the difference between the estimated and observed values and $\overline{y}$ represents the median of the expected values. The MSD is then expressed as $\frac{1}{n} \sum_{i=1}^{n} (\widehat{Y}_i - Y_i)^2$ where $\widehat{Y}$ is a vector of the estimated values, while Y represents the vector of the observed values. It is therefore very useful while comparing different power models with different datasets or scales, and is usually represented as

Figure 4.9: SSD power models.

(a) Power model for the SSD read operation. (b) Power model for the SSD write operation.



a percentage, where a low value indicates low residual errors and thus means a good accuracy. The learned read and write SSD power models exhibit NRMSD percentages of 11.9% and 2.3%, thus assessing our approach.

Summary

Since the publication of the first analytical power model [KZ08], the research community has been intensively investigating the design of power models by considering different hardware architectures, power-aware features, workloads, and modeling techniques. Nevertheless, the state-of-the-art in this area demonstrates that the designed power models are mostly hand-crafted and are based on assumptions that prevent their reuse in other execution contexts and their deployment at scale. Beyond the few power models presented as example in this chapter, our contributions therefore offers different approaches for learning CPU and SSD power models. Our approaches exploit freely available tools and benchmark suites to thoroughly train the dataset used for building the power models. The training dataset are then exploited by a combination of regression analysis techniques to identify the most accurate ones. We also aim to foster the research on power models by proposing adaptive approaches for learning power models. In particular, one has to consider other power consuming components, such as GPU [JSM12], to incrementally learn their power consumption model and to thus provide a wider cartography of the power consumption of a software system.

Chapter 5

Building Software-Defined Power Meters “à la carte”

Table of Contents

Researchers and operators have been proposing solutions to increase energy efficiency at all levels, from application to runtime and to hardware. As surveyed by Orgerie et al. [ODL14], examples include methods for energy-based task scheduling, energy-efficient software, dynamic frequency and voltage scaling and energy-aware workload consolidation. Power monitoring is usually achieved with the support of hardware measurement equipments, such as power meters or specialized integrated circuits. Such solution is not suitable in large-scale environments because requires costly investments. Furthermore, it provides coarse-grained power consumptions—*i.e.*, hardware-level—and cannot be a relevant solution when targeting fine-grained power estimation. Consequently, software-centric and scalable approaches have to be considered. Contrarily to the hardware approach, it requires power models for trying to estimate at best the power consumptions drawn by software. Power estimation of running software processes requires to tackle several challenges and provides indicators to drive scheduling or power capping heuristics. Firstly, the software-centric solution needs to be compatible with the majority of modern architectures, adaptive and easily deployable to foster their usage. Secondly, it needs also to take into account the most recent energy saving features for producing the most accurate power estimation. Finally, the solution has to deliver power estimation at high frequency to make critical runtime decisions.

The rest of this chapter is organized as follows. We first explain why it is so important to have efficient software-defined power meters in Section 5.1. We secondly describe in Section 5.2 our open-source and non-invasive middleware toolkit, PowerAPI, as an efficient solution for assembling software-defined power meters upon requirements. We finally provide an overview of the module components available in PowerAPI and several examples of software-defined power meters built with PowerAPI in Section 5.3 and Section 5.4 respectively.

5.1 The Need For Software-Defined Power Meters

Software-defined power meters are customizable and adaptable solutions that can deliver raw power consumptions or power estimation at various frequencies and granularity, depending on the requirements. Power estimation of running processes is not a trivial task and must tackle several challenges. Several solutions are already proposed by the state-of-the-art:

pTop [DRS09] is a process-level power profiling tool for Linux or Windows platforms. pTop uses a daemon in background for continuously profiling statistics of running processes. pTop keeps traces about component states and stores temporarily the amount of energy consumed over each time interval. This tool displays, similarly to the `top` output, the total amount of energy consumed—*i.e.*, in Joules—per running process. They propose static built-in energy models for CPU, disks, and network components. An API is also provided to get energy informations of a given application per selected component.

PowerScope [FS99] is a tool capable of tracking the energy usage by application for later analysis and optimizations. Moreover, they map the energy consumption to procedures within applications for better understanding the energy distribution. Their approach uses 2 computers for offline analysis: one for sampling system activities (*Profiling* computer) and another for collecting power measurements from external digital multimeter (*Data Collection* computer). Once the profiling completed, the *Profiling* computer is next used to compute all energy profiles for later usage.

PowerTOP [Int15b] is a Linux tool for finding energy-consuming software on multiple component sources (*e.g.*, CPU, GPU, USB devices, screen). Several modes are available, such as the calibrate mode to test different brightness levels as well as USB devices, or the interactive mode for enabling different energy saving mechanisms not enabled by default. This software-defined power meter can only report power estimation while running on battery within an Intel laptop, or only usage statistics otherwise.

SPAN [WCS11] is designed for providing real-time power phases information of running applications. They also propose external API calls to manually allow developers to synchronize the source-code applications with power dissipation. They first design micro benchmarks for sampling the only HPC used—*i.e.*, IPC—and they gather the HPC data and raw power measurements for computing the parameters of their hand-crafted power models. They can next use SPAN for real-time power monitoring and offline source-code analysis.

Different limitations can be thus extracted from these solutions and the state-of-the-art. Most solutions are monolithic, created for specific needs and cannot be easily tuned or configured for assembling new kind of power meters [DRS09; FS99; Int15b; WCS11]. Power models used by such power meters cannot be used or adapted to modern architectures because they have been especially designed [WCS11] or they are using specific metrics [DRS09] that cannot trustfully represent recent features. They can as well lack of modularity [WCS11] and require additional investments [FS99].

In order to detect energy-consuming applications and to apply critical energy decisions at runtime, one rather needs a middleware toolkit fully modular, configurable and adaptive that can report power estimation (or power measurements) at high frequency on heterogeneous systems. For promoting the usage of such solution as a good alternative of physical power meters, the proposed solution has to be freely available for the community.

We consequently propose and design PowerAPI, an open-source middleware[1] for

[1] Freely available from: `http://powerapi.org`

assembling software-defined power meters upon needs.

5.2 PowerAPI, a Middleware Toolkit

The PowerAPI middleware toolkit has been first created during the thesis of Adel Noureddine [Nou14]. PowerAPI is built on top of Scala and the actor programming model using the Akka library. Scala is a relatively new language (2003) that combines object-oriented and functional programming paradigms and has a strong static type system [Ode+04]. Scala code runs on top of the java virtual machine (JVM) and is fully interoperable with Java code. Akka is an open-source toolkit for building scalable, and distributed applications on the JVM and pushes forward the actor programming model as the best programming model for concurrency. This open-source contribution was the first step to a better understanding of the software power consumption. However, this first iteration of PowerAPI has not been designed to support the complexity of nowadays multi-core architectures. This is the reason why we therefore propose the second major iteration of this middleware toolkit to fully support this kind of widely spread architectures. This new iteration was also the opportunity to better support all the features of the Akka library.

The software components of PowerAPI are implemented as actors, which can process millions of messages per second [Nor12], a key property for supporting real-time power estimation. PowerAPI is therefore fully asynchronous and scales both on several dimensions—*i.e.*, the number of input sources, the requested monitoring frequencies, the number of monitoring and the number of targets.

More especially, the PowerAPI toolkit identifies 5 type of actor components:

Clock actors is the entry point of our architecture and allows to meet throughput requirements by emitting ticks at given frequencies for waking up the other components;

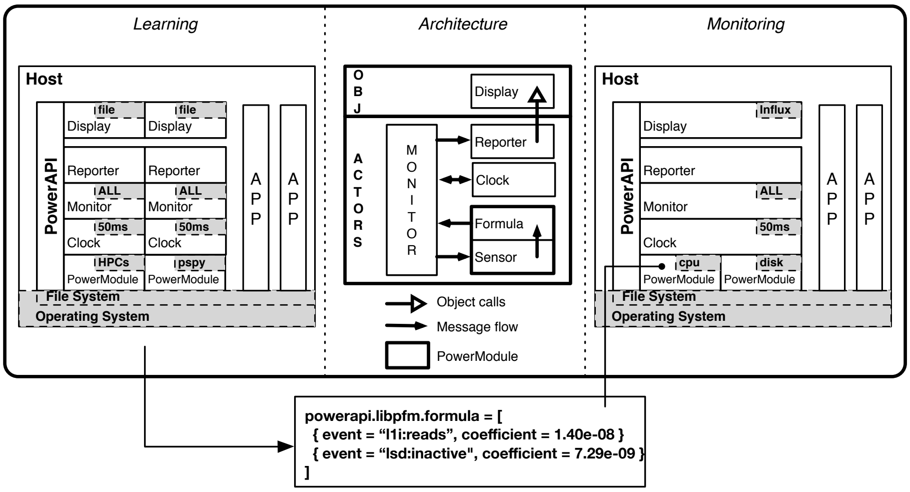
Monitor actors reflect the power monitoring request for one or several processes. They react to the messages published by a clock actor, configured to emit tick messages with a given frequency. The monitor is also responsible for aggregating the power estimation by applying a function (*e.g.*, SUM, MEAN, MAX) defined for the monitoring when needed;

Sensor actors connect the software-defined power meters to the underlying system in order to collect raw measurements of system activity. Raw measurements can be coarse-grained power consumption reported by third-party power meters and embedded probes (*e.g.*, RAPL), or CPU activity statistics as delivered by the process file system (`ProcFS`). Sensors are triggered according to the requested monitoring frequency and forward raw measurements to the appropriate formula;

Formula actors use the raw measurements received from the sensor to compute a power estimation. A formula implements a specific power model [Kan+10; VWT13] to convert raw measurements into power estimation. The granularity of the power consumptions reported by the formula (machine, core, process) depends on the granularity of the measurements forwarded by the sensors;

Reporter actors finally gives the power estimation computed by the aggregating function to a Display object. The Display object is responsible to convert the raw power estimation and the related informations (*e.g.*, the timestamp, the monitoring id or the devices) into a suitable format. The built report is then provided, by example, via a web interface, via a virtual file system (*e.g.*, based on FUSE), or can be uploaded into a database (*e.g.*, InfluxDB).

Figure 5.1: PowerAPI's architecture & deployment.



```
powerapi.libpfm.formula = [
  { event = “l1i:reads”, coefficient = 1.40e-08 }
  { event = “lsd:inactive", coefficient = 7.29e-09 }
]
```

As actors have lightweight CPU and/or memory footprints, a Monitor, a Sensor, a Formula, and a Reporter actors are created per monitoring request and target. Each Sensor and Formula actors being tightly coupled, we grouped them as a PowerModule that represents the link between the input data and the power model. All actors are centralized on a common event bus where they can publish messages or subscribe to topics for actively waiting events. Particularly, we use a Lookup Classification[2] event bus for extracting a classifier per published event, maintaining a set of subscribers attached to each classifier, and thus creating channels between actors.

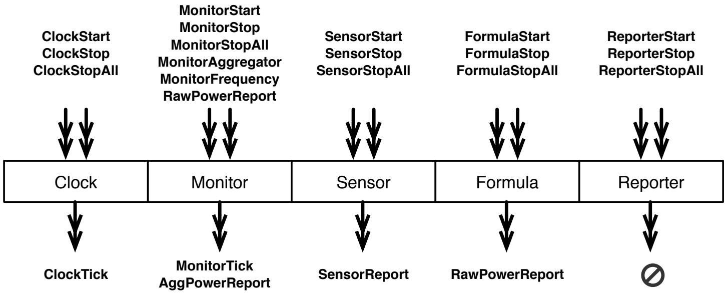
The overall architecture of PowerAPI is described in Figure 5.1. Several PowerModule components can be assembled together for grouping power estimation from multiple sources. For the sake of clarity, we do not detail the event bus described earlier. One can see that PowerAPI is fully modular, can be used to assemble power meters upon needs and to fulfill all monitoring requirements. We can also note that PowerAPI is a non-invasive solution and does not require costly investments or specific kernel updates. An overview of the exchanged messages between actors is depicted in Figure 5.2.

2 instances of software-defined power meters are also depicted in this figure. In the left side, one can find an instance of PowerAPI especially configured to learn the CPU power models. This instance is composed of 2 PowerModule components, one for retrieving raw accurate CPU metrics via `libpfm`, and another, for retrieving the power measurements from the bluetooth power meter. The data are then forwarded to several files to be later processed by our learning approaches. The resulting power model is then written inside a configuration file that can be used later by a new instance of PowerAPI to estimate the power consumption. In the other side, another instance of PowerAPI is configured to use the aforementioned power model for producing fine-grained power estimation.

Akka actors form a kind of natural hierarchy—*i.e.*, each actor acts like a supervisor and implements a fault handling strategy for its children. 2 different strategies can be

[2] `http://doc.akka.io/docs/akka/2.4/scala/event-bus.html#Lookup_Classification`

Figure 5.2: Overview of internal messages exchanged between the PowerAPI's actors.



Snippet 5.1: Supervisor definition.

```
trait Supervisor extends Actor {
  def handleFailure: PartialFunction[Throwable, Directive]

  override def supervisorStrategy: SupervisorStrategy =
    OneForOneStrategy(10, 1.seconds)
      (handleFailure orElse SupervisorStrategy.defaultStrategy.decider)
}
```

Snippet 5.2: Clock supervisor implementation.

```
class Clocks(eventBus: MessageBus) extends Supervisor {
  {...}

  def handleFailure: PartialFunction[Throwable, Directive] = {
    case _: UnsupportedOperationException => Resume
  }

  def receive: Actor.Receive = {
    case msg: ClockStart => start(msg)
    case msg: ClockStop => stop(msg)
    case msg: ClockStopAll => stopAll(msg)
  }

  {...}
}
```

used by a supervisor. The OneForOneStrategy only applies the chosen fault handling directives to the failing actor while the AllForOneStrategy concerns all its children. 4 fault handling directives—*i.e.*, Resume, Restart, Stop, and Escalate—can be chosen to define the actor behavior while failing. The Resume directive allows to stop the processing of the current message, to skip it, and to move to the next message available in the mailbox. The Restart one drops the failing actor and replace it with a new one, thus cleans out all its internal states. To avoid infinite restarting, different limits can be applied. When the Stop directive is applied, the crashed actor is stopped and destroyed. The Escalate directive forwards the failure to the upper supervisor.

The supervisor strategy is described in Snippet 5.1 and allows to define a fault handling strategy per supervisor. Our supervisors have been written to handle—*i.e.*, to create child actors upon request, to forward messages, or to stop a child actor when needed—a hierarchy of children actors. We describe our Clock supervisor in Snippet 5.2. A Clock child actor only fails when a message cannot be processed, represented by the UnsupportedOperationException in the code. Different supervisors have been implemented in PowerAPI, such as the Clock, Monitor, Sensor, and Formula supervisors, for limiting the spreading of exception and allowing PowerAPI to be fault-tolerant.

To produce ticks at given frequencies, we use the `scheduler(...)` method from the ActorSystem actor that allows to call a method at a given rate. This method returns a Cancellable object that we can use to stop the publishing when needed. For limiting the overhead, only one instance of Clock actor is created per requested frequency. This is why we keep an accumulator to know whether a Clock actor has been already created or whether the frequency is not used anymore. To limit its internal complexity and to avoid mutable data-structures that can lead to unpredictable behaviors, we use the `context.become(...)`. This method allows to replace the current actor state with a new one by using immutable structures or values, as described in Snippet 5.3.

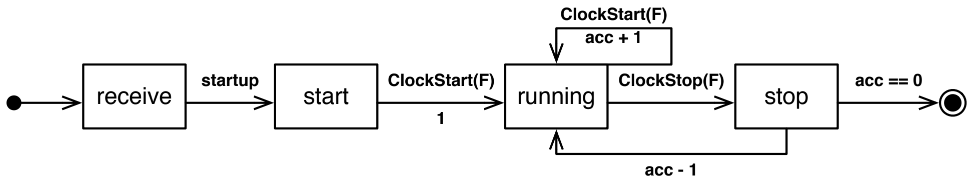
A Clock actor has 3 different states (cf. Figure 5.3):

start state is the entry point of a Clock actor. It allows to check that the ClockStart message is received first and consequently switch with its running state;

running state is invoked when a scheduler has been created and tick messages are published at a fixed rate. The accumulator is then incremented by 1 each time the same clock frequency is requested;

stop state is called when a monitoring has been stopped. The underlying scheduler is then stopped and the underlying actor is killed if the accumulator reaches 0, the accumulator is decremented otherwise.

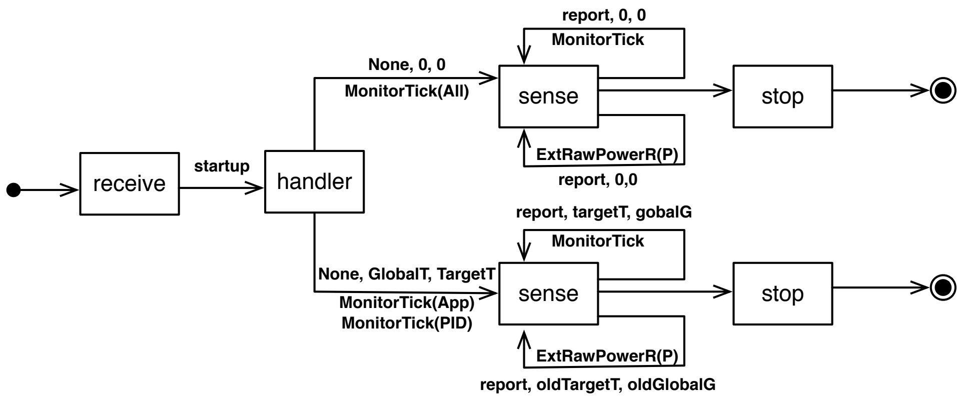
Figure 5.3: Clock actor state diagram.



PowerAPI can also be used as a connector to external probes for retrieving power measurements (*e.g.*, PowerSpy, RAPL, G5K OmegaWatt). To allow such connectivity, Snippet 5.3: Clock actor implementation.

```
class Clock(eventBus: MessageBus, frequency: FiniteDuration) extends Actor {

  def receive: Actor.Receive = starting orElse default

  def starting: Actor.Receive = {
    case msg: ClockStart => start()
  }

  def start(): Unit = {
    val timer = context.system.scheduler.schedule(Duration.Zero, frequency) {
      publishClockTick(frequency)(eventBus)
    }(context.system.dispatcher)

    context.become(running(1, timer) orElse default)
  }

  def running(acc: Int, timer: Cancellable): Actor.Receive = {
    case msg: ClockStart if msg.frequency == frequency =>
      context.become(running(acc + 1, timer) orElse default)
    case msg: ClockStop if msg.frequency == frequency => stop(acc, timer)
    case _: ClockStopAll => stop(1, timer)
  }

  def stop(acc: Int, timer: Cancellable): Unit = {
    if (acc > 1) {
      context.become(running(acc - 1, timer) orElse default)
    }
    else {
      timer.cancel()
      self ! PoisonPill
    }
  }
}
```

Figure 5.4: ExternalSensor actor state diagram.



developers have to create the business code that retrieves and forwards data to PowerAPI. Thanks to the event bus, the external probe pushes data at a given frequency (it can be independent of the frequency chosen in PowerAPI and limited by the hardware) and PowerAPI can pull them when made available. The given base implementation is provided in Snippet 5.4. Each given Sensor component is then responsible to represent the link between the probe and PowerAPI—*i.e.*, represented by the `extPMeter` parameter. Power measurements begin pushed to PowerAPI, we also use the `context.become(...)` method to store the forwarded raw measurement for using it at posteriori.

An ExternalSensor actor has 3 states (cf. Figure 5.4):

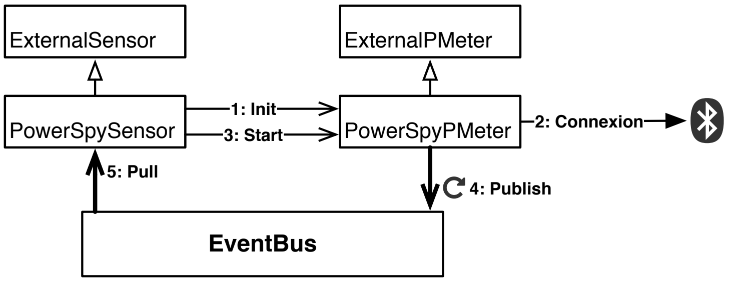
handler state allows to initialize the actor and to directly switch with its active mode;

sense state actively listens the messages published by the external probe—*i.e.*, the ExtRawPowerReport messages—and the monitoring. When the All target is asked, we only forward the raw power measurement to the Formula. Otherwise, when another target is asked—*i.e.*, a Process or an Application, the overall power consumption is downscaled by using an activity ratio, represented by a CPU activity;

stop state is invoked when a monitoring has been stopped and it triggers the cleaning of involved resources.

A schematic example of the PowerSpy's ExternalSensor is given in Figure 5.5. The same principle is also applied for RAPL, or G5K OmegaWatt and could be applied to any other external probes.

Figure 5.5: Link description between PowerAPI and the PowerSpy bluetooth power meter.



To support the utilization of PowerAPI as a middleware toolkit for defining software power meter, or for learning power models, we use the Docker[3] technology. It allows to package our applications inside lightweight images that contains everything needed to run them. These images can be directly downloaded[4] and used without any dependency to install (except Docker itself).

Figure 5.6 describes relevant statistics about the PowerAPI's code repository.

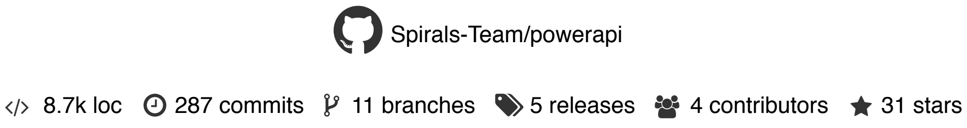
[3] `https://www.docker.com`

[4] Images are freely available from: `https://hub.docker.com/u/spirals/dashboard`

Snippet 5.4: Base implementation to establish a connexion between PowerAPI and an external probe.

```
abstract class ExternalSensor(eventBus: MessageBus, muid: UUID, target: Target,
  extPMeter: ExternalPMeter, idleP: Power) extends Actor {

  {...}

  def handler: Receive = target match {
    case All => sense(None, 0l, 0l)
    case _ =>
      val times = currentTimes
      sense(None, times._1, times._2)
  }

  def sense(report: Option[ExtRawPowerReport], oldTargetTime: Long,
    oldGlobalTime: Long): Receive = {

    case msg: RawPowerReport =>
      context.become(sense(Some(msg), oldTargetTime, oldGlobalTime))
    case msg: MonitorTick if target != All =>
      ...
    case msg: MonitorTick if target == All =>
      ...
  }
}
```

Figure 5.6: PowerAPI's repository statistics.



Spirals-Team/powerapi

</> 8.7k loc 287 commits 11 branches 5 releases 4 contributors ★ 31 stars

5.3 PowerAPI's Modules

As previously mentioned, a software-defined power meter created with PowerAPI can represent an assembly of PowerModule components. In particular, 8 types of PowerModule components have been designed:

procfs-cpu-simple | sigar-cpu-simple modules use the ProcFS and the Sigar API,[5] respectively. The Sigar API provides a common and portable interface for retrieving system statistics, such as, system cpu load, per-process cpu or network metrics, in most operating systems. The core API is written in C, but several bindings are available in most programming languages;

cpu-dvfs module benefits from `cpufreq-utils` to load 2 kernel modules (`acpi-cpufreq` and `cpufreq_stats`) for retrieving the CPU time spent per each available frequency when the DVFS feature is enabled. The CMOS formula ($P = C \cdot f \cdot V^2$) is then used for estimating with better accuracy the CPU power consumption;

[5] https://support.hyperic.com/display/SIGAR/Home

`libpfm` module takes advantage of the `libpfm` library to retrieve accurate metrics about the various CPU activities and uses them as inputs for CPU power models;

`disk` module profits from block IO controller (BLKIO) cgroup subsystem for retrieving global and per-process SSD metrics and thus provides per-process SSD disk power estimation;

`powerspy` module allows to integrate the PowerSpy bluetooth power meter for retrieving the overall power consumption of a node. Different commands and data are exchanging with the Bluetooth protocol.

`g5k-omegawatt` module employs the Kwapi API[6] to collect raw metrics from PDUs at infrastructure level on Grid'5000 nodes when available. PowerAPI can be then used in a wide diversity of machines, thanks to Grid'5000, for performing and assessing experiments;

`rapl` module uses the RAPL counters available on most recent Intel processors for getting different power consumption information. RAPL is divided in different domains, varying across CPUs, to collect power consumptions at different levels. The `package` domain allows to get the whole power consumption of the CPU (core and uncore devices together), while core and uncore devices are represented by the `pp0` and `pp1` packages respectively. In the most recent CPU architectures, it is also possible to retrieve the DRAM power consumption, thanks to the new `dram` package. In PowerAPI, we only use currently the `package` domain in order to represent the overall CPU power consumption.

All the above PowerModule components are fully configurable, thanks to the Typesafe Configuration library.[7]

5.4 PowerAPI's Assemblies

Different ways are proposed for creating software-defined power meters upon needs. Firstly, end users can assemble and build their own power meters by using our customizable and documented API. Secondly, a CLI has been made available online and can be used for testing or doing basic power monitoring that use default components.

An example of software-defined power meter is described in Snippet 5.5 and shows the monitoring created for the per-process power estimation in Section 6.3.3. Such power meter can also be created by using the CLI instead, as demonstrated in Snippet 5.6.

When possible, one can mix several PowerModule components together and get metrics from multiple sources as demonstrated in Snippet 5.7. That allows to retrieve input data as synchronized as possible and thus reduce the errors involved.

In order to implement the empirical approach for learning CPU power models described in Section 4.1.1, we create a special project, called Sampling, that contains different steps.

First, this project uses the software-defined power meter for collecting data during the workload execution for all available frequencies and different CPU load variations. Once the collecting done, it processes all generated files to group and to split data according to these parameters and, to be processed more easily in the next step. Lastly, the files

[6] `https://launchpad.net/kwapi`

[7] `https://github.com/typesafehub/config`

Snippet 5.5: Code used in Section 6.3.3 to demonstrate that PowerAPI is able to achieve accurate per-process power estimation.

```
object SDPowerMeter extends App {
  val pspy = PowerMeter.loadModule(PowerSpyModule())
  val cpu = PowerMeter.loadModule(LibpfmCoreProcessModule())

  val pspyDisplay = new FileDisplay("powerspy.dat")
  val cpuDisplay = new FileDisplay().perTarget()

  val pspyMonitoring = pspy.monitor(All).every(250.milliseconds).to(pspyDisplay)
  val cpuMonitoring = cpu.monitor("bt.C", "cg.C (1)", "cg.C (2)", "freqmine")
                         .every(250.milliseconds)
                         .to(cpuDisplay)

  pspyMonitoring.waitFor(1.minute)

  pspyMonitoring.cancel()
  cpuMonitoring.cancel()
  pspy.shutdown()
  cpu.shutdown()
}
```

Snippet 5.6: CLI command to create the software-defined power meter described in Snippet 5.5.

```
./powerapi modules powerspy \
    monitor --frequency 250 --all --file powerspy.dat \
  modules libpfm-core-process \
    monitor --frequency 250 --apps bt.C,cg.C (1),cg.C (2),freqmine --file-per-target \
duration 60
```

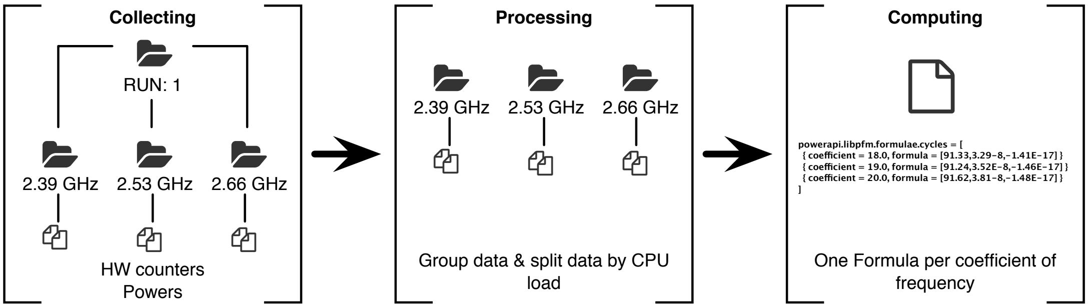
Snippet 5.7: Code used in Section 6.1.1 to prove the performance of PowerAPI compared to RAPL.

```
object SDPowerMeter extends App {
  val sdPMeter =
    PowerMeter.loadModule(PowerSpyModule(), RAPLModule(), LibpfmCoreModule())

  val sdDisplay = new FileDisplay().perDevice

  val sdMonitoring = sdPMeter.monitor(All).every(1.second).to(sdDisplay)

  sdMonitoring.waitFor(250.seconds)

  sdMonitoring.cancel()
  sdPMeter.shutdown()
}
```

from the Processing step are used together with a Regression technique for producing one power model per frequency (represented by its coefficient).

All the above steps are detailed in Figure 5.7.

Figure 5.7: Phases used by PowerAPI to generate the power models and generate the configuration file.



```
powerapi.libpfm.formulae.cycles = [
  { coefficient = 18.0, formula = [91.33,3.29-8,-1.41E-17] }
  { coefficient = 19.0, formula = [91.24,3.52E-8,-1.46E-17] }
  { coefficient = 20.0, formula = [91.62,3.81-8,-1.48E-17] }
]
```

An example of such configuration file is described in Snippet 5.8. The JSON format was chosen to represent the generated power models in order to be human-readable and to be efficiently parsed at runtime.

Snippet 5.8: Example of configuration file generated by PowerAPI that can directly be used at runtime.

```
{
  "powerapi.libpfm.formulae.cycles": [
    { "coefficient": 18.0, "formula": [91.33,3.29E-8,-1.41E-17] }
    { "coefficient": 19.0, "formula": [91.24,3.52E-8,-1.46E-17] }
    { "coefficient": 20.0, "formula": [91.62,3.81E-8,-1.48E-17] }
  ]
}
```

This project is packaged and can be easily launched to learn a new CPU power model in few minutes (cf. Snippet 5.9).

Snippet 5.9: Command to launch the CPU power models learning.

```
./bin/sampling --all collecting processing computing
```

Summary

PowerAPI therefore provides a flexible way to assemble software-defined power meters upon needs. Our middleware toolkit is published as open-source software[8] under AGPLv3 license to foster the wide adoption of software-defined power meters. As demonstrated in Snippets 5.5, Snippet 5.6 and, Snippet 5.7, PowerAPI can be used in various ways

[8] Available from: `http://powerapi.org`

to estimate the power consumption at process, application or machine-level. Several PowerModule components have already been implemented for the CPU, SSDs, or to integrate external probes and thus allowing PowerAPI to be as modular as possible. Our middleware toolkit is non-invasive and acts like a normal application on the system. PowerAPI can scale both on the number of selected modules, the number of monitored targets, the number of monitoring requests, and the defined monitoring frequencies. We consider PowerAPI as a cornerstone to new energy aware scheduling [Bam+13; Bel00; KOS16; Mog+13; Ras15], to energy-proportional computing [BH07; Kri+10; Mei+11; Pre+15], to new kind of optimizations [Sch+14], and to a better understanding of the power consumption drawn by a software [NRS14; NRS15; Ste13]. By establishing a clear knowledge of usual energy leaks, we intend to identify green patterns as a methodological guideline that can assist the developers in building energy-efficient software. Additionally, the emergence of renewable energies is introducing the need for the development of adaptive strategies that can cope with the sporadic nature of these energy feeds.

Part III

Evaluations

Chapter 6

Process-Level Power Estimation in Multi-Core Systems

Table of Contents

In Part II, we present different approaches for learning CPU and SSD power models, and our toolkit, PowerAPI, for building software-defined power meters. Each of the contribution is then validated in this chapter.

The rest of this chapter is organized as follows.

We first validate our learning approaches for CPU power models in Section 6.1 and for SSD power models in Section 6.2. We only use open-source and well-known benchmark suites as input assets to differ with the ones used while learning, and we use the Bluetooth power meter, PowerSpy, as a ground truth to retrieve raw power measurements. We finally describe several applicative scenarios to demonstrate the adaptiveness and effectiveness of our approaches for learning power models, and to be tightly coupled with PowerAPI for providing real-time power estimation of concurrent processes in Section 6.3.

6.1 Assessing CPU Power Models

6.1.1 Empirical Learning

First, to demonstrate that our empirical approach (cf. Section 4.1.1) is able to handle applications with diverse loads, we start with a baseline experiment on the Intel i3 2120 processor (cf. Appendix A). We run the `stress` tool in combination with `cpulimit`[1] on a single core. Every 30 seconds, the load applied by the `stress` command is decreased by 10%. In this experiment, we compare the results not only to PowerSpy, but also to RAPL counters, which are available on recent Intel processors (since the *Sandy Bridge* processor generation). Furthermore, the CPU frequency has been fixed to 1.6 GHz to avoid measurement peaks induced by frequency switches.

Figure 6.1: Decreasing load of `stress` on the Intel i3 2120 processor, compared to RAPL.

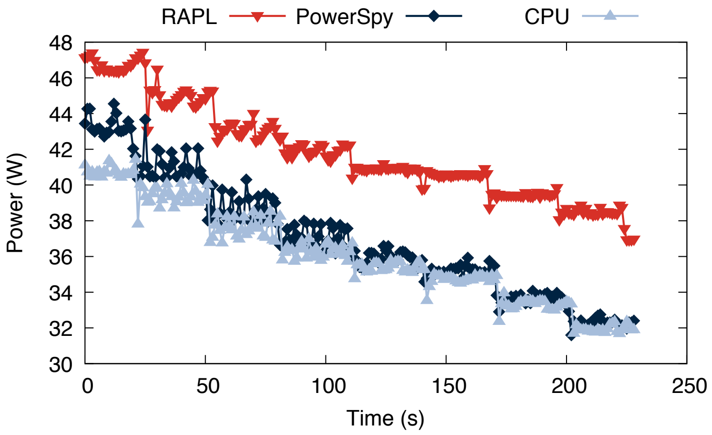


Figure 6.1 shows the results of this experiment. We see that the RAPL counters follow the same trend than the workload, but tend to overestimate the power consumption of a single CPU. Compared to RAPL, the learned power model provides power estimation that is much closer to PowerSpy that we consider as the ground truth. This indicates that our learning approach performs accurate sub-system estimation in various load scenarios, key challenge for tracking the power consumption of everyday software on modern architectures.

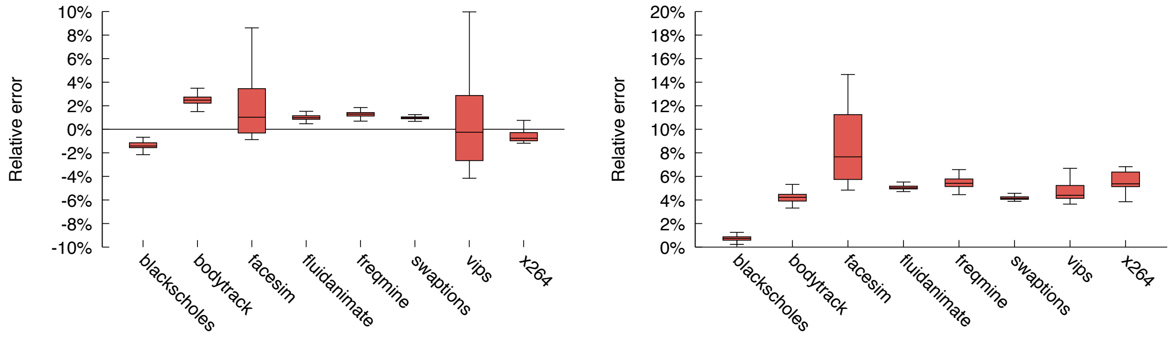
We use the well-known PARSEC [BL09] v2.1 benchmark suite to evaluate our approach in the next scenario.

PARSEC includes emerging recognition, mining, and synthesis (RMS) as well as system applications that mimic large-scale multi-threaded commercial programs. This benchmark suite is diverse in terms of working set, locality, data sharing, synchronization and off-chip traffic, thus making it well-designed to stress multi-core architectures. In particular, we report the power consumption of all benchmarks available on 2 different configurations (cf. Appendix A).

Figures 6.2a and 6.2b report the relative error between the measured and estimated power consumption (by aggregating the power consumption per process using P_{host}) on 2 Intel processors: a Xeon W3520 and a i3 2120.

[1] `https://github.com/opsengine/cpulimit`

Figure 6.2: Relative error distribution of the PARSEC benchmark suite on 2 Intel processors.



(a) On the Intel Xeon W3520 processor.

(b) On the Intel i3 2120 processor.

Synthesis

Even though PARSEC was not included as a workload during the learning phase, one can observe that the power estimation produced by the power models is very close to the power measurements collected for the testbed processors. The closest method, described in [VWT13], adopts an iterative approach to minimize the error rate to at most 5%. However, the key limitations of their approach are *i).* they only consider full usage of cores, and *ii).* they rely on application-specific models. Our solution is application-agnostic, supporting both CPU- and memory-intensive workloads, and are processor-aware, considering different models of CPUs including *multi-cores*, *simultaneous multi-threading*, *dynamic voltage/frequency scaling*, and *dynamic overclocking* features.

While the multi-core CPU power model proposed in this section is only assessed on Intel processors (cf. Appendix A), the approach described does not rely on any Intel-specific extensions. Indeed, the learned power models consider processor features (HT, SS, TB) which are also available under different names from other vendors. In particular, AMD processors also represent a target CPU architecture for our learning approach, but a limitation of the `libpfm4` library currently prevents the access of the reference-cycles counter to compute the frequency. Once this barrier lifted, we expect to be able to demonstrate the validity of the approach on AMD processors with results similar to those reported in this section.

6.1.2 Architecture-Agnostic Learning

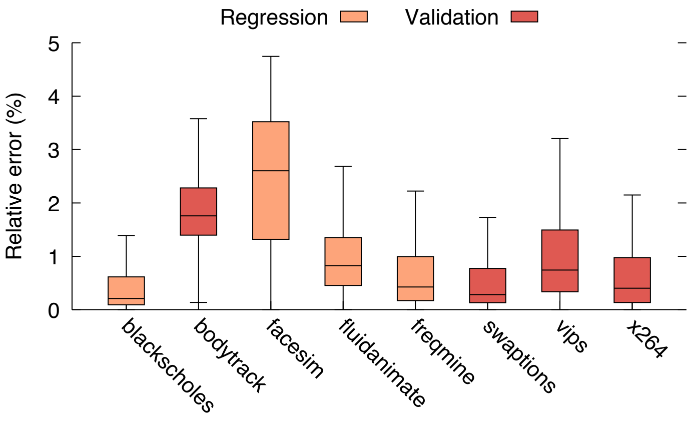
CPU power models that are automatically learned by our architecture-agnostic approach (cf. Section 4.1.1) for each of the 4 CPU architectures described in Appendix A. For assessing the CPU power models, we compare the power estimation from the learned power model with raw power measurements from a physical power meter, PowerSpy. The readings of the power meter are collected simultaneously to the HPC events to ensure that the data is well synchronized. As in Section 6.1.1, we use the PARSEC benchmark suite. The set of 8 benchmarks (W) is then split into two subsets: *i).* those used to learn the CPU power model (R), and *ii).* the remaining ones used for the purpose of validation (E).

Given that we focus on CPU—and memory-intensive—systems, we report the power drawn by a whole host, which is defined as follows: $P = P_{idle} + P_{CPU}$, where P_{idle}

corresponds to the static power consumption and P_{CPU} to the dynamic power consumption drawn by the CPU.

Intel Xeon W3520

Figure 6.3: Relative error distribution of the PARSEC benchmarks on the Intel Xeon W3520 processor ($P_{idle} = 92\ W$).



System configuration

This server is configured with Linux Ubuntu 14.04 (kernel 3.13).

CPU power model

The resulting CPU power model computed with the training subset of benchmarks, R_3, comprises 2 HPC events from the PMU nhm ($e1 =$ l1i:reads, $e2 =$ lsd:inactive):

$$P_{idle} = 92\ W\ ;\ P_{CPU} = \frac{1.40 \cdot e1}{10^8} + \frac{7.29 \cdot e2}{10^9} \tag{6.1}$$

To assess the effectiveness of the robust ridge regression, we inspect the *eigenvalues* of corresponding correlation matrix. Very low values (closed to zero, 10^{-3}) in the resulting matrix denote a collinearity between variables. The selected events have *eigenvalues* of 1.5 and 0.5, confirming the non-collinearity of the HPC events included in this CPU power model.

Model accuracy

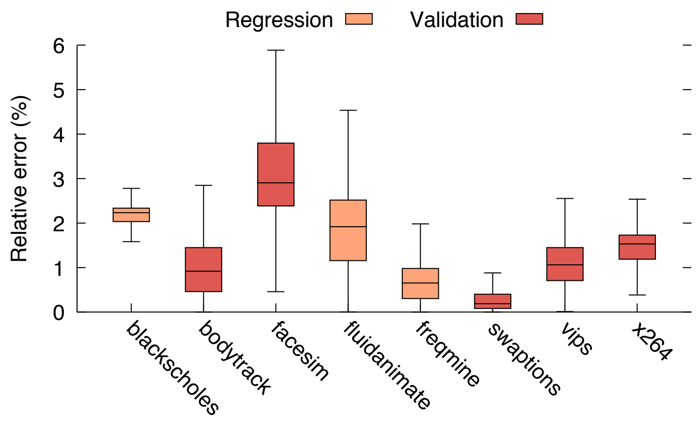
Our approach isolates the idle power consumption of the processor whose relationship to TDP is defined in [Riv+07] as $P \simeq P_{idle} + 0.7 \times TDP$. Figure 6.3 reports on an average relative error of 1.35% (1.60 W), which improves the existing CPU power models on such configuration [Col+15b].

Intel i3 2120

System configuration

This server is configured with Linux Ubuntu 14.04 (kernel 3.13).

Figure 6.4: Relative error distribution of the PARSEC benchmarks on the Intel i3 2120 processor ($P_{idle} = 30\ W$).



CPU power model

The resulting CPU power model computed with a training subset, R_3, is composed of 2 HPC events from PMU snb (e_1 = idq:empty, e_2 = uops_dispatched:stall_cycles) and 1 HPC event from PMU snb_unc_cbo0 (e_3 = unc_clockticks):

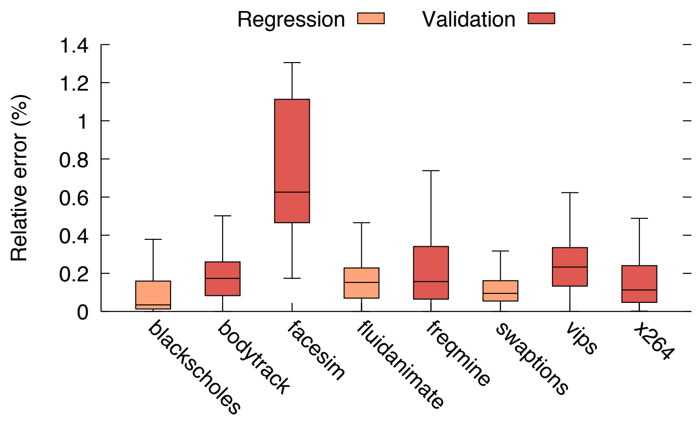
$$P_{idle} = 30\ W\ ;\ P_{CPU} = \frac{1.12 \cdot e1}{10^8} + \frac{4.55 \cdot e2}{10^9} + \frac{6.89 \cdot e3}{10^{10}} \tag{6.2}$$

Model accuracy

Although it has a similar configuration to the Intel W3520 processor described earlier, the resulting CPU power model strongly differs. Yet, Figure 6.4 reports a relative error of 1.57% (0.71 W), on average, which confirms the accuracy of our CPU power models for Intel architectures.

AMD Opteron 8354

Figure 6.5: Relative error distribution of the PARSEC benchmarks on the AMD Opteron 8354 processor ($P_{idle} = 390\ W$).



System configuration

This server is configured with Linux Ubuntu 14.04 (kernel 3.13).

CPU power model

The resulting CPU power model computed from the training subset, R_3, is composed of 3 HPC events from the PMU fam10h_barcelona (e_1 = probe:upstream_non_isoc_writes, e_2 = instruction_cache_fetches, e_3 = retired_mmx_and_fp_instructions:all):

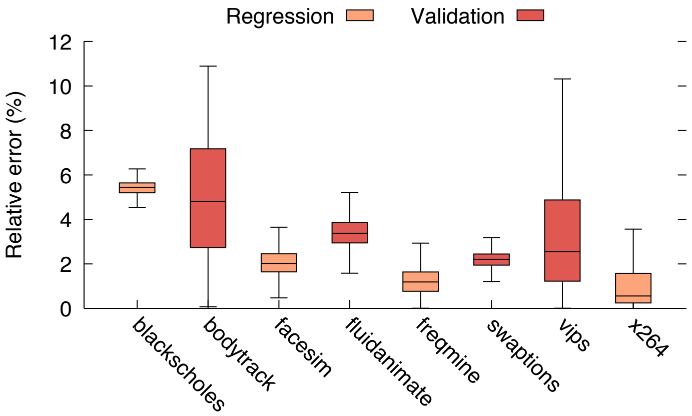
$$P_{idle} = 390\ W\ ;\ P_{CPU} = -\frac{2.63 \cdot e1}{10^4} + \frac{8.20 \cdot e2 + 3.16 \cdot e3}{10^9} \tag{6.3}$$

Model accuracy

Figure 6.5 reports on a relative error of 0.20% (0.81 W), on average. While most of the works in the state of the art focus on Intel architectures 2.1, the accuracy of the CPU power model we generate for the AMD Opteron configuration assesses our capability to cover alternative CPU architectures.

ARM Cortex A15

Figure 6.6: Relative error distribution of the PARSEC benchmarks on the ARM Cortex A15 processor ($P_{idle} = 3.5\ W$).



System configuration

Our last configuration is a Jetson Tegra K1[2] with Linux Ubuntu 14.04 (kernel 3.10). The processor has 4 plus 1 cores on its chip, designed and optimized by NVIDIA. The 4 cores have a standard behavior, while the additional core is designed to be energy efficient. These configurations are exclusive—*i.e.*, we cannot use both together. By default, the 4 cores are enabled.We first use this default behavior for the purpose of validation.

CPU power model

The resulting CPU power model computed from R_4 is composed of 3 HPC events from the PMU arm_ac15 (e_1 = cpu_cycles, e_2 = inst_spec_exec_integer_inst, e_3 = bus_cycles):

$$P_{idle} = 3.5\ W\ ;\ P_{CPU} = \frac{1.18 \cdot e1}{10^9} + \frac{1.26 \cdot e2}{10^{10}} + \frac{1.84 \cdot e3}{10^{11}} \tag{6.4}$$

[2] https://developer.nvidia.com/jetson-tk1

Model accuracy

Figure 6.6 reports on a relative error of 2.70% (0.17 W), on average. This error rate has to be balanced with the low idle consumption of this CPU, compared to previous configurations. Nonetheless, this CPU power model demonstrates that our learning approach can also generate accurate CPU power models for embedded systems, thus going beyond standard server settings.

Synthesis

From all the above observations, we can assess that the generated models perform well on a variety of representative architectures, including Intel, ARM, and AMD. Our solution does not rely on a specific processor extension (*e.g.*„ RAPL) and can use specific workloads during the learning phase to build domain-specific CPU power models. On average, our solution exhibits a relative error of 1.5% (0.8 W), thus clearly outperforming the state of the art.

The closest method to ours, described in [ERK06], adopts a similar approach. Indeed, the authors build a CPU power model for the whole testbed system and use it then inside their software, Mantis, for power estimations. Their power model is composed of 4 metrics, the CPU utilization, the memory accesses count, hard disk data read and written, and network I/O rates, and exhibits an error range between 0% and 15%. However, the key limitations of their solution are that *i).* they use a predefined set of metrics, which can clearly differ between architectures, *ii).* they use an heavy subsystem of power planes to get the power consumption of components for their offline power modeling, and, *iii).* their solution produces only power estimation for components, not at process level.

Feng and Ge [FGC05; Ge+10] describe a solution for computing the power profiles of each component of a subsystem. A component's power profile corresponds to its power footprint over a period of time. Moreover, their solution allows them to get additional insights about the code power consumption. However, this solution tends to be very intrusive by connecting to the hardware pins to collect the power consumption of each component, and they do not propose a proper way to estimate the power consumption of these components.

We strongly believe that our approach is well suited to explore the space of HPC events made available by the CPU and for profiling with accuracy the power consumption drawn by this component.

6.2 Assessing SSD Power Models

6.2.1 Empirical Learning

As demonstrated in Section 4.2.1, our approach for learning SSD power models is able to accurately model the sequential read and write operations. To further demonstrate the validity of our approach in the wild, we firstly check that the learned power models is also able to fit randomized operations. We consequently choose the `filebench`[3] benchmark to perform such operations. The `filebench` benchmark is a flexible file system and storage workload generator that allows to use pre-defined I/O scenarios, such as web, file, or database server, or to write specific ones upon needs. We therefore choose to write specific scenarios in order for randomizing the classical read and write scenarios. While

[3] `https://github.com/filebench/filebench`

running each scenario in isolation during 30 seconds, the CPU and SSD power estimation provided by our power models are collected together with the raw power measurements from PowerSpy.

Figure 6.7: Power consumption of the Intel Xeon E5-2630 host when executing the `filebench` benchmark configured to perform random write and read operations ($P_{idle} = 83\ W$).

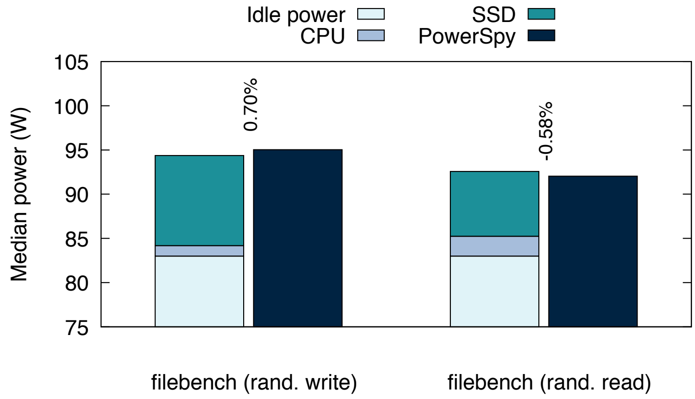


Figure 6.7 shows the result of the workloads executed on the Intel Xeon E5-2630 host (cf. Appendix A) with 1 SSD of 372 GB (Intel SSDSC2BX40). We observe that the power models proposed for the sequential read and write operations are also well-suited to accurately model the same randomized operations with relative errors of 0.58% and 0.70% respectively. While our approach uses the `fio` tool for performing I/O loads, we also demonstrate that it can be used for providing accurate power estimation for another workload. We can also mention the non negligible part of the CPU power consumption while performing I/O operations.

We next assess the previously described power models on 5 benchmarks in comparison to PowerSpy. We consequently choose to use `iozone`,[4] `aio-stress`,[5] `filebench`, `fs_mark`,[6] and `tiobench`[7] for performing various kind of operations, such as sequential read/write, randomized read/write, file allocation/creation/deletion, etc., across a variety of I/O sizes, threads and queue depths.

Figure 6.8 reports the median power consumption compared to the PowerSpy Bluetooth power meter on each benchmark. As can be seen, our approach exhibits less than 1% of relative errors and furthermore validates our empirical learning approach for SSD power models. As previously mentioned, the CPU can have a significant impact on the power consumption while doing I/O activities and nearly represents here 49% for the `filebench` benchmark.

Synthesis

While our learning approach for SSD power models used the `fio` tool for performing sequential I/O operations, we demonstrate in this section that the learned power models

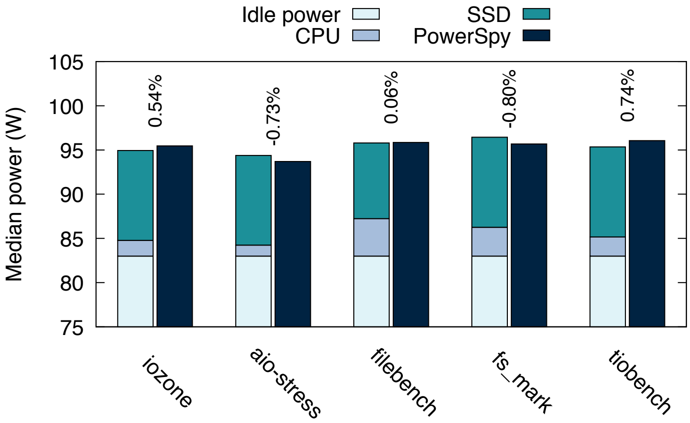
[4] http://www.iozone.org

[5] http://fsbench.filesystems.org/bench/aio-stress.c

[6] https://sourceforge.net/projects/fsmark

[7] https://sourceforge.net/projects/tiobench

Figure 6.8: Power consumption of the Intel Xeon E5-2630 when executing 5 different workloads: `iozone`, `aio-stress`, `filebench`, `fs_mark`, and `tiobench` ($P_{idle} = 83\ W$).



are also able to accurately represent other kinds of operation. We further validate our approach on 5 benchmarks that perform various kinds of I/O activity on the SSD disk and we show that our approach exhibits less than 1% of relative error. Further validations need to be done in order to evaluate the noise introduced by the garbage collector once the SSD is full.

While monitoring SSD power consumption, we prove the necessity of monitoring the CPU component, as I/O operations tend to stress the CPU. Our power models do not rely on specific metrics and are based on 2 generic metrics, the SSD read and write bandwidths, that can be easily computed. This approach can be used to learn any SSD disk power model. Furthermore, if several SSDs are plugged into the system, we can easily extract different power models that can be used at runtime for providing a single or global view of their power consumption.

6.3 Assessing Software-Defined Power Meters

The learning and model generation approaches introduced in Section 4.1 allow to build accurate CPU power models. All the described CPU power models are built to represent the overall power consumption of a node. They can nonetheless be used to produce accurate per-process power estimation when needed, thanks to the different modes exposed by the HPC (cf. Section 6.3.2 and Section 6.3.3). From our CPU power models, we can easily extract the idle power consumption of a node and then show its impact.

In the following sections, we define and study various applicative scenarios and, in this process, attempt to answer specific questions regarding the effectiveness of our approach. We use PowerAPI to build different software-defined power meters based on the CPU power models presented in Section 6.1.2 or especially designed.

For this purpose, we address the following applicative scenarios:

AS1: *Can we build CPU power models that better fit specific domains of applications?*

AS2: *Can we use the derived CPU power models with* PowerAPI *to estimate the power consumption of any workload in real-time?*

AS3: *Can we use the derived CPU power models with* PowerAPI *to estimate the power consumptions of concurrent processes?*

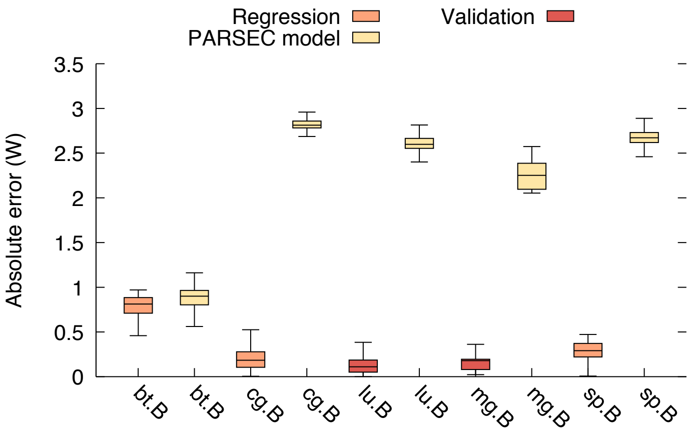
AS4: *Can we adjust the CPU power model depending on an execution profile?*

AS5: *Does the CPU power model depend on the underlying operating system or performance profiles?*

6.3.1 Domain-Specific CPU Power Models

AS1: *Can we build CPU power models that better fit specific domains of applications?*

Figure 6.9: Absolute error distribution of the NPB benchmarks on the ARM Cortex A15 processor by using the PARSEC and NPB power models ($P_{idle} = 3.5\ W$).



In Section 6.1, we identified applications from the PARSEC benchmark suite as representative workloads for characterizing the power consumption of the testbed CPUs. In particular, we focused on delivering generic CPU power models that can estimate the power consumptions of a wide diversity of applications. However, if one knows beforehand that a specific type of workload will be run on a node, our approach can be used to derive domain-specific power models. As an example, we use a set of benchmarks from the well-known NPB suite on the ARM Cortex A15 processor [Bai+91], and derive a new power model specially for this set of applications using the approach described in Section 4.1.2. NPB is designed to take advantage of highly parallel supercomputers and thus the implemented benchmarks represent CPU-intense workloads.

The resulting CPU power model with the lowest average error is composed of 3 HPC events from the PMU arm_ac15: ($e_1 =$ bus_read_access, $e_2 =$ cpu_cycles, $e_3 =$ bus_access):

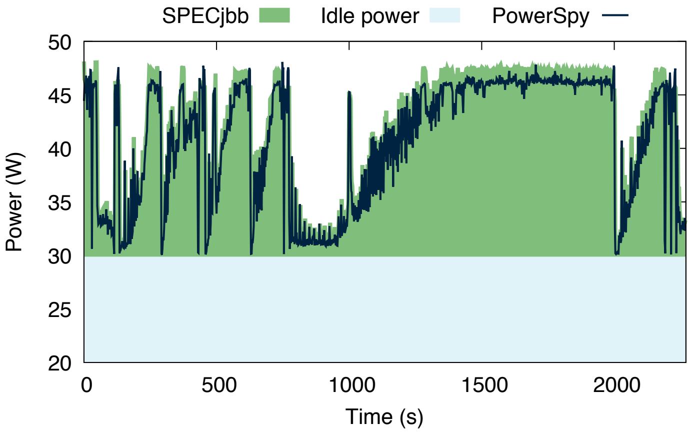
$$P_{idle} = 3.5\ W\ ;\ P_{CPU} = \frac{-1.72 \cdot e1}{10^8} + \frac{1.52 \cdot e2 - 5.08 \cdot e3}{10^9} \tag{6.5}$$

In Figure 6.9, we depict the results and compare them with the original model derived in Section 6.1.2. We can see that a domain-specific model may improve the original one (PARSEC power model) with an average relative error of 4% (0.41 W). In comparison, the PARSEC power model has an average relative error of 20% (2.34 W), which demonstrates the benefits of building domain-specific CPU power models. We are thus able to derive accurate CPU power models with our approach despite the wide diversity of benchmarks. To the best of our knowledge, our solution is the first to be open-source, configurable and directly usable to build efficiently CPU power models without any deep-knowledge of the underlying architecture.

6.3.2 Real-Time Power monitoring

AS2: *Can we use the derived CPU power models with* PowerAPI *to estimate the power consumption of any workload in real-time?*

Figure 6.10: Power estimation delivered by PowerAPI in real-time (4 Hz) for SPECjbb 2013 on the Intel i3 2120 processor ($P_{idle} = 30\ W$).



To further evaluate the applicability of PowerAPI in a real-world and multi-threaded environment, we run the SPECjbb 2013 benchmark [SPE13]. This benchmark implements a supermarket company which handles distributed warehouses, online purchases, as well as high level management operations (data mining). The benchmark is written in Java and consists of *controller* components for managing the applications and *backends* for performing the work. A run takes approximatively 45 minutes; it has varying CPU utilization levels and requires at least 2GB memory per backend to finish properly.

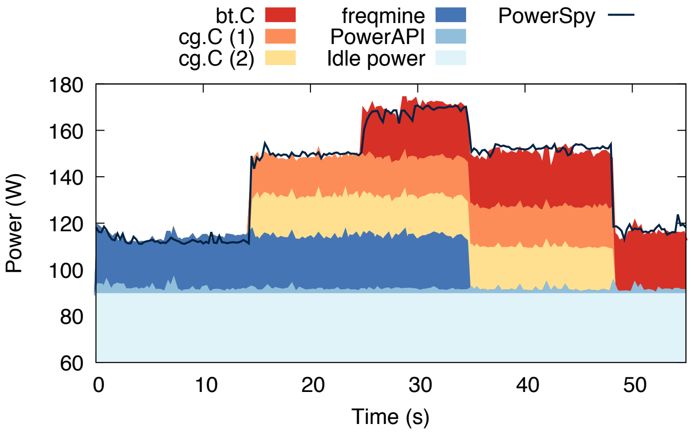
We use the Intel i3 2120 processor for this experiment with the CPU power introduced in Section 6.1.2. Figure 6.10 illustrates the per-process power consumption, focused on the SPECjbb process, compared to physical power measurements. We can see that our system is capable of monitoring varying workloads with an average error of 1.6% (1.70 W).

Regarding the monitoring frequency, PowerAPI is mostly limited by the frequency of hardware and software sensors used to collect runtime metrics. In particular, PowerAPI can report the power consumption of software processes up to 10 Hz when using the `libpfm4` library—*i.e.*, limitation to accurately retrieve raw HPC values. However, by increasing the monitoring frequency, one can observe that the stability of power estimation is affected, that does not help to properly identify the power consumption of processes.

6.3.3 Process-Level Power Monitoring

AS3: *Can we use the derived CPU power models with* PowerAPI *to estimate the power consumptions of concurrent processes?*

Figure 6.11: Process-level power estimation delivered by PowerAPI in real-time (4 Hz) on the Intel Xeon W3520 processor ($P_{idle} = 92\ W$).



PowerAPI is an efficient toolkit that allows to build software-defined power meters in order to perform fine-grained power estimation. We now show that our solution can be used to estimate the power consumption on concurrent processes running on the same CPU. For this experiment, we use the Intel Xeon W3520 processor with the power model detailed in Section 6.1.2.

Figure 6.11 illustrates the ability to estimate with accuracy the power consumption of several processes running concurrently. In particular, it shows the power distribution between the idle power consumption, one benchmark from the PARSEC benchmark suite (`freqmine`), and 2 others from the NPB suite (`bt.C` and `cg.C` configured to run with 2 message passing interface (MPI) processes in this experiment).

Compared to physical power measurements from PowerSpy, when running at a frequency of 4 Hz—*i.e.*, one report every 250 milliseconds, our solution exhibits a relative error of 2% (2.92 W), thus competing with the state-of-the-art solutions [Bir+05; Col+15b; CM05; IM03]. One can also notice the effectiveness of our solution even on the NPB benchmark suite, not used here during the learning phase.

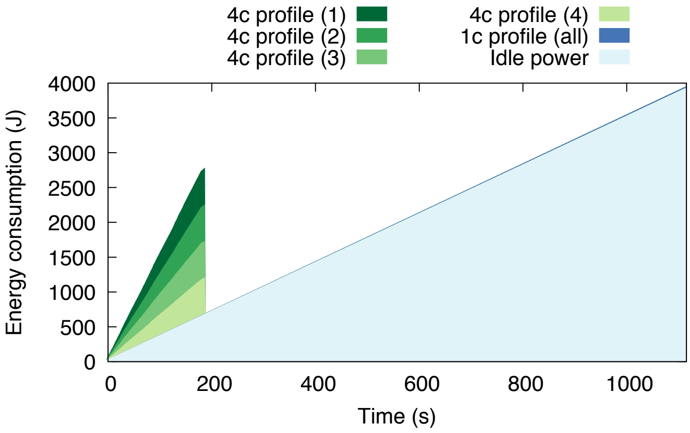
Additionally, Figure 6.11 reports on the power consumption of PowerAPI along its execution. The power consumption of 2 W on average confirms that our implementation of the CPU power model has a reasonable energy footprint and is weakly impacted by the number of processes being monitored. This footprint acknowledges the design and the implementation of PowerAPI as a scalable system toolkit to build efficient software-defined power meters.

6.3.4 Adaptive CPU Power Models

AS4: *Can we adjust the CPU power model depending on an execution profile?*

We use the 4-plus-1-core processor available on the Tegra K1 card developed by NVIDIA. Based on the approach described above, we build different CPU power models in order to model the different modes—*i.e.*, when the 4 cores are enabled, or when the low power core is used. The first CPU power model is described in Section 6.1.2 and represents the processor power consumption when the 4 cores are in action. We are now interested in

Figure 6.12: Energy consumption of the host by using the 4-plus-1 power profiles on the ARM Cortex A15 processor and `cg.b` ($P_{idle} = 3.5\ W$).



modeling the low power core. To trustfully represent the underlying optimizations, we therefore build a separate CPU power model for being able to distinguish the different profiles.

For the low power core, the power model with the lowest absolute error is composed of 3 HPC events from the PMU arm_ac15 (e_1 = cid_write_retired, e_2 = ttbr_write_retired, e_3 = inst_spec_exec_load):

$$P_{idle} = 3.5\ W\ ;\ P_{CPU} = \frac{7.82 \cdot e1 + 4.38 \cdot e2}{10^4} + \frac{3.67 \cdot e3}{10^{10}} \tag{6.6}$$

This model is very different from the one presented in Section 6.1.2 as the number and type of events differ.

To better understand the difference between both profiles, we plot the energy consumption of each profile for the benchmark `cg.B` from NPB spawned on 4 processes in Figure 6.12. The energy consumption is either shared between 4 cores (*4c profiles*) to optimize the performance, or 1 core (*1c profile*) when the low power mode is enabled. One can observe that the 4-cores profile completes 6 times faster by exploiting the parallelism of the underlying architecture, resulting in much lower energy consumption ($\simeq 1\ KJ$). The 1-core profile exhibits a low power consumption, but is penalized by the idle power accumulating over time.

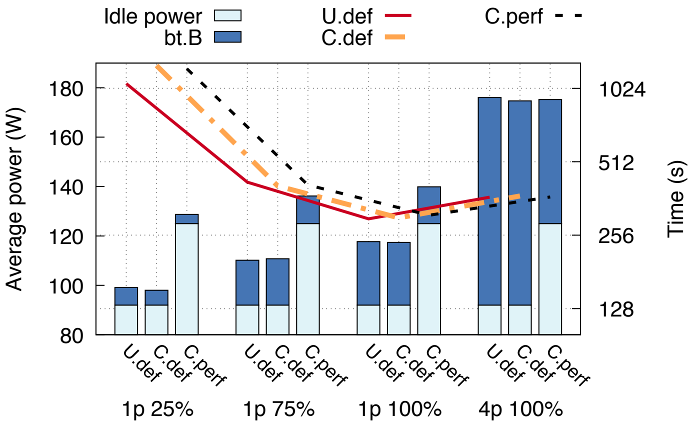
In comparison to existing solutions, these power profiles were derived automatically without a deep expertise of the underlying architectures. Our solution is then able to detect which mode is enabled and to adapt the CPU power model accordingly at runtime for estimating the power consumption of software assets. Our approach therefore captures all the features enabled on the processor to build adjusted CPU power models.

6.3.5 System Impact on CPU Power Models

AS5: *Does the CPU power model depend on the underlying operating system or performance profiles?*

We already showed that the optimizations made at CPU level can have a non-negligible impact on the power consumption. Additionally, several hardware features are controlled

Figure 6.13: Avg. power consumption of the Intel Xeon W3520 in UBUNTU, CENTOS with default settings ($P_{idle} = 92\ W$) and CENTOS with `latency-performance` profile enabled ($P_{idle} = 125\ W$).



by the operating system, such as dynamic voltage/frequency scaling or simultaneous multi-threading.

We now show the power consumption of two popular open-source operating systems: UBUNTU and CENTOS. UBUNTU is known as user-friendly, with a huge community of users and the philosophy of supporting a large variety of devices, from server to mobile. CENTOS is derived from the open-source version of red hat enterprise linux (RHEL) and hence targets productivity systems that require a stable and dependable operating system. Some very useful tools are available on this system for optimizing the hardware and software components.

A version with UBUNTU 14.04 with a Linux kernel 3.13 and a version of CENTOS 7 with a Linux kernel 3.10 were installed on the Intel Xeon W3520 server.

We use the `bt` benchmark from NPB suite for this experiment and we compare 3 CPU power models. One is already described in Section 6.1.2 and represents the default settings without any customization (`U.def`).

The second power model represents the default CPU settings of CENTOS (`C.def`).

The CPU power model is composed of 4 HPC events from the PMU nhm (e_1 = uops_retired:active_cycles, e_2 = uops_issued:any, e_3 = ssex_uops_retired:scalar_single, e_4 = uops_retired:retire_slots):

$$P_{idle} = 92\ W\ ;\ P_{CPU} = \frac{2.02 \cdot e1}{10^8} + \frac{7.76 \cdot e2 + 4.43 \cdot e3 + 2.70 \cdot e4}{10^9} \tag{6.7}$$

The third power model covers the performance optimizations provided by CENTOS (`C.perf`). We use the `tuned-adm` tool for improving performance in specific use cases and for interacting with the power saving mechanisms. This command comes with different tuning server profiles depending to the use of the underlying system and hardware. We use here the `latency-performance` profile, which allows the operating system to reduce drastically the latency of the system and thus to increase the performance. The CPU power model designed for CENTOS with the `latency-performance` profile is composed of 4 HPC events from the PMU nhm (e_1 = l1d_prefetch:triggers, e_2 = uops_decoded_dec0, e_3 = fp_comp_ops_exe:sse_fp_scalar, e_4 = l1i:reads):

$$P_{idle} = 125\ W\ ;\ P_{CPU} = \frac{8.86 \cdot e1}{10^8} + \frac{7.93 \cdot e2 + 6.33 \cdot e3 + 5.38 \cdot e4}{10^9} \tag{6.8}$$

The average power consumption values reported by the 3 CPU power models are shown in Figure 6.13.

In particular, we compare the duration and the power consumption of each profile while changing the utilization ratio of a core and increasing the number of allocated cores. With default settings (`U.def` and `C.def`), the choice between Ubuntu and CentOS does not impact the power consumption and none of them pulls out of the game in terms of execution duration. However, more interesting reports are delivered when the `latency-performance` profile is chosen. Indeed, when one process stresses a full core, the power difference between the default settings (`C.def`) and this profile can be greater than 20 W. The difference is partially due to the idle power consumption that represents a non-negligible part of the power drawn by the system with this profile. Actually, the `latency-performance` profile turns all cores of the processor in the `C0` state, which means that the cores are always turned on for minimizing the latency while waking up.

Moreover, one can see that the activation of the performance profile does not decrease the execution duration of the benchmark. Hence, we can clearly target Ubuntu or CentOS with default settings (`U.def`, `C.def` resp.) to obtain the best compromise between performance and power efficiency.

These experiments show that the optimizations made automatically by the operating system have to be carefully selected since it may cause power losses.

Synthesis

Several applicative scenarios have been reported in this chapter. We first show that universal CPU power models do not exist to accurately represent any domain-specific workloads that have not been included during the learning phase. Thanks to our adaptive learning approach, new power models can be learned to better fit the needs and requirements. We next demonstrate the ability of PowerAPI to use the learned power models for providing accurate per-process power estimation in real-time of a CPU-intense workload that has not been used during the learning phase but with similar characteristics than the input workloads. We go further by showing that our approach is not limited to only one application, but can also be used to monitor several concurrent processes in real-time with a low energy footprint. We illustrate afterwards that different power models can be learned for representing different execution profiles and be automatically switched at runtime to better represent hardware features enabled. We finally show that optimizations made by the OS have an important impact on the power consumption and the resulting power models, but can be easily modeled with our approach for better analyzing what optimizations represent the best compromise between performance and energy efficiency.

Summary

In this chapter, we demonstrate the validity of our approach for learning CPU and SSD power models. Our empirical approach for learning CPU power models exhibits an average accuracy of 97% while our architecture-agnostic approach, 98.5%. As for our empirical learning approach for SSD power models, it exhibits an accuracy greater than 99%. We therefore demonstrate that our approaches are very efficient to accurately model modern and future CPU architectures (Intel, AMD, ARM included), but also recent SSD disks. We finally present PowerAPI as a “swiss army knife” for building software-defined power meters with a low energy footprint ($\simeq 2$ W).

Chapter 7

SaaS-Level Power Estimation

Table of Contents

7.1 Process-Level Power Estimation in VMs

Virtualization offers environment and performance isolation and, hence, is the basis for many data centers and cloud management frameworks. In order to improve their energy efficiency, such cloud management frameworks need to know the resource requirements of the running entities.

For data center providers and users, it is particularly useful to identify which applications are the largest power consumers. However, physical power meters and components with embedded energy sensors are often missing, and they require significant investments and efforts to be deployed a posteriori in data centers. Moreover, these hardware facilities usually only provide system-level or device-level granularities. Hence, software-based power estimation is becoming an economical alternative [ODL14]. Power estimation is relatively accurate when one has full control over the underlying hardware and detailed knowledge of its properties. It typically works by sampling the activity of applications and measuring the power consumption of the whole system using hardware-specific probes.

In virtualized environments, one does not have direct access to the physical CPUs and one can only observe the processor emulated by the virtual machine's hypervisor. Furthermore, the physical resources available to the emulated CPU may change dynamically as a result of VM scheduling—a VM may run alone on some physical core(s) for some time and later compete with other VMs—or even migrate to another host.

Current approaches providing power estimation remain poorly adapted to virtualized environments and do not provide acceptable measures. The few existing approaches either consider the VM as a black-box running a single application [Kan+10; KVN10], or they require specific extensions to the hypervisor or to the host and guest operating systems for being operational [She+13; SLB07].

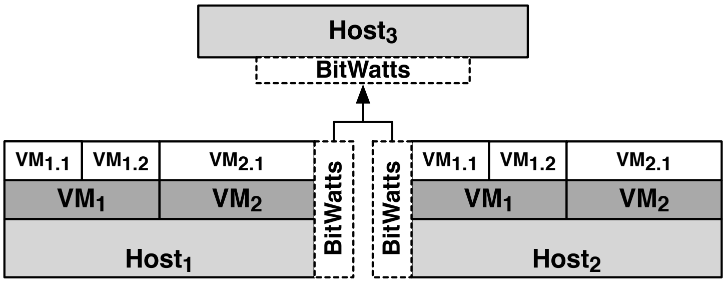
The introduction of fined-grained power monitoring within virtualized environments opens up for new scenarios.

Platform-as-a-Service (PaaS) infrastructures such as Google App Engine allow developers to create programs that run in sandbox mode [San09]. Request and database handling is performed outside of an application in separate tasks. To isolate which application draws the most power, it is necessary to cover each individual process. This does not only allow for new power-aware pricing models [KOS16], but also helps to improve energy proportional mechanisms.

In cases of dedicated cloud offers[1] or nested virtualization, such as proposed by Ben-Yehuda et al. [Ben+10], an Infrastructure-as-a-Service (IaaS) provider could offer user-controlled hypervisors within a VM. This allows cloud users to run their favorite types of hypervisors and VMs. However, the management of VMs in such environments can become deeply complex and, with current solutions, makes impossible to monitor the power consumption of a single VM at the highest level of nesting and thus prevents typical tasks, such as resource and power provisioning. Such use cases therefore require a flexible solution that can operate on local, nested, and distributed levels without extra efforts.

More specifically, consider a distributed setup with nested virtualization in which we would like to track the power consumption per VM and per user in order to apply power-aware pricing. Such setup is illustrated in Figure 7.1.

Figure 7.1: Example for BitWatts acting in a multi-tenant virtual environment.



One VM per user can be initially started on each node, and the user can subsequently launch additional VMs running multiple processes within the provided environment. In such settings, it is desirable to be able to monitor the power consumption of each of the user's processes and VMs separately. Furthermore, as a user might operate on multiple

[1] https://www.ovh.com/ca/en/dedicated-cloud

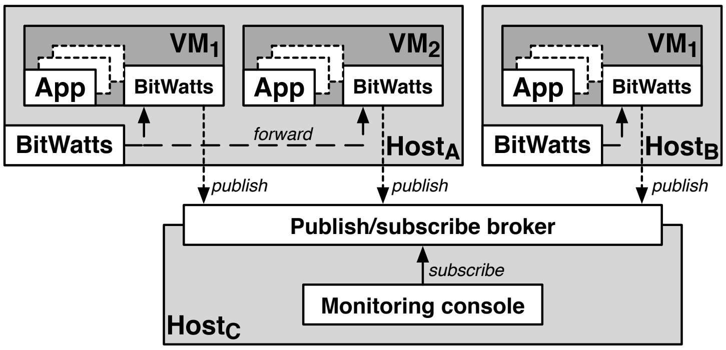
nodes, distributed monitoring of the energy consumption of all his processes is rather a new challenge and begins to be addressed.

In this section, we describe our middleware toolkit, BitWatts, for providing such facilities in Section 7.1.1. In Section 7.1.2, we present our communication protocol for exchanging data between a host and its VMs. Given the new level of abstraction, we propose a new virtual CPU power model for VMs in Section 7.1.3. We then characterize our experimental setup in Section 7.1.4 and show the effectiveness of BitWatts, and the virtual CPU power model, in Section 7.1.5 and Section 7.1.6.

7.1.1 BitWatts, Middleware Toolkit for VMs

Power estimation of processes running in virtualized environment is not a trivial task, since several factors have to be considered. In particular assumption, such as the presence of a single application running in a single VM on a single core, do not hold anymore. One has to deal with complex scenarios with a number of VMs that may exceed the number of physical cores and several applications that run within each VM. To cope with these different dimensions of scaling, we designed and implemented the BitWatts middleware framework on top of PowerAPI. BitWatts relies on a multi-tier architecture, depicted in Figure 7.2, that shares the power consumption of the VMs running on the host to application processes running within the VM. Since the VM does not have direct access to the hardware, we use a fast communication interface to connect instances of BitWatts running on the host and in the VMs. Similarly, BitWatts also supports communication across machines using publish/subscribe communication channels to report consolidated power estimation of distributed applications spanning multiple nodes (*e.g.*, in a cluster).

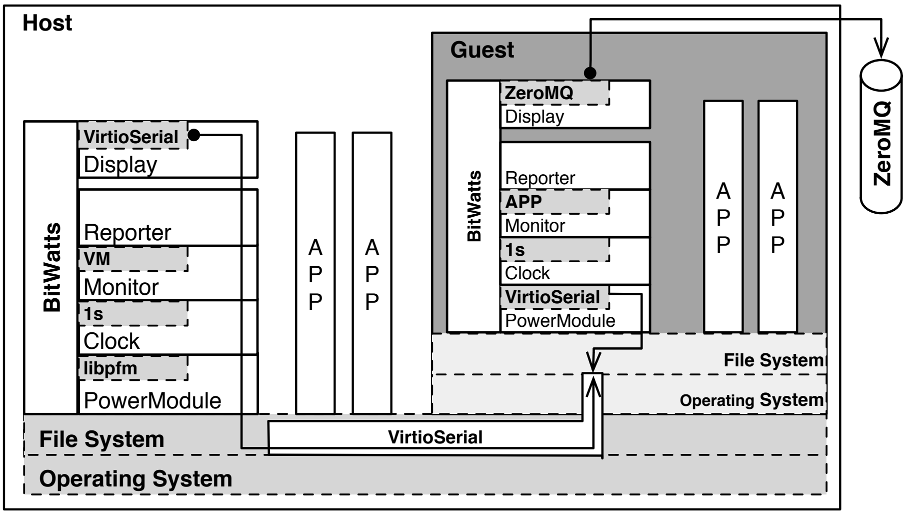
Figure 7.2: BitWatts middleware overview.



Our solution builds on PowerAPI, which adopts the actor programming model as a solution that can scale with the frequency and the number of applications to be monitored. As the BitWatts middleware framework supports process estimation in VM-based systems, implementations of the *Sensor*, *Formula*, and *Reporter* actors are assembled in different configurations on the hosts as well in the VMs (see Figure 7.3).

Additionally, to improve the accuracy of the state-of-the-art power estimation, we use also a power model that uses `libpfm4` on the host to collect raw HPC values associated to the monitored VM process. The formula consumes the measurements collected on the host by this `libpfm` sensor to estimate the power consumption of the VM process. The resulting power estimation are next automatically published by 2 reporter actors through 2 different communication channels: VirtioSerial[2] and in a distributed setup also with ZeroMQ.[3] The data forwarded through these channels is consumed by *Sensor* actors within the VM.

Figure 7.3: BitWatts middleware implementation.



7.1.2 Power Consumption Communication Channels

Exchanging data between instances of BitWatts requires 2 levels of communication. Firstly, we need to exchange data between the host and the VM for estimating the power consumption of a process within the VM. Secondly, in a distributed setup, we want BitWatts to report the power estimation to another server, *e.g.*, to aggregate the data monitored on multiple physical or virtual nodes.

For the hierarchical communication between instances of BitWatts running on the host and a VM, a lightweight transport mechanism is required to exchange messages at high rate while crossing the VM boundaries. As we also want to choose the communication medium with the higher bandwidth, we therefore decide de compare the traditional Socket implementation with VirtioSerial. To assess the performance, we transfer a large file from a host to a VM on the Intel i3 2120 server.

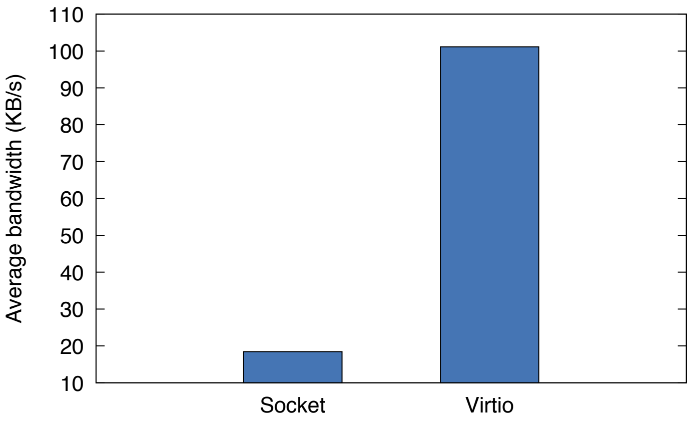
The results are described in Figure 7.4. Socket communication has a maximum bandwidth of 18 KB/s approximatively whereas VirtioSerial reaches more than 100 KB/s (5 times higher). These results are expected because VirtioSerial is directly based on the file system and has been developed for the very purpose of inter-VM communication, while Socket has to traverse the whole network stack. We choose consequently VirtioSerial as a transport mechanism between a host and VMs and provides thus the performance required to reduce the likelihood of synchronization errors of power measurements between host and VM.

The VirtioSerial communication channel is implemented in BitWatts as a reporter component on the host and is included inside a *Sensor* component inside the VM (see Figure 7.3).

[2] http://www.fedoraproject.org/wiki/Features/VirtioSerial

[3] http://www.zeromq.org

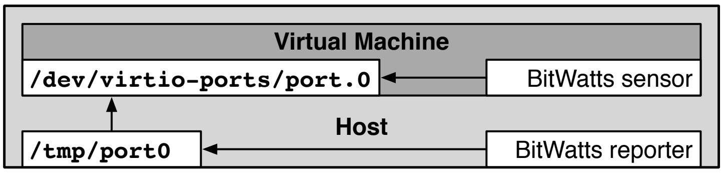
Figure 7.4: Average bandwidth (KB/s) for communication using Socket and VirtioSerial.



Multiple instances of BitWatts are running concurrently: one in the host, and one per VM. For the host, the VirtioSerial reporter communicates the power consumption of the VM process to the `virtio-pci` device. In the VM, the VirtioSerial sensor connects to the VirtioSerial port and reads power consumption reported by the host through a file (detailed in Figure 7.5). It also monitors the utilization of the application under observation and of all processes running in the VM. The BitWatts formula (cf. Section 7.1.3) uses these values to compute the per process-level power consumption within the VM and forwards the result to a reporter.

In a distributed setup, we need to communicate across machines, typically to aggregate the power measurements from distributed application components running on different VMs and hosts. Our distributed communication channel therefore consists of a publish/subscribe system using ZeroMQ. ZeroMQ is a networking API that supports complex messaging patterns and provides bindings for various programming languages while being lightweight. The key component of the publish/subscribe system is the broker. It forwards messages received from the distributed BitWatts instances to interested subscribers, for example loggers or the monitoring console (see Figure 7.2). Messages exchanged between BitWatts, the broker, and the subscribers are serialized using Apache Thrift,[4] an efficient interface definition language and binary serialization protocol.

Figure 7.5: `virtio-pci` interface in action.



7.1.3 Virtual CPU Power Model

Unlike the architectures observed at the host level, virtual CPUs tend to be simpler: they map physical cores to logical processors (sockets) and typically do not support any

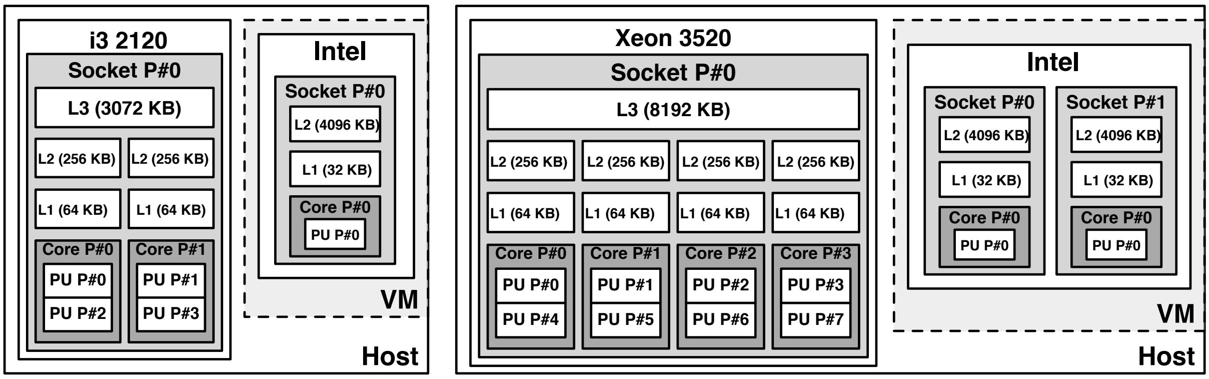
[4] `http://thrift.apache.org`

dynamic voltage/frequency scaling, *simultaneous multi-threading*, or *dynamic overclocking* features, as illustrated in Figure 7.6. Hence, when pinning a single-core VM on a physical core of the host, the power consumption of a process running in the VM is proportional to the CPU utilization of the VM on the host.

To estimate the power consumption of an application running in the VM $P_{vm}(app)$, we need therefore to know the consumption of the VM process $P_{CPU}(vm)$ on the host machine, as well as the CPU utilization of the application $U_{vm}(app)$ relatively to the other applications running in the VM $U_{vm}(total)$:

$$P_{vm}(app) = P_{cpu}(f, uc_{vm}^{1}...uc_{vm}^{N}) \cdot \frac{U_{vm}(app)}{U_{vm}(total)}. \tag{7.1}$$

Figure 7.6: Intel i3 2120 and Intel Xeon W3520 VM topologies.



7.1.4 Experimental Setup

The experimental setup consists of 2 types of servers (i3 and Xeon) with different hardware characteristics, as shown in Appendix A. For the distributed setups, we use 3 identical servers of type i3.

We rely on `kvm` [Kiv+07] for virtualization. `kvm` turns the Linux kernel into a hypervisor without need of additional software. In addition to the typical process operating modes (kernel space, user space) of Linux, `kvm` adds a *guest mode* for programs running in a virtualized environment. This feature helps for measuring the CPU time used by a virtual process.

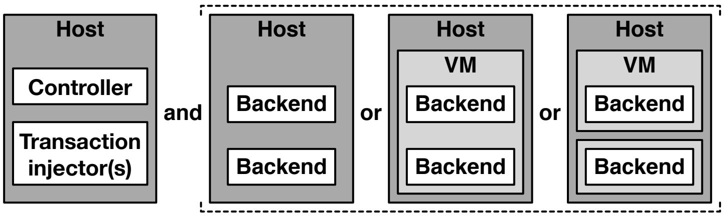
As `kvm` does not perform any emulation to operating systems on various architectures, we combine it with `QEMU`[5] to emulate different CPU and device types. With `kvm/QEMU`, the VM runs as a normal user process and is hence controlled by the Linux scheduler. By default, the scheduler tries to keep a process on the same CPUs, notably to maximize cache efficiency. We run `kvm/QEMU` with an off-the-shelf Ubuntu 13.11 on both server types (i3 and Xeon).

We want to investigate in our experiments the accuracy and applicability of BitWatts at different scales. Therefore, we first consider the execution of benchmarks on a single host, with an increasing number of concurrently running VMs, for observing the impact of VM scheduling on the host. As a first benchmark, we use PARSEC for our experiments, as it is multi-threaded and CPU-intensive. PARSEC contains a variety of applications

[5] `http://www.qemu.org`

implemented in C. We experiment with all except 2 (`raytrace`, `ferret`) that were not readily supported by our hosts. We use the PARSEC *native* workload as it yields sufficiently long execution times. We allocate 2 threads per VM, thus allowing the execution of 4 concurrent VMs on the Intel Xeon W3520 server.

Figure 7.7: Possible setup of SPECjbb (only backends are part of the evaluation).

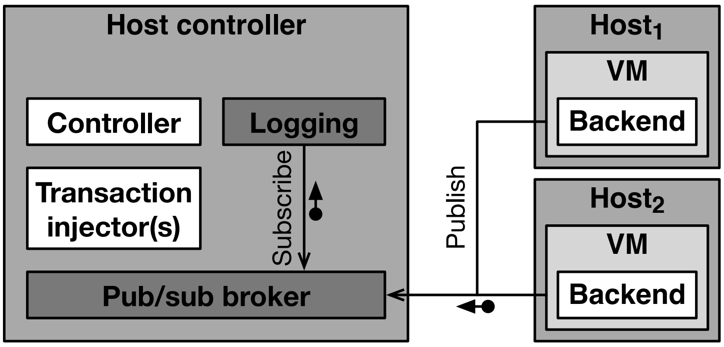


Then, to further evaluate BitWatts in a real-world, multi-threaded and distributed environment, we use the SPECjbb 2013 benchmark. In our experiments, we focus on evaluating the power consumption of the backends, since they can be scaled arbitrarily in virtualized environments. A run takes approximatively 45 minutes.

In order to have more than one backend run on our instances of i3, we apply the following parameter changes to the `specjbb2013.conf` file: we reduce the number of customers and products to 50,000, increase the step-size, and reduce the maximum and minimum duration for the phase 2 of the benchmark.[6]

Since we only have several identical servers of type i3, the SPECjbb experiments are only executed on these machines. We compare different setups, running on 1 or 2 backends on the host or in a VM (cf. Figure 7.7). The distributed setup consists of a controller host and 2 virtualized or non-virtualized backend hosts (cf. Figure 7.8). Note that in virtualized scenarios one BitWatts instance runs on the host and another one in the VM.

Figure 7.8: Distributed SPECjbb setup.



7.1.5 Scaling the Number of VMs

We already assessed our power models on a host machine in Section 6.1, by comparing the power estimation of PARSEC to the values reported by PowerSpy. We first evaluate

[6] Note that these changes make our runs non-compliant, therefore we do not use the SPEC-specific metrics.

the power model, described in Section 7.1.3, by comparing the BitWatts estimation of PARSEC running in the VM to the values reported by PowerSpy on the host. In this experiment, PARSEC will run inside a single VM, which takes 2 cores on the host. As the activity of the other active processes is comparably negligible, we compute the BitWatts estimation as the sum of the power estimation in the VM with the idle power ($P_{idle}(f)$) of the corresponding host machine.

Figure 7.9: Power consumption of the host when scaling PARSEC on multiple VMs.

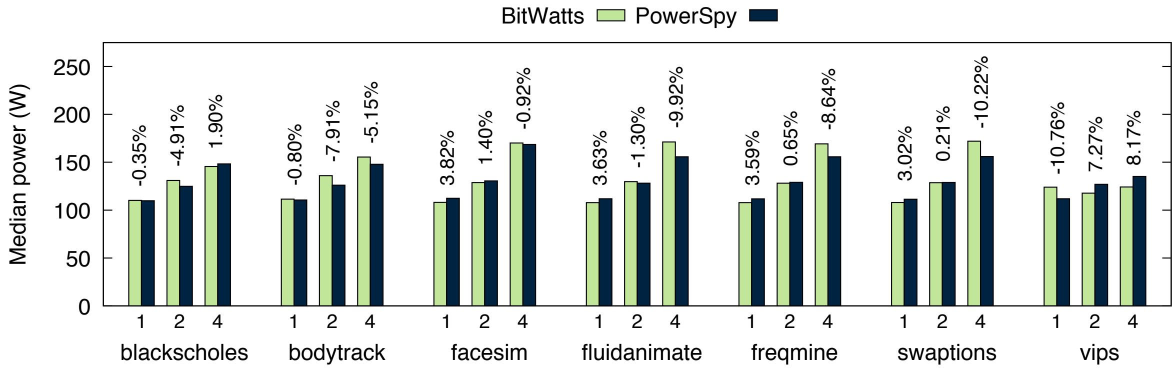


Figure 7.9 reports on the median power error observed between BitWatts and PowerSpy. The overall PARSEC experiment resulted in roughly 10,000 power values with a runtime of 1 hour per VM. Note that we did not pin the VM to any specific cores on the host, hence we rely on the native `kvm` scheduling. When running a single VM on the host, power estimation within the VM by BitWatts has a similar precision to that on the host (see Section 6.1.1 and Section 6.1.2). This measure assesses that the CPU power models we propose properly captures the *guest mode* used by `kvm` to execute the VMs on the host.

Then, given that nowadays VM-based systems tend to be consolidated to minimize the number of active hosts [KF14], we evaluate the precision of our software-defined power meter when scaling the number of VMs to be executed on the host. For each of the available PARSEC benchmarks, we evaluate the median power error when scaling the number of VMs from 1 to 4 on the Xeon server. As we do not try measure the side effects of host over-provisioning on power, we do not exceed the number of physical cores available on the host. The relative error reported in Figure 7.9 spans from less than 1% (`fluidanimate`) up to around 10% (`swaptions`) with increasing errors if the cores used by the VMs reach the number of physical cores on the host. In comparison to existing solutions like VMeter [BC10], we are not only able to report the whole power consumptions of VMs when multiple VMs are running, but the per-process power consumption inside them. This experiment demonstrates that the virtual CPU power model introduced in Section 7.1.3 holds in virtual environments, given the simplified architecture of the virtual processor exposed by the hypervisor (see Figure 7.6).

7.1.6 Scaling the Number of Hosts

In this section, we evaluate the power consumption of a real-world application (SPECjbb) using BitWatts. In particular, we further show the possibility of estimating workloads on several nodes such as commonly used in cloud environments.

Table 7.1 summarizes the experiments we performed using 1 or 2 instances of the SPECjbb backend. The controller runs on a separate host and is not part of our evaluations (see Figure 7.7). We use `taskset` tool to control the CPU affinity of the multi-threaded backend, which we pin to 2 physical threads in the execution.

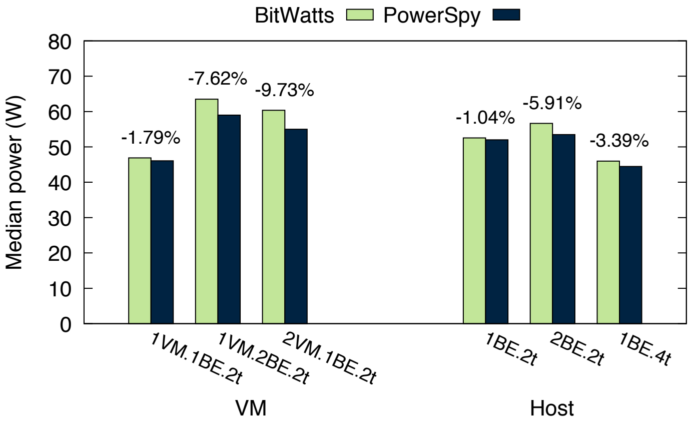
Table 7.1: Experiments performed using SPECjbb (`BE`: backend, `VM`: virtual machine, `t`: threads).

Name	Description
Host	
`1BE.2t`	1 backend pinned to 2 threads
`2BE.2t`	2 backends, each pinned to 2 threads
`1BE.4t`	1 backend with 4 threads
VM	
`1BE.1VM.2t`	1 backend, 2 threads, 1 VM
`1BE.2VM.2t`	1 backend, 2 threads, 2 VMs
`2BE.1VM.2t`	2 backends, each 2 threads, 1 VM
Distributed	
`1BE.4t`	2 hosts, 1 backend, 4 threads
`1BE.1VM.2t`	2 hosts, 1 backend, 2 threads, 1 VM

As a comparison, we also run a non-pinned version of the backend on the host (using all available threads) to the difference in resource utilization. 2 dedicated threads are assigned to each VM.

Single node setup. Applications are usually evaluated in isolated runs. However, due to resource sharing, process-level power estimation becomes more difficult. We further investigate the impact of virtualization as well as interference of concurrently running applications, first on the host and then in virtual machines.

Figure 7.10: Median power consumption for SPECjbb on an Intel i3 2120 server with different resources assigned to a single or multiple VMs on one host.

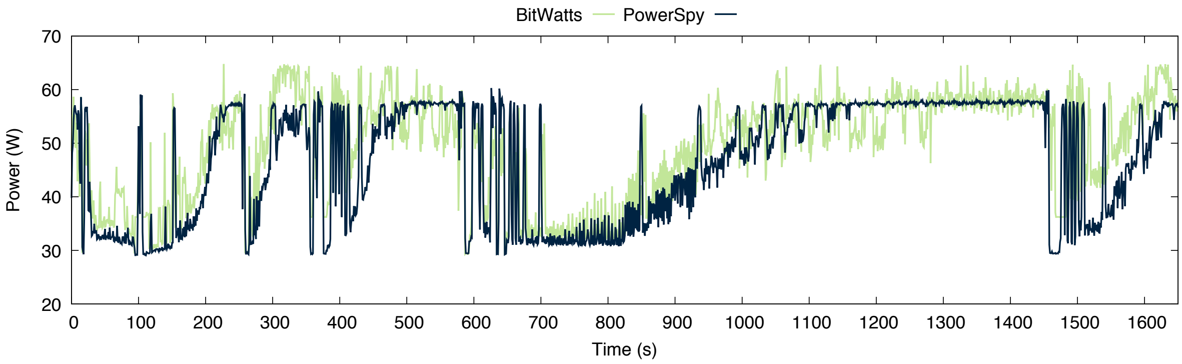


In Figure 7.10, we report on the median power consumption of the overall SPECjbb run and the median relative error compared to PowerSpy.

On the host, we run once a backend with all available threads and once pinned to 2 threads to ensure that only some of the CPU cores are used. We can see that the accuracy is not influenced if only a part of the CPU is dedicated to a process. In this experiment, we additionally show that we can monitor 2 processes at the same time, when running on the host as well as within the VM. Note that we are monitoring both processes separately and only sum up the process power consumptions for comparing to PowerSpy power measurements. As HPC interfere when more than one process is running, the isolation of the power consumption for each of the process is harder. This is also reflected by the increasing median error if we monitor more than one process at the same time (*e.g.*, when we run 2 backends on the host or within 1 or 2 VMs).

In the case of the host running only a single backend, we are underestimating the high-load phases (as it can be seen in Figure 7.11). In general, however, the estimation error is below 10% and one can mention that the 2 curves follow exactly the same trend.

Figure 7.11: Power consumption during the execution of SPECjbb on the Intel i3 2120 processor with 2 threads.



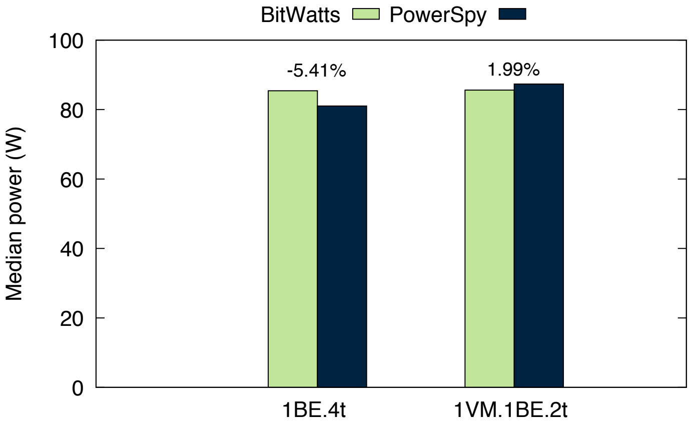
BitWatts can therefore also estimate real-world applications with load variations and sub-system scenarios when only parts of the CPU are used. We can further observe that virtualization does not cause power consumption overhead, as can be seen in the single VM run with 2 backends and the 2 VMs run with one backend each. `kvm` is hence very power efficient. We can finally see that the backend can use the available resources more efficiently when it has all threads available (see `1BE.4t` vs. `1.BE.2t`) because the highest possible throughput in the workloads is reached faster than when the backend has limited resources.

Distributed setup. Placing application components in different VMs allows us to execute across multiple hosts. We consequently extend our experiments to a distributed setup, showing that BitWatts can be applied in realistic data center settings. Experiments have been executed on 3 i3 identical servers as shown in Figure 7.8.

We first run 1 backend on each host, once with 4 available threads, using BitWatts. We also execute 1 backend on 2 hosts, each with a VM and 2 threads. The reporting interval to the broker is 1 second. Based on our observations, the contribution of the network interface to the power consumption is very low and is mainly bound to the CPU activity for sending data. Furthermore, the impact of disk accesses is not covered by the SPECjbb benchmark. At the broker, the values are aggregated and forwarded to the logger that sums the results and writes them to a file.

The results are shown in Figure 7.12.

Figure 7.12: Median power consumption for SPECjbb on Intel i3 2120 servers for a distributed setup, virtualized and non-virtualized.



As expected, the absolute power consumption increases with running on 2 hosts because we have to account for both idle values. The median error, however, does not increase as there are instances of BitWatts on each server and they report the local values to a broker. Furthermore, the single VM experiment shows high accuracy, although underestimating the power consumption. Overall, results are comparable to a single host experiment.

Summary

We showed that BitWatts performs well in various situations, notably when scaling VMs and in distributed environments. Power consumption tend to be application dependent. This is the reason why developers start to consider the potential energy footprint of their software. Since the trend is to run software not only locally, but also in data centers and clouds, additional levels of abstraction have to be considered. Based on our architecture—and application—agnostic power models that supports the power-aware features of modern processors, we deliver a middleware toolkit, BitWatts, to estimate the power consumption of distributed and virtualized setups, which are commonly used in cloud environments. We also demonstrated that our solution is accurate in most cases, even when compared with native power measurements from PowerSpy or RAPL.

7.2 Software-Defined Power Monitoring of Distributed Systems

The design of energy-efficient distributed systems is a challenging task, which requires software engineers to consider all the layers of a system, from hardware to software. While the state-of-the-art in green computing proposes solutions to increase energy efficiency at all levels, from applications over run-time to hardware [ODL14], it remains difficult to evaluate the power consumption of a distributed system *i)* spanning several—potentially heterogeneous—nodes and *ii)* composing several distributed algorithms and/or protocols. In this domain, PDUs are often shared amongst nodes to deliver aggregated power consumption reports, in the range of hours or minutes. However, in order to improve the energy efficiency of distributed systems, one needs to offer novel power monitoring solutions that go beyond the node's granularity and at a higher frequency, therefore surpassing the actual capabilities of PDUs [Tan+15].

To build such distributed power monitoring solutions, or software-defined power meters, the CPU—considered as the major power consumer [ERK06; NRS15] within a node—requires to be accurately modeled for capturing the activity of a distributed service. Designing power models that can accurately cover the power-aware features of a CPU (*e.g.*, multi-threading, frequency scaling) is a complex task. We therefore build on the expertise we developed in Chapter 4 to automatically learn the power model of the nodes supporting the execution of a distributed system. The resulting power models are then used to build a software-defined power meter, named WattsKit, which can report on the power consumption of complex distributed systems by aggregating and processing in real-time the power measurements collected from multiple hosting nodes.

As a matter of validation, we illustrate the benefits of WattsKit on the monitoring and the analysis of the power consumption of a distributed system stack composed of Docker Swarm, Weave, Elasticsearch, and Apache ZooKeeper, which we deploy in a cluster of 6 nodes grouping 3 generations of CPUs (cf. Appendix A).

The remainder of this section is organized as follows. Section 7.2.1 introduces the experimental case study of this section. Section 7.2.2 details our approach for enabling a service-level power monitoring, while Section 7.2.3 describes the service-level power model we proposed. Section 7.2.4 depicts the general architecture overview of our software-defined power meter for services, WattsKit. We finally revisits the initial case study with WattsKit in Section 7.2.5 and Section 7.2.6 by offering new perspectives on the power consumption of distributed systems.

7.2.1 Case Study

Distributed systems are generally composed of several protocols and algorithms to implement the various services that are required across the system. In this chapter, we study a distributed system stack composed of legacy distributed services, which are widely deployed nowadays. In particular, we consider an instance of distributed search engine based on Elasticsearch[7], which is—at the time of writing this manuscript—the most popular enterprise full-text search engine with an HTTP web interface and schema-free JSON documents.[8]

More specifically, Elasticsearch builds on Apache ZooKeeper[9] to implement the discovery and coordination services, which are required to operate in a distributed configuration. The deployment of instances of Elasticsearch onto nodes is achieved by Docker Swarm, which can be considered as the *de facto* standard for building a cluster of Docker hosts.[10] Both Elasticsearch and ZooKeeper are therefore packaged as Docker containers and we use Weave to network and manage their network configuration.[11] Figure 7.13 summarizes the deployment of this distributed system on 6 hosts composed of 1 master node (Intel Xeon W3520) and 5 slaves nodes (2 Intel Xeon W3520, 1 Intel Core2 Q6600, and 2 Intel Core2 E8400) described in Appendix A.

[7] `https://www.elastic.co`

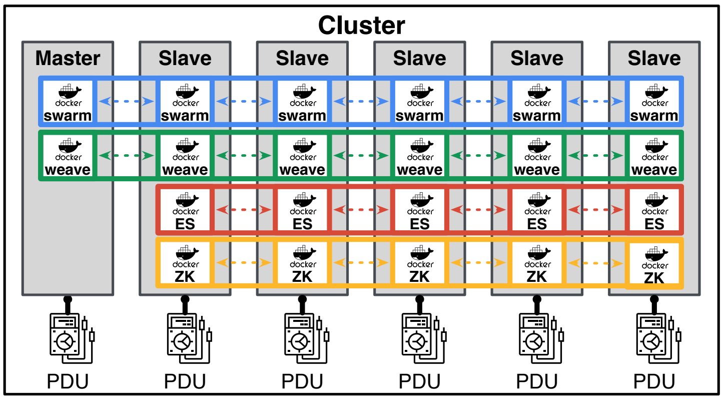
[8] `http://db-engines.com/en/system/Elasticsearch`

[9] `https://zookeeper.apache.org`

[10] `https://docs.docker.com/swarm/overview`

[11] `https://www.weave.works`

Figure 7.13: Overview of the distributed search engine based on ELASTICSEARCH.



Given the distributed nature of each of these services (ELASTICSEARCH, ZOOKEEPER, SWARM, WEAVE) and their entanglement due to respective dependencies, monitoring and analyzing the power consumption of individual services is a particularly tedious task. For example, Figure 7.14 illustrates the measurements reported by a PDU physically connected to each node of the cluster (cf. Figure 7.13). In this experiment, we deploy and sequentially stress each of these services by running the ZOOKEEPER benchmark[12] and Yahoo! Cloud Serving Benchmark (YCSB)[13] [Coo+10] while logging the power consumption per node reported by the associated PDU. In particular, we run the *update heavy workload* (Workload A) of YCSB, which has a mix of 50/50 reads and writes. An application example is a session store recording recent actions. We complete the scenario by killing sequentially each node of the cluster to observe the impact of nodes' leaves on the distributed system's behavior.

If one can observe some variations in the power consumption of individual nodes, it remains difficult to analyze how this power consumption is distributed across services (and not hosts). Furthermore, the heterogeneity of nodes (Intel Xeon W3520, Intel Core2 Q6600 and E8400), which is the rule in modern production systems, complicates the power analysis due to the diversity in idle powers and CPU power features (HT, TB, etc.). One therefore needs to manually tag the nodes to services and to ideally find the relevant scenarios that isolate the execution of services in order to obtain a better insight on their individual power consumption, in order to identify potential energy leaks or optimize the whole system's configuration.

We therefore introduce WATTSKIT as a solution to this key limitation and we propose in particular to introduce a modular approach to monitor—in real-time—the power consumption of all the services involved in a distributed system. In the following sections, we first define and assess a service-level power model before revisiting the above case study with our solution.

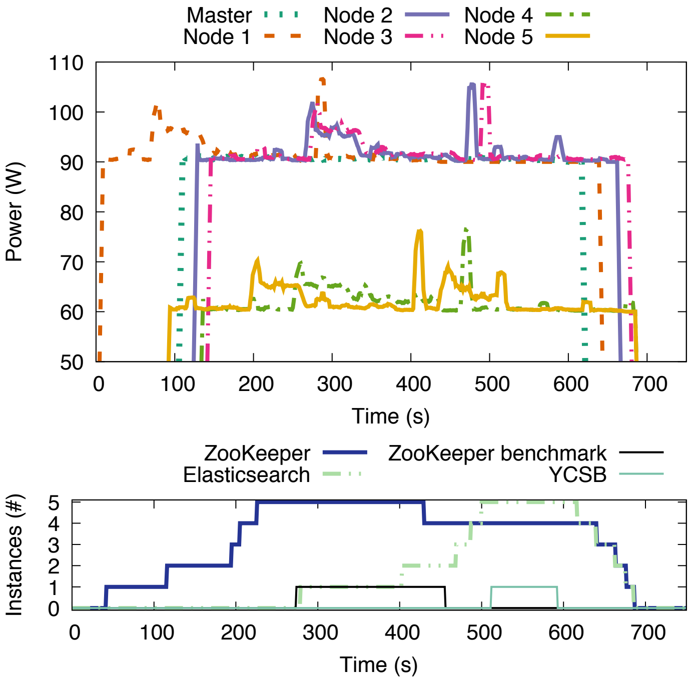
7.2.2 Enabling Service-Level Power Monitoring

To deliver service-level power measurements, our approach consists in tracking the power consumption—per node—of the system processes associated to the services of a given distributed system before aggregating these power measurements at the scale of the cluster.

[12] https://github.com/brownsys/zookeeper-benchmark

[13] https://research.yahoo.com/news/yahoo-cloud-serving-benchmark

Figure 7.14: Power consumption of the distributed search engine based on Elasticsearch.



Achieving such process-level power measurements therefore requires a software-defined power meter as physical power meters are limited to the boundaries of nodes and hardware components. For example, Intel's RAPL fails to support this process-level granularity.[14]

Given the diversity of nodes and services, we further decide to extend the empirical learning approach described in Section 4.1.1 for handling the service-level granularity. Unlike the state-of-the-art in this domain [ZA12], our power model is service-agnostic, which means that it can be used to track a wide diversity of distributed systems. Once defined, this power model can therefore be used in production by WattsKit to monitor the power consumption of the individual services composing the distributed system in real-time.

To build this service-level power model, we adopt a bottom-up approach, therefore estimating the power consumption of the instances of the services running on the hosting nodes, before aggregating them into a service-level power model. Regarding network-intensive workloads, we have previously demonstrated in [NRS15] that the power consumption of network-intensive systems were dominated by the activity of the CPU spent on I/O operations. By carefully modeling such I/O operations, we are able to deliver accurate estimations of both memory-intensive and network-intensive workloads (cf. Section 6.1.1).

7.2.3 To a Service-Level Power Model

As described in Section 4.1.1, we are able to first model the power consumption of a node as the sum of its idle power consumption and the consumptions of its individual processes that are running on it. Given the power aware features already available in modern processors, we demonstrate in Section 6.1.1 that our approach is able to accurately estimate the dynamic power consumption of a node by using HPCs as input metrics for

[14] https://01.org/rapl-power-meter

the power models. The empirical learning approach is therefore used to generate the power models per type of node, resulting in 3 different configurations (cf. Appendix A) for our case study.

Once the node-level power models are inferred and deployed, we can define the service-level power model as:

$$P_{service}(s) = \sum_{n \in N(s)} \sum_{p \in P_n(s)} P_n^{dyn}(p) \tag{7.2}$$

where $P_{service}(s)$ aggregates all the power measurements for each instance p of the service s running on the set of hosting nodes n.

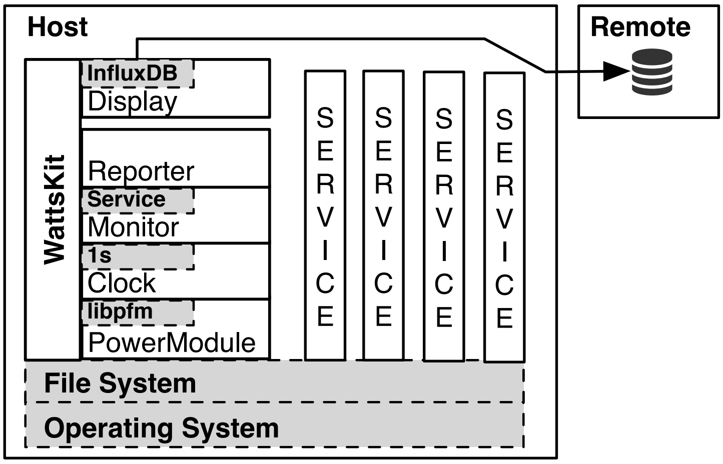
The learning phase is an offline process that is realized when a new node is deployed into the cluster. This phase is therefore the only one where a PDU is required to be connected to the node since the resulting power models are intended to be integrated into the instances of software-defined power meters in order to offer a finer monitoring granularity than physical PDUs.

The following sections describe the integration and the assessment of this service-level power model within WattsKit.

7.2.4 WattsKit, a Software-Defined Power Meter for Distributed Services

We build WattsKit on top of PowerAPI and we pair the libpfm module (cf. Section 5.3) with a remote instance of an InfluxDB database, as described in Figure 7.15.

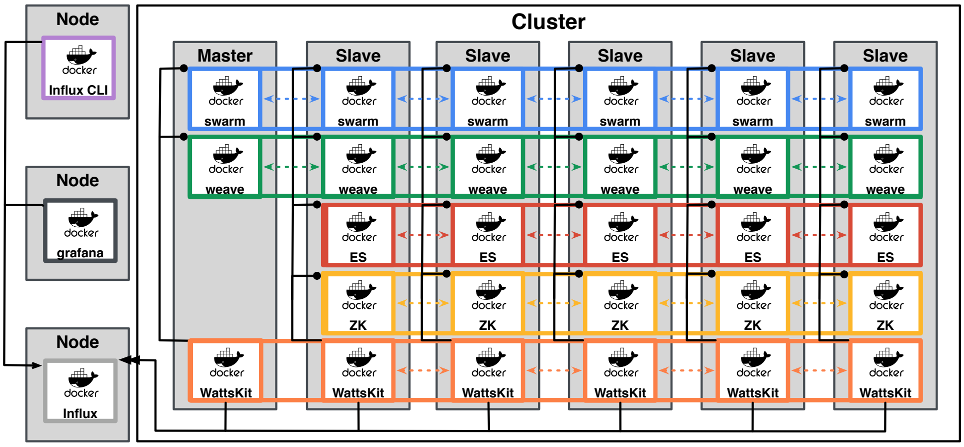
Figure 7.15: Software-defined power meter built with WattsKit.



In this section, we revisit the case study introduced in Section 7.2.1 to offer a new perspective on the power consumption of the distributed services deployer as part of this system, thus overcoming the limitations previously described.

As part of this validation, we deploy WattsKit as a Docker container, which runs along the other services we previously deployed as containers (cf. Figure 7.16). This configuration only differs from Figure 7.13 by unplugging the physical PDUs, which are replaced by the WattsKit containers running on each node and an instance of the InfluxDB time-series database[15] running on a third-party node.[16] WattsKit is configured to automatically monitor all the containers deployed within a node with a sampling frequency of 1 Hz. While WattsKit can use Swarm and ZooKeeper to coordinate the deployment and the execution of software-defined power meters on the nodes, we decided to disable these features to avoid any side-effect on the power consumption analysis of these distributed services.

Figure 7.16: Overview of the experimental deployment of WattsKit.



All the power measurements aggregated by the instance of InfluxDB can be easily queried from any client application, like InfluxDB CLI or Grafana,[17] to monitor, explore and analyze the power consumption of the distributed services in real-time. In the following sections, we execute the same benchmarks as in Section 7.2.1 and we introduce new perspectives on the distribution of power consumption per service and across nodes.

7.2.5 Monitoring the Service-Level Power Consumption

We start by delivering, in Figure 7.17, a new view focusing on the service-level power consumptions, independently of the nodes hosting instances of these services. This view reports on the overall power consumption of the distributed system, masking the idle power consumption of nodes as well as other systems running within the cluster. Within this distributed system, one can observe the limited impact of Swarm and Weave on the power consumption of the cluster along the execution, while Elasticsearch and ZooKeeper can be considered as particularly power-consuming services. Beyond the peaks of activity due to the execution of the ZooKeeper and YCSB benchmarks, one can also observe that each of these services exhibits some residual power consumption along the scenario to maintain their distributed state. More generally, ZooKeeper imposes a larger energy footprint than any other distributed services, consuming 49.27% of the distributed system, due to the consensus algorithm it implements [JRS11]. Additionally, when sequentially killing the nodes, one can observe the energy impact of running the leader election process (at $t = 630$ sec. and $t = 670$ sec.).

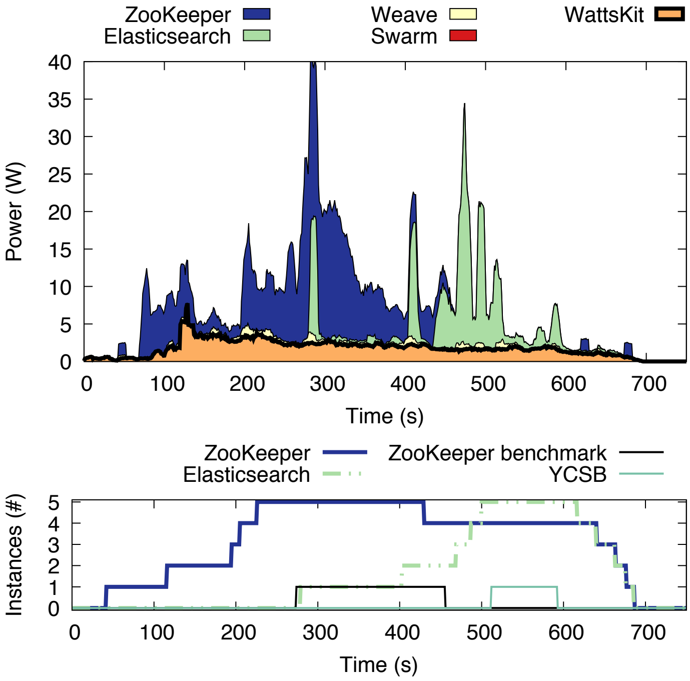
Additionally, we also report in Figure 7.17 on the power consumption of WattsKit. One can observe that the overhead of WattsKit is reasonable (4 W on average for the

[15] `https://influxdata.com`

[16] The backend services of WattsKit can be deployed within the cluster or on any remote node.

[17] `http://grafana.org`

Figure 7.17: Monitoring the distribution of the power consumption of a distributed system in a cluster.



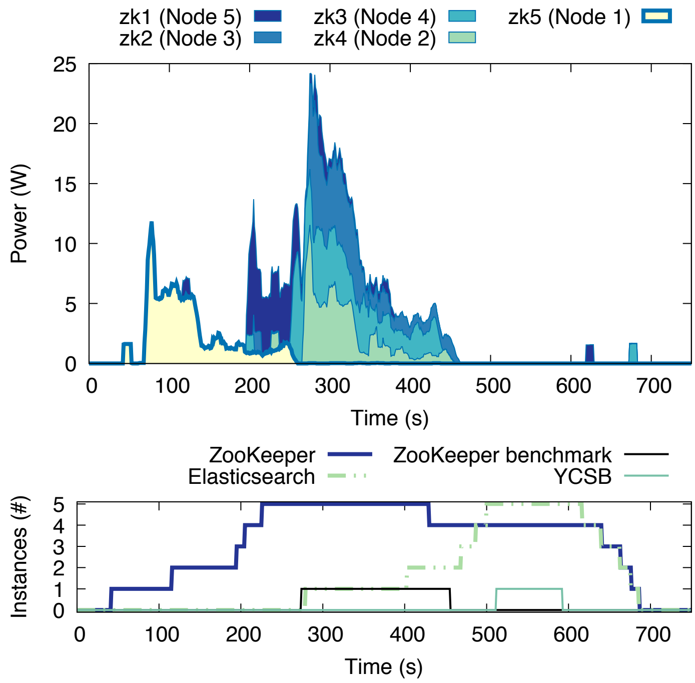
whole cluster) and constant along the execution. WattsKit can therefore be considered as a relevant and cheap alternative to physical PDUs by providing an accurate and fine-grained power meter for distributed services.

7.2.6 Analyzing the Power Consumption Per Service

By taking a closer look to individual services composing the distributed system, we can also use WattsKit to zoom in the power consumption analysis of one of these services. In particular, we report in Figure 7.18 on the distribution of the power consumption of ZooKeeper across the nodes that we use for its deployment. This perspective on ZooKeeper illustrates that the power consumption of such a service is distributed but not equally balanced across the nodes. It also illustrates that the nodes 5 (`zk1`) and 4 (`zk3`) are running the two leader elections we identified towards the end of the scenario.

This granularity of power consumption understanding was particularly difficult to achieve using the coarse-grained power measurements (cf. Figure 7.14) and WattsKit clearly advances the state-of-the-art with respect to that. In particular, we believe that WattsKit can help software engineers to better understand the energy footprint of their services once deployed in production, thus investigating potential optimizations in order to minimize this impact. WattsKit can also benefit to system administrators by investigating the impact of the configuration parameters exposed by the individual services on the power consumption of the distributed system.

Figure 7.18: Analyzing the distribution of the power consumption of ZooKeeper across nodes.



Summary

We presented a software-defined power meter, WattsKit, for monitoring the power consumption of distributed systems. Such software power meters provide an accurate alternative to dedicated hardware systems or embedded probes by estimating power consumption in the large—*i.e.*, at the granularity of services running across several nodes. With WattsKit, we cross the boundaries of physical hosts and we provide an estimation of the power consumption of applications spanning several (or virtual) machines.

Thanks to the learning approach we describe in Section 4.1.1, we extend the learned power model for services that conciliates the heterogeneity and the complexity of modern processors. This power model runs in WattsKit without hardware support or system alterations to accurately deliver power estimation. To the best of our knowledge, WattsKit is the first approach to provide such an accurate service-level power model.

This power model is exploited within an instance of software-defined power meter, which can be deployed across the nodes of a cluster to monitor the power consumption of distributed systems in real-time. It is noteworthy that the proposed solution can be scaled to multiple services and nodes, depending on the complexity of the environment. We evaluated the applicability of WattsKit on 3 processor architectures, and we demonstrated that it performs well for different kinds of distributed protocols and algorithms we considered.

The code of WattsKit is freely available as open-source.[18]

[18] `http://wattskit.powerapi.org`

Chapter **8**

Code-Level Power Estimation in Multi-Core Systems

Table of Contents

In the previous chapters, we built several software-defined power meters and described few approaches for better understanding by which software systems and/or services the energy is consumed. However, these tools and approaches are thus limited to debug or monitor the software's energy consumption, thus lacking of precision trying to optimize software. We therefore claim that a finer level of energy monitoring is required to understand how the power is really consumed by a software. This fine-grained level of power monitoring will allow to find out the root cause of energy leaks and thus helping developers to improve their software energy-efficient. As the software-level is not enough acute, we therefore need to go one step further and thus leveraging source-code level power optimization. Such level of power estimation requires in-depth analysis and remains a challenging task. It is particularly tedious to scale down the upper level (node or software) to the lowest level (source-code) of power monitoring. Such level requires a complete analysis to understand how the source-code of a software is crossed and to therefore find out its contribution to the overall energy consumption.

We therefore propose an approach, codEnergy, for leveraging source-code level power monitoring and paving the way for future software energy optimizations.

The remainder of this chapter is organized as follows. Section 8.1 describes our approach, codEnergy, for analyzing the energy consumption at the source-code level. Section 8.2 demonstrates the lightness of our approach while monitoring source-code at high rate. Section 8.3 finally reports on a case study that analyzes the energy distribution of methods in a well-known in-memory storage solution, `redis`.

8.1 codEnergy, In-Depth Energy Analysis of Source-Code

Figure 8.1: General overview of the proposal for analyzing the energy distribution of software methods.

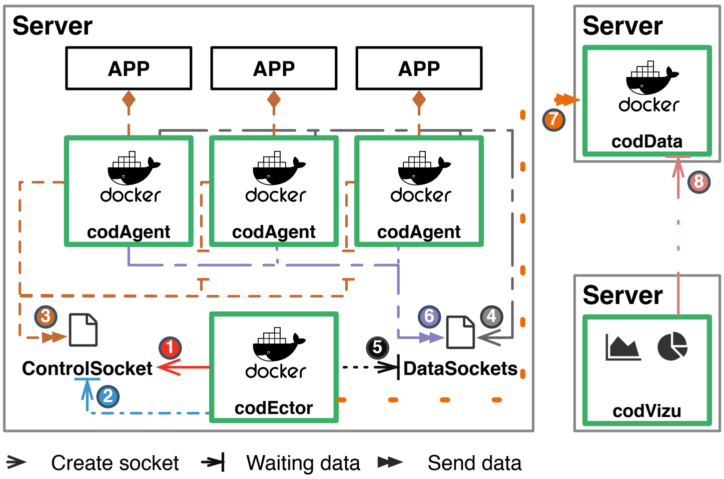


Figure 8.1 depicts the overall architecture of our approach for proposing an in-depth energy analysis. The remaining of this section describes the underlying components of our approach, named codEnergy, and follows a common theme—*i.e.*, the energy analysis of `redis`. `redis` is a well-known open-source in-memory key-value data structure storage solution. To be efficient and reliable, `redis` only works with data in memory, that can be persisted on disk. Different strategies are available for persisting data and one has to choose between dumping the data every once a while, or to log each command to a file. Upon starting, `redis` automatically loads the dump inside the memory, or replays all commands read from log. Different modes of execution are proposed and `redis` can be configured to automatically shard data between nodes.

8.1.1 codAgent, the Runtime Observer

The codAgent is a non-intrusive program that is attached to a target application for collecting runtime code-level metrics. As `OProfile`[1] and `gprof`[2] do, our agent uses statistical sampling to retrieve which method is currently being called. Both of these tools, when configured at high sampling rate, allows to get the distribution of the code executed while running.

While `OProfile` uses HPC for triggering the samples and recording all the symbols involved when calling a method, `gprof` uses special system calls that are only limited to the current monitored process. In other words, the first tool provides a general overview and helps to know whether the bottlenecks occur in the kernel or inside external shared

[1] `http://oprofile.sourceforge.net`

[2] `https://sourceware.org/binutils/docs/gprof`

libraries whereas the second one can not. That is why our approach mimics the behavior of `OProfile` and we therefore made the choice to use HPC as best input metrics to trigger samples.

Firstly, our agent is attached to an external program, thanks to the `ptrace`[3] system call that allows to observe the execution of another process. This approach does not require any instrumentation of the target program and therefore does not affect its runtime performances (cf. Section 8.2). Thanks to the `libpfm` library, an interruption is triggered when a given HPC reaches a configured threshold. This threshold is then directly linked to the sampling rate that will allow to get a representative distribution of the code executed. Our agent then catches the triggered interruption to retrieve current HPC values and to build the current local call graph. As demonstrated in Section 6.1.1, we therefore select the `unhalted cycles` HPC for accurately represent the activity of a given application.

Whereas the `OProfile` tool builds statistical call graph, which means inferring which method are calling which, our approach is able to compute the real call graph in real-time. Thanks to the `libunwind`[4] library, we unwind the stack from the current frame and retrieve the raw address informations. We use next the `libdwarf`[5] library for converting these addresses into human-readable names. The call stack is then dynamically built and linked to the raw HPC values collected when the interruption was triggered.

Given we need an efficient mechanism to serialize structured data for forwarding them to the codEctor component, we therefore choose the `protobuf` library to serialize messages.[6]. We then define an universal Payload message that can be read by both codAgent and codEctor components. This message will be later used by the second component for code-level power estimation (cf. Snippet 8.1). The Payload message contains all the informations we can monitor when an interruption is caught.

Snippet 8.1: Payload message definition.

```
syntax = "proto2";

option java_package = "org.powerapi.module.libpfm";
option java_outer_classname = "PayloadProtocol";

message MapEntry {
  required string key=1;
  required uint64 value=2;
}

message Payload {
  required uint32 core=1;
  required uint32 pid=2;
  required uint32 tid=3;
  required uint64 timestamp=4;
  repeated MapEntry counters=5;
  repeated string traces=6;
}
```

[3] http://man7.org/linux/man-pages/man2/ptrace.2.html

[4] http://www.nongnu.org/libunwind

[5] https://www.prevanders.net/dwarf.html

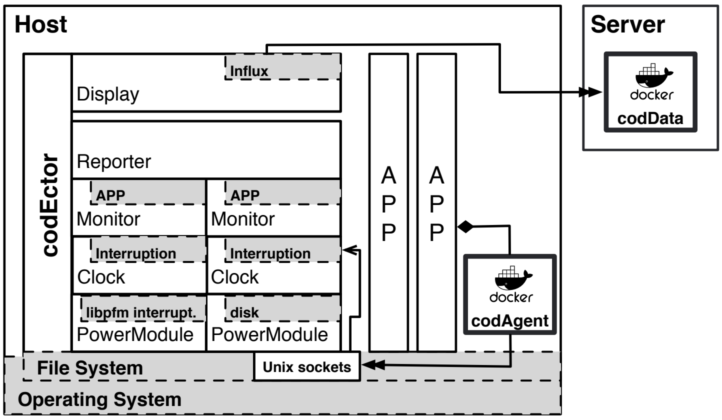
[6] https://developers.google.com/protocol-buffers

We choose Unix domain sockets as an efficient communication mechanism for sharing data between our components (cf. Section 7.1.2).

8.1.2 codEctor, the Code-Level Software-Defined Power Meter

As described in Figure 8.2, the codEctor component is a software-defined power meter built on top of PowerAPI that collects raw informations about the applications that have been attached by a codAgent. It first creates an Unix domain socket server and will be consequently used by each codAgent to establish a connexion—*i.e.*, via the ControlSocket. Once the connexion established, new server sockets will be created for parallelizing the data sending via DataSockets. When a Payload message is available, the codEctor component uses the internal mechanisms of PowerAPI to convert raw informations into power estimation. We then use the raw values that come from the `unhalted cycles` HPC to estimate the software's CPU power consumption, and we instantaneously capture the SSD power consumption we attach to the method that triggered the interruption. Consequently, our codEctor is not limited to some hardware components and can be easily extended upon requirements. As our approach uses samples—*i.e.*, we capture informations when an interruption is caught—to limit the overhead, the codEctor component builds upon each local view sent by codAgent to aggregate the power consumptions over the time—*i.e.*, several methods can simultaneously be executed in different threads. Consequently, each estimation only represents the net power consumption of a method without its dependencies—*i.e.*, we assign the power consumption to methods that have been on top of a call stack. Estimation are next forwarded in real-time to the codData component.

Figure 8.2: Overview of the codEctor architecture.



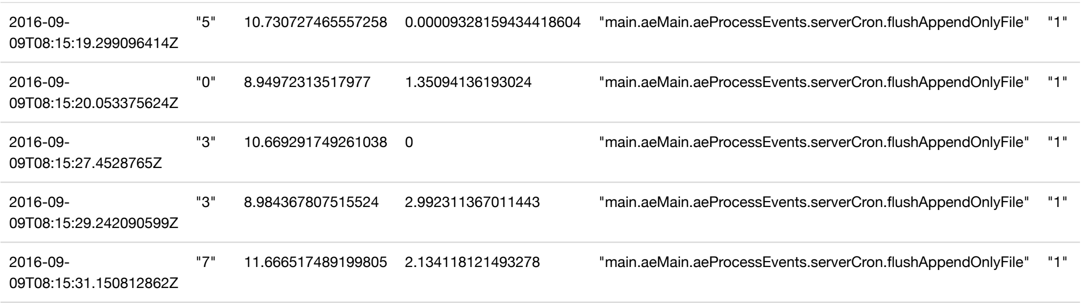
8.1.3 codData, the Storage Solution

codData is responsible for storing all data sent by the codEctor component into a remote InfluxDB[7] instance. A time-series database is then used to store all power estimation per method and application. As an application can be run more than once, a software execution is automatically tagged and can be easily found while querying. The codData component can be queried on demand to learn the energy usage of a software.

[7] `https://influxdata.com`

Figure 8.3 represents the results while querying the power consumption over the time of the flushAppendOnlyFile method from a `redis` execution. Each row associates the tick origin of the interruption in nanoseconds, the core id where the method was running, the CPU and SSD power estimation, the method name, and the tag described above. As several methods can be run simultaneously via threads or forks, the results can be stacked to represent the power consumption over the time. Thanks to the InfluxDB technology, we can automatically aggregate data by time interval and thus directly get the required stacked view.

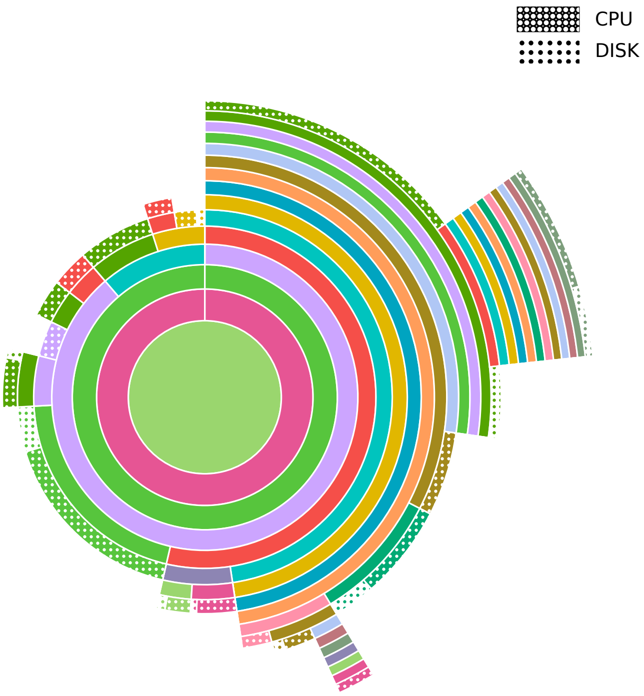
Figure 8.3: Data registered for the flushAppendOnlyFile method of a `redis` execution while querying the InfluxDB service.



2016-09-09T08:15:19.299096414Z	"5"	10.730727465557258	0.00009328159434418604	"main.aeMain.aeProcessEvents.serverCron.flushAppendOnlyFile"	"1"
2016-09-09T08:15:20.053375624Z	"0"	8.94972313517977	1.35094136193024	"main.aeMain.aeProcessEvents.serverCron.flushAppendOnlyFile"	"1"
2016-09-09T08:15:27.4528765Z	"3"	10.669291749261038	0	"main.aeMain.aeProcessEvents.serverCron.flushAppendOnlyFile"	"1"
2016-09-09T08:15:29.242090599Z	"3"	8.984367807515524	2.992311367011443	"main.aeMain.aeProcessEvents.serverCron.flushAppendOnlyFile"	"1"
2016-09-09T08:15:31.150812862Z	"7"	11.666517489199805	2.134118121493278	"main.aeMain.aeProcessEvents.serverCron.flushAppendOnlyFile"	"1"

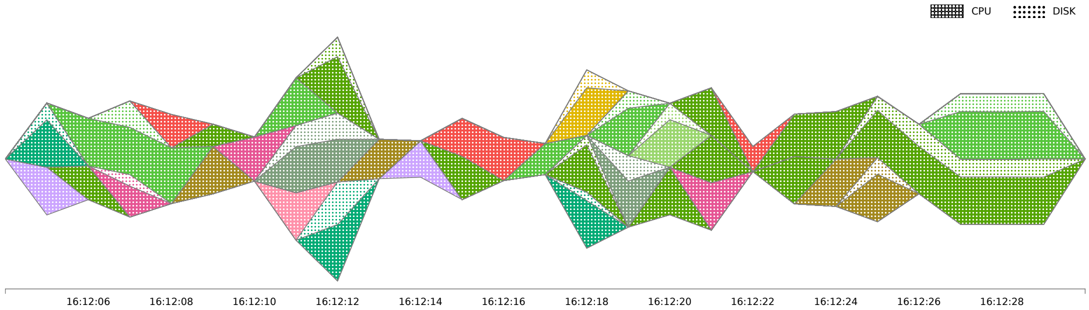
8.1.4 codVizu, the Visualizer for Code Energy Distribution

Figure 8.4: Sunburst chart available via codVizu for a `redis` execution.



codVizu is our built-in web application that queries the codData component and builds dynamic charts of software energy distribution. We first propose an interactive chart to representing the software energy distribution by method over each of its runs. We choose to use a Sunburst chart as the best representation to graphically describe the energy distribution through methods and the relationships between them. The central inner circle represents the root method of a program, while the others moving outwards represent the hierarchy from parent to children methods. Each method has its own unique color and we apply a different pattern filling for each hardware component. Rings are sliced and each angle is proportional to the energy consumed by the underlying method. The Sunburst chart allows the user to get a general overview of the software energy distribution and can be considered while trying to target optimizations at code-level (cf. Figure 8.4).

Figure 8.5: Streamgraph chart available via codVizu for a `redis` execution.



It is sometimes useful to get deeper insights by observing the energy distribution over the time. We decide then to create a dynamic Streamgraph chart that belongs to the family of stacked area charts. This kind of chart replaces the traditional way to plot values against fixed axis by using a varying baseline and displays changes over the time by using organic shapes for each displayed category. The size of the shape represents the importance of the associated value at each given time t. The benefits of such interactive diagrams are described in [BW08; HHN02]. This Streamgraph is tightly coupled with the Sunburst and uses the same colors and patterns (cf. Figure 8.5).

Thanks to the D3.js[8] library, all these charts are dynamics and interactives. Users can go deeper in the call graph by zooming into the charts while clicking on a method from the Sunburst chart.

codVizu is available online and is freely available via Github Pages.[9] Github Pages uses `jekyll` as internal engine to statically build the website. A local website can be thus easily deployed for testing purpose or privacy issues.

Summary

All the components especially designed for codEnergy have been packaged as Docker containers and can be thus easily configured and deployed upon requirements. The code of the described components is available online.[10]

As depicted in Figure 8.1, all these components are tightly linked together. To summarize, codEctor creates an Unix socket server and actively waits connexions (❶, ❷). Once a codAgent is attached to an application, it sends basic informations—*i.e.*, PID,

[8] `https://d3js.org`

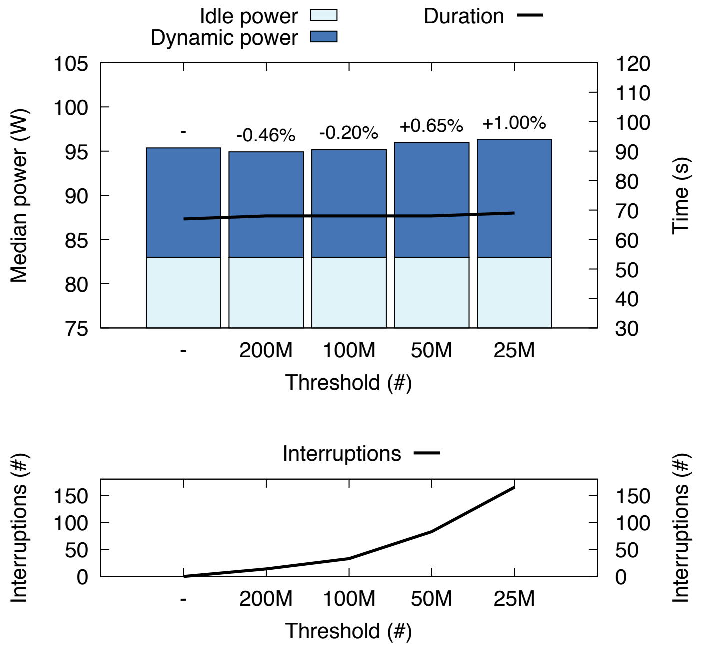
[9] Freely available at: `http://spirals-team.github.io/codEnergy/charts`

[10] Available from: `http://codenergy.powerapi.org`

name, label etc.—via sockets for starting the monitoring, and waits interruptions (❸). New Unix socket servers are created for being exclusive to the target application (❹), and codEctor listens to trigger the computation of new power estimation (❺). When an interruption is caught, the codAgent component dynamically retrieves the call stack of the attached program and runtime informations—*i.e.*, core id, TID, PID, HPC values, and serializes data to forward them via sockets (❻). codEctor uses the serialized data read from sockets to trigger new power estimation that will be sent to the codData component (❼). codData will be queried upon needs via a CLI, or will be used by codVizu for displaying software's energy usage distribution (❽).

8.2 codEnergy's Overhead

Figure 8.6: Impacts of codEnergy on the power consumption and time completion of a `fio` workload while decreasing the codAgent's threshold.



As previously described, codEnergy uses statistical sampling for retrieving data—*i.e.*, informations are retrieved once a threshold is reached and an interruption based on it is caught. Our approach being fully configurable, the codAgent's threshold can be changed upon requirements. Indeed, as we used the `unhalted cycles` HPC as input metric for the threshold, an interruption is only triggered once the software has reached a certain CPU activity. This threshold has to be carefully configured according to the type of the target software.

As a lower threshold means potentially a higher number of interruptions, we first check the effectiveness of our approach while decreasing it. We use the `fio` tool to define an heavy I/O workload. We already demonstrate in Section 4.2 that any I/O activity leads to a certain CPU activity.

The results of the experiment are shown in Figure 8.6. As a matter of a comparison, the normal execution of the workload is graphically represented at the leftmost part of the chart (respectively named -). Different thresholds are tested (200M, 100M, 50M, 25M) to show the evolution of the overhead that can be induced by codEnergy. As one can mention, the overhead of our approach is very low and only represents 1% of power consumption increase when using 25M as threshold (respectively 165 interruptions over 70 seconds). One can also see that the median power consumption while using the thresholds 200M and 100M is lower than the default one. We can explain these results because it is impossible in the facts to get exactly the same power consumption for several runs of an application. We therefore consider that the overheads of codEnergy, configured with 200M and 100M as thresholds, are almost non-existent. While the number of interruptions, ranging from 14 to 165, increases as the same time as the thresholds, ranging from 200M to 25M, the impact of codEnergy in the target program remains very negligible, although the number of interruption is 12× higher.

These results confirm the efficiency of our approach for retrieving critical runtime informations for later code-level power estimation.

The remainder of this section covers a case study analyzing the energy distribution of a well-known in-memory storage solution, `redis`.

8.3 Study the Methods Energy Distribution of `redis`

codEnergy allows to analyze the software energy distribution over methods and therefore allows to find out which methods are the most energy consuming. However, it is still tedious to compare different executions together (cf. Figure 8.4 and Figure 8.5) and thus finding out the code changes that may positively or negatively impact the energy consumption. We therefore propose an extension to the codEnergy approach for allowing direct energy comparison between programs and executions. As an example, one may want to use this extension for checking if a patch correctly fixes an energy leak.

This extension is available online and allows to choose which software systems and versions to compare.[11] For perception reasons, we therefore choose to generate dynamic horizon charts that have already been considered as an efficient solution by the research community [PVF13]. Horizon charts allow to drastically reduce the vertical space used for displaying time series by combining position and color gradients. This kind of charts can be thus used to better display a large number of time series in parallel and to improve the readability.

As codEnergy builds dynamic charts upon needs, we consequently use the `cubism`[12] javascript library which is fluently used by the community.

In the remaining of this section, we use this extension to compare the energy evolution of 2 releases of `redis` and different configurations.

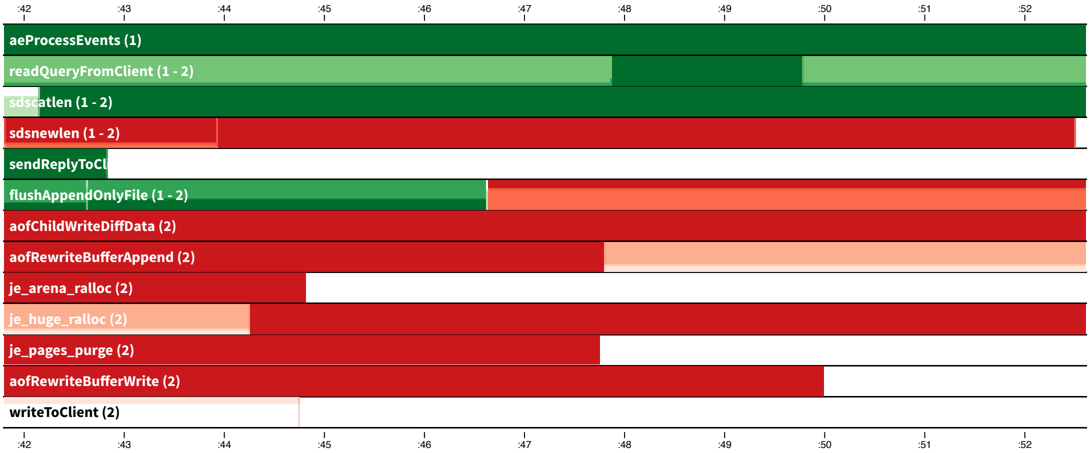
8.3.1 Comparing the Energy Evolution of `redis` Over Versions

We choose in this section to compare 2 different versions of `redis`, the first and last releases—*i.e.*, versions 2.2 and 3.2, respectively. The version 2.2 has been released in 2011,

[11] `http://spirals-team.github.io/codEnergy/compare`

[12] `https://square.github.io/cubism`

Figure 8.7: Energy comparison of methods between `redis` (2.2) and `redis` (3.2).



while the most recent one in 2016. A lot of patches have been proposed and merged by the community for adding features and improving performances (more than 4,000 commits). We therefore decide in this section to compare them for finding out energy improvements or deteriorations.

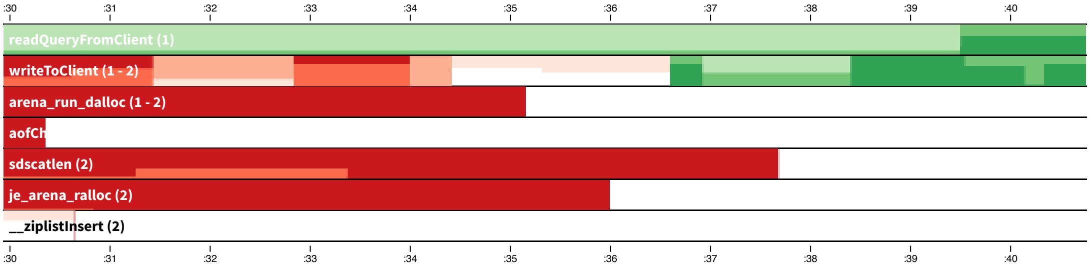
To benchmark `redis`, the developers already include `redis-benchmark` as the official benchmarking tool. The `redis-benchmark` tool simulates N clients configured to send M requests. Each client simulates chosen existing commands (`PING`, `SET`, `INCR`, `LPUSH`, `LRANGE`, etc.) and therefore loads the server with realistic scenarios.

For this experiment, we use `redis-benchmark` to create 50 clients that will send $10,000$ commands each. Each `redis` instance are configured with the default parameters and we force each instance to flush data on disk every second (described as the best compromise between speed and data safety in the documentation).

The energy comparison of the version 2.2 and 3.2 is detailed in Figure 8.7. While a green gradient depicts a positive energy impact—*i.e.*, the old version (1) consumes much power than the new one (2), the red one show an energy leakage. The darker the color, the higher the impact. Thanks to Javascript, the horizon charts are interactive and power consumptions are dynamically displayed while crossing the time series over the time. For clarity reasons, some methods have been removed in the charts.

As expected, one can observe a lot of energy variations, making the comparison difficult, but several insights can nevertheless be extracted. First of all, it confirms that the `redis` server has been clearly evolved since the first release. Indeed, several methods were not implemented in the version 2.2 (je_arena_ralloc, je_huge_ralloc, je_pages_purge, aofChildWriteDiffData, writeToClient, etc.), while the others have not been sampled by our approach (sendReplyToClient) because of different execution models. Secondly, we can see major energy improvements over versions in readQueryFromClient, sdscatlen, and flushAppendOnlyFile methods. We can make the assumption of a better management of requests and replies, and I/O operations. However, while the energy efficiency of sdscatlen were improved ($\approx 12\ W$ on average), the energy efficiency is worse for the sdsnewlen ($\approx 24\ W$ on average), thus mitigating these improvements.

Figure 8.8: Energy comparison of methods between 2 configurations of `redis` (3.2) while sending acks after each command or after 50 commands respectively.



8.3.2 Comparing the Energy Impacts of `redis` Configurations

`redis` acts like a TCP server that uses a client-server mode and adopts a Request/Response protocol. By default, each command sent by a client is then acknowledged by the server. As an example, if a client sends the command `INCR`, the server will answer with the incremented value of the selected variable. In most cases, the server and clients are connected over the network. Whatever the network latency is, the request packets have to be transferred from client to server, and the reply packets from server to client. The time required to transfer these packets is called round trip time (RTT). Higher is the RTT, less the number of requests processed per second by the server is important. To increase performance, one needs to have the lowest possible RTT.

For this purpose, the Request/Response protocol of `redis` can be configured by the client to send multiple commands and to read replies in a single step once executed. This principle is called pipelining and is widely used in many POP3 protocol implementations.

We therefore use the `redis-benchmark`, described in the previous experiment, to send a bunch of commands and to pipeline 50 commands before replying. We use the same parameters and the same commands as the previous section.

The energy impacts between the classic and the pipelining modes are described in Figure 8.8. As the previous experiment, several methods were not sampled during the runtime acquisition (*e.g.*, readQueryFromClient, sdscatlen, je_arena_realloc, ...). We can explain this behavior because the runtime executions are not exactly the same because of configuration changes. However, we can clearly observe energy savings in the writeToClient method ($\approx 25\ J$). These energy improvements are therefore expected and are due to a fewer number of replies sent to the client.

Summary

In this chapter, we propose an approach, codEnergy for analyzing the energy consumption at source-code level. Our approach is composed of several components. The first component, codAgent, uses statistical sampling to retrieve critical runtime informations (metrics, call stack, etc.) and sends the acquired data via sockets to our code-level software-defined power meter, codEctor. Once data collected, codEctor is able to estimate the power consumption of the call stack for the CPU and SSD hardware components. The power estimation are then stored inside the codData component for later use. codVizu will be requested on-demand to display interactive charts on a website. We demonstrate the effectiveness and the lightness of our approach by exhibiting an overhead of 1% in the worst case. Thanks to the case study describes on `redis`, we therefore believe that our approach will be used for finding out the most power consuming methods and thus guiding the developers. By allowing direct energy comparison of the energy distribution of software methods in codVizu, developers can test different energy bug fixes or different configurations and thus improve their software energy footprint.

Chapter 9

Conclusion & Perspectives

Table of Contents

In this manuscript, we describe 3 techniques for learning power models of CPU and SSD hardware components. In addition, we present PowerAPI, our middleware toolkit for building software-defined power meters “à la carte”, thus allowing energy monitoring of concurrent running applications in multi-core architectures. Several approaches/tools have been proposed on top of PowerAPI for, *i).* leveraging SaaS-level power estimation for virtualized environments and distributed services, *ii).* analyzing the source-code energy distribution of a software system. This thesis leverages several new exciting research opportunities.

The remaining of this chapter is organized as follows. We summarize this thesis in Section 9.1 by discussing the challenges and the goals addressed. Section 9.2 describes our contribution. We finally present short-term ideas in Section 9.3, while Section 9.4 presents long-term research directions.

9.1 Summary of the Dissertation

The research community has been intensively investigating the design of power models by considering different components, characteristics, workloads and regression techniques. Nevertheless, the state-of-the-art in this area demonstrates that the proposed power models are mostly based on assumptions that prevent their reuse in other execution contexts and their deployment at scale. We therefore propose different automatic learning techniques that can be used by the community for accurately learning the power models of the CPU and SSD components.

To foster the adoption of power models, we describe our middleware toolkit solution, PowerAPI, for assembling software-defined power meters “à la carte”. PowerAPI allows to propose fine-grained power estimation at system-level, component-level and software-level, thus allowing developers to better analyze the energy efficiency of their solution.

Among the deployment of virtualized environments and the need to find critical indicators to drive power capping heuristics, state-of-the-art solutions only provide coarse-grained power estimation, typically treating the VMs as a black-box. However, these solutions are not suitable in common scenarios while considering that VMs host multiple applications for cost and energy savings. To outcome these limitations, we propose BitWatts that leverages process-level power estimation in VMs.

Monitoring and analyzing of a distributed system spanning several nodes becomes particularly tedious when aiming at a finer granularity than observing the power consumption of hosting nodes. While state-of-the-art fails to deliver such level of power estimation, we propose WattsKit, a dedicated software-defined power meter to be at the forefront of per-service energy monitoring solutions.

Finally, to further guide developers to develop energy-efficient software, one need to have a deeper level of power estimation. However, the few existing solutions remain invasive and are not suitable for common usage scenarios. For this purpose, we propose codEnergy, an approach for leveraging source-code level energy analysis.

9.2 Contributions

The contributions of this thesis are summarized as follows:

Learning Power Models Automatically. We introduce 3 approaches that can automatically learn the power models for CPU and SSD hardware components. 2 learning techniques are proposed for the CPU. Inspired by the state-of-the-art, we select the HPC as input parameters for our power models. The first technique—*i.e.*, the empirical method—uses predefined HPCs extracted from the state-of-the-art: the unhalted-cycles and reference-cyles. Contrarily to the first approach, the second one—*i.e.*, the architecture-agnostic method—does not use an *a priori* knowledge and rather automatically find the most correlated HPCs with the power consumption. The empirical method for CPU power models is adapted for the SSD component and we rather use OS statistics as input metrics for the power models. All of these learning techniques follow the same principles. The targeted hardware component is stressed with publicly available workloads. During the stress, we gather the selected metrics with the power measurements that come from an external power meter. Once collected, the values are injected inside various regression techniques to compute the power models.

[Col+15b] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. “Process-level Power Estimation in VM-based Systems”. In: *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. 2015.

[Col+16] M. Colmant, R. Rouvoy, M. Kurpicz, P. Felber, A. Sobe, and L. Seinturier. “The Next 700 CPU Power Models”. In: *ACM Trans. Model. Perform. Eval. Comput. Syst. (ACM TOMPECS)* (2016).

[CRS14] M. Colmant, R. Rouvoy, and L. Seinturier. “Improving the Energy Efficiency of Software Systems for Multi-Core Architectures”. In: *Middleware 2014 Doctoral Symposium*. 2014.

[CRS15a] M. Colmant, R. Rouvoy, and L. Seinturier. "Estimation de la consommation des systèmes logiciels sur des architectures multi-coeurs". In: *Conférence d'informatique en Parallélisme, Architecture et Système (Compas).* 2015.

[CRS17] M. Colmant, R. Rouvoy, and L. Seinturier. "codEnergy: an Approach For Leveraging Source-Code Level Energy Analysis". In: *To be chosen.* 2017.

Building Software-Defined Power Meters "à la carte". We propose, PowerAPI, our middleware toolkit for assembling software-defined power meters. We define a software-defined power meter as a software solution that can be used for estimating the power consumption of processes, software systems or system with minimal hardware investments. PowerAPI therefore implements the described learning techniques and can be thus used for automatically learning the CPU and SSD power models. Once learned, these power models can be used in PowerAPI for defining accurate and various software-defined power meters, thus allowing to foster their wider adoption. We built several software-defined power meters in this thesis for learning power models or monitoring purposes (Section 4.1, Section 4.2, Section 6.1, Section 6.2, etc.). PowerAPI is published as open-source software[1] under AGPLv3 license thus promoting its adoption by the research and development communities.

[Col+15b] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. "Process-level Power Estimation in VM-based Systems". In: *Proceedings of the 10th European Conference on Computer Systems (EuroSys).* 2015.

[Col+16] M. Colmant, R. Rouvoy, M. Kurpicz, P. Felber, A. Sobe, and L. Seinturier. "The Next 700 CPU Power Models". In: *ACM Trans. Model. Perform. Eval. Comput. Syst. (ACM TOMPECS)* (2016).

[CRS14] M. Colmant, R. Rouvoy, and L. Seinturier. "Improving the Energy Efficiency of Software Systems for Multi-Core Architectures". In: *Middleware 2014 Doctoral Symposium.* 2014.

[CRS15a] M. Colmant, R. Rouvoy, and L. Seinturier. "Estimation de la consommation des systèmes logiciels sur des architectures multi-coeurs". In: *Conférence d'informatique en Parallélisme, Architecture et Système (Compas).* 2015.

Leveraging SaaS-Level Power Estimation. The learned power models, together with PowerAPI, can further be used to estimate the power consumption at different levels. In this contribution, we demonstrate the ability of PowerAPI to estimate the power consumption of concurrent running applications. With the emergence of data centers, it is particularly useful for providers and users to identify which applications are the largest power consumers within such environments. With this aim in mind, we propose BitWatts, a fine-grained software-defined power meter for virtualized environments. BitWatts can be directly deployed upon existing data centers and be used to estimate the software power consumption through all levels of virtualization. To further promote PowerAPI as the middleware to use for all energy studies, we propose WattsKit, a built-in software-defined power meter to

[1] Available from: `http://powerapi.org`

allow the power monitoring of distributed systems. WattsKit therefore helps to consider all energy layers while designing an energy-efficient distributed systems.

[Col+15b] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. "Process-level Power Estimation in VM-based Systems". In: *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. 2015.

[Col+17] M. Colmant, R. Rouvoy, M. Kurpicz, P. Felber, A. Sobe, and L. Seinturier. "WattsKit: Software-Defined Power Monitoring of Distributed Systems". In: *To be chosen*. 2017.

Leveraging Source-Code Energy Monitoring. To better understand how the power consumption can be distributed among software assets, we propose a novel approach, codEnergy. codEnergy is an efficient and non-invasive approach that helps the developers to analyze the software energy distribution through its methods. For this purpose, it creates dynamic and interactive charts that can clearly assist the developers to find out the most power consuming method in the large. Once found, developers can optimize their application and then directly compare the differences between 2 versions by using codEnergy. codEnergy contributes a cornerstone to better energy analysis and optimizations, thus making software more energy efficient in the future.

[CRS17] M. Colmant, R. Rouvoy, and L. Seinturier. "codEnergy: an Approach For Leveraging Source-Code Level Energy Analysis". In: *To be chosen*. 2017.

9.3 Short-Term Perspectives

Defining a New Scheduler for Saving Energy in Cloud Data Centers

This topic is introduced as the most advanced perspective related to this thesis as it is currently under conference proceedings [Hav+ar]. In this paper, we propose to extend the classic scheduling problem that can occur while provisioning resource-intensive jobs in data centers. However, such scheduling fails for services, and more particularly system containers, when the required resources are not known in advance. To address this limitation, we propose a framework, GenPack, for better scheduling containers in cloud data centers. GenPack leverages its strengths from generational *garbage collection* and monitors the runtime containers to dynamically learn their requirements for later scheduling decisions. The underlying scheduler manages several generations of servers to better place containers upon needs. All the machines can be turned-off (turned-on resp.) according to the load and thus saving (increasing resp.) energy. Coupled with PowerAPI for energy monitoring issues, we demonstrate that GenPack is able to be up to 23% more energy-efficient than the classic scheduling policies.

[Hav+ar] A. Havet, V. Schiavoni, P. Felber, M. Colmant, R. Rouvoy, and C. Fetzer. "GENPACK: A Generational Scheduler for Cloud Data Centers". In: *IEEE International Conference on Cloud Engineering (IC2E)*. 2017. (To appear).

Leveraging Distributed Source-Code Energy Monitoring

We describe in Chapter 8 an approach for estimating the power consumption at source-code level and thus providing a complete view of the energy consumed by an application. This approach has been assessed on local applications—*i.e.*, not spread among several machines—but can be used with few modifications on distributed applications. Indeed, we first design the components of this approach as Docker containers that ease its deployment across nodes. We already demonstrate in Section 7.2 the ability of PowerAPI to monitor such distributed services. We therefore claim that codEnergy can be extended to distributed applications, thus leveraging distributed source-code level power monitoring and analysis. Such finer level of power monitoring implies new challenges. As an example, one can mention that a method call can trigger remote calls across nodes. To address this problem, we would like to explore different learning techniques for trying to infer such implicit calls. To the best of our knowledge, codEnergy will be the first solution to leverage this level of power monitoring granularity.

Extend codEnergy to Other Programming Languages

As codEnergy is currently limited to C-family programs. Indeed, we do not have already a solution to retrieve the call stack of a program that uses the just-in-time (JIT) compilation. Once this barrier lifted, we therefore believe that this approach can easily be extended to other languages with few modifications. Indeed, only the codAgent component will evolve according to the proposed solution.

Self-Optimization of the Power Models in a Cluster

Another short-term research directions could be to optimize the previously described learning phase once applied to a cluster of nodes. As demonstrated in 7.2, we already propose a software-defined power meter to monitor distributed services among nodes of a cluster. The software-defined power meter uses the learning techniques described in this thesis to learn the power model of each heterogeneous node. These power models are next used to propose an accurate view of the power consumption of each distributed services. Among other things, the learning approaches require a power meter to retrieve power measurements that can be correlated with the input metrics. That means that a power meter can potentially be required on each node and thus requiring important hardware investments. Furthermore, it will be time consuming to use few power meters and to share manually the power models between homogeneous nodes. We therefore believe in an approach that can continuously learn the power models of a node and dynamically spread them to the nodes that share the same architecture. This solution will then only imply few power meters—*i.e.*, one per type of node, and therefore limiting the cost of hardware investments.

Turning-off Nodes of a Cluster during Inactivity Periods

As shown in this thesis, the idle power consumption represents a large part of the overall power consumption of a machine. If we consider that the nodes among a cluster are always turned on and not used 24 hours a day, the total amount of the energy consumed while idling is therefore non-negligible. We believe in an approach that can cleverly learn the activity periods when the nodes are used or not, and thus inferring usage models that can be later used to anticipate the overall load and turn-on/turn-off the number of required nodes. This approach could used the Wake-On-Lan network protocol to remotely act on a given node. It will therefore save a lot of energy during periods of inactivity.

9.4 Long-Term Perspectives

Proposing a Wider Energy Cartography of a System

We already demonstrate the scalability of our learning approaches by generating accurate power models for the CPU and SSD hardware components. In the future, these approaches could be extended to automatically learn per-component power models of a system (*e.g.*, RAM, Network). To this end, PowerAPI will be able to pick up and to use synthetic workloads from a common dictionary that can trustfully represent the activity of each available component. To have the best fitted power models, we therefore think to propose incremental learning approaches that can set things right if the errors are too hight and therefore create new power models. These approaches could be fully integrated by default in modern operating systems and thus directly exposing different views of power estimation from the ProcFS virtual filesystem. With such solutions, we thus pave the way for the spreading of PowerAPI as a transparent software power estimation tool directly integrated inside the OS.

The Power Rising of GPU cards

GPU components are more and more specialized, powerful, and are now able to replace the processor for several complex tasks. GPU cards already integrate hardware power saving mechanisms as modern CPU architectures do (*e.g.*, the DVFS feature [Abe+12; Mei+13]). For this purpose, a deep understanding of the similarities shared with modern CPU architectures have to be done before adapting the learning approaches proposed in Chapter 4. Among other things, one can mention the presence of specific NVIDIA GPU HPC[2] that can already be used. The above learning approaches, coupled with specific GPU-intensive workloads, may possibly be used to learn GPU power models. The future approach will have to be compared to the few existing power models proposed by the state-of-the-art [HK10; Len+13].

Using Genetic Programming to Improve the Energy-Efficiency at Source-Code Level

As shown in Chapter 8, we already provide a wider cartography of the energy distribution of software methods, thus helping developers to identify the most power consuming methods. Once identified, one needs to optimize them, thus leading to energy decrease or improvements. We already identified a technique used by Chen *et al.* [CV16] that needs to be further explored. The authors use the genetic programming principles for mutating code with predefined mutation operators (*e.g.*, sign conversion, commutativity, merge). Among the conclusions drawn by the authors, they already demonstrate the benefits of this technique. Coupled with developer tools, such as Eclipse or IntelliJ, we therefore believe that is a valuable solution for helping developers to better optimize their software.

[2] https://developer.nvidia.com/nvidia-perfkit

Defining Solutions to Automatically Optimize the Software Energy-Efficiency

Amongst the energy monitoring solutions and specific optimizations that can already be applied on software, one may want to propose automatic and integrated solutions. Several solutions have already been published and need further analysis. As an example, we can mention the post-compiler method, proposed in [Sch+14], to optimize non-functional properties of assembly programs—*i.e.*, by considering energy instead of time or binary size. Other methods are interesting to consider energy issues during compilation [Chi+11]. Some other directions are also taken and propose to generate identical (in term of functionalities) applications [All+15; Bau+14], or to use approximate computations [SR16].

Bibliography

[Abe+12] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, and S. Kato. “Power and Performance Analysis of GPU-accelerated Systems”. In: *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*. 2012.

[All+15] S. Allier, O. Barais, B. Baudry, J. Bourcier, E. Daubert, F. Fleurey, M. Monperrus, H. Song, and M. Tricoire. “Multi-tier diversification in Web-based software applications”. In: *IEEE Software* (2015).

[Arj+14] J. Arjona Aroca, A. Chatzipapas, A. Fernández Anta, and V. Mancuso. “A Measurement-based Analysis of the Energy Consumption of Data Center Servers”. In: *Proceedings of the 5th International Conference on Future Energy Systems*. 2014.

[Bai+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. “The NAS parallel benchmarks”. In: *International Journal of High Performance Computing Applications* (1991).

[Bam+13] M. Bambagini, J. Lelli, G. Buttazzo, and G. Lipari. “On the energy-aware partitioning of real-time tasks on homogeneous multi-processor systems”. In: *Energy Aware Computing Systems and Applications*. 2013.

[Bau+14] B. Baudry, M. Monperrus, C. Mony, F. Chauvel, F. Fleurey, and S. Clarke. “DIVERSIFY: Ecology-Inspired Software Evolution for Diversity Emergence”. In: *IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. 2014.

[BC10] A. E. Bohra and V. Chaudhary. “VMeter: Power Modelling for Virtualized Clouds”. In: *Proc. of IEEE International Symposium on Parallel & Distributed Processing*. 2010.

[Bed+10] D. Bedard, M. Y. Lim, R. Fowler, and A. Porterfield. “PowerMon: Fine-grained and Integrated Power Monitoring for Commodity Computer Systems”. In: *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*. 2010.

[Bel00] F. 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*. 2000.

[Ben+10] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. "The Turtles Project: Design and Implementation of Nested Virtualization". In: *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*. 2010.

[Ber+10] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade. "Decomposable and Responsive Power Models for Multicore Processors Using Performance Counters". In: *Proceedings of the 24th ACM International Conference on Supercomputing*. 2010.

[Ber+12] R. Bertran, Y. Becerra, D. Carrera, V. Beltran, M. Gonzílez, X. Martorell, N. Navarro, J. Torres, and E. Ayguadé. "Energy Accounting for Shared Virtualized Environments Under DVFS Using PMC-based Power Models". In: *Future Generation Computer Systems* (2012).

[BH07] L. Barroso and U. Holzle. "The Case for Energy-Proportional Computing". In: *Computer* (2007).

[Bin+14] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. "ORBS: Language-independent Program Slicing". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014.

[Bir+05] W. L. Bircher, M. Valluri, J. Law, and L. K. John. "Runtime identification of microprocessor energy saving opportunities". In: *Proceedings of the International Symposium on Low Power Electronics and Design*. 2005.

[BJ07] W. Bircher and L. John. "Complete System Power Estimation: A Trickle-Down Approach Based on Performance Events". In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software*. 2007.

[BL09] C. Bienia and K. Li. "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors". In: *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*. 2009.

[BW08] L. Byron and M. Wattenberg. "Stacked Graphs – Geometry & Aesthetics". In: *IEEE Transactions on Visualization and Computer Graphics* (2008).

[Chi+11] D. Chillet, E. Senn, C. Belleudy, R. Ben Atitallah, O. Zendra, and A. Fritsch. "Open power and energy optimization platform and estimator (open-people)". In: *International Workshop on Power and Timing Modeling, Optimization and Simulation*. 2011.

[CM05] G. Contreras and M. Martonosi. "Power Prediction for Intel XScale® Processors Using Performance Monitoring Unit Events". In: *Proceedings of the International Symposium on Low Power Electronics and Design*. 2005.

[Col+14] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. *BitWatts: A Process-level Power Monitoring Middleware*. Middleware- Poster session. 2014.

[Col+15a] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. *Process-level Power Estimation in VM-based Systems*. European Conference on Computer Systems (EuroSys) - Poster session. 2015.

[Col+15b] M. Colmant, M. Kurpicz, P. Felber, L. Huertas, R. Rouvoy, and A. Sobe. "Process-level Power Estimation in VM-based Systems". In: *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. 2015.

[Col+16] M. Colmant, R. Rouvoy, M. Kurpicz, P. Felber, A. Sobe, and L. Seinturier. "The Next 700 CPU Power Models". In: *ACM Trans. Model. Perform. Eval. Comput. Syst. (ACM TOMPECS)* (2016).

[Col+17] M. Colmant, R. Rouvoy, M. Kurpicz, P. Felber, A. Sobe, and L. Seinturier. "WattsKit: Software-Defined Power Monitoring of Distributed Systems". In: *To be chosen.* 2017.

[Coo+10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing.* 2010.

[Coo12] G. Cook. *How Clean is Your Cloud?* 2012. URL: `http://www.greenpeace.org/international/Global/international/publications/climate/2012/iCoal/HowCleanisYourCloud.pdf` (visited on 09/23/2016).

[CRS14] M. Colmant, R. Rouvoy, and L. Seinturier. "Improving the Energy Efficiency of Software Systems for Multi-Core Architectures". In: *Middleware 2014 Doctoral Symposium.* 2014.

[CRS15a] M. Colmant, R. Rouvoy, and L. Seinturier. "Estimation de la consommation des systèmes logiciels sur des architectures multi-coeurs". In: *Conférence d'informatique en Parallélisme, Architecture et Système (Compas).* 2015.

[CRS15b] M. Colmant, R. Rouvoy, and L. Seinturier. *Process-level Power Estimation in Multi-core Architectures.* Conférence d'informatique en Parallélisme, Architecture et Système (Compas) - Poster session. 2015.

[CRS17] M. Colmant, R. Rouvoy, and L. Seinturier. "codEnergy: an Approach For Leveraging Source-Code Level Energy Analysis". In: *To be chosen.* 2017.

[CV16] J. Chen and G. Venkataramani. "enDebug: A hardware–software framework for automated energy debugging". In: *Journal of Parallel and Distributed Computing* (2016).

[Dol+15] M. F. Dolz, J. Kunkel, K. Chasapis, and S. Catalán. "An analytical methodology to derive power models based on hardware and software metrics". In: *Computer Science - Research and Development* (2015).

[DRS09] T. Do, S. Rawshdeh, and W. Shi. "pTop: A process-level power profiling tool". In: *Proceedings of the 2nd Workshop on Power Aware Computing and Systems.* 2009.

[ERK06] D. Economou, S. Rivoire, and C. Kozyrakis. "Full-System Power Analysis and Modeling for Server Environments". In: *In Workshop on Modeling Benchmarking and Simulation.* 2006.

[FGC05] X. Feng, R. Ge, and K. Cameron. "Power and Energy Profiling of Scientific Applications on Distributed Systems". In: *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium.* 2005.

[FS99] J. Flinn and M. Satyanarayanan. "PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications". In: *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications.* 1999.

[FWB07] X. Fan, W.-D. Weber, and L. A. Barroso. "Power Provisioning for a Warehouse-sized Computer". In: *Proceedings of the 34th Annual International Symposium on Computer Architecture.* 2007.

[Ge+10] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. Cameron. “PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications”. In: *IEEE Transactions on Parallel and Distributed Systems* (2010).

[Hav+ar] A. Havet, V. Schiavoni, P. Felber, M. Colmant, R. Rouvoy, and C. Fetzer. “GENPACK: A Generational Scheduler for Cloud Data Centers”. In: *IEEE International Conference on Cloud Engineering (IC2E).* 2017. (To appear).

[Hea+05] T. Heath, B. Diniz, E. V. Carrera, W. Meira Jr., and R. Bianchini. “Energy Conservation in Heterogeneous Server Clusters”. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 2005.

[HHN02] S. Havre, B. Hetzler, and L. Nowell. “ThemeRiver™: In Search of Trends, Patterns, and Relationships”. In: *Proceedings of IEEE Symposium on Information Visualization.* 2002.

[HK10] S. Hong and H. Kim. “An Integrated GPU Power and Performance Model”. In: *SIGARCH Comput. Archit. News* (2010).

[IM03] C. Isci and M. Martonosi. “Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data”. In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture.* 2003.

[INB16] S. Islam, A. Noureddine, and R. Bashroush. “Measuring Energy Footprint of Software Features”. In: *24th IEEE International Conference on Program Comprehension.* 2016.

[Int15a] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual.* 2015. URL: `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf` (visited on 08/01/2016).

[Int15b] Intel. *PowerTOP.* 2015. URL: `https://01.org/powertop` (visited on 08/02/2016).

[Jan+12] S. Janacek, K. Schroder, G. Schomaker, W. Nebel, M. Ruschen, and G. Pistoor. “Modeling and approaching a cost transparent, specific data center power consumption”. In: *Proc. of International Conference on Energy Aware Computing.* 2012.

[JRS11] F. P. Junqueira, B. C. Reed, and M. Serafini. “Zab: High-performance broadcast for primary-backup systems”. In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks.* 2011.

[JSM12] W. Jia, K. Shaw, and M. Martonosi. “Stargazer: Automated regression-based GPU design space exploration”. In: *Performance Analysis of Systems and Software.* 2012.

[Kan+10] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya. “Virtual Machine Power Metering and Provisioning”. In: *Proceedings of the 1st ACM Symposium on Cloud Computing.* 2010.

[KF14] T. Knauth and C. Fetzer. “DreamServer: Truly On-Demand Cloud Services”. In: *Proc. of ACM SIGOPS International Conference on Systems & Storage.* 2014.

[Kiv+07] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. “`kvm`: the Linux Virtual Machine Monitor”. In: *Proc. of the Linux Symposium.* 2007.

[KOS16] M. Kurpicz, A.-C. Orgerie, and A. Sobe. “How Much Does a VM Cost? Energy-Proportional Accounting in VM-Based Environments”. In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2016.

[Kri+10] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. H. Katz. “NapSAC: Design and Implementation of a Power-proportional Web Cluster”. In: *Proceedings of the First ACM SIGCOMM Workshop on Green Networking*. 2010.

[Kri+11] B. Krishnan, H. Amur, A. Gavrilovska, and K. Schwan. “VM Power Metering: Feasibility and Challenges”. In: *ACM SIGMETRICS Performance Evaluation Review* (2011).

[KTG11] E. Krevat, J. Tucek, and G. R. Ganger. “Disks Are Like Snowflakes: No Two Are Alike”. In: *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*. 2011.

[Kur+14] M. Kurpicz, M. Colmant, L. Huertas, A. Sobe, P. Felber, and R. Rouvoy. *How energy-efficient is your cloud app?* Conférence d’informatique en Parallélisme, Architecture et Système (Compas) - Poster session. 2014.

[KVN10] R. Koller, A. Verma, and A. Neogi. “WattApp: An Application Aware Power Meter for Shared Data Centers”. In: *Proc. of ACM International Conference on Autonomic Computing*. 2010.

[KZ08] A. Kansal and F. Zhao. “Fine-grained Energy Profiling for Power-aware Application Design”. In: *SIGMETRICS Perform. Eval. Rev.* (2008).

[LBL07] C.-H. Lien, Y.-W. Bai, and M.-B. Lin. “Estimation by Software for the Power Consumption of Streaming-Media Servers”. In: *IEEE transactions on instrumentation and measurement* (2007).

[LeB+15] M. LeBeane, J. H. Ryoo, R. Panda, and L. K. John. “WattWatcher: Fine-Grained Power Estimation for Emerging Workloads”. In: *Computer Architecture and High Performance Computing (SBAC-PAD), 2015 27th International Symposium on*. 2015.

[Len+13] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. “GPUWattch: Enabling Energy Optimizations in GPGPUs”. In: *SIGARCH Comput. Archit. News* (2013).

[LJ03] T. Li and L. K. John. “Run-time Modeling and Estimation of Operating System Power Consumption”. In: *SIGMETRICS Perform. Eval. Rev.* (2003).

[LPD13] J. H. Laros, P. Pokorny, and D. DeBonis. “PowerInsight - A Commodity Power Measurement Capability”. In: *Green Computing Conference, 2013 International*. 2013.

[LPF10] M. Y. Lim, A. Porterfield, and R. Fowler. “SoftPower: Fine-grain Power Estimations Using Performance Counters”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. 2010.

[McC+11] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta. “Evaluating the Effectiveness of Model-based Power Characterization”. In: *Proceedings of the USENIX Annual Technical Conference*. 2011.

[Mei+11] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. “Power Management of Online Data-intensive Services”. In: *SIGARCH Comput. Archit. News* (2011).

[Mei+13] X. Mei, L. S. Yung, K. Zhao, and X. Chu. “A Measurement Study of GPU DVFS on Energy Conservation”. In: *Proceedings of the Workshop on Power-Aware Computing and Systems*. 2013.

[MGW09] D. Meisner, B. T. Gold, and T. F. Wenisch. “PowerNap: Eliminating Server Idle Power”. In: *SIGARCH Comput. Archit. News* (2009).

[Mog+13] N. Moghaddami Khalilzad, J. Lelli, G. Lipari, and T. Nolte. “Towards Energy-aware Multiprocessor Hierarchical Scheduling of Real-time Systems”. In: *19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 2013.

[NIB16] A. Noureddine, S. Islam, and R. Bashroush. “Jolinar: Analysing the Energy Footprint of Software Applications (Demo)”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016.

[Nor12] P. Nordwall. *50 million messages per second - on a single machine*. 2012. URL: `http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single` (visited on 08/01/2016).

[Nou14] A. Noureddine. “Towards a Better Understanding of the Energy Consumption of Software Systems”. PhD thesis. Université des Sciences et Technologie de Lille - Lille I, 2014.

[NR15] A. Noureddine and A. Rajan. “Optimising Energy Consumption of Design Patterns”. In: *International Conference on Software Engineering*. 2015.

[NRS14] A. Noureddine, R. Rouvoy, and L. Seinturier. “Unit Testing of Energy Consumption of Software Libraries”. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 2014.

[NRS15] A. Noureddine, R. Rouvoy, and L. Seinturier. “Monitoring energy hotspots in software - Energy profiling of software code”. In: *Autom. Softw. Eng.* (2015).

[Ode+04] M. Odersky, P. Altherr, V. Cremet, E. Burak, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *An Overview of the Scala Programming Language*. Tech. rep. EPFL Lausanne, Switzerland, 2004.

[ODL14] A.-C. Orgerie, M. Dias De Assuncao, and L. Lefèvre. “A Survey on Techniques for Improving the Energy Efficiency of Large Scale Distributed Systems”. In: *ACM Computing Surveys* (2014).

[Pre+15] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. “Energy Proportionality and Workload Consolidation for Latency-critical Applications”. In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 2015.

[PVF13] C. Perin, F. Vernier, and J.-D. Fekete. “Interactive Horizon Graphs: Improving the Compact Visualization of Multiple Time Series”. In: *Proceedings of the 2013 Annual Conference on Human Factors in Computing Systems*. 2013.

[Ras+15] M. Rashti, G. Sabin, D. Vansickle, and B. Norris. “WattProf: A Flexible Platform for Fine-Grained HPC Power Profiling”. In: *2015 IEEE International Conference on Cluster Computing*. 2015.

[Ras15] M. Rasmussen. *sched: Energy cost model for energy-aware scheduling.* 2015. URL: `https://lwn.net/Articles/650426/` (visited on 08/01/2016).

[Riv+07] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. "JouleSort: a balanced energy-efficiency benchmark". In: *Proceedings of the ACM SIGMOD international conference on Management of data.* 2007.

[RL87] P. J. Rousseeuw and A. M. Leroy. *Robust Regression and Outlier Detection.* 1987.

[Rot+12] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. "Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge". In: *IEEE Micro* (2012).

[RRK08] S. Rivoire, P. Ranganathan, and C. Kozyrakis. "A Comparison of High-level Full-system Power Models". In: *Proceedings of the Conference on Power Aware Computing and Systems.* 2008.

[San09] D. Sanderson. *Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure.* 2009.

[Sch+14] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer. "Post-compiler Software Optimization for Reducing Energy". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems.* 2014.

[She+13] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. "Power Containers: An OS Facility for Fine-grained Power and Energy Management on Multicore Servers". In: *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems.* 2013.

[SLB07] J. Stoess, C. Lang, and F. Bellosa. "Energy Management for Hypervisor-Based Virtual Machines". In: *Proc. of USENIX Annual Technical Conference.* 2007.

[SPE13] SPEC. *SPECjbb2013 Design Document.* 2013. URL: `https://www.spec.org/jbb2013/docs/designdocument.pdf` (visited on 08/01/2016).

[SR16] P. Stanley-Marbell and M. Rinard. "Reducing Serial I/O Power in Error-tolerant Applications by Efficient Lossy Encoding". In: *Proceedings of the 53rd Annual Design Automation Conference.* 2016.

[Ste13] C. Sterling. *Energy Consumption tool in Visual Studio 2013.* 2013. URL: `http://blogs.msdn.com/b/visualstudioalm/archive/2013/07/10/energy-consumption-tool-in-visual-studio-2013.aspx` (visited on 08/01/2016).

[Tan+15] G. Tang, W. Jiang, Z. Xu, F. Liu, and K. Wu. "Zero-Cost, Fine-Grained Power Monitoring of Datacenters Using Non-Intrusive Power Disaggregation". In: *Proceedings of the 16th Annual Middleware Conference.* 2015.

[The08] The Climate Group. *SMART 2020: Enabling the low carbon economy in the information age.* 2008. URL: `http://gesi.org/article/43` (visited on 09/23/2016).

[VAN08] A. Verma, P. Ahuja, and A. Neogi. "pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems". In: *Proc. of Middleware Conference.* 2008.

[VGS97] V. Vapnik, S. E. Golowich, and A. Smola. "Support Vector Method for Function Approximation, Regression Estimation, and Signal Processing". In: *Advances in Neural Information Processing Systems.* 1997.

[VWT13] D. Versick, I. Wassmann, and D. Tavangarian. “Power Consumption Estimation of CPU and Peripheral Components in Virtual Machines”. In: *SIGAPP Appl. Comput. Rev.* (2013).

[WCS11] S. Wang, H. Chen, and W. Shi. “SPAN: A software power analyzer for multicore computer systems”. In: *Sustainable Computing: Informatics and Systems* (2011).

[Yan+14] H. Yang, Q. Zhao, Z. Luan, and D. Qian. “iMeter: An integrated VM power model based on performance profiling”. In: *Future Generation Computer Systems* (2014).

[ZA12] R. Zamani and A. Afsahi. “A Study of Hardware Performance Monitoring Counter Selection in Power Modeling of Computing Systems”. In: *Proceedings of the 2012 International Green Computing Conference*. 2012.

[Zha+14] Y. Zhai, X. Zhang, S. Eranian, L. Tang, and J. Mars. “HaPPy: Hyperthread-aware Power Profiling Dynamically”. In: *Proceedings of the USENIX Annual Technical Conference*. 2014.

Appendices

Appendix A

Hardware Architectures

This chapter is an in-depth description of the hardware used inside this manuscript.

Table A.1: Examples of PMUs detected for 5 processors from 3 manufacturers, including numbers of generic counters and available events.

Manuf.	CPU	PMU	# Generics	# Events
Intel	**Xeon E5-2630 v3**	hsw_ep	4	418
		hswep_unc_cbo0	4	190
	Xeon W3520	nhm	4	338
		nhm_unc	8	176
	i3 2120	snb	4	336
		snb_unc_cbo0	2	19
		snb_unc_cbo1	2	18
AMD	**Opteron 8354**	fam10h_barcelona	3	421
ARM	**Cortex A15**	arm_ac15	6	67

Table A.1 describes the PMU available on the testbed CPUs together with the associated number of generic events and the number of available events. It can varies considerably across architectures and even among CPUs of the same manufacturer.

Table A.2 reports on the configuration of 5 Intel processors, 1 AMD processor and 1 ARM processor that exhibit different type of architectures and different features.

Table A.2: Processor architecture specifications.

Manuf.	Intel	Intel	Intel	Intel	Intel	AMD	ARM
CPU	Xeon	Xeon	i3	Core2 Quad	Core2 Duo	Opteron	Cortex
Model	E5-2630 v3	W3520	2120	Q6600	E8400	8354	A15
Freq.	2.40GHz	2.66 GHz	3.10 GHz	2.4 GHz	3.0 GHz	2.20 GHz	2.32 GHz
Design	16 cores × 2 threads	4 cores × 2 threads	2 cores × 2 threads	4 cores	4 cores	16 cores	4 cores + 1 lp core
L1 cache	64KB/core	64KB/core	64KB/core	64KB/core	64KB/core	128KB/core	64KB/core
L2 cache	256KB/core	256KB/core	256KB/core	8MB	6MB	512KB/core	4MB
L3 cache	40MB	8MB	3MB	✗	✗	8MB	✗
TDP	80 *W*	130 *W*	65 *W*	105 *W*	65 *W*	N/A	N/A
SMT	✓	✓	✓	✗	✗	✗	✗
DVFS	✓	✓	✓	✓	✓	✗	✗
Turbo	✓	✓	✗	✗	✗	✗	✗
CS	✓	✓	✓	✓	✓	✓	✓