Towards a Better Understanding of the Energy Consumption of Software Systems

Adel Noureddine

Département de formation doctorale en informatique
UFR IEEA

École doctorale SPI Lille

Towards a Better Understanding of the Energy Consumption of Software Systems

THÈSE

présentée et soutenue publiquement le 19/03/2014

pour l'obtention du

Doctorat de l'Université Lille 1

(spécialité informatique)

par

Adel Noureddine

Composition du jury

Rapporteurs :	Ivica Crnkovic, *Mälardalen University - Suède* François Taïani, *Université de Rennes 1 - France*
Examinateurs :	Pierre Boulet, *Université Lille 1* – France Jean-Marc Pierson, *Université Paul Sabatier* – France
Invité :	Alain Anglade, *ADEME* – France
Directeurs :	Lionel Seinturier, *Université Lille 1 - France* Romain Rouvoy, *Université Lille 1 - France*

Laboratoire d'Informatique Fondamentale de Lille – UMR Lille 1/CNRS 8022 – Inria Lille - Nord Europe

Acknowledgement

Foremost, I would like to express my gratitude to my advisors, Prof. Lionel Seinturier, and Dr. Romain Rouvoy, for their continuous support during my Ph.D. study. Their advices, knowledge and support have been invaluable on both academic and personal levels.

Besides my advisors, I would like to thank Prof. Laurence Duchien for welcoming me in the team, and for her continuous advices throughout my thesis years.

In addition, I thank all my colleagues at the research team, and my colleagues at Inria and at University Lille 1, for the challenging talks and inspiring discussions. A big thank for my friends for their help and tips during my years in Lille.

My biggest gratitude goes to my family, my father, my mother, my sister and my brother, for their enduring support and understanding. This thesis would not have been possible without your encouragements.

Abstract

With the rise of the usage of computers and mobile devices, and the higher price of electricity, energy management of software has become a necessity for sustainable software, devices and IT services. Energy consumption in IT is rising through the rise of web and distributed services, cloud computing, or mobile devices. Therefore, energy management approaches have been developed, ranging from optimizing software code, to adaptation strategies based on hardware resources utilization. However, these approaches do not use proper energy information for their adaptations rendering themselves limited and not energy-aware. They do not provide an energy feedback of software, and limited information is available on how and where energy is spend in software code.

To address these shortcomings, we present, in this thesis, energy models, approaches and tools in order to accurately estimate the energy consumption of software at the application level, at the code level, and for inferring energy evolution models based on the method's own input parameters. We also propose JALEN and JALEN UNIT, two energy frameworks for estimating how much energy each portion of a code consumes, and for inferring energy evolution models based on empirical benchmarking of software methods. By using software estimations and energy models, we are able to provide accurate energy information without the need of power meters or hardware energy investment. The energy information we provide also gives energy management approaches direct and accurate energy measurements for their adaptations and optimizations. Provided energy information also draws a model of energy consumption evolution of software based on the values of their input parameters. This gives developers knowledge on energy efficiency in software leading to choose some code over others based on their energy performance.

The experimentations using the implementations of our energy models offer important information on how and where energy is spend in software. In particular, we provide empirical comparison of programming languages (PL), algorithms implementations, the cost of using a virtual machine in PL, compilers' options, and I/O primitives. They also allow the detection of energy hotspots in software, therefore focusing on the main spots where further lookups are needed for energy optimizations. Finally, we demonstrate how our benchmarking framework can detect energy evolution patterns based on input parameters strategies.

With our contributions, we aim to advance knowledge in energy consumption in software by proposing models, approaches and tools to accurately measure energy at finer grains. In a nutshell, we build a software-centric energy *microscope* and conduct experiments aimed to understand how energy is being consumed in software, and directions to be taken for energy optimized software.

Résumé

Avec l'augmentation de l'utilisation des ordinateurs et des appareils mobiles, et la hausse du prix de l'électricité, la gestion énergétique des logiciels est devenue une nécessité pour des logiciels, appareils et services durables. La consommation énergétique augmente dans les technologies informatiques, notamment à cause de l'augmentation de l'utilisation des services web et distribuée, l'informatique dans les nuages, ou les appareils mobiles. Par conséquent, des approches de gestion de l'énergie ont été développées, de l'optimisation du code des logiciels, à des stratégies d'adaptation basées sur l'utilisation des ressources matérielles.

Afin de répondre à ces lacunes, nous présentons dans cette thèse, des modèles énergétiques, approches et outils pour estimer fidèlement la consommation énergétique des logiciels, au niveau de l'application, et au niveau du code, et pour inférer le modèle d'évolution énergétique des méthodes basé sur leurs paramètres d'entrées. Nous proposons aussi JALEN et JALEN UNIT, des frameworks énergétiques pour estimer la consommation énergétique de chaque portion de code de l'application, et pour inférer le modèle d'évolution énergétique des méthodes en se basant sur des études et expériences empiriques. En utilisant des modèles énergétiques et d'outils d'estimations logicielles, nous pouvons proposer des informations énergétiques précises sans avoir besoin de wattmètres ou d'investissement de matériels de mesures énergétiques. Les informations énergétiques que nous proposons, offrent aussi aux approches de gestion énergétique des mesures directes et précises pour leurs approches d'adaptations et d'optimisations énergétiques. Ces informations énergétiques établissent aussi un modèle d'évolution énergétique des logiciels en se basant sur leurs paramètres d'entrées. Cela offre aux développeurs une connaissance plus profonde sur l'efficacité énergétique dans les logiciels. Cette connaissance amènera les développeurs à choisir un certain code au lieu d'un autre en se basant sur son efficacité énergétique.

Les expérimentations utilisant l'implémentation de nos modèles énergétiques offrent des informations importantes sur comment et où l'énergie est consommée dans les logiciels. Plus particulièrement, nous proposons des comparaisons empiriques des langages de programmation (LP), des implémentations d'algorithmes, du coût de l'utilisation d'une machine virtuelle dans les LP, des options des compilateurs, et des primitives d'entrées/sorties. Nos outils permettent aussi de détecter les hotspots énergétiques dans les logiciels, permettant ainsi de focaliser sur les principaux endroits où davantage d'études sont nécessaires pour l'optimisation énergétique. Finalement, nous démontrons comment notre framework d'étude empirique permet de détecter les modèles d'évolution énergétique en se basant sur les stratégies d'évolution des paramètres d'entrées.

Grâce à notre contribution, nous visons d'évoluer la connaissance dans le domaine de la consommation énergétique dans les logiciels, en proposant des modèles, des approches et

des outils pour mesurer avec précision la consommation énergétique à des grains plus fins. En un mot, nous avons construit un *microscope* logiciel et énergétique, et avons mener des expérimentations afin de comprendre comment l'énergie est consommée dans les logiciels, et les chemins à prendre pour produire des logiciels optimisés énergétiquement.

موجز

مع إزدياد إستخدام أجهزة الحاسوب و الأجهزة النقالة، و مع إرتفاع أسعار الكهرباء، أصبحت إدارة إستهلاك الطاقة في التطبيقات و البرمجيات ضرورة من أجل تطبيقات، أجهزة و خدمات مستدامة. إستهلاك الطاقة في تكنولوجيا المعلومات يزداد بسبب إرتفاع إستخدام خدمات الويب، الخدمات الموزعة، الحوسبة السحابية، و الأجهزة النقالة. لذلك، تم تطوير مناهج لإدارة الطاقة في البرمجيات، و تتراوح هذه المناهج بين تحسين شفرة البرمجيات، إلى خطط تكيّف تعتمد على مدى إستخدام موارد الإجهزة. و لكن، هذه المناهج لا تعتمد على معلومات الطاقة بشكل مناسب من أجل خطط التكيّف، فتصبح هذه المناهج محدودة التأثير في مجال الطاقة. لا تقدم هذه المناهج معلومات عن إستهلاك الطاقة في البرمجيات، و هناك معلومات قليلة متوفرة حول كيف و أين يتم إستهلاك الطاقة في شفرة البرمجيات.

لمعالجة هذه القصور، نقدم في هذه الأطروحة صيغ، مناهج و برامج من أجل تقدير بشكل دقيق إستهلاك الطاقة في البرمجيات على مستوى البرنامج و على مستوى شفرة البرمجيات، و عبر إستدلال نموذج تطور الطاقة في البرمجيات طبقا لمعلومات الإدخال. نقدّم أيضا Jalen وJalen Unit، منصات برمجيات من أجل معرفة نسبة إستهلاك الطاقة في كل جزء من شفرة البرمجيات، و من أجل إستدلال نموذج تطور الطاقة إعتمادا على إختبارات تجريبية. عبر إستخدام التقديرات البرمجية من أجل معرفة إستهلاك الطاقة، نستطيع أن نقدم معلومات دقيقة حول إستهلاك الطاقة من دون إستخدام آلات لقياس الطاقة أو الإستثمار في هذه الآلات. معلومات الطاقة التي نقدمها تعطي مناهج إدارة الطاقة قياسات دقيقة و مباشرة للطاقة من أجل خطط تكيفها و مقاربات تحسين البرمجيات. هذه المعلومات تقدم أيضا نموذج تطور الطاقة معتمدة على معلومات الإدخال للبرمجيات. هذه المعلومات تعطي المطوّرين معرفة حول كفاءة البرمجيات، و تؤدي إذن إلى إختيار شفرة برمجية ما على حساب أخرى طبقا لأدائها في مجال إستهلاك الطاقة.

الإختبارات التي قمنا بها بإستخدام برامجنا التي تنفذ صيغ الطاقة التي طوّرناها، تقدّم معلومات هامة حول كيف و أين يتم إستهلاك الطاقة في البرمجيات. بشكل خاص، نقدّم مقارنات تجريبية حول لغات البرمجة، تطبيق الخوارزميات، ثمن إستخدام آلة إفتراضية في لغات البرمجة، خصائص مترجمات لغات البرمجة، و وظائف المدخلات و المخرجات. نستطيع أيضا إكتشاف النقاط الساخنة في إستهلاك الطاقة في البرمجيات حيث يجب التدقيق أكثر من أجل تحسين إستهلاك الطاقة. أخيرًا، نبرهن كيف أن منصة برمجياتنا الإختبارية تستطيع أن تكتشف نموذج تطور إستهلاك الطاقة في البرمجيات إعتمادا على خطط تطور معلومات الإدخال.

عبر إسهاماتنا، نهدف إلى تطوير المعرفة في إستهلاك الطاقة في البرمجيات عبر تقديم صيغ، مناهج و برامج من أجل قياس الطاقة بشكل دقيق. بإختصار، طوّرنا مجهر برمجي من أجل قياس إستهلاك الطاقة، و قمنا بإختبارات تهدف إلى فهم كيف يتم إستهلاك الطاقة في البرمجيات، و الإتجاهات التي يتعيّن إتخاذها من أجل بناء برمجيات كفوءة في إستهلاك الطاقة.

Contents

List of Figures

List of Tables

Part I

Context and Problem Statement

Chapter 1

Introduction

"Innovation is hard. It really is. Because most people don't get it. Remember, the automobile, the airplane, the telephone, these were all considered toys at their introduction because they had no constituency. They were too new."-Nolan Bushnell

Contents

Energy consumption of computers and software is becoming a major factor in designing, building and using sustainable technologies. The topic of energy is becoming mainstream, as more approaches, software, hardware and technologies are being proposed for energy management, optimization or measurement. Information and Communication Technology (or ICT) accounted for 2% of global carbon emission in 2007 [Gar07] or 830 $MtCO_2e$ (Metric Tonne Carbon Dioxide Equivalent), and is expected to grow to 1,430 $MtCO_2e$ in 2020 [Web08]. However, the Climate Group also estimates that ICT solutions could reduce 15% of carbon emissions in 2020 [Web08]. On the other hand, in term of energy consumption, ICT consumed up to 7% of global power consumption (or 168 Gigawatt, or GW) in 2008 [VVHC+10]. This number is expected to double by 2020 to 433 GW, or more than 14.5% of global power consumption [VVHC+10]. These numbers show that although ICT could help reduce energy consumption and carbon emissions of other domains, its own carbon footprint and energy consumption is predicted to rapidly grow. The need to accurately measure and optimize energy efficiency of ICTs is therefore a necessity for the next years and decades.

In addition, software and computer configurations are using distributed services and are constantly powered up. Mobile devices (such as smartphones), servers in data centers, and

desktop computers, consume a rising amount of energy. The diversity of software and hardware configurations makes the middleware layer a good candidate for managing energy consumption. However, managing energy and adapting software and devices for energy concerns require to accurately measure this energy consumption. Existing approaches are limited to using resources utilization information, or coarse-grained hardware-based measurements. A new generation of models, approaches and tools is therefore needed for measuring the energy consumption of software.

In the remainder of this chapter, we outline the problems that motivate this research in Section 1.1. Next, in Section 1.2, we present our research goals. Then, we summarize the contribution of our thesis in Section 1.3. Finally, we conclude with a brief introduction of each of the chapters of this document.

1.1 Problem Statement

Existing approaches for energy management and energy measurements are subject to many limitations hampering the efforts for building efficient and accurate energy management approaches. A number of open problems have limited the efficiency of energy management and measurement. During this dissertation, we have identified the following issues.

Lack of Context-Adapted Energy Measurement Approaches

Existing energy management middleware and software do not use direct energy measurements. They use resources utilization as a key metric for energy estimation. This methodology lacks accurate energy measurements, therefore adaptations have, at best, gross efficiency when managing software. The diversity and evolution of hardware and software in term of energy efficiency implies that energy modeling and estimations need to be as accurate as possible. Modern CPUs, for example, have multiple energy optimization features. Therefore, simple resources utilization mapping is not enough to get accurate measurements (*e.g.*, DVFS, multi-cores, etc.). In addition, existing energy management approaches base their adaptations and optimizations on configuration and domain-specific algorithms, protocols or rules. Therefore, their approaches can only be applied in a restrained set of configurations. For example, many require the usage of a hardware power meter for measuring energy, thus limiting the usability of such approach to an additional hardware investment. The difficulty of the latter, in terms of additional financial, usability, and energy costs, makes it a difficult choice for efficient and context-adapted energy measurements.

Limited Granularity in Software Energy

Measuring energy consumption in software is also a question of granularity. The problem here is that coarse-grained approaches provide little usable information on the energy consumption of software. Hardware power meters give accurate energy consumption results

but for a whole device or hardware, such as the energy consumption of a computer laptop (including the screen, keyboard, mouse, etc.). This makes it difficult for software developers, engineers and scientists to extrapolate and calculate the energy consumption of particular software. It is even nearly impossible to use such hardware meters to measure the energy consumption of software code, such as methods in an application. The latter adds additional problems, such as how to isolate the energy consumption of methods, and how to provide accurate estimation at this code level that is several layers away from the hardware, *i.e.*, the actual material consuming the electric energy. There is limited granularity in existing approaches for measuring the energy consumption of software.

Limited Understanding of Software Energy Consumption

Beside the lack of energy measurement approaches based on context information and their limited granularity, the green computing community have limited understanding of how software consumes the energy at the code level. Lots of research has been done into understanding the internal functioning of software in term of execution time or memory management, but few empirical-based research exists for energy consumption in software. Understanding how energy is being consumed by software, in particular at the code level, is a key for developing better energy efficient software. We also need to experiment and thus understand where energy is being spent by the internal components of software, which part is consuming more, and most importantly, why this energy consumption is happening as it is? What is the impact of modifying the programming language, the compiler or evolving the parameters of a method on the energy consumption? The answers to these questions provide valuable knowledge into writing energy efficient code and software.

1.2 Research Goals

Given the problems we identified in the previous section, our goals in this dissertation are focused on bringing solutions for accurately measuring energy in software, at a finer-level, and understanding this energy consumption and distribution. Such solutions allow us to more accurately measure the energy consumption in software, with an adaptable and scalable approach, and to better understand energy in software. The main goals of our approach are as follows:

- **Energy modeling for software**. The first step to measure energy without drawbacks of hardware investment or gross estimations is to properly model energy in software. The modeling should take into account the different hardware components involved in the energy consumption (such as the CPU, network card, disk, etc.), and resources utilization by the monitored software.

- **Software-only approach**. In addition to modeling energy, the approach needs to be software-only. Therefore, acquiring all the information required for the energy models needs to be done only through software means (such as calling an operating system routine or API, a virtual machine function or reading a configuration file).
- **Accuracy**. Estimating the energy consumption of software without the need of a hardware power meter is not relevant if the estimations are not accurate. Therefore, the margin of error in the estimations needs to be negligible or low within an acceptable margin. The estimations also need to be aware of the diversity of hardware components and their characteristics (such as DVFS and multi-cores in modern CPUs).
- **Fine-grained measurements**. It is important that estimations, measurements and models provide results at finer levels. In particular, offering energy information at code level, such as classes and methods in software, is mandatory to understand energy consumption and efficiently energy optimize software.
- **Experimentations on software energy**. Experimentation results and analysis on energy consumption and distribution in software are strongly missing in the green computing community. Our goal here is to empirically benchmark and experiment on software, and at code level, in order to understand how energy is being consumed, why the said energy pattern is happening, and how energy is evolving.

1.3 Contribution

In this section, we summarize the contributions of our thesis. As stated before, the goal of our research if to provide models, approaches and tools for measuring the energy consumption of software. And also, we aim to understand the energy consumption and distribution in software. The main contributions of our work are summarized as follows.

Energy Models for Software. We define and propose energy models for estimating the energy consumption of software at the application level (*e.g.*, software as black box), and at the code level (*e.g.*, at the granularity of classes and methods). We also propose a methodology to infer the energy evolution model of software methods based on their input parameters and empirical benchmarking.

Energy Measurement Tools. Based on our energy models, we develop tools and applications for measuring the energy consumption of software code. The first software system, JALEN, measures the energy consumption of software methods and classes. It has a low overhead, therefore does not impact the user experience, provides accurate, fine-grained measurements, and follows a software-only approach. The second software system, JALEN UNIT, is a framework for inferring the energy evolution model of software code based on their input parameters. It generates and executes empirical benchmarks for software methods and their applications.

Lessons on Software Energy Consumption. Our final contribution is a serie of experimentations aimed into understanding the energy consumption and distribution in software. Based on our experimentations, we verify common belief on energy consumption in software. And we provide learnings that help software developers into writing more energy efficient software.

1.4 Organization of the Document

This dissertation is organized in three parts. In the first part, *Context and Problem Statement*, we discuss the context behind our work, motivations and problem statement of our thesis. We also conduct a study of the state-of-the art approaches and tools for energy management and energy measurement. The second part, *Energy Measurement and Evolution*, presents our contribution, both in term of energy modeling, methodology approach and validation and experimentation results. Finally, we conclude and present our perspectives in the last part, *Conclusion and Perspectives*. In the remainder of this section, we summarize each chapter in the dissertation.

Part I: Context and Problem Statement

Chapter 1: Introduction. In this chapter, we introduce the problematic of energy management and energy consumption in software, and describe the context of our work. We also present the motivation behind our work and the problem statement of our thesis and contributions.

Chapter 2: State of the Art. In this chapter, we conduct a study of the state-of-the art approaches for managing energy of software and services at the middleware layer. The former layer is a good candidate for managing energy of distributed software and heterogeneous software services. We also present the challenges of managing energy in software and the necessity of having direct measurement for optimal energy management. This requirement motivates us to further study the energy measurement approaches, models and tools in software. We discuss our findings and deduce the challenges and limitations of existing approaches. Therefore, our contribution tackles these challenges into providing energy measurement models, approaches and tools.

Part II: Energy Measurement and Evolution

Chapter 3: Energy Measurement at the Application Level. In this chapter, we present our energy models and approach for measuring the energy consumption of software at the application level, *i.e.*, as black box. We propose energy models for estimating energy consumption of software using only information collected through software means. Therefore,

our approach does not require any hardware power meter, and offers estimation accuracy similar to hardware measurements. Finally, we conduct a series of experimentation on software in order to understand the impact of software on energy consumption. In particular, we study the impact of programming languages, their compilers, the usage of a virtual machine, the implementation algorithms, and of I/O primitives.

Chapter 4: Energy Measurement at Code Level. This chapter complements our research on energy measurement with the introduction of a lower layer of measurement, *e.g.*, code level. We present our energy models and approach for measuring the energy consumption of software code, *i.e.*, classes and methods. We also introduce our code level measurement tool, JALEN. This tool provides accurate estimations of the energy consumption of Java applications at the granularity of their methods and classes. Finally, we validate our approach and tool, and conduct a study on the energy distribution between methods in software. In particular, we detect energy hotspots in software and discuss our findings.

Chapter 5: Energy Evolution Modeling. In this chapter, we propose an approach to model the energy consumption evolution of software code based on their input parameters. We introduce our energy evolution software framework, JALEN UNIT. The framework benchmarks methods in an application and infers its energy consumption evolution based on the evolution of its parameters. We conduct experimentations in order to validate and report on different energy evolution strategies. Finally, we discuss our findings and the impact of software, programming language's infrastructure (in particular, virtual machine), and side effects on software energy evolution.

Part III: Conclusion and Perspectives

Chapter 6: Conclusion and Perspectives. In this chapter, we summarize our work and contributions in the dissertation. Finally, we present our short-term and long-term perspectives in term of energy measurements, energy evolution modeling and energy management of software.

Chapter 2

State of the Art

"We are embedded in a biological world and related to the organisms around us."-Walter Gilbert

Contents

2.1 Introduction

In this chapter, we review approaches, models and tools related to energy management and measurement of software. The goal of this chapter is to study the existing energy-related approaches, compare and discuss the latters in order to outline the limitations of the current approaches into providing energy management and energy measurement platforms.

We start by studying energy management approaches in software, and in particular at the middleware layer in Section 2.2. We argue that middleware platforms are relevant candidates for managing energy of software and hardware in the context of distributed services, multi-devices configurations, heterogeneous programming languages, software and

devices, and connected devices. Then in Section 2.3, we motivate our approach focusing on software and on energy measurement as they are requirements for efficient middleware-based energy management platforms. We study and compare energy measurement, estimation and modeling approaches in Section 2.4. Finally, we summarize our findings in Section 2.5.

2.2 Energy Management Approaches

Reducing the energy consumption of software and devices requires a comprehensive view of the different layers of the system. Sensors and actuators, used to monitor energy consumption and modify devices' options, need to be controlled by intelligent software. Applications running on the devices and the hardware itself also need to be monitored and controlled in order to achieve efficient energy savings. Many approaches have been proposed to manage the energy consumption of the hardware, operating system, network or software layers. However, with the widespread usage of ubiquitous devices and the high coverage of networks (Wi-Fi, 3G, 4G, Bluetooth), a new generation of communicating and mobile devices is emerging. The energy consumption of this diversity of devices, and subsequently applications developed in different programming languages, is better managed through a layer capable of monitoring and managing both the hardware, operating system, network and application layers. Therefore, the middleware layer positions itself as a relevant candidate for hosting energy-aware approaches and solutions.

Many middleware platforms, architectures, optimization techniques and algorithms already exist for energy management of hardware and software. We therefore chose several approaches based on the priority given to energy management in the proposed solutions. Our study on middleware approaches focuses on architectures and frameworks that emphasize energy management in distributed environments. Only references and recent works of the last years have been considered. Lot of efforts have been spent on energy management and optimization. Middleware approaches can adapt their core modules and/or the environment (*e.g.*, hardware, software) following energy objectives and we considered both approaches in our review. As most of nowadays distributed systems are connected with other applications and services, any viable solution should therefore incorporate solutions for energy awareness that emphasize and take advantage of the distributed nature of such systems. We selected the middleware approaches based on this criterion.

In Section 2.2.1, we review middleware approaches for energy management in distributed environments, proposing a detailed overview of the energy management issues in each approach. Finally, we compare the reviewed middleware approaches based on an energy taxonomy that we introduce in Section 2.2.2.

2.2.1 Middleware Approaches for Energy Management

In this section, we review 12 middleware solutions targeted or specifically build for energy management. Middleware platforms provide an abstraction of the underlying hardware, network, and operating system interfaces to the applications. S. Krakowiak defines middleware as [Kra07]:

> In a distributed computing system, *middleware* is defined as the software layer that lies between the operating system and the applications on each site of the system.

The main goal of architectures and platforms for energy management is to optimize or reduce the energy consumption of hardware devices or software services. These approaches do not only optimize the energy consumption of applications and devices, but also optimize the consumption of the middleware platform itself. For a platform to manage energy efficiently, energy should not be considered as a non-functional requirement. It should rather be the core of the approach, eventually taking into account other requirements (quality of service, quality of context, user preferences, usability).

Many approaches integrating energy awareness or energy optimizations exist at the middleware layer. From the wide range of approaches, we select 12 middleware platforms responding to the following criteria:

1. *Middleware architectures or frameworks emphasizing energy management in distributed environments.* We skipped middleware approaches that do not integrate the distributed dimension, or energy management. With the democratization of cloud-computing, the widespread usage of connected mobile devices (*e.g.*, smartphones, tablets, laptops), and with the reduction of network costs for end-users, distributed usage scenarios are frequent. Managing energy of this rising usage is, thus, crucial for their success and market adoption.

2. *Recent work of the last years only.* Energy-aware approaches applied at different system layers have been proposed since the early days of computer science. Since 2005, more than 20,000 research papers related to energy management have been published [LQBC11]. Also, with the progress of technologies and the evolutions in customers' usages, approaches that were valid a decade ago may not offer today the same level of accuracy or energy saving as they were offering. Many approaches and energy related technologies became deprecated. The utilization of technologies and devices also change and evolve. This makes the need for new solutions a necessity (*e.g.*, the decline of the desktop PC and the rise of mobile devices and servers). With the high number of available publications, we argue that limiting the study to the most recent approaches and technologies allows us to provide a more representative view of the usable energy management approaches at the middleware layer.

We select 12 middleware platforms integrating energy management approaches: Transhumance [PMND07, PlDR08, DPHu+08], Grace/2 [SYH+04, VYI+09], CasCap [XHSYJ11], DYNAMO [MDNV07], PARM [MV03], ECOSystem [ZEL05], SANDMAN [SB07, SHB08], SleepServer [ASG10], GreenUp [SLH+12] and the approaches reported in [XKYJ09, BS09, RGKP10]. Table 2.1 and 2.2 on pages 18 and 19 summarize the positive and negative points of these middleware approaches for energy management.

Transhumance

Transhumance [PMND07, PlDR08, DPHu+08] is a power-aware middleware platform for data sharing on *Mobile Ad hoc Networks* (MANets). It supports collaborative applications and provides a set of communication facilities such as a publish-subscribe event system. Transhumance targets small networks (up to 20 nodes), moving at pedestrian speed (up to 5 km/h). The energy management in the platform is policy driven with adaptation policies defining battery level thresholds at which adaptations are triggered. These adaptation policies follow the conditions/actions paradigm.

Energy management in Transhumance focuses mostly on adapting the middleware platform's modules. Applications adaptations follow the middleware platform's own adaptations. As such, if the middleware platform does not adapt its modules in order to save energy (*e.g.*, enable/disable messages encryption), then applications will not be adapted to the environment's changes. Transhumance does not check for conflicts between applied actions. This is left for the user or administrator to guarantee that adaptations actions are not in conflict and do not damage the system integrity or are counterproductive (*e.g.*, wasting energy instead of saving it).

GRACE and GRACE-2

Global Resource Adaptation through CoopEration (GRACE) [SYH+04, VYI+09] is a hierarchical adaptive framework for energy savings. It combines adaptations at different levels: *seldom and expensive global adaptation*, *frequent and cheap per-application adaptation*, and *internal adaptation on a single system layer*. GRACE uses a hierarchical approach that invokes expensive global adaptation (which considers all applications and all system layers), and inexpensive scoped adaptations (per-application adaptation where only one application is considered at a time), and internal adaptation where only a single system layer—but not necessarily one application—is considered.

The approach of GRACE provides a flexible middleware platform that fits for distributed environments. The three layers system allows different granularity adaptations, from global and expensive adaptations to local per-application ones. However, this approach requires a centralized global coordinator that needs a resources-rich device to run on. This limits the benefits of the framework in environments where a large, but resource-poor, number of devices are present (*e.g.*, wireless sensor networks). The use of predictions

in the global adaptation severely limits its frequency, thus limiting these adaptations to large changes in the system. This approach has also a drawback when the framework is used in a volatile environment. As applications and devices may appear or disappear frequently, the global adaptation is also invoked frequently, leading to more energy-consuming global adaptations (and ultimately to energy losses instead of savings).

Middleware for Energy-awareness in Mobile Devices

In [XKYJ09], the authors propose an energy-aware middleware platform for mobile devices that is based on application classifications and power estimations to accomplish application-specific energy optimizations. The middleware platform uses a policy manager to choose adaptive policies to apply on the system.

The major weakness of the architecture is the absence of any conflict resolving mechanism. Adaptation policies are added by the user/administrator and may conflict. The approach also requires a training period for the application classifier to be anywhere effective. This may not fit very well in volatile environments where applications and hardware components change frequently. The dual power estimation (component hardware level and application level) may incur a non-negligible energy overhead in small sensor networks and very low capacity devices (embedded sensors and devices). On the other hand, policy-based rules written in semantic languages allow flexible adaptations. The approach can cover different environments with only having to modify policy rules.

CasCap

CasCap [XHSYJ11] is a framework for context-aware power management. The framework is based on three concepts: crowd-sourcing of context monitoring, functionality offloading and providing adaptations as services. Its architecture is composed of three components: mobile devices, Internet services, and clones. Mobile adaptations are based on adaptation policies that are also offered as services, while clones allow the mobile device to offload some processing to them for energy savings.

The proposed architecture uses the cloud in order to propose adaptation services and offloading. Adaptation as a service fits well in a ubiquitous environment where several devices may use a similar service. In this case, one implementation in the cloud is needed and allows energy savings by offloading all the processing to the cloud. However, the extensive usage of the cloud and Internet services requires high usage of the network interface of mobile devices. The latter is one of the most energy expensive component in a mobile device. Therefore, optimizing the usage of the Internet service by calculating processing/network costs tradeoffs is a most needed requirement for the architecture.

The weakness in the approach is more present in the experimentations rather then the theoretical architecture. The authors did not fully evaluate the implementation of the approach or any cloud-based services. Only minimal validations of trivial experiences were

conducted (such as the WLAN energy consumption on a YouTube video). However, the authors identify where the major points of the architecture fit in the experiences. All what is left is to validate these points with a global implementation of the CasCap architecture.

DYNAMO

DYNAMO [MDNV07] is a cross-layer framework for energy optimizations based on quality of service tradeoffs for video streaming in mobile devices. The framework uses a distributed middleware layer in order to adapt all levels of the system (*e.g.*, software, middleware, operating system, network and hardware). Its architecture uses a proxy server in order to perform *end-to-end* global and network adaptations with the mobile device (*e.g.*, dynamic video transcoding). *On-device* adaptations complement the *end-to-end* adaptions with local adaptations specific to the hardware and software of the mobile device (*e.g.*, LCD backlight intensity adaptation). The approach is built around the usage of a proxy in order to offload some energy-consuming functionalities from the mobile device. This approach is adapted to a network intensive scenario (such as video streaming) where the network overhead for communicating with a proxy is leveraged with the intensive use of the network for the video stream. Adapting the video stream on the proxy, thus allowing to reduce the streamed bandwidth and the required computation for processing it on the device, provides energy savings.

However, the proposed model and architecture will probably not perform as good as the paper's results in different scenarios. In a CPU intensive application, or a lighter network case (*e.g.*, web browsing), the energy gains are quickly overrun by the network overhead, and in particular in mobile devices (where the network card is in the top 3 of the energy consuming components [CH10]). On the other hand, the approach is based on utility functions. The reasoning for adaptation is based on the evaluation of this utility function, and the tradeoffs between energy consumption and the allowed variations of the quality of service parameters.

PARM

PARM [MV03] is a *Power-Aware Reconfigurable Middleware* for low-power devices. It dynamically reconfigures the component distribution on these devices and migrates components to proxy servers in order to save energy on the mobile client. The PARM framework is a flexible, reflective message oriented middleware platform. In addition to typical middleware platform components, it provides a set of additional independent components (*e.g.*, encryption/decryption, caching, clock synchronization) that are used for energy optimization. Components are migrated to a proxy in order to save energy on the mobile client.

The PARM algorithm has a worst case execution time of $O(n^3)$, which may lead to high energy consumption of brokers and longer execution decision times when the environment is composed of a large number of mobile clients. PARM goal is also limited to an environment where proxies, brokers and servers are abundant. It has one main core adaptation technique: component migration from mobile clients to proxy servers.

Green Computing: Energy Consumption Optimized Service Hosting

In [BS09], the authors propose a dispatch algorithm for data centers in order to consolidate services dynamically into a subset of servers and temporarily shut down the remaining servers in order to save energy. The goal of the approach is to minimize the number of running servers while still being able to respond to clients' requests and respect the QoS requirements described in SLAs.

The approach has the advantage of respecting the SLA and QoS of client's requests while still trying to minimize the energy consumption. Although not rule-based or policy-based reasoning, the probabilistic algorithmic approach is not penalizing with regards to energy or time. Data centers are a resource-rich environment and the overhead of the algorithm should be minimal compared to the energy consumption of the servers. However, the proposed approach and algorithm poses many assumptions, thus being limited to a small pool of case studies. It is also limited to data centers that host and handle services offered to clients. The authors identify several limitations of their algorithm, such as limited optimization criteria (only energy consumption and service response time are considered to date), centralized algorithm which may become a performance bottleneck, absence of fault tolerance and management of sudden fluctuations in service requests.

ECOSystem

Energy Centric Operating System (ECOSystem) [ZEL05] is a framework that manages energy consumption at the OS level. The framework is based on a new unit: *currentcy*, which is an abstraction of energy currency. *Currentcy* is a unified abstraction for the energy a system can spend on devices. A unit of *currentcy* represents the right to consume an amount of energy during a fixed amount of time.

The *currentcy* model is a step towards unified energy management. The model, inspired from human's financial transactions, allows a flexible and generic approach to energy allocation. However, the model and framework does not reduce or optimize energy consumption. It is only limited to providing a modeling and architectural infrastructure for energy transactions and allocation, but without reducing the energy utilization *per se*. Nevertheless, proposing a currency model for energy helps in raising awareness on the price of energy consumption, not only for the end-user, but also for developers and system administrators.

SANDMAN

SANDMAN [SB07, SHB08] is an energy-efficient middleware platform built upon the BASE middleware platform [BSGR03]. BASE is a minimal communication middleware platform for pervasive computing based on peer-to-peer principles. SANDMAN is built as several extensions to BASE, and relies on three main concepts: *i)* reducing data transfer energy consumption by selecting the most efficient communication protocol, *ii)* switching idle devices

to low power mode (sleep) in order to save devices' energy during their idle time, *iii)* and allowing clients to select the most energy efficient service. In order to switch devices to idle mode while still being discoverable by users, SANDMAN uses a self-adaptable discovery protocol that can handle deactivated devices.

The SANDMAN approach targets energy consumption of idle devices in a communicating network. It achieves this by grouping devices with similar mobility patterns in a cluster and temporarily migrating nodes advertisement to the cluster head (CH). The approach however, has two weaknesses: *i)* the cluster head (CH) has to answer discovery requests on behalf of devices of its cluster, thus having to be awake all the time and consuming more energy. No energy-aware approach is specified in the election of the CH. This may lead to small battery devices being elected as a CH, which may also lead to a quicker failure of the CH. *ii)* the CHs act as single point of failure in the platform. The failure of a CH causes the devices of its cluster to be temporarily undiscoverable. On the other hand, SANDMAN manages energy when devices are idle. It does not optimize the energy consumption of running applications, thus making the approach limited to situations where devices are used for short periods of time, and in non-critical environments.

SleepServer

SleepServer [ASG10] is a software approach allowing energy management in desktop PCs. The network-proxy based approach allows machines to migrate to low power sleep states while still allowing their network connectivity. This is done using virtual machine proxy servers and virtual LANs.

The approach of SleepServer is similar to SANDMAN's. Both use additional proxy machines (a cluster head in SANDMAN and a sleep server in SleepServer) to maintain availability and network presence of the host while it is in sleep mode. Thus, it has the same weaknesses: *i)* the sleep server becomes a single point of failure for the platform. Its failure may cause host machines to become unavailable until they are wake up again; *ii)* the sleep server has to be awaken all the time therefore consuming energy (albeit the energy consumed is compensated by the energy saved of the system). On the other hand, the usage of SleepServer still requires additional hardware investment (in the form of the sleep servers themselves). It only manages energy for idle devices, allowing energy savings if frequent or long periods of sleep time occurs. SleepServer itself does not offer energy optimizations to applications or devices, but rather to the overall functioning of connected network of computers. The approach, nevertheless, allows a transparent and heterogeneous migration of hosts' availability and network presence. Virtual images offer great flexibility for grouping images into a smaller number of sleep servers, or migrating them again to another sleep server (in particular in case of failure of a sleep server). They also offer security and isolation between the different virtual images.

Sleepless in Seattle No Longer

In [RGKP10], the authors present a system aimed to preserve the network accessibility of user machines in an enterprise network while still allowing them to go to sleep, and therefore save energy. The system is based on two components: a *sleep proxy* for each subnet, and a *sleep notifier* program that runs on the user machine. The latter alerts the proxy when the machine goes to sleep.

The main advantage of this approach (and other similar ones) is allowing machines to go to sleep while still keeping them available for user requests. The only investment needed for user machines is the installation of a sleep notifier daemon. The latter doesn't need to know the address of the proxy (it broadcasts its notifications), thus making adding/removing machines easy to manage for the proxy. The usage of an independent proxy frees the user machines from additional loads incurred from managing sleeping machines. However, the proxy is a single point of failure in the system. If it goes down, sleeping machines cannot be woken up again because their network traffic is redirected to the proxy. In addition, the sleeping policies are left for users or administrators to implement. A machine will go to sleep by its own (or the user's) initiative. Therefore, the proposed system cannot save energy if machines (through their OSs, applications or users) do not go to sleep at all.

GreenUp

GreenUp [SLH+12] is a software-only approach for providing high-availability to sleeping enterprise workstation machines. GreenUp allows any workstation machine to act as a proxy for other machines so the latter can go to sleep while preserving their presence in the network subnet. Therefore when a machine goes to sleep, another one starts acting as a proxy for it.

GreenUp offers a software-only approach where existing machines are used as proxies to manage each other's sleep and network presence. The advantage is the absence of any hardware investment or modification of existing software (unlike for example SleepServer), making deployment easy. Another aspect is the absence of bottlenecks or central proxy servers. Because each machine can act as a proxy, the failure of a machine or even a proxy is quickly remediated by having another machine acting as a proxy for the newly unmanaged ones. *Guardian* machines ensures that there is always a minimum number of proxies available, thus limiting a complete blackout of the system. However, although the approach manages the network presence of sleeping machines, it does not include policies to determine when machines go to sleep. This is left for users or administrator. Thus, GreenUp hopes to *induce users to choose more aggressive sleep policies and thereby save energy*, while still allowing machine availability.

Table 2.1 and 2.2 present the pros and cons of each of the previous middleware approaches. In particular, we summarize the advantages of these approaches, and outline the main drawbacks and limitations for efficient energy management.

Middleware Approach for Energy Management	Pros	Cons
Rule-based approaches		
Transhumance	Uses the intuitive conditions/actions paradigm	Focus on adapting the middleware platform's modules
		No conflict management for adaptation policies
		Can only be applied on a limited environment
CasCap	Offloads functionalities to the cloud to relieve the mobile device	Network (energy consuming component) usage required
		No complete experimentation yet
Middleware for Energy-awareness in Mobile Devices	Usage of applications classification	No conflict management
	Based on semantic policy rules	Requires training periods
DYNAMO	Combines adaptations at different system levels	Fits well only in a network intensive scenario
	Vision of the global and local context for adaptations	Requires hardware investment in a proxy
Proxy-based approaches		
SANDMAN	Usage of an auto-adaptable discovery protocol	Cluster head consumes maximum energy
	Nodes can go to sleep and still be discoverable	Cluster heads act as bottlenecks
	Usage of various techniques for detecting if a device is unused or not	
SleepServer	PCs can go to sleep and still maintain network presence	Sleep servers act as bottlenecks
	Virtual Images allows flexibility and security	Sleep servers and state transitions consume energy
Sleepless in Seattle No Longer	PCs can go to sleep and still maintain network presence	Proxy server acts as bottlenecks and is a single point of failure
	A proxy frees machines from additional loads	Sleeping and energy policies left for users to define
GreenUp	PCs can go to sleep and still maintain network presence	Sleeping and energy policies left for users to define
	Software-only approach	
	Absence of bottlenecks or central server	

Table 2.1: Summary of rule-based and proxy-based approaches

Middleware Approach for Energy Management	Pros	Cons
Other approaches		
GRACE/2	Combines adaptations at different system levels	Requires a centralized global coordinator
	Flexible approach: infrequent and expensive global adaptations, and frequent and cheap local adaptations	Limited advantages in volatile environments
PARM	Efficient component migration and redistribution for energy optimization	Worst case execution time of $O(n^3)$
		Necessity of proxies, brokers and servers presence
	Works well for computation intensive applications	Moderate or negative gains in communication intensive applications
Energy Consumption Optimized Service Hosting	Energy savings while respecting SLAs and QoS	Requires many assumptions, limited to data centers
	Minimizing the number of running servers	Centralized algorithm, limited optimization criteria, no fault tolerance
ECOSystem	Proposes a currency model to quantify energy management	No energy optimization *per se*
	Allocation and scheduling following Time, Tasks and Devices dimensions	

Table 2.2: Summary of other middleware approaches

2.2.2 Comparison and Discussions

In order to compare the solutions reported in this review, we defined a taxonomy to describe the various properties of each middleware platform solution. Figure 2.1 summarizes this taxonomy. In particular, we introduce five comparison terms: *System Levels* which refer to the level of the system where the energy adaptation takes places (such as adapting hardware characteristics, or software components); *Applied Environment* which refers to the main user environment that the middleware platform targets (such as wireless sensor networks or

data centers); *Type* which refers to the type of the energy management approach (such as an adaptation algorithm, or an architectural pattern); *Degree of Autonomy* which refers to the autonomous nature of the energy management approach (such as using predefined algorithm or a rule-based approach); and *Sizing* which refers to the scalability of the approach (such as being applied to a limited or specific environment or a generic approach). In Table 2.3, we compare the middleware approaches we reviewed in this section based on the taxonomy we introduce.

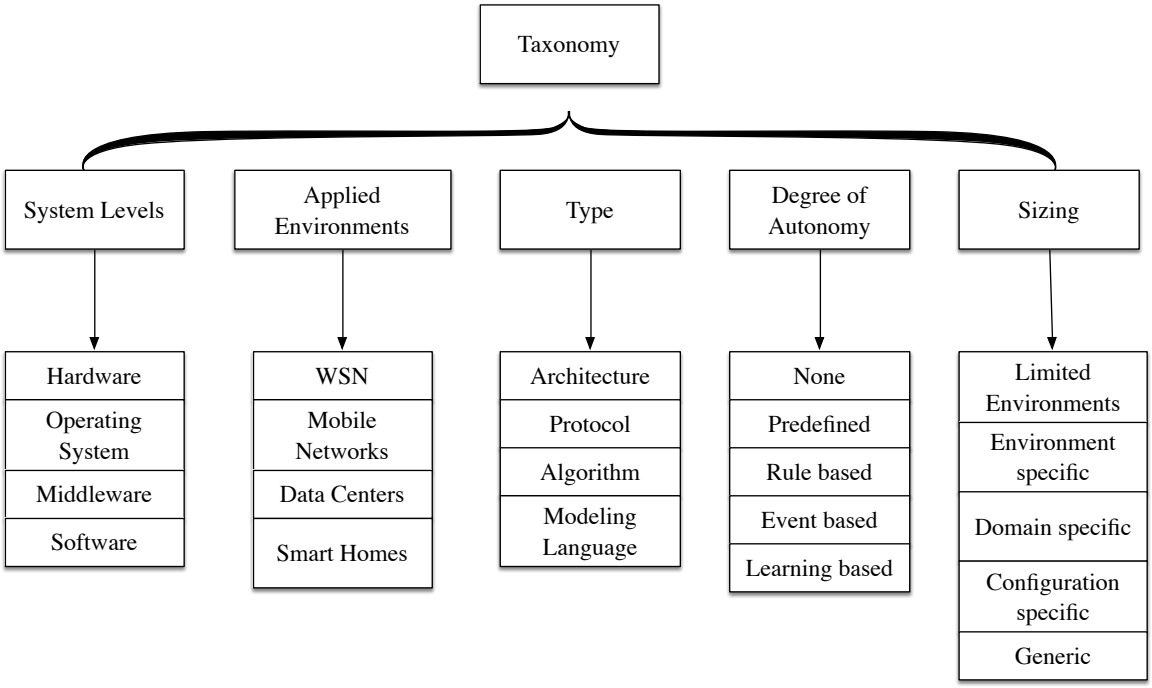


Figure 2.1: The comparison taxonomy.

System Levels

By this term, we refer to the level of the system where the energy adaptation takes place. We take into account the following system levels: Hardware, Operating System, Middleware, and Software. Most proposed solutions manage the software, hardware, or operating system levels in addition to the middleware level. Several solutions apply energy adaptations at the software level. Transhumance focuses on adapting the middleware platform, but the software is also adapted when the middleware platform adapts itself. GRACE applies multi-layer adaptations including per-application adaptations. Application-specific energy optimizations based on application classification are used in [XKYJ09]. While CasCap adapts software and services in mobile devices. DYNAMO targets video streaming scenarios (*e.g.*, software) while trying to perform quality of service tradeoffs. The approach in [BS09] uses

a dispatch algorithm to consolidate services in data centers. SANDMAN integrates several middleware platform techniques (middleware platform interfaces, service sessions or protocol selection), however it also manages the hardware level by techniques such as switching devices' power mode or grouping devices in clusters based on similar mobility patterns. SleepServer targets also the hardware and software levels by managing energy through the creation of virtual images in order to keep the network availability of computers. The approach in [RGKP10] and GreenUp both target the hardware and software levels by managing the sleep of machines and their network presence through the usage of hardware and software proxies. GRACE and ECOSystem manage energy consumption and adaptations also at the operating system level. GRACE can apply adaptations at an OS level granularity, while ECOSystem is built on the *currentcy* model and used in OS scheduling and allocation. Other approaches presented in this review adapt the system mainly on the middleware level, without major energy oriented adaptations in other levels of the system.

Applied Environments

This refers to the main user environments that the proposed middleware platform approach targets. Environments range from *Wireless Sensor networks* (WSN) to other mobile networks, to large-scale systems, and to data centers. Because we limited our study to distributed environments, the approaches presented in this chapter involve mobile networks or large-scale systems. Transhumance targets *Mobile Ad Hoc Networks* (MANETs). SANDMAN, on the other hand, targets larger environments: pervasive mobile networks and Wireless Sensor Networks (WSN). It seeks to reduce the energy consumption of idle devices in communicating networks. SleepServer, GreenUp and the approach in [RGKP10] are applied on computers in an enterprise environment (networked PCs, server availability). PARM also applies to mobile networks but its architecture and algorithmic approach are fit for large-scale systems. The approach in [BS09] is built for data centers. GRACE adopts a hierarchical approach for energy saving in different levels of computers (applications, all system layers). The principle of the approach can be applied to distributed systems in general. DYNAMO's approach follows some of GRACE's ones, in particular the usage of different information layer to perform end-to-end adaptations. DYNAMO, however, targets mobile devices. ECOSystem manages energy at the Operating System (OS) level, particularly managing computer tasks using a new energy abstraction mixing energy with currency: *currentcy*. CasCap and [XKYJ09] target applications in mobile devices running using network services or devices.

Type

This defines the type of the energy management approach. Types can be architectural, protocol, algorithm, or a modeling based. Architectural approaches are middleware platforms or frameworks that propose energy optimization techniques as architectural solutions. Protocol approaches refer to the cases where an energy efficient protocol is proposed in the middleware layer. Algorithmic approaches refer to the cases where the latter is constructed around

an energy-aware algorithm in the middleware layer. Modeling approaches are when a model is proposed for energy-awareness in the middleware layer. Most approaches presented in this review are architectural-based, with few exceptions: SANDMAN and PARM, in addition, propose protocol-based and algorithmic approaches, respectively. In SANDMAN, a self-adaptive discovery protocol is used as the basis of the energy management approach. PARM uses an algorithm to determine the components to migrate. This algorithm is crucial for the PARM framework. For the other approaches, [BS09] is a dispatch algorithm for data centers, and ECOSystem's approach is based on a new modeling definition: *currentcy*.

Degree of Autonomy

This indicates the degree of autonomy of the energy management approach (and not the overall degree of autonomy of the middleware platform itself). Autonomic computing refers to "*computing systems that can manage themselves given high-level objectives from administrators*" [KC03]. These systems are self-manageable where this *self-management* encompasses four main aspects: *self-configuration*, *self-optimization*, *self-healing*, and *self-protection*. They are formed by *autonomic elements*, which contain resources and deliver services. We choose several keyword expressions to describe the degree of autonomy. The expressions are: *i) None*, where the approach is not autonomous at all. This can be a system that applies energy adaptations through question/answer interactions with the user. *ii) Predefined*. In this case, the system uses predefined strategies, such as an algorithm, a protocol, static rules or a finite-state automaton. *iii) Rule-based*. Here, energy management adaptation is based on rules (or adaptive policies) that can be added and modified by an external entity (user, administrator). *iv) Event-based*. Systems that use an (complex) event processing engine falls in this category. The system collects data (context or energy data) and takes an adaptation decision as a result of events processing. And *v) Learning based*. Here, energy management evolves by learning from context and energy information, and from the user habits. A level of artificial intelligence is also required for systems to be considered in this category.

Transhumance uses adaptation policies for middleware platform energy adaptations, and adaptation profiles for applications. Both techniques are based on a rule-based approach with conditions on the energy levels (local to one node or global to the network) and adaptation actions. The solutions proposed in [XKYJ09] and in [XHSYJ11] (CasCap) uses also adaptive policies for energy management. DYNAMO is also based on rules and on policies that can take the shape of utility factors. ECOSystem proposes a modeling definition, *currentcy*, and allocation and scheduling approaches that are predefined. PARM and [BS09] are based on algorithms that are defined prior to the execution of the system. SANDMAN uses an adaptable protocol and different predefined approaches for its energy awareness. SleepServer, GreenUp and [RGKP10] use a predefined approach allowing computers to go to sleep while preserving their network presence. GRACE is build around a multi-layer architecture. It is based on global and per-application algorithms to apply adaptations, and profiling for resource usage predictions, all of which are predefined prior to the execution of the system.

Sizing or Scalability

This describes the scale of the proposed approach. This means, how well can we apply the approach on different environments other than the ones that were presumably defined for. We use several concepts for this taxonomy: *i) Limited Environment*, where the approach can be applied to an environment with very specific conditions. *ii) Environment-specific*. The approach is limited to one or few environments (such as approaches limited to WSN). This taxonomy is a subset of the wider domain-specific taxonomy. *iii) Domain-specific*. Here, the scope is wider than environment-specific. For example, an approach that can be applied on WSN and other mobile networks, fits in this taxonomy. *iv) Configuration-specific*, where the approach can be sized if a specific configuration is met, regardless of the applied environments or domains. And *v) Generic*. Here, the approach can be scaled on different domains, and thus can be considered as generic enough for a high degree of sizing.

Most of the reviewed approaches are environment-specific or domain-specific, with two more specific solutions: Transhumance, which requires additional preconditions, and PARM that has a wide domain scope but requires a specific configuration. Transhumance targets MANets with a maximum of 20 nodes and moving at a pedestrian speed (up to 5 km/h), which makes this a limited environment. ECOSystem emphasizes on the various computer tasks and networks. Thus, these approaches are domain-specific, same for SANDMAN, GRACE and [XKYJ09]. SANDMAN targets pervasive networks in general, including WSN. SleepServer, GreenUp and [RGKP10] are specific to computer network within an enterprise environment (where servers are available to host virtual images for desktop PCs). The approach in [XKYJ09], CasCap and DYNAMO are applied on mobile networks, while GRACE targets the very wide scope of distributed environments. Finally, PARM can be applied on different domains, from mobile networks to large-scale systems. However, it requires a specific configuration with the presence of proxies, brokers and servers.

Discussions

Based on our comparison we discuss how well the reviewed solutions fit in an area of research where contributions are still needed for a fully autonomous middleware platform for energy management. Most of the solutions presented in this review follow two main approaches of autonomy: *rule-based* and *proxy-based* approaches, while the others use *predefined* techniques.

Rule-based approaches offer an high degree of architectural autonomy, but with a limited decisional autonomy. The architecture of the middleware platform is flexible and evolutive, and can easily cope with changes in the environment. These architectures are based on autonomic control loop design [KC03], with subsystems designed to monitor, analyze, plan and execute energy-aware adaptations. Rules, on the other hand, need to be predefined

Name	System Levels (+ Middleware)	Applied Environments	Type	Degree of Autonomy	Sizing (Scalability)
Transhumance	Software	MANets	Architecture	Rule-based	Limited Environment
SANDMAN	Hardware	Pervasive Mobile Networks, WSN	Architecture, Protocol	Predefined	Domain-specific
PARM	–	Mobile Networks, Large Scale Systems	Architecture, Algorithm	Predefined	Configuration-specific
GRACE/2	Operating System, Software	Distributed Systems	Architecture	Predefined	Domain-specific
Middleware for Energy-awareness in Mobile Devices	Software	Mobile Networks	Architecture	Rule-based	Domain-specific
Energy Consumption Optimized Service Hosting	Software	Data Centers	Algorithm	Predefined	Environment-specific
ECOSystem	Operating System	Computer Tasks	Modeling	Predefined	Domain-specific
CasCap	Software	Mobile Networks	Architecture	Rule-based	Domain-specific
DYNAMO	Software	Mobile Networks	Architecture	Rule-based	Domain-specific
SleepServer	Hardware, Software	Desktop PCs, Enterprise Networks	Architecture	Predefined	Domain-specific
Sleepless in Seattle No Longer	Hardware, Software	Desktop PCs, Enterprise Networks	Architecture	Predefined	Domain-specific
GreenUp	Hardware, Software	Desktop PCs, Enterprise Networks	Architecture	Predefined	Domain-specific

Table 2.3: Comparative table of middleware platform solutions for energy management

and updated on environment's evolutions. None of the rule-based middleware platform approaches [PMND07, PlDR08, DPHu+08, XKYJ09, XHSYJ11, MDNV07] incorporate conflict management strategies. Transhumance [PMND07, PlDR08, DPHu+08], CasCap [XHSYJ11] and the approach in [XKYJ09] all use *Event-Condition-Action* rules. The rule selection strategy is therefore simply a condition checked when an event occurs, and actions are applied regardless of their impact on the system or the conflicts that they may produce. The creation and the update process of these rules are left to the user, administrator or another application. The latters are responsible for verifying the consistency of the rules and the absence of rules conflicts and incompatibilites. DYNAMO [MDNV07] follows a similar process, although the usage of utility functions is complementary to ECA rules. This allows better modularity and flexibility in rule creation and evolution, but also allows the rule selection process to alleviate some of the conflicts. Conflicts may be prevented by using different rule selection strategies, such as fuzzy logic [Zad65]. Or they can be dealt with a conflict management process based on system knowledge (through, for example, machine learning techniques), crowd-sourcing or modeling of the impact of applied actions. The distributed nature of the environment gives developers the opportunity to utilize the processing power and knowledge repositories that lies in the cloud. Finally, Transhumance is the only approach here where rule actions are applied on the middleware platform modules only. The other approaches apply their rule actions on the managed system (software, system or hardware parameters).

Proxy-based approaches have the main advantage of keeping a device *available* for the user (or for other devices), while the device is in sleep mode. Therefore, this approach achieves both energy savings and limited disruption of the availability of the device. The reviewed proxy-based approaches [SB07, SHB08, ASG10, RGKP10, SLH+12], although sharing a similar proxy functioning, differ in a key aspect : the proxy. In SANDMAN [SB07, SHB08], nodes regroup in clusters following similar criteria, then elect one of their own as a cluster head (or proxy). The cluster head is then responsible, not just for maintaining the presence of its nodes in the network, but also in deciding when a node is unused and thus can be put to sleep. On the contrary, the proxy in SleepServer [ASG10], GreenUp [SLH+12] and in [RGKP10], is only responsible for keeping the availability of their sleeping machines in the network. The proxy does not incorporate autonomic or intelligent functioning in relation to deciding when a machine should go to sleep. The nature of the proxy in the previous 3 approaches is different: *i)* a virtual machine image for SleepServer and unique to its original machine. Therefore, proxies are not shared as each machine have a unique proxy in the form of its virtual image. *ii)* an independent hardware proxy responsible for managing all machines in [ASG10]. And *iii)* in GreenUp [RGKP10], a proxy is a normal machine that is elected using a distributed management technique. Although not similar, the selection process share some principles with SANDMAN's selection process.

The other approaches reviewed in this study [SYH+04, VYI+09, MV03, BS09, ZEL05] have little autonomic functioning. They all use predefined techniques to manage energy, such as algorithmic adaptations and prediction algorithms (PARM [MV03], GRACE [SYH+04, VYI+09] and [BS09]) or energy models (ECOSystem [ZEL05]).

Although these approaches offer a certain degree of autonomic management for energy consumption, a full autonomous energy management is yet to be defined. Autonomous context learning approaches for energy management at the middleware are missing. An energy management autonomous approach should therefore imitate the human body metabolism [1]: the platform needs to be transparent to the user and to devices and applications, but without limiting users' high-level decisions. In the human body, when energy becomes low, the system starts by using its reserves and notifying the human about the situation (*e.g.*, the human feels hunger). Therefore, the human could apply high-level decisions, such as eating (to recharge his energy and reserves), or reduce his activity, or go to sleep (low power mode). We therefore believe that middleware platform approaches should take inspiration from biologic systems and provide a similar autonomous functioning for energy-awareness because the complexity of systems is rapidly increasing. Autonomous functioning means that users, in particular home users, do not need to do the job of a system administrator in managing the energy consumption of their software and devices. The absence of a system administrator, albeit for managing energy, is a challenge for ubiquitous computing, in particular in smart homes [EG01]. Energy, as a non-functional requirement, needs, more than other business logics, to be less disruptive for users, thus the need for autonomous energy management.

In addition to autonomic functioning, all the reviewed approaches are specific to a domain or to an environment. Transhumance requires very specific conditions on the environment (small networks of up to 20 nodes and moving at pedestrian speed of up to 5 km/h). The platform focuses mainly on adapting the middleware platform itself with limited adaptability for applications. On the contrary, PARM requires a specific configuration (the presence of proxies, brokers or directory services), but with a wider domain scope (*e.g.*, mobile networks, large scale systems). The algorithm presented in [BS09] can only be applied on data centers, thus is environment-specific. Similar to the previous approach, ECOSystem proposes a currency definition and a framework that targets the energy consumption at the OS level. SANDMAN targets energy consumption of idle devices in a communicating network, SleepServer, GreenUp and [RGKP10] manage energy consumption of idle devices in an enterprise network, while the architecture in [XKYJ09] is valid for mobile networks. Finally, GRACE combines adaptations at different levels of the system in order to optimize the energy and resource utilization in a distributed environment. We believe that middleware platform approaches should allow some degree of scaling, not just inside a specific domain (such as the proxy-based approaches in enterprise networks [ASG10, RGKP10, SLH+12]), but to a wider scope of applications. In the diversity of devices, networks and environments, an ideal middleware platform for energy management should be able to manage energy for multiple domains, ranging from mobile devices and desktop machines to distributed software in data center.

But in order to efficiently manage energy consumption in software and devices, the said energy need to be calculated. In the next sections, we study the energy measurements and estimation approaches for software and hardware.

[1] The human body metabolism is an analogy commonly used in autonomic computing.

2.3 From Managing to Measuring Energy of Middleware and Software

We argue that energy optimization at the middleware layer is best suited for large-scale energy reduction in our modern context. Devices are more and more heterogeneous and vary in characteristics and utilization. From mobile devices, to servers and data centers, and to desktop and laptop computers, the range of hardware and its execution domain vary greatly. Managing energy per device or for a specific execution context or platform provides more precise energy management but with the cost of lower scalability. These types of approaches are limited, specially with the advent of cloud computing and distributed environments.

Although managing energy for this diversity of hardware and software at the middleware layer is a good choice, energy should be accurately measured at the software layer. Distributed devices would send the energy consumption of their software to the middleware layer. The latter would then take energy-related adaptation decisions. Energy management and optimization need software energy measurement, whether at the middleware layer or at lower layers, such as at hardware, software or even at code level. Middleware approaches for energy management require in-depth knowledge of software and hardware in order to efficiently manage energy and optimize its consumption.

Our study of the energy management approaches at the middleware layer shows that two groups define these approaches: *rule-based* management, and *predefined* management. Each has its own advantages and drawbacks as we discussed in the previous section. However, both approaches share a common drawback: the lack of energy metrics. The first component of the MAPE-K autonomic control loop [KC03] is *monitoring*. Therefore, achieving autonomic energy management of middleware and software requires accurate energy monitoring of the system.

Although existing approaches achieve energy adaptations based on non-energy metrics (mostly using resources utilization, and usage assumptions), these adaptations could be improved by using direct and accurate energy measurements. What if, instead of using the CPU percentage usage of an application, an energy management middleware would use the real energy consumption of the application on the CPU hardware? The same logic applies to other hardware resources, such as the network card or hard disk.

Providing direct energy measurements does not only offer more accurate monitoring and management, but also it allows more autonomic management. *Rule-based* and *predefined* approaches use common software assumptions about energy consumption in their adaptations. These assumptions are based on developers' experience and are limited to certain contexts. Aiming for full autonomic management requires separation of contexts and configurations' logics within the autonomic management platform. Having direct energy metrics allows achieving this separation of concerns for energy related managements.

In the next section, we study the state-of-the art approaches in energy measurement, estimation and modeling. We discuss approaches for measuring the energy consumption of

hardware components and software applications, and report the limitation of the existing work. Finally, we propose criteria for efficient energy measurement approaches based on the limitations of the state-of-the art solutions and on the requirements of measuring energy for software.

2.4 Energy Measurement Approaches

Managing energy at any system level while providing a minimum accuracy requires measuring the energy available and consumed. In particular, monitoring or estimating the energy and/or resources consumption of hardware and software is a *sine qua non* condition for energy management at a higher level. This understanding of energy is however rudimentary [Kan09], but also depends on hardware, software and execution context. New power models taking into account both computation and power management are therefore needed. Ultimately, systems should be designed to be energy adaptive (*e.g.*, being able to adapt their behavior depending on energy concerns) not just energy efficient [Kan09].

Distributed systems add an additional layer of complexity in measuring energy. Energy efficiency can be improved by considering the end-to-end energy use of a task in all involved systems [LQBC11]. Metrics should take into consideration that energy consumption affects and is affected by other factors such as reliability, performance. Therefore, new metrics, models and new measurement techniques are needed to support scientific evaluation of end-to-end energy management [LQBC11].

Monitoring energy consumption of hardware components usually requires a hardware investment, like a multimeter or a specialized integrated circuit. For example in [MSK07], the energy management and preprocessing capabilities is integrated in a dedicated ASIC (*Application Specific Integrated Circuit*). It continuously monitors the energy levels and performs power scheduling for the platform. However, this method has the main drawback of being difficult to upgrade to newer and more precise monitoring and it requires that the hardware component is built with the dedicated ASIC, thus making any evolution impossible without replacing the whole hardware.

On the other hand, an external monitoring device provides the same accuracy as an ASIC circuits and does not prohibit energy monitoring evolutions. Devices, such as AlertMe Smart Energy [Ale], monitor home devices and allow users to visualize their energy consumption history through application services, such as the now defunct Google Powermeter [Goo].

The previously mentioned monitoring approaches allow getting energy measures about hardware components only. However, knowing the energy consumption of software services and components requires an estimation of that consumption. This estimation is based on formulas such as the ones presented in [SMM07] and [KZ08].

Next, we details the main state of the art approaches for energy modeling and energy measurement and estimation.

2.4.1 Energy Modeling

Estimating the energy consumption of hardware and software is often achieved through modeling resource usage for energy information. In this section, we outline the main approaches for energy models at software and middleware layers.

Energy Cost of Software

In [SMM07, SMM08], the authors propose formulas to compute the energy cost of a software component as the sum of its computational and communication energy costs. For a Java application running in a virtual machine, the authors take into account the cost of the virtual machine and eventually the cost of the called OS routines. The energy cost of a software component is calculated based on the following formula:

$$E_{component} = E_{computational} + E_{communication} + E_{infrastructure} \tag{2.1}$$

where $E_{computational}$ is the computational cost (*i.e.*, CPU processing, memory access, I/O operations), $E_{communication}$ is the cost of exchanging data over the network, and $E_{infrastructure}$ is the additional cost incurred by the OS and runtime platform (*e.g.*, Java VM).

More specifically, the *computational energy cost* of a component is determined as the computational energy cost of its interfaces (in component-based software engineering sense). The latter is calculated as the aggregation of the energy costs of executing its bytecodes, native methods and the cost of threads synchronization (via a monitor mechanism in the Java Virtual Machine). *Communication energy cost* is calculated based on the size of transmitted and received data while accounting for the cost of transmission/receiving a unit of data. The authors rely on previous research [FN01, XLWN03] to assert their argumentation that *the energy consumption of wireless communication is directly proportional to the size of transmitted and received data* [SMM08]. Finally, *infrastructure energy overhead cost* is calculated as the energy cost of the garbage collector thread, process scheduling, context switching, and paging.

In [KZ08], the authors take into account the cost of the *wait* and *idle* states of the application (*e.g.*, an application consumes energy when waiting for a message on the network). The following model is proposed:

$$E_{App} = E_{Active} + E_{Wait} + E_{Idle} \tag{2.2}$$

where E_{Active} is the energy cost of running the application and the underlying system software, E_{Wait} is the energy spent in wait states (when a subsystem is powered up while the application is using another), and E_{Idle} is the energy spent while the system is in idle state. This general model is also derived for the CPU using the following formula:

$$E_{CPU} = \{p_{Active} \times f_{Active} + P_{Idle} \times (1 - f_{Active})\} \times T \tag{2.3}$$

where P_{Active} is the power consumption of the CPU in active mode, P_{Idle} is the power consumption in idle mode, f_{Active} represents the CPU percentage time spent on running the application, T is the time spent running the application workload.

In [TT13], the authors use sensors between the power source and the system in order to measure its energy consumption. The sensors capture at regular intervals the power line conditions, such as voltage and current. The captured information is then stored in a central data collection server. After application execution, the readings from both sensors and application are correlated and energy consumption is estimated using the following power model:

$$E = \int_T P_S dt - P_I T \tag{2.4}$$

where P_S is the instantaneous power profile of the system, T is the execution time and P_I is the idle power of the system. The idle power is calculated when the system is idle while a minimum number of applications are running.

Energy aware middleware

In addition to the previous approaches, Petre [Pet08] proposes an energy-aware model for the MIDAS middleware platform language [PSW06]. The author proposes to model energy-awareness using the MIDAS middleware platform language [PSW06]. MIDAS is a resource-centric language based on a previous framework developed also by the author [PSW00] for location-aware computing. The framework defines a language for topological action systems, which is used for resource notation. The language assists the network manager on issues like resource accessibility and mobility, replicated resources and node failure and maintenance.

The author models data resources, code resources, and computation unit resources (a combination of data and code). A resource is defined as a unit that has a location and other properties. The location of these resources is modeled as a node of a network. The author distinguishes two networks: the electricity network containing the electricity sockets (modeled as electricity resources or energy supply); and the resource network of devices and resources.

In this model, energy is defined as a quantity that is consumed by the hardware devices (and indirectly by the software). The author considers that data resources and their storage do not consume energy. However, writing and reading data do consume energy. Code resources, on the other hand, need hardware to execute on. And hardware needs a power supply to work. Therefore, the author distinguishes three computation units: software or code (the unit that requests energy to run), hardware (the unit that consumes energy in order to execute the code), and electrical socket (the energy provider).

The author gives an example scenario of a user walking in a city with a mobile phone and interacting with context and location-aware elements: a statue that sends multimedia

information about itself, and a restaurant that sends an SMS about its menu and price. The example is modeled using the energy-aware additions to the MIDAS language. For example, the phone will have energy and functionality variables, as well as action modeling for charging using the electric sockets. Actions to apply when receiving an SMS or a video message are also included in the phone modeling.

Adding energy awareness in the MIDAS middleware platform language allows a uniform approach to modeling resources in a network. This is done by having energy modeled using the same formalism of network nodes, location or other properties. However, this modeling does not offer tools to optimize or reduce the energy consumption directly. Instead, it provides a modeling infrastructure that helps in managing energy-aware applications and networks.

Energy consumption estimation based on workload in servers

In [LGT08], the authors propose a model for estimating the energy consumption of servers. For that, they use hardware performance counters (collected through software and operating system tools), and experimental results. A linear regression model is also proposed for predicting the energy consumption of computer jobs.

In particular, the total energy consumed by the system for a given workload is calculated using the following combined model:

$$E_{system} = \alpha_0(E_{proc} + E_{mem}) + \alpha_1 E_{em} + \alpha_2 E_{board} + \alpha_3 E_{hdd} \tag{2.5}$$

where α_0, α_1, α_2, and α_3 are constants that are determined using experimental results on a given server architecture (*e.g.*, linear regression analysis).

E_{proc} is the energy consumed by the processor, E_{mem} the energy consumed by the DRAM memory, E_{em} the electromechanical energy, E_{board} the energy consumed by the support chipsets, and E_{hdd} is the energy consumed by the hard drive while operating.

The energy consumption of these resources is calculated using resource-specific models. For example, the energy consumption of the hard disk is the sum of the power required to spin the disk, the idle power, and the power to read and write data. The model is thus represented using the following formula:

$$\begin{aligned} E_{hdd} = P_{spin-up} \times t_{su} + P_{read} \sum N_r \times t_r \\ + P_{write} \sum N_w \times t_w + \sum P_{idle} \times t_{idle} \end{aligned} \tag{2.6}$$

where $P_{spin-up}$ is the power required to spin-up the disk from 0 to full rotation, t_{su} is the time required to achieve spin-up, P_{idle} the power consumed by the disk when in idle, P_{read}

and P_{write} are the power consumed per kilobyte of data read and write from the disk, and N_r the number of kilobyte read or written.

E_{mem} is calculated using a combination of the counts of highest level cache misses in the processor combined with the read/write power and the DRAM memory activation power. E_{em} is calculated based on the energy consumed by the cooling fans and the optical drives. E_{board} uses probe based measurements to calculate the energy required by the support chipsets. Finally, the processor energy E_{proc} is calculated as a function of its workload. The workload manifests by the CPU core temperature and the ambient system temperature. The temperature is measured using *ipmitool* [ipm] through sensors in the path of the outgoing airflow from the processor.

2.4.2 Energy Measurement and Estimation

Managing and optimizing energy consumption in software while providing a minimum accuracy requires measuring the energy available and consumed. In particular, monitoring or estimating the energy and/or resources consumption of hardware and software is a *sine qua non* condition for energy management at a finer grain. In this section, we review the main approaches and tools for measuring and estimating the energy consumption of software systems.

PowerScope

In [FS99], the authors propose a tool, *PowerScope*, for profiling energy usages of applications. This tool uses a digital multimeter to sample the energy consumption and a separate computer to control the multimeter and to store the collected data. PowerScope can sample the energy usage by process. This sampling is more accurate than energy estimation, although it still requires a hardware investment.

In particular, *PowerScope* maps energy consumption to program structure. It can therefore determine the energy consumed by a specific process, and even down to the energy consumption of different procedures within the process. The implementation of the tool uses statistical sampling of both the power consumption and the system activity. The tool generates an energy profile that is analyzed *post mortem*. Thus, the tool has no profiling overhead, but with the price of no online values. During the sampling, a multimeter is used to sample the current drawn of the profiled computer. A separate computer is also used to store the collected information and controls the multimeter (although this can also be done on the same computer).

In more details, *PowerScope* uses three software components:

1. a *System Monitor* that samples system activity by using a user-level daemon and OS kernel modifications. The monitor records the value of the program counter (PC) and

the process identifier (PID). It also records, through instrumentation, additional system information such as the pathname associated with executing processes, or the loading of shared libraries.

2. an *Energy Monitor* that runs on a separate machine and collects current samples from the multimeter. The latter transmits asynchronously the current samples to the monitor where they will be stored.

3. and an *Energy Analyzer* that uses the collected data by the system monitor and the energy monitor to generate the energy profile of the system activity. Energy usage is calculated using the formula in equation 2.7, and the analyzer then generates a summary of energy usage per process. The analyzing process is done offline after the execution of the program.

$$E \approx V_{meas} \sum_{t=0}^{n} I_t \Delta t \tag{2.7}$$

where E is the total energy over n samples using a single measures voltage value V_{meas}, I_t is the current and Δt is the interval of time.

Using their tools on adaptive video scenarios, the authors managed to obtain a 46% reduction in total energy consumption when applying video compression, smaller display size, network and disk power optimizations. However, the tool is relatively old *i.e.* 1999. Many modern hardware, operating system and software energy management techniques were not available more than a decade ago. On a modern system the energy reduction may be lower than the number the authors got in their research.

pTop

pTop [DRS09] is a process-level power profiling tool. Similar to the GNU/Linux *top* program [Linc], the tool provides the power consumption (in Joules) of the running processes. For each process, it gives the power consumption of the CPU, the network interface, the computer memory and the hard disk. The tool consists in a daemon running in the kernel space and continuously profiling resource utilization of each process. It obtains these information by accessing the */proc* directory [Mou01]. For the CPU, it also uses Thermal Design Power (TDP) – which is the maximum amount of power the cooling system requires to dissipate – provided by constructors in the energy consumption calculations. It then calculates the amount of energy consumed by each application in a *t* interval of time. It also consists of a display utility similar to the Linux *top* utility. A Windows version is also available, so called *pTopW*, and offers similar functionalities, but using Windows APIs.

pTop's energy model is a sum of the energy consumed by individual resources in addition to energy consumed by the interaction of these resources. The following formula

presents the energy consumed by an application E_{appi}:

$$E_{appi} = \sum U_{ij} \times E_{resourcej} + E_{interaction} \tag{2.8}$$

where U_{ij} is the usage of application i on resource j, $E_{resourcej}$ is the amount of energy consumed by resource j, and $E_{interaction}$ is the indirect amount of energy consumed by the application because of the interaction among system resources, in the time interval t.

The authors also propose a general model for energy consumption of a particular resource. The model is a function of the states (*e.g.*, read, write) and transitions of the resource. The formula is as follows:

$$E_{resourcej} = \sum_{jinS} P_j t_j + \sum_{kinT} n_k E_k \tag{2.9}$$

where S defines the states of the resource j, T its transitions, P_j the power consumed by resource j in a time interval t, n_k the number of transitions k, and E_k is the energy consumed by this transition.

From the general model, resource specific models can be generated. For the CPU the formula is therefore:

$$E_{CPU} = \sum_{j} P_j t_j + \sum_{k} n_k E_k \tag{2.10}$$

where P_j and t_j are the power consumption and the time the processor running at a particular frequency, respectively; n_k is the number of times transition k occurs, and E_k is the corresponding energy of that transition. This calculation is based on an assumption that CPU energy is proportional to the process CPU time.

For the network interface:

$$E_{Neti} = t_{sendi} \times P_{send} + t_{recvi} \times P_{recv} \tag{2.11}$$

where t_{sendi} and t_{recvi} are the amount of time process i sends and receives packets, P_{send} and P_{recv} are the power consumption of the wireless card at sending and receiving states.

For the hard disk:

$$E_{Diski} = t_{readi} \times P_{read} + t_{writei} \times P_{write} \tag{2.12}$$

where t_{readi} and t_{writei} are the amount of time process i writes to the disk and reads from the disk, P_{read} and P_{write} are the power consumption of the disk writing and reading states.

The authors tested their model using their process-level profiling tool. The average median error is less than 2 Watts when compared to direct energy values by a wattmeter (in their case a Watts Up Pro meter [wat]) in a random workload sample taken every 10 seconds. The tool's overhead is relatively low, although not negligible, at 3% of the CPU and 0.15% of memory in a 1 second sampling interval of more than 60 processes running in the systems.

Other Energy Tools

In addition to the previous approaches, other tools offer energy information. We present here a selection of other major energy measurement tools in the remainder of this section.

PowerTop PowerTop [powc] is a Linux tool to *diagnose issues with power consumption and power management*. It reports an estimation of the energy consumption of software applications and system components. It also offers an interactive mode where users can apply different power management settings not enabled by default in the Linux distribution. PowerTop therefore shares similarities with pTop but also limitations such as the lack of fine-grained results.

Energy Checker Energy Checker [ene] is an SDK provided by Intel and offers function for exporting and importing counters from an application. These counters measure the time spent for a particular event or process, such as reading a file, or converting a video. The counters are then used to estimate the power consumption of the application. However, the power estimation requires a additional powermeter, thus limiting the flexibility of the approach.

Joulemeter Joulemeter [jou, RGKP10] is a software tool that estimates the energy consumption of hardware resources and software applications in a computer. It monitors resources usage, such as the CPU utilization or screen brightness, in order to estimate the energy consumption of these resources. Joulemeter uses machine specific power models for hardware configuration. Their current model takes into account processor Pstates, power utilization, disk I/O levels and whether the monitor is turned on or off. The models, however, are learned through calibration. This draws a limitation in term of flexibility as power models cannot be estimated without previous laboratory benchmarks.

Other System Tools

In addition to *pTop*, several utilities exist on Linux for resource profiling. For example, *cpufrequtils* [The] which consists of multiple tools such as *cpufreq-info* to get kernel information about the CPU (*i.e.*, frequency), and *cpufreq-set* to modify CPU settings such as the frequency. *iostat* [Linb] that is used to get devices' and partitions' input/output (I/O) performance information, as well as CPU statistics. Other utilities [Git] also exist with similar functionalities, such as *sar*, *mpstat*, or the system monitoring applications available in Gnome, KDE or Windows. However, all of these utilities only offer raw data (*e.g.*, CPU frequency, utilized memory) and do not offer direct energy information. These raw data can, nevertheless, be used to feed power models with information needed for estimating the energy consumption.

Application Profiling Tools

Several open-source or commercial profiling tools already propose some statistics of applications. Profilers are generally programming language dependent. GNU gprof [gpr] and C Profiler [cpr] as an example of profilers in C. For .NET languages, profilers exist such as ANTS Performance profiler [ant], AQtime Pro [aqt] or SlimTune [sli]. In Java, tools such as VisualVM [Vis], *Java Interactive Profiler* (JIP) [jip], JProfiler [ej-], or the Oktech Profiler [OKT], offer coarse-grained information on the application and fine-grained resource utilization statistics. However, they fail to provide energy consumption information of the application at the granularity of threads or methods. For example, the profiler of VisualVM only provides self wall time (*e.g.*, time spend between the entry and exit of the method) for its instrumented methods. However these tools do not provide network related information, such as the number of bytes transmitted by methods and thus the energy consumed.

2.4.3 Discussions

Although many approaches exist for measuring various resources metrics, energy metrics are still lacking. Few approaches offer energy models or tools for calculating the energy consumption of software or hardware.

Energy measurement nowadays can be grouped into three categories: hardware measurement as for example in [MSK07, Ale], power models as in [SMM07, KZ08, Pet08, LGT08, TT13], and software measurement (as in many tools [DRS09, powc, ene, jou, FS99, cpr, gpr, aqt, sli, jip, ej-, OKT, Vis]).

Hardware measurement offers high precision but at a coarse-grained level. It also requires, as it name states, additional hardware whether embedded or not. The main limitation of such approach is the inability for evolution and the difficulty to scale.

Power models provide models to calculate or estimate the energy consumption of hardware and software. Models are either too generic and coarse-grained [Pet08, KZ08], or platform dependent (in particular Java) [SMM07, SMM08]. Tools based on energy models suffer also from platform dependency [DRS09, FS99, LGT08]. The model in [LGT08] offers a combined model to calculate the energy consumption of the system. However, their resources-specific models varies from fine-grained software-based models, such as the hard disk energy model, to coarse-grained hardware-based models, such as for the processor. The model presented in [LGT08] uses statistical methods in their formulas, thus a tradeoff arises between precision and software overhead.

The most promising approach in software measurement is energy application profiling. Profilers help in understanding the system and decomposing the energy consumption of each resource. For example, in [CH10], the authors determined that on an Openmoko Neo Freerunner mobile phone [fre], the GSM module and the display (LCD panel, touchscreen, graphics accelerator and driver, and backlight) consume the majority of power. Still, current approaches are either coarse-grained (provide energy values at the process level) as in

[DRS09, FS99], or profile some system resources without providing energy values such as [cpr, gpr, aqt, sli, jip, ej-, OKT, Vis]. *PowerScope* [FS99] does not offer energy information in real time unlike *pTop* [DRS09]. Similar to a number of other profilers, *PowerScope* collects resources information at runtime then calculates energy values offline at a later stage of the measurement. The advantage of real time solutions such as *pTop* is the ability for adaptive middleware platforms to use energy measurements for runtime energy-aware adaptations. *PowerScope* also requires hardware investment in the form of a digital multimeter while *pTop* provides similar per-process energy information using only software means.

Software profilers use software statistical sampling or software code instrumentation. Both approaches have advantages and limitations [LV99, Pro]. Instrumentation offers two main advantages: *i) accuracy* where exact resources values are provided; and *ii) repeatability* as bytecode instrumentation produces similar results with the same environment and parameters. Sampling, on the other hand, *i)* have a lower overhead as it only occurs at sampling intervals (unlike instrumentation where the overhead is permanent); and *ii)* does not require application source (or byte-) code modification. Although bytecode instrumentation has a non-negligible overhead for very large applications, we argue that supporting precise and accurate per-method energy profiling is suited for diagnosing energy leaks in applications.

Table 2.4 presents a general comparison between the main energy measurement approaches studied in this chapter.

Based on our review of state-of-the-art approaches, we argue that work still need to be done for accurate and invisible energy measurement approaches. New metrics and models on both system and software levels need to be defined. These new measurements should adopt in our opinion the following criteria:

1. *Accurate measurements*. Energy consumption measurement is key for energy-aware adaptations. On higher system levels (middleware and software), more accurate measurement provides better information for relevant energy management and adaptation. Measurements at a finer-granularity need to be defined, not only by providing system resources values, but rather by providing fine-grained energy consumption values for applications.

2. *Fine-grained power models*. Energy models and formulae need to be precise enough to offer energy consumption values at finer-granularity. State-of-the-art software and middleware platform models have either energy precision limitations (providing coarse-grained energy values), or provide finer-grained resources (not energy) values. We argue that finer-grained power models, without unnecessary mathematical or architectural complexity, are needed for better energy measurements.

Name	Energy Measurement	Monitored Resources	Energy Precision	Operating System Modification	Software Modification	Hardware Investment
ASIC, Powermeter	✓	Hardware Energy	Hardware	×	×	✓
OS utilities	×	Hardware and OS Resources	×, but Hardware, Software, Process for Other Resources	×	×	×
Software Profilers	×	Software Resources & Parameters	Software, Classes & Methods	Depends on Profiler	Depends on Profiler	×
PowerScope	✓	Current, Program Counter	Process, Procedure	✓	×	✓
pTop	✓	Hardware Resources (CPU, Disk, Network)	Process	× for CPU, ✓ for Network	×	×
PowerTop	✓	Hardware resources	Application	×	×	×
Energy Checker	✓	Hardware resources, Application counters	Application	×	✓ through counters	✓
JouleMeter	✓	Hardware resources	Process	×	×	×

Table 2.4: Comparative table of energy measurement approaches

3. *Reduce user experience impact*. Adding an additional layer of computation in order to measure energy consumption has a non-negligible impact on user experience. Approaches implementing energy models and formulae need to be invisible for the user, the application and the underlying system. Therefore, tools need to have low or negligible overhead (in particular in term of time and energy impact). The scalability and evolution (in addition to practical usage) of the system is greatly impacted with additional hardware. Thus, no additional hardware investment needs to be used for energy measurement. Finally, measurement tools should not require the manual modification of applications source code. We need to measure legacy or newer software without requiring the availability of their source code, or their modification by the user/developer. Instrumentation (in particular bytecode instrumentation at runtime), in this case, provides a tradeoff between accuracy and independence of source code modification.

4. *Software-centric approaches*. Hardware meters, although offering a precise value of the energy consumption of the device, have numerous limitations:

   - They only monitor hardware devices, not software.
   - They do not offer flexibility as they require hardware investment.
   - The impact of energy meters on energy efficiency have also been found to decrease over time [KG09].

   Measuring energy consumption of devices and software is relevant when the collected information is reusable. Raw energy consumption values are hardware dependent, therefore they cannot be used *as is* in different hardware or configurations. They also may, to a lesser extend, fluctuate even on the same machine and configuration due to electro-mechanical imperfections. We argue that a software-only methodology offers enough advantages to yield this limitation of reusability, while still maintaining accuracy, fine-grained results and with little user experience impact.

2.5 Summary

In this chapter, we reviewed the state-of-the art approaches in energy management and energy measurement in software. The middleware layer is a good choice for managing the energy consumption of software and devices (see Section 2.2). This layer offers assets for managing the diversity of software and hardware, their inherent heterogeneous nature (*e.g.*, programming languages, devices characteristics, applied environments, etc.), and the distributed platforms that devices and services are nowadays using. However, existing solutions only offer limited energy management, such as providing an architecture or algorithms applied to an environment-specific domain or to a certain execution context (see our discussions in Section 2.2.2). In addition, managing energy cannot be achieved efficiently if energy is not directly measured.

Existing approaches manage energy based on resources utilizations, and rarely use direct energy measurement and monitoring. The latters, on the other hand, are also limited. They require hardware power meters, or coarse-grained estimations of the energy consumption of software, and are limited in granularity (see Section 2.4). Measuring or estimating the energy consumption of software needs to be accurate with a low margin of error, and also fine-grained. Devices, software and even portions of software code need to be measured for energy consumption, without impacting the user experience (for example with a high execution overhead), and while using software-only approaches.

Our goal in this thesis is, therefore, to provide energy measurement approaches and tools, to accurately estimate the energy consumption of applications, and software code, at a finer level, without impacting the user experience, and using only software approaches. We also propose to model the evolution of energy consumption of software code (such as methods in modern programming languages) based on their input parameters. We conduct experiments using our approaches and tools, and report on the findings and learnings we got from our work about software energy consumption, distribution of energy in software code, and the impact of software parameters and execution platforms on energy evolution.

Our contribution is detailed in the next part of this thesis in Chapters 3, 4, and 5.

Part II

Energy Measurement and Evolution

Chapter 3

Energy Measurement at the Application Level

"The higher your energy level, the more efficient your body. The more efficient your body, the better you feel and the more you will use your talent to produce outstanding results."-Tony Robbins

Contents

3.1 Introduction

Energy consumption of software is a rising issue in the green computer community as well as for computer scientists and developers. Energy needs to be optimized in order to efficiently use software without any useless waste. Developers try to produce energy efficient software by stripping features, limiting software's capabilities, produce smaller feature-centric software, or reduce the usage of their applications. However, a first step of optimizing the energy consumption in software is to accurately measure it. State-of-the art solutions (see Chapter 2) provide limited approaches for efficient and accurate energy measurement. Accurate approaches require hardware investment in the form of power meters, while software approaches lack accuracy, usability and flexibility, and software profilers do not address the energy dimension. Energy meters have the drawback of being difficult, if not impossible, to upgrade. They also only provide information at hardware level, not software. Finally, hardware meters *fall into oblivion* as users would stop using them after a period of time [KG09]. We argue that efficient solutions require software-only energy measurement approaches, and that model-based software estimation of energy consumption provides accurate estimation for energy management and optimization approaches.

In addition, using resources utilization as a metric for energy consumption (as state-of-the art energy management approaches do) is not an efficient or accurate approach for modern hardware and software. Resources utilization is not 1-to-1 and linear to energy consumption. Modern CPUs feature Dynamic Voltage and Frequency Scaling or DVFS, therefore providing different amount of energy consumption of the CPU between its various frequencies and voltages. An application using the same percentage utilization in two different CPUs will not consume the same amount of energy. The same problematic is also present for other hardware components: hard disks, RAM memory or network cards feature different energy consumption specifications. Ideally, a software-only approach measures or estimates the energy consumption of hardware and software resources. This approach would have a low margin of error and provides an important addition to energy management solutions at middleware and software layers. It provides its energy measurements or estimations to middleware energy management platforms. The latter then uses this information in order to adapt and optimize software and devices based on real energy values (instead of resources utilizations). We argue that using direct energy values for energy management is a more efficient approach as to adapt and optimize software, but also to understand where and how the energy is being consumed. This knowledge is key into developing energy efficient software and energy management middleware platforms.

In this chapter, we propose in Section 3.2 energy models in order to estimate the energy consumption of software without the need of additional hardware meters. In Section 3.3, we then validate our models in a software implementation and conduct experiments in order to understand the energy consumption of software while varying programming languages, algorithms, parameters and compilers. The lessons we learnt from these experiments provide key insights into energy consumption in software and lead to a better understanding of energy consumption at deeper levels. We details thoses lessons and discussions in Section 3.4.

3.2 Energy Models

3.2.1 Methodology of Measurement

To measure the energy consumption of applications, we adopt a software-based estimation methodology. Our approach is summarized as follows: hardware resource utilization are measured before being mapped to software through energy models. Concretely, we measure hardware resource utilization, either directly from the devices, or through the operating system primitives. This information is then exposed in defined energy models in order to estimate the energy consumption of software. In details, our methodology is composed of four steps that are summarized in Figure 3.1:

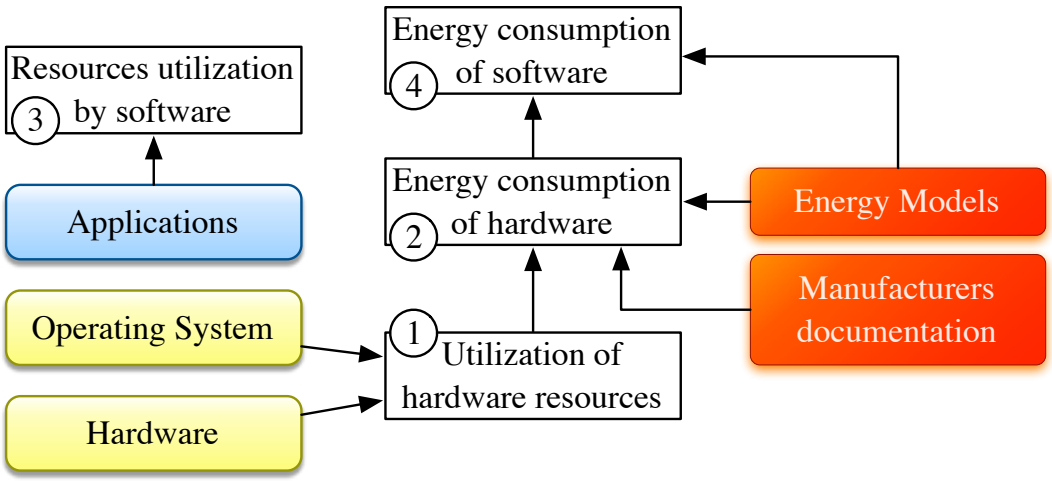


Figure 3.1: Methodology of measurement at application level.

1. First, we **collect utilization data of hardware resources**. For example, the current CPU frequency and voltage, or the network card throughput mode. This step is necessary for modeling the total energy consumption of the monitored hardware resources. For each resource, data is collected directly from the hardware interface (if available), or through the operating system APIs or tools.

2. Next, we **use energy models in order to estimate the total energy consumption of the hardware resource**. These models compute the energy consumed by a hardware component based on its runtime characteristics (*e.g.*, voltage and frequency for a CPU), and its physical properties (*e.g.*, Thermal Design Power or TDP for a CPU). The former are collected in the first step (from hardware itself or through the operating system), while the latter are provided by hardware manufacturers.

3. The third step is to **collect resource utilization of the hardware resources by software applications**. We consider that each process (identified by a unique PID) is a separate

application. Software running using multiple processes is managed by calculating the sum of the energy consumption of its processes. Resource utilization is monitored at runtime through the operating system. For example, we use the `proc` file system (or `procfs`) in Unix systems in order to collect information about the percentages of utilization of each CPU frequency by the monitored application.

4. Finally, we **apply our energy models to estimate the energy consumed by the application on a specific hardware resource**. The total energy consumption of software is therefore the sum of its energy consumption on each monitored hardware component.

Our approach uses energy models in order to estimate the energy consumption of software. In our work, we concentrate on several key hardware components, in particular the CPU and the network. These two components are widely present in most of modern devices, from desktop computers, data centers and mobile devices. Therefore, we concentrate on these two components as a first step in our energy modeling. The next sections present our models for the CPU and the network card.

3.2.2 Model for CPU

In our methodology described in Section 3.2.1, we use energy models in order to compute and estimate the energy consumption of software. Several previous works have proposed energy and power models for the CPU (see Chapter 2). However, these models are limited in their scope and accuracy. In this section, we extend the state-of-the art models and present our energy models to estimate the energy consumption of software for the CPU hardware resource. The model is summarized in Figure 3.2.

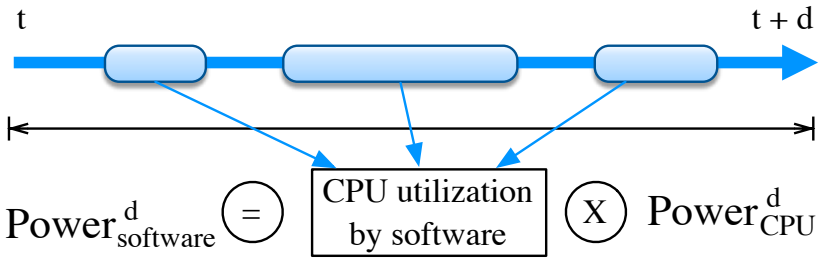


Figure 3.2: CPU model for software.

Building the CPU Energy Model

We propose a power model for modeling the energy consumption of the CPU. This model is based around the standard equation for modeling the power consumption of Complementatry Metal Oxide Semiconductor (CMOS) components, and in particular modern processors. We use *power* as measurement unit instead of *energy* in parts of our models for two reasons:

- Most state-of-the art models use *power* as unit for their models. Because we base and improve our own models on standard CMOS equations and models, we also use *power* as a measurement unit. This facilitates comparison between models and widespread usage of our models.
- Energy consumption is strongly related to execution time. We want to abstract this relation by providing the total energy consumption, and the energy consumption by a unit of time (in particular, per second–*power*).

The standard model for the power consumption on a CPU is as follows:

$$P_{CPU}^{f,V} = c \times f \times V^2 \tag{3.1}$$

Where f is the frequency, V the CPU voltage and c a constant value depending on the hardware materials (such as the capacitance and the activity factor). Thanks to this relation, we note that power consumption is not always linearly dependent to the percentage of CPU utilization. This is due to Dynamic Voltage and Frequency Scaling (DVFS) and also to the fact that power depends on the voltage (and subsequently the frequency) of the processor. For example, a process at 100% CPU utilization will not necessarily consume more power than a process running at 50% CPU utilization but with a higher voltage. Therefore, a simple CPU utilization profiler is not enough in order to estimate the power consumption of the CPU or software.

Two values in the model in Formula 3.1 can be accessed at runtime from the operating system: the CPU frequency f, and the voltage V. Each frequency runs at a certain voltage, with some cases of multiple close-range frequencies running at the same voltage. As an example, Table 3.1 outlines the supported frequencies of an Intel Pentium M processor with their related voltages [pen04].

Frequency (GHz)	Voltage (V)
1.6	1.484
1.4	1.420
1.2	1.276
1.0	1.164
0.8	1.036
0.6	0.956

Table 3.1: Frequencies and voltages for Intel Pentium M processor

The dynamic variables in the standard model in Formula 3.1, frequency f and voltage V, are obtained through the operating system at runtime. However, the static variable c cannot be obtained at runtime. The latter value is a set of data describing the physical CPU characteristics (*e.g.*, capacitance or activity factor). Manufacturers may provide this constant although in most cases it is missing. In order to calculate this value, we use the existing relation between the overall power of a processor and its *Thermal Design Power* (or TDP)

value. TDP represents the power required by the cooling system of a computer to dissipate the heat generated by the processor during execution. It is generally related to a certain state, such as the maximum frequency and voltage. However, TDP is not a perfect estimation of the power consumption of a processor. According to [RSRK07], a factor of 0.7 is to be applied to the relation between the TDP and the power consumption. Therefore, the power consumption of the processor can be modeled as follows:

$$P_{CPU}^{f_{TDP},V_{TDP}} \simeq 0.7 \times TDP \tag{3.2}$$

Where f_{TDP} and V_{TDP} represent the frequency and the voltage of the processor within the *TDP state*, respectively. The benefit of using the TDP in our model is that TDP is a value provided by most manufacturers.

Based on Formula 3.2, our model in Formula 3.1 can be used to calculate the constant *c* as follows:

$$P_{CPU}^{f_{TDP},V_{TDP}} = c \times f_{TDP} \times V_{TDP}^2 \simeq 0.7 \times TDP \tag{3.3}$$

thus *c* is modeled as:

$$c \simeq \frac{0.7 \times TDP}{f_{TDP} \times V_{TDP}^2} \tag{3.4}$$

Therefore, to compute the power consumption of a CPU, we apply the following model:

$$P_{CPU}^{f,V} = \frac{0.7 \times TDP}{f_{TDP} \times V_{TDP}^2} \times f \times V^2 \tag{3.5}$$

The Intel Pentium M processor have a TDP of 24.5 W for the maximum frequency of 1.6 GHz and voltage 1.484 V [pen04]. Thus, its constant *c* is calculated using Formula 3.4 to be equal to $4.86716803 \times 10^{-6}$.

Process CPU usage and power consumption

In order to estimate the power consumption of an application, we need to monitor its resources usage, in particular its CPU usage. We choose to identify applications by their processes, and the latters by their Process IDentifiers (PID). We calculate the process CPU usage as a ratio between CPU time for the PID and the global CPU time (*i.e.*, the time the processor is active for all processes), during a duration *d*, as follows:

$$U_{CPU}^{PID}(d) = \frac{t_{CPU}^{PID}}{t_{CPU}}(d) \tag{3.6}$$

Finally, the power consumption of a process is the product of the power consumption of the CPU for all applications, with the CPU usage of the monitored PID. This product is modeled for a certain frequency as follows:

$$P_{CPU}^{f} = \frac{0.7 \times TDP}{f_{TDP} \times V_{TDP}^2} \times f \times V^2 \times \frac{t_{CPU}^{PID}}{t_{CPU}}(d) \tag{3.7}$$

When the processor supports dynamic scaling of the frequency and voltage (DVFS), the CPU power consumption for a process P_{CPU} is equal to the average of the CPU power of each frequency balanced by the CPU time of all frequencies:

$$P_{CPU} = \frac{\sum_{f \in frequencies} P_{CPU}^{f} \times t_{CPU}^{f}}{\sum_{f \in frequencies} t_{CPU}^{f}} \tag{3.8}$$

3.2.3 Model for Network Card

The network power of a process is calculated using a formula similar to the CPU power formula. We focus our modeling on Ethernet network cards because we first target desktop and servers. Our estimation for network cards follows a cross-multiplication energy model, and is summarized in Figure 3.3.

Mainly, the network power consumed by software is the total power consumed by the network card $P_{network}$ multiplied by the duration of the monitoring d, then divided by the transmission time of data $t_{transmission}$. The following equation presents the model:

$$Power_{process}^{network} = \frac{P_{network} \times d}{t_{transmission}} \tag{3.9}$$

Even though the transmission time is gathered at runtime, the power consumption of the network card is generated through manufacturers' data. In particular, network cards use different states or throughput mode (*e.g.*, 1 Mbps, 10 Mbps) for sending or receiving data. Each of these states yields different power consumption for transmitting bytes for certain duration. Typically, manufacturers provide these values for a one second duration transmission.

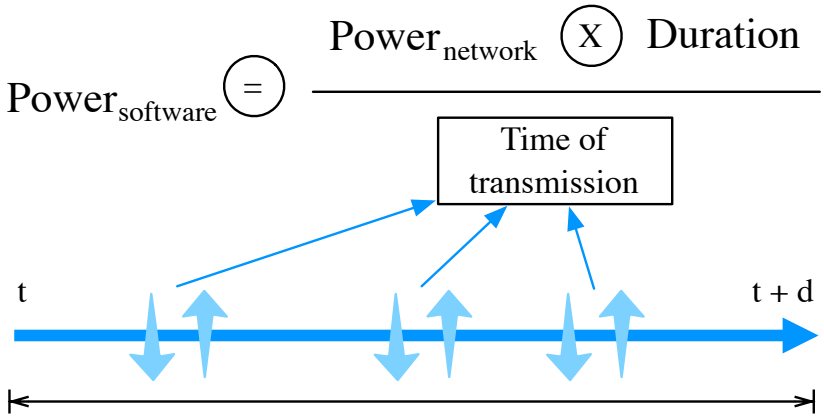


Figure 3.3: Network model for software.

Our network power model is therefore defined as:

$$Power_{process}^{network} = \frac{\sum_{i \in states} t_i \times P_i \times d}{t_{total}} \tag{3.10}$$

Where P_{state} is the power consumed by the network card in the state i (provided by manufacturers), d is the duration of the monitoring cycle, and t_{total} is the total time spent in transmitting data using the network card.

3.3 Experimentations

In this section we present the measurement tools we use to implement and deploy our energy models for software. First, we evaluate the validity of our energy models by comparing the values they generate to direct hardware measurement (using power meters). Then, we study the impact of programming languages and the impact of algorithms on the energy consumption in software. Both studies offer insights into energy consumption variability in software, when varying programming languages or implementation algorithms.

3.3.1 Measurement tools, PowerAPI

We use a system level library, called POWERAPI [BNRS13] and developed in our team by Aurelien Bourdon. It implements our energy models in order to measure the energy consumption of software. POWERAPI is a system library providing a programming interface (API) to monitor at runtime the power consumption of software at the granularity of system processes. Each process can therefore be monitored for its power consumption with a good estimation in comparison to using hardware power meters. The library also offers energy differentiation values based on hardware resources, such as giving the energy consumed by the process on the CPU, or on the network or on other supported hardware resources.

POWERAPI's architecture is modular as each of its components is represented as a *power module* (see Figure 3.4). These power modules implement our energy models in order to estimate the energy consumption of software at the application level. POWERAPI uses an event bus (with the publish/subscribe paradigm) for exchanging information between its modules. Listeners are used for gathering then providing energy information to the API. A sensor module is responsible for gathering operating system related information for the module. For example, it gathers the number of bytes transmitted by the network card, and the time spent by the CPU at each of the processor frequencies (when DVFS is supported). A formula module is responsible for estimating the power consumed for each process by using both information gathered by the sensor module and information based on hardware characteristics. This formula module uses our energy models specificed in Section 3.2 for the calculations. Therefore, the formula module is hardware independent.

POWERAPI is implemented in Scala and is based on an event-driven architecture as described in its website [powa]:

> PowerAPI is based on a modular and asynchronous event-driven architecture using the Akka library. Architecture is centralized around a common event bus

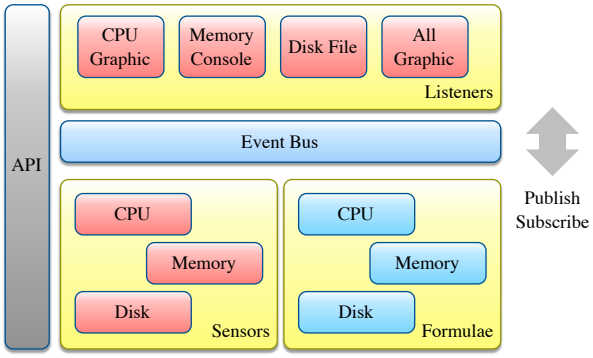


Figure 3.4: PowerAPI architecture.

where each module can publish/subscribe to sending events. One particularity of this architecture is that each module is in passive state and reacts to events sent by the common event bus.

We also developed another monitoring application, called JOLINAR [jol]. This tool is implemented in Java, and implements our energy models in order to estimate the energy consumption of single monitored applications. JOLINAR is therefore a *quick and easy* tool for application level measurements, however it does not provide a similar scalability as sc PowerAPI.

Power meters

For comparing energy results calculated through our energy models (implemented in POWERAPI), we use a Bluetooth power meter. The power meter, POWERSPY 2 [powb], is a measurement unit coupled with power visualizer and analyzer software. It consists of a built-in power meter in an electrical outlet, and measures the power consumption of hardware devices plugged to the outlet. The main limitation of POWERSPY (and other hardware power meters) is its limited granularity, at the level of hardware devices as a whole. Individual hardware components cannot be measured (unless they have separate power units that can be plugged in, independently, in an electrical outlet), neither can software. However, POWERSPY offers direct measures about the power consumption of devices. We use this information, while normalizing them, in order to compare our energy models to hardware energy measurements.

3.3.2 Validation of Models

We validate the accuracy of our energy models (defined in Section 3.2) by comparing the power values they generate with power values given by a hardware power meter. We

use POWERAPI as an implementation tool for our energy models, and POWERSPY as the hardware power meter. Experimentations are done on two computer: Dell Precision T3400 workstation computer with an Intel Core 2 Quad processor (Q6600); and a Dell OptiPlex 745 workstation computer with an Intel Core 2 Duo processor (E6600). Both computers runs Ubuntu Linux version 11.04 and version 1.6 of POWERAPI.

3.3.3 CPU model

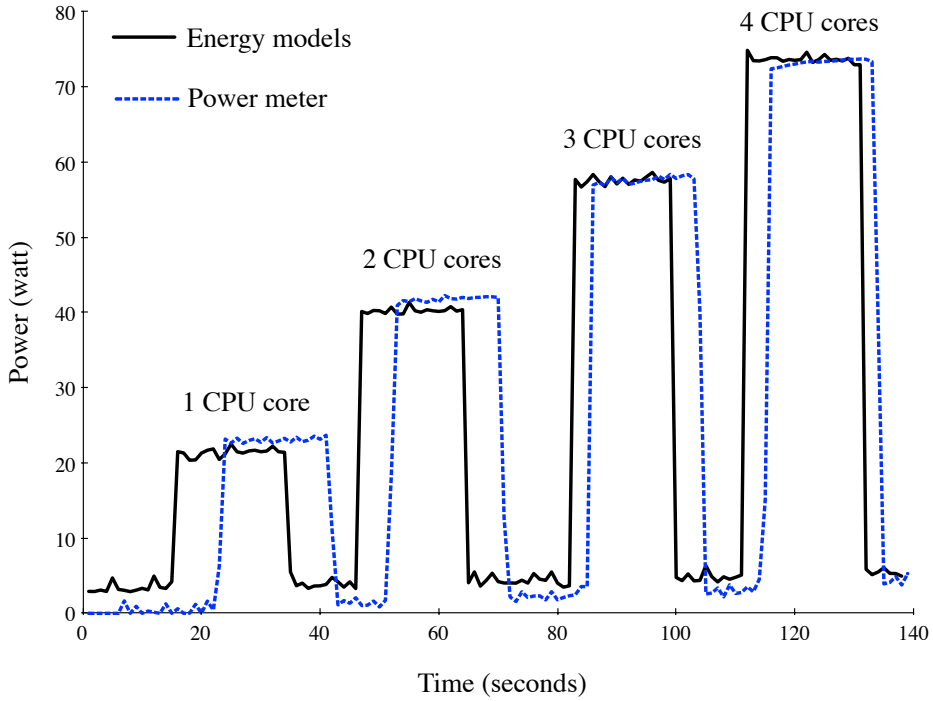
To validate the CPU model, we use three types of applications: a CPU-intensive application (the Linux stress command [lina]), a video player (MPlayer [mpl]), and two application web servers (Jetty [jet] and Tomcat [tom]). These applications cover different types of CPU utilization: CPU-intensive for the stress command where the CPU is utilized up to 100%, another CPU-intensive application, MPlayer, but where the CPU is utilized at different percentages and with different frequencies and voltages, and finally a web server that uses the CPU moderately and depending on user activities. We choose these categories of applications in order to validate our models on different workloads, and using all the parameters of our Formulas (*e.g.*, time, frequency and voltage).

First, we stress the processor using the Linux stress command [lina]. The command is a *tool to impose load on and stress test systems*. Basically, we stress the cores of the multi-cores processor one at a time, while adding a new core to the test incrementally. Figure 3.5a depicts the results as an evolution of the CPU power consumption along time (normalized values). We normalize these values by subtracting for each measured value of the power meter, the average of the differences between the values measured by the power meter and provided by our energy models. The peaks correspond to stressing 1, 2, 3 and 4 cores, respectively. Note that the slight synchronization shift on the graph between the two series of values is due to the communication lag between the Bluetooth power meter and the computer.

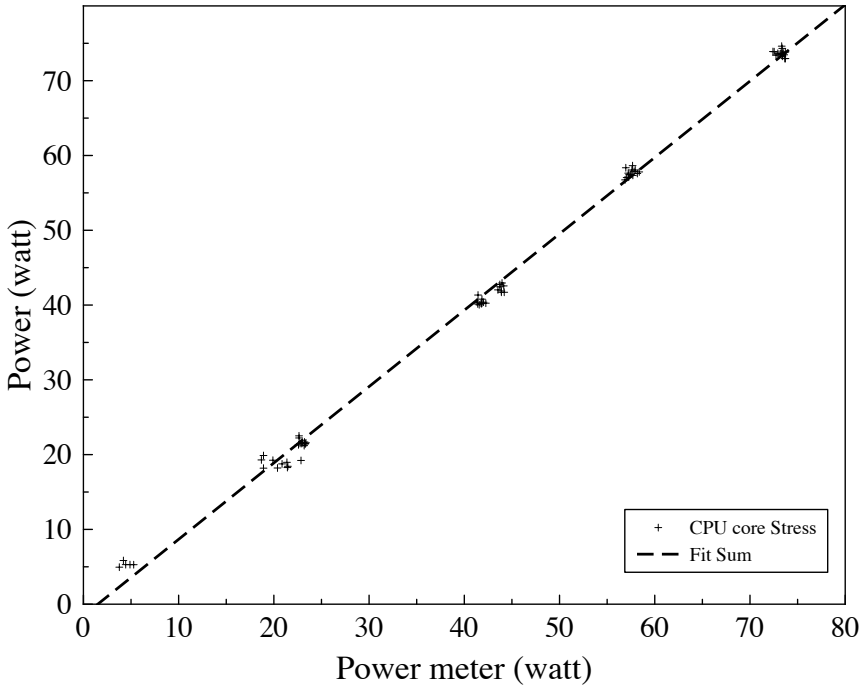
Figure 3.5b shows the accuracy of our library where we outline the measured values using the power meter and the values provided by our energy models, excluding the preliminary synchronization values. The results show minor variations between the estimations of our energy models and the power consumption values provided by the hardware power meter. The margin of error is small in the CPU core stressing experimentation, and is around 0.5% of the normalized and averaged values.

The margin of error grows up to 3% in more complex software. We stress the Jetty web server [jet] and the Apache Tomcat web server [tom] using Apache JMeter [jme]. In both experimentations, the build-in scripts and web applications are stressed and results are shown in Figures 3.6a and 3.6b for Jetty and Tomcat respectively. Finally, we play a video using MPlayer [mpl] free media player. Figure 3.6c shows the results. These figures show two advantages of our energy models:

- First, they validate our energy models on complex software, and in different domains. We validate our model on CPU intensive Linux commands (stress command in Figure 3.5a), on web servers such as Tomcat web server (see Figure 3.6b) or the lightweight



(a) Stressing the processing core using the Linux stress command.



(b) The accuracy of our energy models when stressing the processor cores using the Linux stress command.

Figure 3.5: Accuracy of CPU model

Jetty web server (see Figure 3.6a), and on a video player application (MPlayer, see Figure 3.6c).

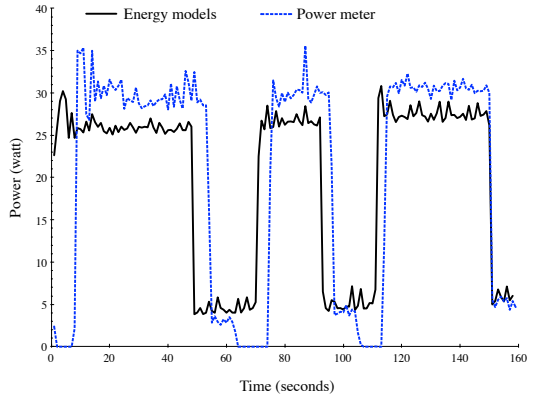
- MPlayer's results (see Figure 3.6c) are particularly interesting as they outline the effects of DVFS. Our energy model takes into account this variability of frequencies and voltages in the CPU, thus the variability of energy consumption in complex software. Also, the Linux stress experiment in Figure 3.5a shows the importance of multi-core in energy consumption. For a similar period of time, energy consumption varies greatly when running one, two, three or all the cores of the CPU.

  On another example, we compared the energy consumption of VLC player decoding a video, and an execution of the Tower of Hanoi program, both running on the Dell OptiPlex 745 workstation. The results in Figure 3.7 show the impact of running applications under multiple frequencies (*i.e.*, 1.6 GHz and 2.4 GHz). Although both executions of VLC the two frequencies run the same program and decode the same video, the difference in energy consumption shows clearly the energy impact of DVFS as an approach to lower energy consumption (*i.e.*, 3325 and 1716 joules, repectively). In contrast, executing a CPU intensive application, such as the Towers of Hanoi Java program, while forcing a specific frequency of the CPU also outlines the impact of DVFS on energy consumption. Results show the difference in CPU power consumption and execution time while running a special version of the Tower of Hanoi algorithm (that writes to a file each step of the algorithm) on two different frequencies. On the faster 2.4 GHz frequency, the program consumes more energy per second and executes faster (totaling 3948 joules), while on the lower 1.6 GHz, it executes longer but with lower power consumption (totaling 5092 joules). In this example, running faster even at a higher frequencies results in better energy consumption than running at a lower speed but nearly 40% longer.

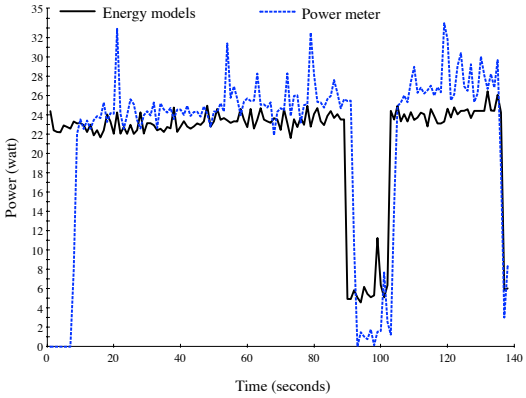
  These results show the validity of our approach and that a simple time profiler isn't enough to get energy insights because its does not take into account DVFS and multi-core CPUs.

3.3.4 Network model

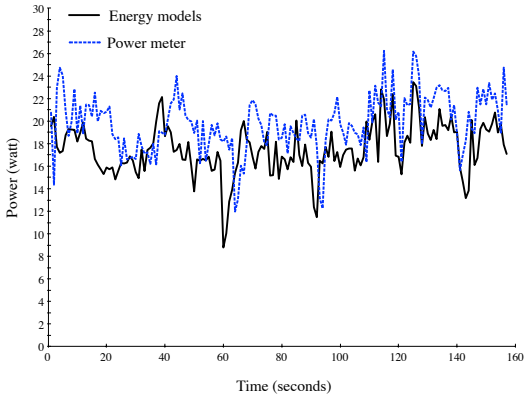
In addition to stressing the CPU, we also stress the network card, an integrated Broadcom 5754 Gigabit Ethernet controller. We use the Iperf command [ipe] that *performs network throughput tests*, and measure the power consumption of Iperf's own CPU consumption, and the power consumption of the Ethernet network card. We send two sets of TCP packets of 100 MB each from a distributed client to our host server. We use the default settings of Iperf while its CPU server executes following a one second cycle. The results show very low network power consumption in comparison to the processor's consumption of the Iperf process. The difference is around 192% between the network's 0.017 watt and the CPU's 0.9 watt. Figure 3.8 outlines these numbers.



(a) Running stress test on Jetty web server using JMeter.



(b) Running stress test on the Tomcat web server using JMeter.



(c) Running a video under MPlayer media player.

Figure 3.6: Stressing Jetty and Tomcat web servers and MPlayer

These numbers show that, although CPU power is quite low (average around 0.9 watt) and the network card uses all its capacity, the consumed network power is largely negligible compared to the consumed CPU power on our test server. This observation is in correlation with the literature [RSRK07].

These CPU and network experimentations show the validity of our energy models. Therefore, we can reasonably argue that using a software-centric approach provides values that are accurate enough to acknowledge the energy cost of software.

3.3.5 Impact of Programming Languages

Monitoring the energy consumption of software helps in understanding **how** energy is being consumed, **where** and **why**. These questions, which we outlined in our main introduction (see Chapter 1), are essential for energy optimization and management. In order to provide

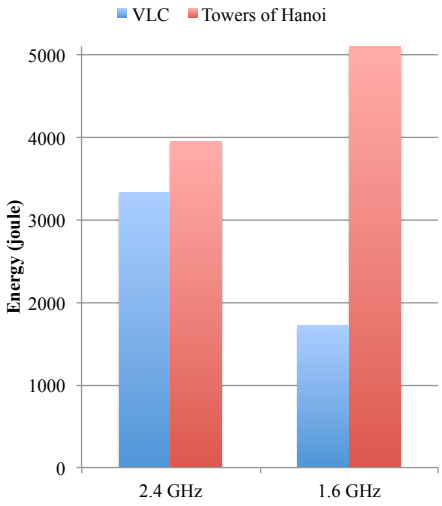


Figure 3.7: An example of the impact of CPU frequencies on energy consumption.

insights into these questions, we decide to conduct a comparison of the energy cost of programming languages. Using our energy models and the tools implementing these models, we run and compare an algorithm implemented in different programming languages. This experiment is relevant for two reasons:

- First, it allows us to identify **how** energy is being consumed when changing the implementation language. Therefore, we can have a view of the impact of programming languages on energy consumption of software.
- Second, programming languages are diverse in term of design and specifications. These languages can provide native code, byte code, compile at runtime, use a virtual machine, etc. Our experiment on programming languages with this diversity gives us clues on **why** energy consumption varies between languages.

Concretely, we run a similar implementation of the Tower of Hanoi program in different programming languages. We choose this algorithm because of its simplicity (a few lines of code), therefore we can quickly identify **how**, **why** and most importantly **where** energy is being spend. The algorithm hold two main implementations: recursive, and iterative, which we compare later in this section (see Section 3.3.6).

Next, we measure the energy consumption of these programs using the same set of parameters. In order to limit the impact of developers' knowledge in programming languages on the algorithm's optimization, we use implementations of Tower of Hanoi made by Amit Singh of Hanoimania project [Sin]. The project contains more than a hundred different implementation of the Tower of Hanoi program in various programming languages, either using the iterative or recursive algorithm for solving the Tower of Hanoi problem.

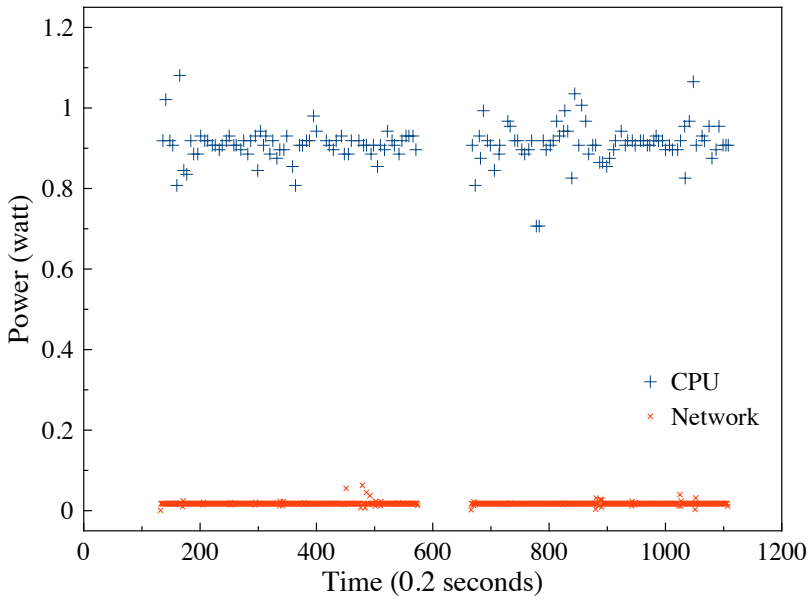


Figure 3.8: CPU and network power consumption in Iperf stress test.

We chose 8 implementations representing three groups of programming languages that compile into:

- native code such as C, C++ and Pascal.
- byte code that is run using a virtual machine such as Java and OCaml.
- and dynamic scripting languages where the program compiles and executes at runtime such as Prolog, Python and Perl.

For each programming language, we use the default compiler with the default parameters and options. We also use PowerAPI to gather power values (in watt) for the program execution and we report the energy consumption in joule (a function of time). Measurements are collected at a 200 ms interval (or 5 Hz), and we run the Tower of Hanoi program with 30 disks and 3 towers. The experimentations are done on the same configuration as in Section 3.3.2, *i.e.*, a Dell Precision T3400 workstation computer with an Intel Core 2 Quad processor (Q6600), and running Ubuntu Linux version 11.04.

Figure 3.9 outlines the results of our experimentation. We observe that the energy consumed by the same algorithm varies from language to language. The three implementations in native code (C, C++, and Pascal) show similar energy consumption. C and C++ have a near equal energy consumption (0.9% difference, at 325 and 322 joules respectively), which is explained by the fact that both implementations are similar (C++ is considered as an increment of C language). The difference with Pascal language is slightly higher, but sill relatively small (with a 5.9% difference at 341 joules).

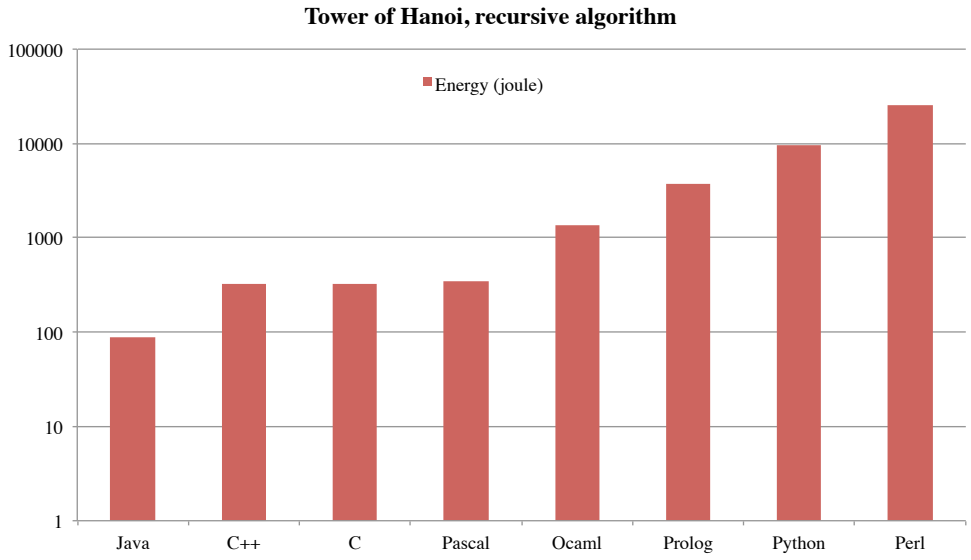


Figure 3.9: Energy consumption of the recursive implementation of the Tower of Hanoi program in different languages (using a base 10 logarithmic scale).

The results also show that scripting languages have significantly higher energy consumption. Perl is the most consuming at 25516 joules, while Python consumes 9450 joules and Prolog 3673 joules. The difference between Prolog and C is high, at around 1000%, and culminates to 47863% when comparing Perl and C. This is due to the scripting nature of Prolog and Perl languages, compared to the native code produced by the gcc compiler of C.

Finally, virtual machine-based languages, Java and OCaml, are both at the opposite side in term of energy consumption. Java has a low consumption at 88 joules, while OCaml consumes 1375 joules.

Based on these results, we note the following observations:

- The first notable observation is the **low energy consumption of the Java implementation** of the Towers of Hanoi program (88 joules). The increase of the C implementation to the Java's one is around 269%. The energy cost of the virtual machine is therefore minimal thanks to optimizations and code predictions in the Java virtual machine (JVM). One such optimizations that is used in our Towers of Hanoi experimentation is Just-in-time (JIT) compilation in the JVM [Ora]. Sun's JVM combines both interpretation of byte code and JIT compilation. The Java byte code is initially interpreted, and if portions of the code are being frequently executed, the JVM compile these portions to native code. In our algorithm, the recursive version clearly outlines repetitive code that the JVM detects and compiles to native code at runtime (JIT compilation).

  In comparison, OCaml compiler produces a self-executable byte code. The execution energy cost (1375 joules) shows the cost of starting and running OCaml virtual machine

(the executable file launches the OCaml byte code interpreter by itself [oca]). In the next section (see Section 3.3.5), we conduct in-depth experiment of the impact of compilers, and in particular of the impact of the OCaml byte code compared to native code.

- The second observation is the **high cost of scripting languages**. These implementations have a high cost at 7751% (Perl), 2807% (Python) and 1030% (Prolog) increase in energy consumption compared to the C energy consumption. This cost is explained by the need to interpret then execute the program. This additional step clearly has a high cost in term of energy consumption.

- Lastly, **the relation between execution time and energy consumption is shown here as linear**. Figure 3.10 reports on the relation between energy consumption and execution time. We find that execution time and energy consumption of the different programming languages is similar. However, **this observation is not universal**, and can be explained by the fact that the Towers of Hanoi program is a CPU-intensive one. The CPU usage is nearly around 100% of one of the processor cores during the execution of the program. In average, this equals to around 18 watts in our host configuration (ranging from 17.5 to 18.2 watts).

  On more complex software and scenarios, such as playing a video in MPlayer, the relation between execution time and energy consumption is not clear anymore. Our experiment on MPlayer (see Figure 3.6c), VLC and Towers of Hanoi (see Figure 3.7) outline the impact of DVFS and using multiple CPU frequencies. Decoding and rendering video frames requires different amount of CPU power utilization (thanks to DVFS), with differences up to 130% between peaks and valleys in the graph. The same rendering is bound to a fixed duration (the length of the video), therefore using different frequencies results in energy gains (although execution time is unchanged). Finally, when using a CPU intensive application, the impact of running faster with a higher frequency rather than running on a lower frequency but taking longer execution times.

Impact of Compilers

In Section 3.3.5, we noticed that Java and OCaml energy cost is clearly different. Energy differences between both implementations are important. We also noticed that C and C++ versions have higher energy consumption cost than Java. We decide to investigate the energy consumption of the C and C++ versions using different options in the GCC/G++ compilers, and compile the OCaml implementation using the native OCaml compiler: `ocamlopt`.

Figure 3.11 shows the energy consumption cost of the recursive implementation of the Tower of Hanoi program using the O2 and O3 optimization flags. The difference in term of energy consumption is notable. The O2 optimization option turns on more than 50 optimizations flags in GCC and G++ compilers. These allow a more optimized code, thus a lower energy consumption: 21% and 27% decrease respectively for C and C++ when using the O2 flag. The O3 flag turns all optimizations flags of O2 in addition to 8 more flags [gcc],

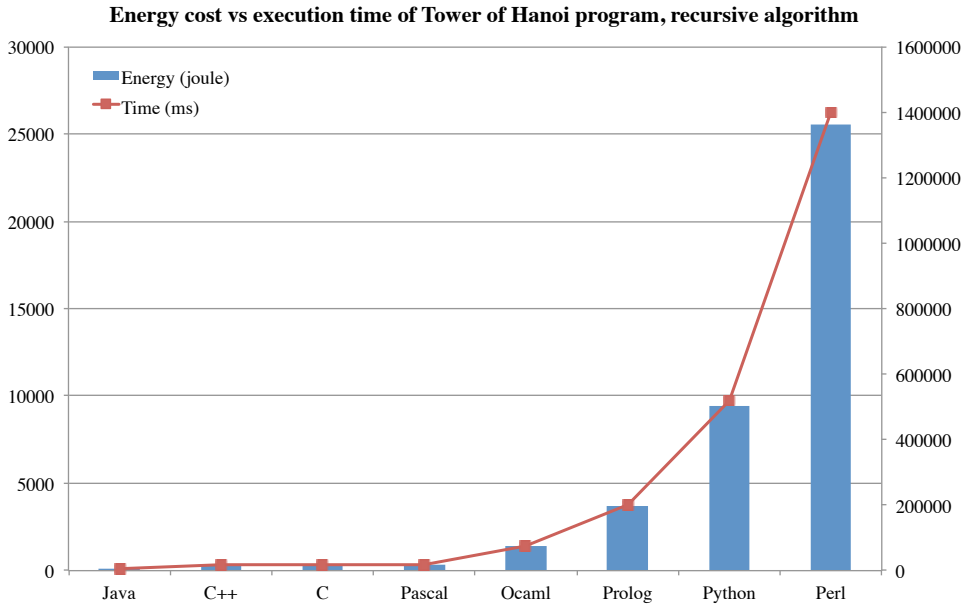


Figure 3.10: Energy consumption cost and execution time of the recursive implementation of the Tower of Hanoi program in different languages

including the *Predictive Commoning* optimization flag. The latter eliminates redundancies across the iteration of a loop [pre], therefore increasing performance and energy saving by 83% for C and C++ with the O3 flag (78% and 77% decrease compared to the compilation with O2 flag).

When using the default OCaml compiler, `ocamlc`, the program consumes 1375 joules. In comparison, the native code OCaml compiler, `ocamlopt`, produces an executable program that consumes 130 joules. Ocamlc compiler produces a self-executable byte code, which is interpreted by the OCaml byte code interpreter. Ocamlopt, on the other hand, produces native code that is executed directly by the host machine. The main advantage of byte code to native code is typical: universal execution on all hardware configuration (if a virtual machine is implemented), or as the Sun slogan for Java says: *Write once, run everywhere*. But this advantage has a cost that is outlined on our experiment with ocamlc and ocamlopt. Byte code interpretation for OCaml has an increase in energy consumption of 957% compared to the native code version.

3.3.6 Impact of Algorithms

We then compare the energy consumption difference between the same program, Towers of Hanoi, but using two different implementations: a recursive algorithm and an iterative algorithm. Figure 3.12 outlines the energy consumption cost of both recursive and iterative algorithms implemented in C and C++ (and with different compilers options).

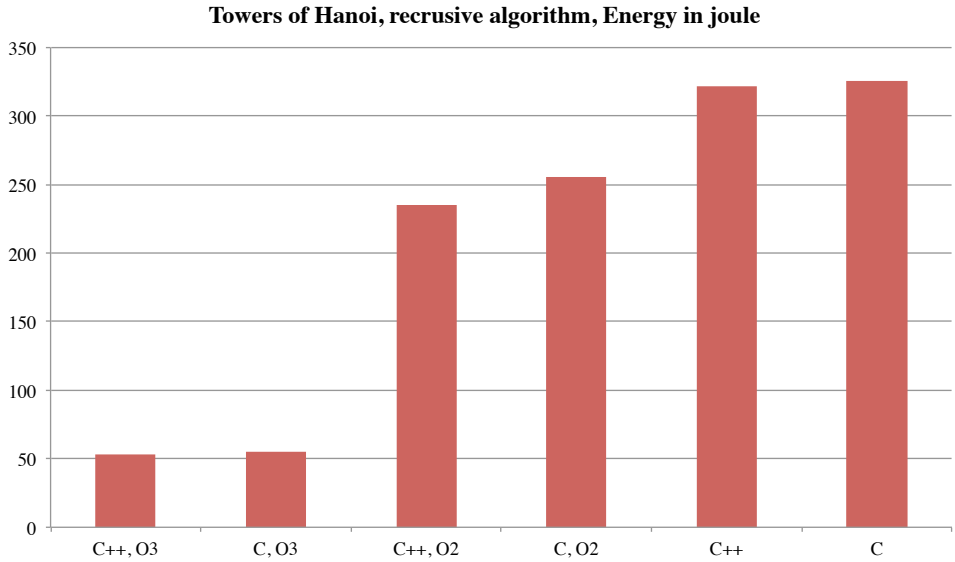


Figure 3.11: Energy consumption of the recursive implementation of Tower of Hanoi program in C and C++ using O2 and O3 GCC and G++ compilers' options.

The recursive algorithm implemented in C++ consumed in average 322 joules while its iterative version consumed 1656 joules, which amounts to more than 400% increase. When using the O2 optimization option during compilation, both versions exhibit similar energy consumption (on average 7% difference), however with the O3 option, the iterative version does not save any energy. The recursive version shows a 78% decrease in energy consumption using the O3 option in comparison with the O2 option. The *Predictive Commoning* optimization in O3 eliminates redundancies across the iterations of a loop, which benefits recursive algorithms more rather than iterative ones. Our results show that the recursive version of the Towers of Hanoi program is more energy efficient than its iterative version.

Impact of I/O Primitives

All the previous experimentations on the Tower of Hanoi algorithm do not print any message on the Linux terminal. We carefully removed all print commands from the implementation code provided by Hanoimania. However, we noticed when we fist executed the unmodified implementations from Hanoimanai, that the power consumption and execution time was high. We decided to investigate this difference of energy consumption due to the I/O primitives. Therefore, we measure the impact of adding an additional print line to the program. We add a print command to print the action of moving a disk from one tower to another. Listing 3.1 shows the implementation code in OCaml with the additional *print_endline* command.

```
1 t movedisk f t =
```

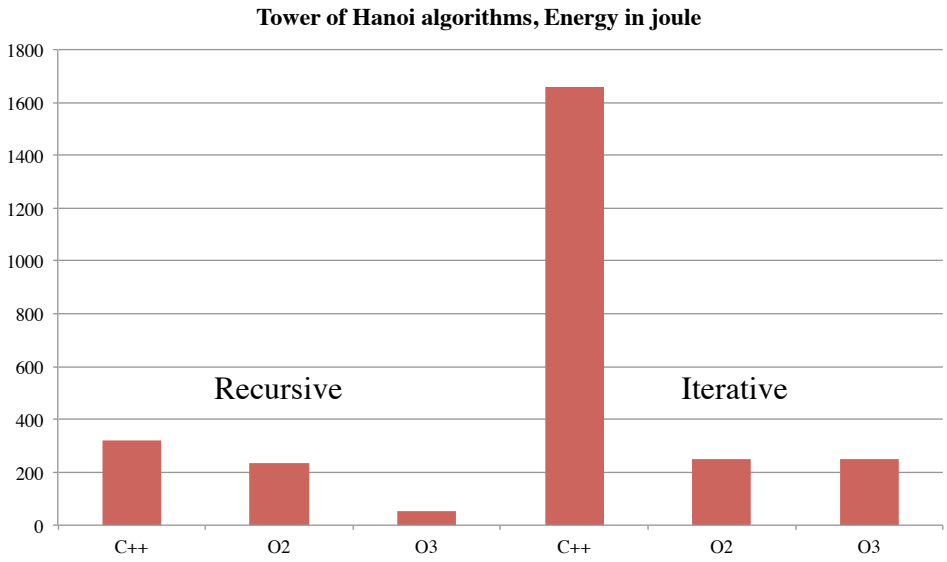


Figure 3.12: Energy consumption of the recursive and iterative implementation of Tower of Hanoi program in C++.

```
2     print_endline(f ^ " --> " ^ t);;
3
4let rec dohanoi n f u t =
5     if n = 1 then
6         movedisk f t
7     else
8         begin
9             dohanoi (n - 1) f t u;
10            movedisk f t;
11            dohanoi (n - 1) u f t;
12        end;;
```

Listing 3.1: Towers of Hanoi recursive algorithm in OCaml with print command

With the print command, the programs consumes 17852 joules, compared to 1375 joules without the print command. The difference is nearly a 1200% increase when adding a print command. The experiment shows that printing to the terminal, a relatively simple task, is costly. In particular if it is executed repetitively in a recursive algorithm (1 073 741 823 times for 30 disks). The tasks of opening a pipe and sending data to the terminal are not negligible efforts in term of performance and energy efficiency.

These numbers show the impact of I/O primitives on energy consumption. Writing to a file, to the terminal, or other activities using I/O primitives use non-negligible amounts of energy. Therefore, we argue that inputs and outputs should be used wisely in software, such as when using log systems for recording the execution of software, or printing debugging information in applications.

3.4 Discussions and Limitations

The learning we gained from the experiments conducted on our proposed energy models, allows us to better understand software energy concerns. We summarize these learning in the following points and limitations.

3.4.1 Model-based software energy estimations are accurate and valid

The first conclusion of our experiments is that our energy models are accurate and valid for estimating the energy consumption of software (Section 3.3.2). The error margin we obtain is low, between 0.1% and up to 3% on complex applications. The results in experiments (Sections 3.3.5 and 3.3.6) also show the potential of our approach: our models allow energy estimations on different programming languages, configurations and implementation algorithms. We can reasonably argue that model-based software energy estimations are accurate enough to replace hardware power meters. Another advantage is the absence of hardware investment, and also being able to upgrade the models when needed. In addition, energy models allow decomposing energy consumption per hardware component, and per application.

3.4.2 Ethernet network energy is negligible to CPU

Physically, energy is directly consumed by hardware. Software energy consumption is the energy consumed by hardware following software requests. As such, hardware components have different energy consumption levels. CPU is, by far, the largest energy-consuming component on computers. Our experiments show that Ethernet network energy consumption is negligible compared to CPU's energy consumption (Section 3.3.2). The difference we found is about 192% for a network-intensive benchmark. Our results are in correlation with state-of-the art results [RSRK07].

3.4.3 Energy and execution time are not linear

A common belief is that a simple CPU profiler can provide enough information to understand the energy consumption of software. This belief is contradicted by our experiment (Section 3.3.2) where energy consumption for decoding a video on MPlayer (Figure 3.6c) is not linear to the execution time of reading this video (this is due to multiple CPU usage values and DVFS). Figure 3.7 also show the impact of DVFS and executing the same task (decoding a video in VLC, and solving the Towers of Hanoi algorithm) but under multiple frequencies. However, on a group of CPU-intensive applications, this relation between energy consumption and time execution is linear. In particular, some CPU-intensive uses up to 100% of the CPU nearly all the time (that is the case in the Tower of Hanoi program). Therefore, CPU utilization is considered constant, thus the CPU usage approaches to 1. Our

model in Equation 3.7 becomes also a constant value, rendering the energy consumption of software only a function of time.

We determine that these two groups of CPU-intensive applications have different energy consumption patterns:

- Software that run at 100% utilization of the CPU have linear relation between energy consumption and execution time.
- Software where its execution is impacted by DVFS, different CPU frequencies or multi-cores, do not have linear relation. They rather are impacted by multiple factors (*e.g.*, CPU frequency and voltage, multi-cores) in addition to execution time, therefore rendering the use of only a time profiler useless for energy profiling.

3.4.4 Code, algorithms, languages, and parameters impact energy consumption

Our experiments show also that how software is being written and the language used are important for energy efficiency. Using scripting languages have clearly drawbacks in term of energy efficiency (Section 3.3.5) that should be taken into consideration when using them (trade-offs would then be made between ease-of-use of the language, universality, and energy efficiency).

The experiments also validate that native code languages are the most efficient (Section 3.3.5). The example of `ocamlc` and `ocamlopt` compilers showed the benefit in term of energy efficiency of native code compared to byte code. However Java does a great job mainly due to optimizations in the JVM (in particular, JIT compilation). Byte code languages have a good trade-off between energy efficiency and versatility. The advantage of using byte code (*Write once, run everywhere*) is increased with good energy efficiency (Java is better than C and C++ without GCC or G++ optimizations; OCaml is 300% worst than Pascal while Perl is 7382% worst).

The impact of the compiler and compilation options is outlined in our experiments. The GCC and G++ compilers have optimization options that do a great job in term of energy efficiency (Section 3.3.5). Energy saving varies from 21% with the O2 optimization flag, to up to 83% with the O3 optimization flag.

Finally, the design of the algorithm used to solve a problem is not to be forgotten. For example, comparing iterative and recursive algorithms shows the advantages of the latter in a CPU-intensive mathematical game puzzle. However, this cannot be generalized to all problems as it also depends on the mathematical problems and solutions.

3.4.5 Software as a black-box is not sufficient

Our application level energy models provides the energy consumption of software. Although this allows understanding the energy consumption of applications as a whole, monitoring at application level does not provide detailed information on how the energy is being spent internally. Worst, if abnormal energy consumption is being noticed in an application, there is little information on where and why this is happening. The experiment we did on the impact of the print command (Section 3.3.6) illustrates such a scenario: a single print command is responsible of nearly 1200% increase in energy. A code-level measurement tool would allow detecting the code responsible for this energy increase. We managed to detect this difference thanks to a small and comprehensible code base to review. On more complex software, only detailed and fine-grained energy measurement would have detected this behavior. Therefore, coarse-grained measurement of the energy consumption of software is not sufficient for later energy management and optimization. Finer-grained approaches are needed for energy efficiency.

3.5 Summary

In this chapter, we present energy models in order to estimate the energy consumption of software. Our approach is software-only, therefore we do not need the usage of any additional hardware meter (such as a multimeter or a power meter). The models we present offers a high accuracy (with a margin of error of up to 3% as seen in Section 3.3.2). In addition, our energy models allow measuring the energy consumption per hardware resource, such as the CPU or the Ethernet network card energy consumption. They also are used to measure the energy consumption of individual applications (for example, processes in a Linux system). We also outline the validity of our models and the software implementation of these models, in term of accuracy of the models and the viability of the approach, and against a power meter.

Based on our models, we investigate the energy consumption cost of different programming languages in a CPU-intensive application. Our results show that scripting languages have a much higher energy consumption compared to virtual machine based languages, and to native code languages (see Section 3.3.5). They also show the impact of the optimization options and parameters in compilers as energy consumption varies greatly in GCC and G++ compilers based on their options (see Section 3.3.5). The impact of the virtual machine is also outlined when comparing `ocamlc` and `ocamlopt` compilers, where native code surpasses byte code in term of energy efficiency (Section 3.3.5). However, this is relative as Java byte code is energy efficient in our experiment, mainly due to optimization in the Java Virtual Machine.

Finally, we study the impact of coding algorithms, where changing the algorithm to solve a problem can reduce the energy consumption of the problem (*i.e.*, using recursive

instead of iterative implementation of the Towers of Hanoi program in C and C++, in Section 3.3.6). The cost of printing a line on the system terminal is also shown to be non-negligible, as shown in out results (see Section 3.3.6).

3.6 The Need for Code Level Measurement

Our experimentations in Section 3.3 shows the limits of software-only energy profiling, and also the potential of measuring energy in software code. In Section 3.3.6, we outline a glimpse of code optimization that is achieved thanks to software measurement. However, this experiment requires running the measurement twice (once with the original code, and once with the modified software), and also requires modifying the software itself. What if we could measure the energy consumption of software at code level? And without software modification or complex experimentation scenarios? Our experiments show the potential and the necessity for such code level measurement energy models and tools.

In the next chapter, we present our energy models for measuring the energy consumption of software at code level. We also present our implementation tool, called JALEN, and our experimentations in detecting energy hotspots in software.

Chapter 4

Energy Measurement at the Code Level

"Is the minor convenience of allowing the present generation the luxury of doubling its energy consumption every 10 years worth the major hazard of exposing the next 20,000 generations to this lethal waste?"-David R. Brower

Contents

4.1 Introduction

In Chapter 3, we showed that monitoring energy at application level is limited in order to understand how and where energy is being consumed in software. In particular, application level measurements do not provide in-depth information on how the energy is being spend, and which portion of code is responsible for the most energy consumption. An abnormal energy consumption of an application would be better detected, understood and corrected if needed, if detailed energy information were provided. Code level energy information is thus useful for software developers for producing power efficient code. The Green Challenge for USI 2010 [gcu] has identified that profiling applications to detect CPU hotspots is a winning strategy for limiting the power consumption of applications. Although CPU profiling can help in getting an idea of the energy consumption, the relation is not linear: for example, DVFS in CPU and the energy consumption of other hardware components impact the global energy consumption of software as we saw in Section 3.3. Therefore, we argue that a fine-grained approach for proposing power-aware information is a keystone for future power-aware systems and software.

However, monitoring at code level holds additional challenges that need to be correctly addressed for measuring energy consumption:

- First, a major problem of code level monitoring is **accuracy**. Not only accurate energy models are needed (as with application level monitoring), but also measuring energy consumption at a fine grain requires precision. Due to the fine granularity of the measurements, a moderate margin of error could lead to a false snapshot of the energy distribution between methods. For example, if two methods consume 80 and 100 joules each and with a measurement tool with 10% margin of error, the values provided to the user would be between 70 and 90 joules and 90 and 110 joules, respectively. Therefore, such tools may estimate an equal result (90 joules each) where in reality a method consumes 25% more energy than the other.

- When monitoring methods that may take as little as few milliseconds to execute, the overhead of the measurement platform should not be high. The cost of the measurement approach itself leads to higher overhead that pollutes energy results. The overhead is not to be confused with the margin of error. Measurement tools with good accuracy but high overhead, produces precise values but an additional cost in term of energy or execution time is present, rendering the usability of the profiling limited in production.

- Finally, a hard challenge is to correctly attribute the energy consumption of hardware to the portion of code responsible for its consumption. With higher level programming languages, and the complexity of modern software (*e.g.*, multiple methods and threads, methods overloading, children methods and call tree), this correlation between hardware's energy consumption and software code is a key element to measuring energy at code level.

In this chapter, we tackle these challenges and propose energy models in order to estimate the energy consumption of software at code level, *e.g.*, at elementary code block such as methods (see Section 4.2). We then validate our models in a software implementation called JALEN in Section 4.3. The latter is a software measurement tool for Java applications that provides the energy consumption per method. Finally, in Section 4.4 we conduct experiments in order to detect energy hotspots in applications. Lessons learned from energy hotspot detections allow developers to better understand the energy distribution in their code and produce energy efficient software, and are discussed in Section 4.5.

4.2 Energy Models

4.2.1 Methodology of Measurement

Our methodology to measure the energy consumption at code level bears similarities with our methodology at application level (Section 3.2.1). In particular, our methodology is summarized as follows: first, we measure hardware resource utilization by software code, then we gather the energy consumption of the application (obtained through application level measurement tools), and finally we map the previous data together through our energy models in order to estimate the energy consumption of software code by hardware resources.

The granularity of our measurement at code level is that of methods. Concretely, we measure the energy consumed by each method of the application on hardware resources. In details, the methodology is composed of three steps that are summarized in Figure 4.1:

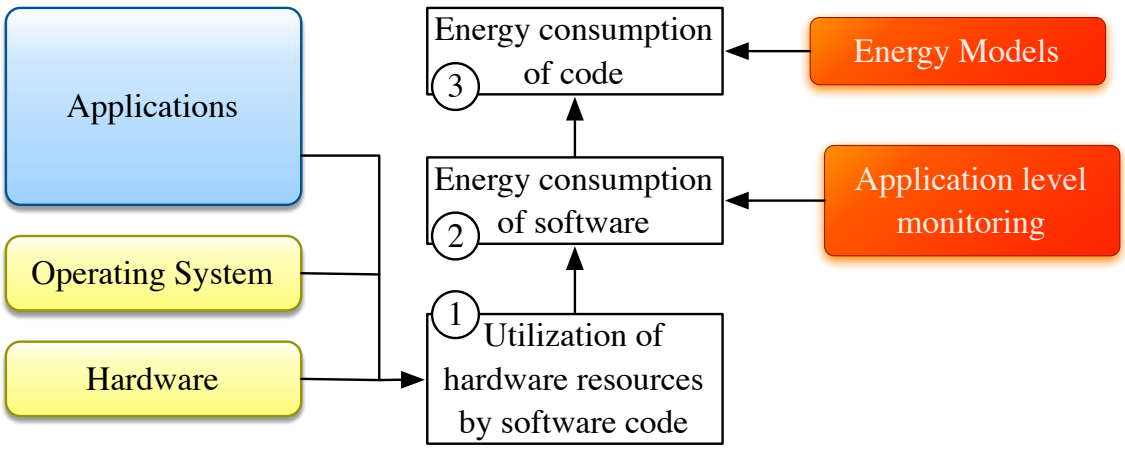


Figure 4.1: Methodology of measurement at code level.

1. First, we **collect utilization data of the hardware resources by software code**. For example, we collect the CPU utilization of the executing method in an application, or the number of bytes sent on the network card by a method, the number of times a method

access the hard disk or the times spend reading or writing data. These data are gathered, either directly from hardware and software, or through a *middleman architecture* if available, such as a virtual machine.

2. Second, we **gather the total energy consumption of the application, by hardware resource**. This information is obtained using our application level models and measurement tools (Chapter 3). Basically, in this step, we *isolate* the energy consumption of the application, therefore allowing energy models to be specified by the energy consumption of software (thus, modeling hardware resources is not required in our energy models at this step; they are already modeled at the application level in Chapter 3).

3. Finally, we **map the collected data of software code and energy consumption at application level with our code level energy models**. This mapping allows our energy models to estimate the energy consumption of software code (methods, classes, etc.) by hardware resource.

4.2.2 Model for CPU

As with the application level models (see Chapter 3), we use power instead of energy in our models. Using the information collected from the application level monitoring system, we calculate the power consumption of software code or methods using the following formula:

$$Power^{CPU}_{method} = Power^{CPU}_{software} \times Utilization^{CPU}_{method} \tag{4.1}$$

$Power^{CPU}_{software}$ is obtained through our application level model, in praticular by using Formula 3.8.

Concretely, the power consumption of a method is a percentage of the power consumption of the whole application based on resource utilization. This cross-multiplication equation allows us to abstract the hardware when measuring power consumption of software code. At code level, our approach assumes that the power consumption of the application is known (*i.e.*, this information is gathered from application level monitoring). Therefore, our code level model is hardware independent (unlike application level model).

Using the information we gather from our application level monitoring, we can estimate the power consumption of software code at the granularity of methods (thus, classes, packages or components). The first step is to calculate the CPU utilization of the code (*i.e.*, method). As application code is generally executed inside threads (this is the case for Java), we start by calculating the power consumed by each thread, then we measure power at code level. The power consumed by a thread is calculated through the following formula:

$$Power^{CPU}_{thread} = \frac{Power^{CPU}_{software} \times Utilization^{CPU}_{thread}}{Duration_{cycle}} \tag{4.2}$$

where $Power^{CPU}_{software}$ is the power consumed by the application in the last monitoring cycle, $Utilization^{CPU}_{thread}$ is the CPU time of the thread in the last monitoring cycle, and $Duration_{cycle}$ is the duration of the monitoring cycle. $Power^{CPU}_{software}$ is obtained from the application level models (from tools such as PowerAPI), $Utilization^{CPU}_{thread}$ is obtained from the environment, such as the OS or a virtual machine).

The next step is to calculate the CPU utilization for a method executing in the thread against the total CPU utilization by the thead. For each thread, and for the monitoring duration, we get the list of all methods executing and estimate their CPU utilization. Threads can execute one method at a time following an execution stack. For the duration of monitoring, multiple methods can be executed by the thread. To estimate the CPU utilization of these methods, we use their execution time and this formula:

$$Utilization^{CPU}_{method} = \frac{Duration_{method} \times Utilization^{CPU}_{thread}}{\sum_{m \in Methods} Duration_m} \tag{4.3}$$

Where $Duration_{method}$ is the execution time of the method in the last monitoring cycle, and $\sum Duration_{methods}$ is the sum of the execution time of all methods in the last monitoring cycle.

Finally, we can estimate the power consumption of software methods of formula 4.1 by the following:

$$Power^{CPU}_{method} = \frac{Utilization^{CPU}_{method} \times Power^{CPU}_{thread}}{Duration_{cycle}} \tag{4.4}$$

4.2.3 Models for Network Card

The network power model at code level also uses power information from the application level. The similar modular approach is applied here for the network. We calculate the network power consumption per method using the number of bytes transmitted by the application. First, we gather the number of bytes read and written by each method in the last monitoring cycle. Then, we collect the network power consumption of the application using application level models (and tools such as PowerAPI). Finally, the power consumed by a method is a percentage of the power consumption of the application based on the number of transmitted bytes (*i.e.*, a cross-multiplication).

$$Power^{Network}_{method} = \frac{Bytes_{method} \times Power^{Network}_{process}}{Bytes_{process}} \tag{4.5}$$

Where $Bytes_{method}$ is the number of bytes read and written by the method, $Power^{Network}_{process}$ is the power consumed by the application, and $Byte_{process}$ is the number of bytes read and written by all methods of the application. $Power^{Network}_{process}$ is calculated using our application level model, in particular Formula 3.10.

The network power consumption per thread is therefore the sum of the network power of all methods running in the thread as shown in the following formula:

$$Power_{thread}^{Network} = \sum Power_{methods}^{Network} \tag{4.6}$$

At the application level, we choose to use the duration of transmission, while at code level we use the number of bytes transmitted. This is motivated the **availability of data**. At code level, it is difficult to collect the duration of network transmission per method, while transmitted data size can be available easily. The usage of the transmitted data size is also valid because the duration of transmission is related to the number of bytes transmitted [Ing] as summarized in the following formula:

$$Time_{transmission} \simeq \frac{Size_{packet}}{Throughput} \tag{4.7}$$

4.3 Jalen: Measuring Energy Consumption of Java Code

JALEN is a Java implementation of our energy models at the code level [jal]. The availability of a virtual machine in the Java programming language helps us to retrieve information used in our model. Although we use Java as a main language for our implementation and studies, our approach is programming language agnostic and can be applied to different programming languages having similar concepts (such as the concept of methods).

4.3.1 Approach and Architecture

We develop JALEN, a runtime measurement software for estimating the energy consumption at code level for Java applications. JALEN uses power information provided by application level monitoring tools (such as PowerAPI, or implementing application level monitoring directly inside JALEN), in order to estimate the energy consumption of software code. JALEN provides energy information at a finer grain, *i.e.*, at the level of threads and methods (therefore, estimations can be aggregated and offered at a higher code level, *e.g.*, classes and packages).

The architecture follows our methodology of measurement (see Section 4.2.1) and is specified in Figure 4.2:

- We first collect statistics about the application's software and hardware resources utilization (see first and second steps in our methodology in Section 4.2.1). Information, such as methods durations, CPU time, or the number of bytes transferred through the network card, are collected and classified at a finer grain, *e.g.*, for each method of the application.

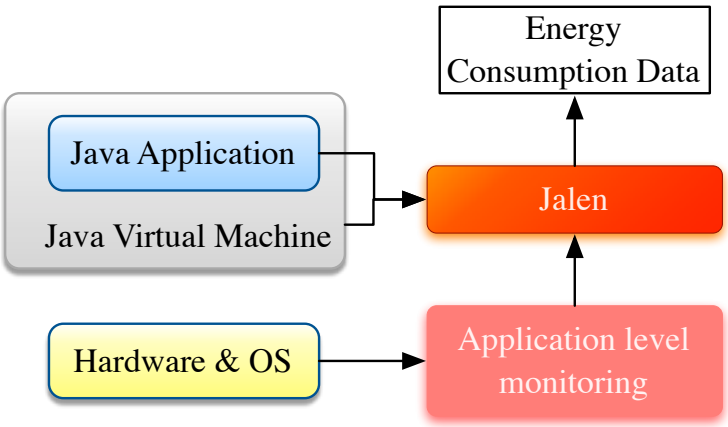


Figure 4.2: Jalen's architecture.

- Next, a correlation phase takes place to correlate the application-specific statistics with the application level energy information (see third step in Section 4.2.1). Per-method energy consumption information is calculated using our energy models (see Section 4.2).
- Finally, energy consumption per method is displayed to the user and can be exposed as a service (to be used, for example, in an application's autonomous adaptation cycle).

Next, we detail the implementation of JALEN.

4.3.2 Implementation

JALEN is implemented as a Java agent that hooks to the Java Virtual Machine during its start, and monitors and collects energy related information of the executed application. We develop two versions, each with different approaches on collecting and correlating information. The first version uses byte code instrumentation, while the second uses statistical sampling. Each holds advantages and disadvantages when monitoring energy consumption of software, and can be better applied than the other in certain contexts. We explain in details each of these implementations in the next sections, where we also compare these implementation approaches.

Instrumentation Version

The instrumentation version of JALEN uses byte code instrumentation technics, in order to collect resources usage information. In particular, we use ASM [Kul07, asm] to inject monitoring code into the methods of legacy applications. The instrumentation process goes as follows and is described in Figure 4.3.

Byte code injection First, we inject monitoring code at the beginning and the end of each instrumented method. The latters are instrumented based on their name, class, package or other characteristics such as their number of parameters. This filtering is specified in the settings of JALEN agent.

This injection can either be at runtime, where the JALEN agent injects code when a class is first loaded; or offline, where a special tool is used to inject code to the .class files of the program. We build both versions: *1)* the first version is an agent that instruments byte code at runtime and estimates the energy consumption; and *2)* the second is composed of two tools, a software that instruments offline the byte code of Java classes, and an agent that estimates the energy consumption. Both versions inject the same code and provide the same calculations. Differences are:

- The all-in-one agent has an additional overhead due to the cost of instrumenting Java classes at runtime. However this cost is limited as the instrumentation happens only once at the first class load.
- The two-tools version instruments all files of the program including files loaded with a different class loader at runtime. The all-in-one agent cannot instrument classes loaded by a different class loader.

Information collection The code injected at the beginning of methods collects information such as the full name of the method, the timestamp of method's execution and the depth in the method call tree (to detect children method, *i.e.*, methods that are started by an instrumented method). This information is useful for acknowledging where energy is being consumed (*e.g.*, the energy spend by a method excluding the energy spend by its children methods).

Call tree The code injected at the end of methods also collects the timestamp of method's end, and takes into account the restoration of the call tree up one level (when a method ends, the *hand* is given back to its parent method, or to the *main* method). This is because Java uses *stack frames* in its Java stack. Stack frames *contain the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual machine pops and discards the frame for that method.* [Ven99]. Therefore, our implementation monitors the energy consumption of a method excluding its called methods (*e.g.*, children methods).

Network information Network information are gathered by using a delegator to route all method call of sockets methods to a custom implementation where we add counters to count the number of bytes send and received by each method. We use a delegator class to route calls from the class `SocketImpl` [soc] to a custom implementation. We override the methods `getInputStream()` and `getOutputStream()` to monitor the number of bytes

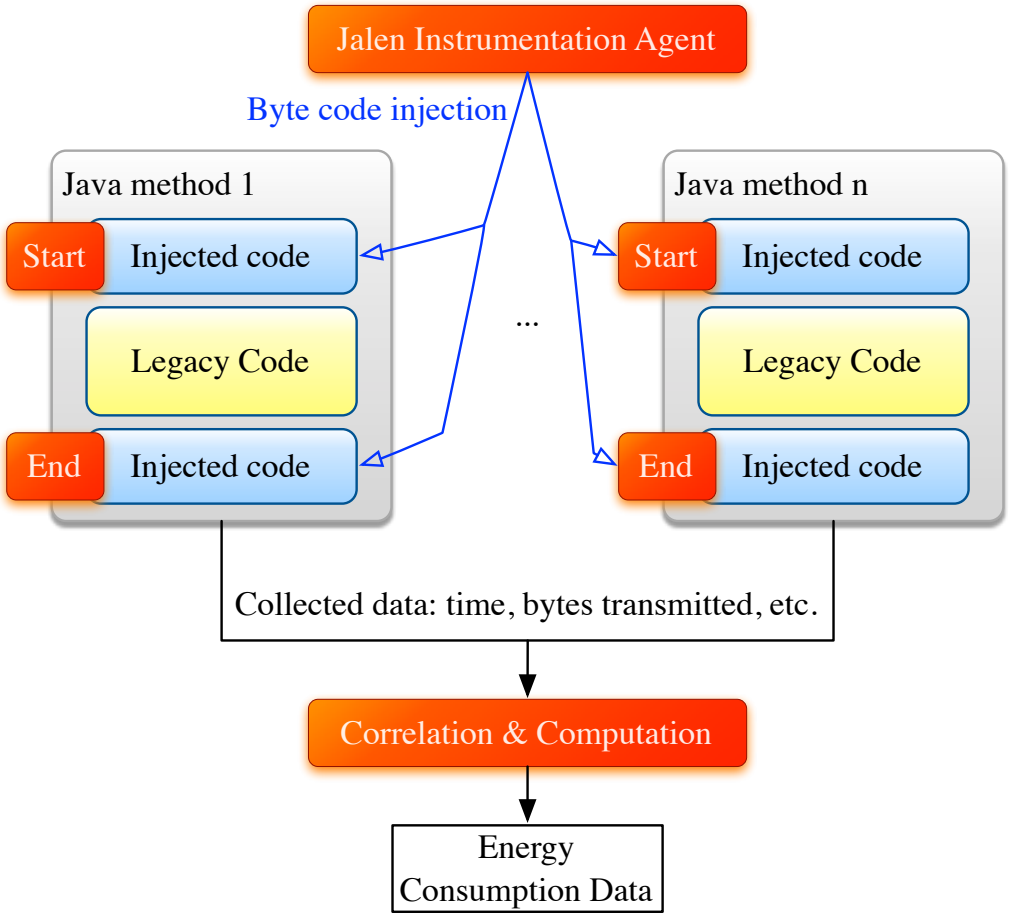


Figure 4.3: The approach of the instrumentation version of Jalen.

read and written to sockets. This information is then correlated with the method names invoking the methods `getInputStream()` or `getOutputStream()`, in order to get the number of bytes read/written by method.

Applying energy models Periodically, on each monitor cycle, or at the end of the program's execution, the JALEN agent processes the gathered data on each method invocation. It applies our energy models (see Section 4.2) and provides the energy consumed by each method on the terminal or saved in a file.

In the next section, we detail the statistical sampling version of JALEN.

Statistical Sampling Version

The statistical sampling version of JALEN collects information about running methods from the JVM, and correlates them with our energy models (see Section 4.2). This version follows

a sampling strategy that is outlined in Figure 4.4:

- We first follow a two cycle approach: a *big* monitoring cycle where power consumption of software is gathered from application level monitoring (see Chapter 3); and a *small* monitoring cycle where statistical information is collected on each running method.
- During the *small* monitoring cycle, we collect the number of times a method appears in our statistical sampling (measured at a higher frequency). For example, two method `AT` and `BT` are executing for 10 seconds, and the *big* cycle is 1 second and the *small* cycle is 10 milliseconds. The method `AT` is captured 7 times during the *small* cycle while `BT` is captured 3 times. Each of these methods have different execution times and CPU utilization, therefore both methods are scheduled and executed accordingly (for example, method `BT` waits for a network answer, thus the JVM executes `AT` during the wait).
- We then correlate theses statistics with the CPU time of threads (gathered from the JVM), in order to estimate the energy consumption of methods.
- For disk and network energy, we detect and count the calls to Java's JDK methods responsible for input/output and network (such as java.io, java.nio or java.net methods)

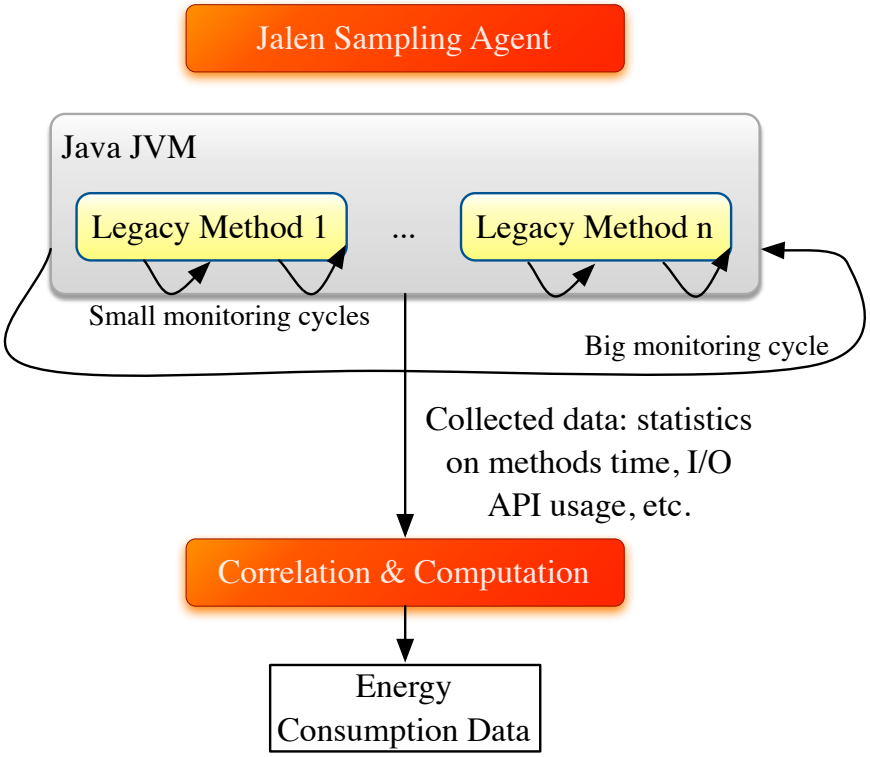


Figure 4.4: The approach of the sampling version of Jalen.

JALEN provides three types of information as shown in Figure 4.5:

- **All energy**: aggregated energy consumption, which means the energy consumption of each method including all the methods it calls (children methods). Here, the energy consumption of all methods running in the JVM is monitored, including Java's JDK methods (such as java.* methods).
- **Net energy**: energy consumption of methods excluding their children. Net energy is similar to all energy as it provides the energy consumption of all running methods in the JVM, but excluding the energy cost of children methods.
- **Net library energy**: energy consumption of certain methods (filtered by canonical name) excluding children methods. This type of information provides only the energy consumption of certain methods (filtered in the settings of JALEN), excluding their children, and excluding all other methods. This provides the energy consumption of methods of the monitored software without polluting results with Java's JDK methods. For example, for monitoring the energy consumption of a software library used by another application.

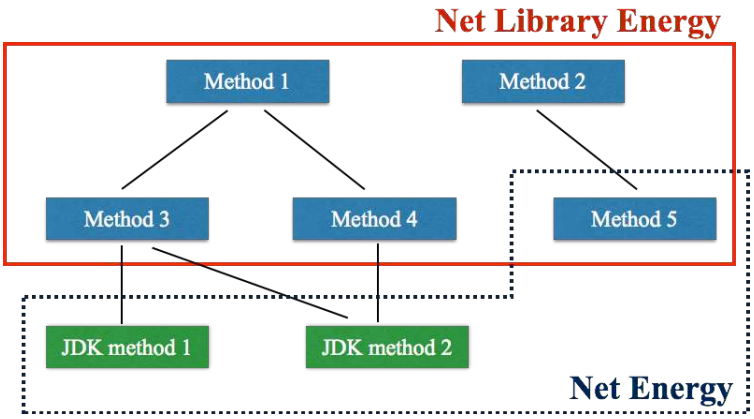


Figure 4.5: The energy information call tree provided by Jalen.

Byte code Instrumentation versus Statistical Sampling

Our two implementations of JALEN differ in many ways, whether in the monitoring strategy, granularity of measurements, or the cost overhead of the Java agent. Both approaches haves advantages and limitations that we outline in the next paragraphs.

Code modification Byte code instrumentation (in short, we will use *BCI*) requires the injection of additional byte code to the application's methods. This injection implies that *1)*

byte code modifications is available and allowed, and *2)* modifications is done when the class is first loaded, therefore resulting in additional load time overhead (typically at the start of the application). In contrast, statistical sampling (*STS* in short) does not require any modification of the application code.

Overhead In *BCI*, the additional code added at the start and end of each method is executed *each time* the method is executed. In particular, the cost of the injected code itself is constant throughout all methods, as this injected code does the same calculations independently to which method it is running in. Thus, in term of percentages, the overhead is small in big consuming methods, while it is high in smaller ones. However, the overhead is also strongly related to the number of method calls. A small method called lots of times will results in having an overall high overhead. We measure this overhead to be around 129% for Tomcat web server's individual requests.

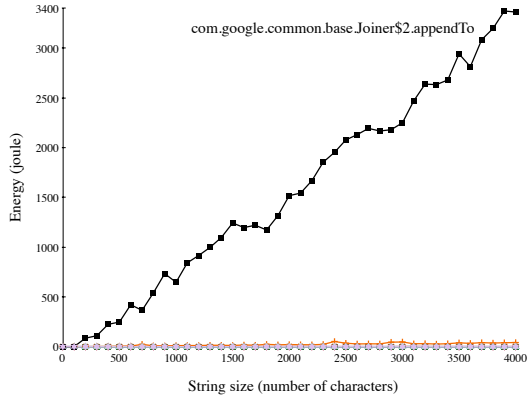
On the other hand, *STS*'s overhead is limited to the computations the JALEN agent does, which are run in a separate thread of the main application. The overhead to the application is therefore null, but the agent itself consumes energy and its overhead is measured to around 3%.

We also note that special care is required when analyzing information provided by the instrumentation version of JALEN. Instrumentation adds a fixed and constant overhead to all instrumented methods, therefore it is more visible (in total percentage) on small methods, or frequently executed methods. Values should be normalized by removing this constant overhead by a factor of the number of times the method is called.

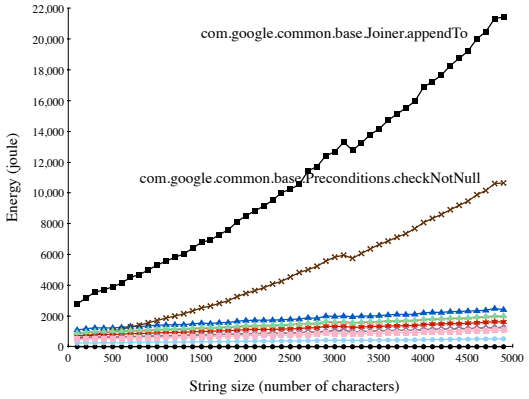
To outline this situation, we compare the energy consumption of Google Guava's [gua] `Joiner.join` method using the instrumentation version of JALEN (see Figure 4.6b), and the statistical sampling one (see Figure 4.6a). The figure reports the energy distribution of the methods called by `Joiner.join` method. Experiments are done on a Dell OptiPlex 745 workstation with an Intel Core 2 Duo 6600 processor at 2.40 GHz and running Lubuntu Linux 13.04 64 bits, version 1.6 of PowerAPI, and Java 7. We use version 14.0.1 of Google Guava library. Energy data are calculated each 500 milliseconds.

Both versions show similar energy trendline and linear evolution of the energy consumption. However, in the instrumentation version, small methods have high energy consumption, in particular, com.google.common.base.Preconditions.checkNotNull method. This latter is called four times in each join call (once in Joiner.iterable, once in Joiner.appendTo, and twice in Joiner.toString), and does nothing other than comparing if a parameter is equal to null. However, due to instrumentation, checkNotNull has an energy consumption ranging from 3% to 10% of the total energy when varying the size of the strings to join.

Measured resources The *BCI* approach measures directly from the method its resources consumption (*i.e.*, execution time, number of bytes sent or received in network sockets, etc.);



(a) Energy consumption of methods called by Google Guava's join method when varying its string parameter size, using statistical sampling version.



(b) Energy consumption of methods called by Google Guava's join method when varying its string parameter size, using instrumentation version.

Figure 4.6: Energy consumption of methods called by Google Guava's join method when varying its string parameter size, using statistical sampling and instrumentation versions of Jalen.

while *STS* samples the exposed data by the Java JVM and estimates the resources utilization based on statistics. Therefore, the *BCI* approach have better results as resources consumption are directly monitored instead of being statistically sampled. However, we find in our experiments that both exhibits acceptable results, and that the main differences are for methods that are fast to execute and that consume little energy. The latters have a small impact to the overall energy consumption, and due to their execution time and energy consumed, they may be missed by our statistical sampling version. Tuning the statistics parameters (in particular to allow shorter monitoring cycles), allows better resources monitoring for these methods. Figures 4.6a and 4.6b show that both *STS* and *BCI* present similar energy trendlines and evolution growth, with similar energy distribution between methods (when excluding the overhead cost of the instrumentation).

Method filtering In our implementation, *BCI* first filters methods to measure prior to the monitoring. Therefore, only selected methods are measured and instrumentation code is added to just the required methods. This is in contrast to *STS* where all methods are monitored, and filtering is done later at the correlation calculations. These two different strategies adhere better to the specifications to each approach: the high overhead of instrumentation pushes us to optimize *BCI*'s agent whenever possible, while *STS* is better achieved where all available information is collected then energy information is more accurately estimated.

In the next section, we conduct experiments in order to validate our models and our software implementation, JALEN. We also detect energy hotspots in software and the distribution of energy in software code.

4.4 Experimentations

4.4.1 Validation

Both versions of JALEN, byte code instrumentation and statistical sampling, uses PowerAPI as an application level library. PowerAPI uses our application level energy models that has been validated in Chapter 3. We also have developed an independent version of JALEN that implements both our application level and code level energy models directly in JALEN. In Section 3.3.2, we showed that the margin of error of our application level measurements is up to 3% on complex applications (*i.e.*, Tomcat and Jetty web servers or MPlayer). As the version of JALEN we use during our experimentations relies on PowerAPI for estimating the energy consumption of hardware at application level, it has the same margin of error and accuracy as PowerAPI for hardware estimations.

For code level validation, we run two sets of experiments: first, we measure the overhead and compare it to the overhead of software profilers (also due to the absence of similar code level energy profilers); second, we assess their accuracy by comparing the energy evolution with the CPU time evolution of CPU intensive applications running at 100% CPU, and with comparisons with another software profiler. The latter is relevant because we showed in Chapter 3 that *energy and time are linear for CPU-intensive applications* (see Section 3.4.3): for software that run at 100% utilization of the CPU, they have linear relation between energy consumption and execution time.

We run our experiments on a Dell OptiPlex 745 workstation with an Intel Core 2 Duo 6600 processor at 2.40 GHz and running Lubuntu Linux 13.04 64 bits, version 1.6 of PowerAPI, and Java 7. Energy data are calculated each 500 milliseconds. Sampling intervall for *STS* and HPROF is at 10 ms.

Accuracy

As no other software profiler provides energy consumption of software code (see Chapter 2), we validate the accuracy of our approach by comparing the energy consumption provided by our agent with the CPU time returned by our profiling and the results returned by the HPROF profiler [hpr]. As discussed earlier in Section 3.4.3, energy and time are linear for CPU-intensive applications when they use 100% of the CPU. Therefore, we use the same CPU-intensive application, the recursive Java version of the Towers of Hanoi program, in order to demonstrate that the results provided by JALEN are accurate.

Instrumentation version We compare the energy information provided by *BCI*'s version of JALEN with the estimated CPU time of methods returned by our profiler. Method TowerOfHanoi.moveDisk consumes 83.34% of the CPU and of its energy, while TowerOfHanoi.solveHanoi consumes 16.58%. Finally, the main method consumes 0.06% of

the energy. Results in Figure 4.7 show similar match between CPU time and energy, which is what we anticipated as we validate in Section 3.4.3 that time and energy are linear in this context.

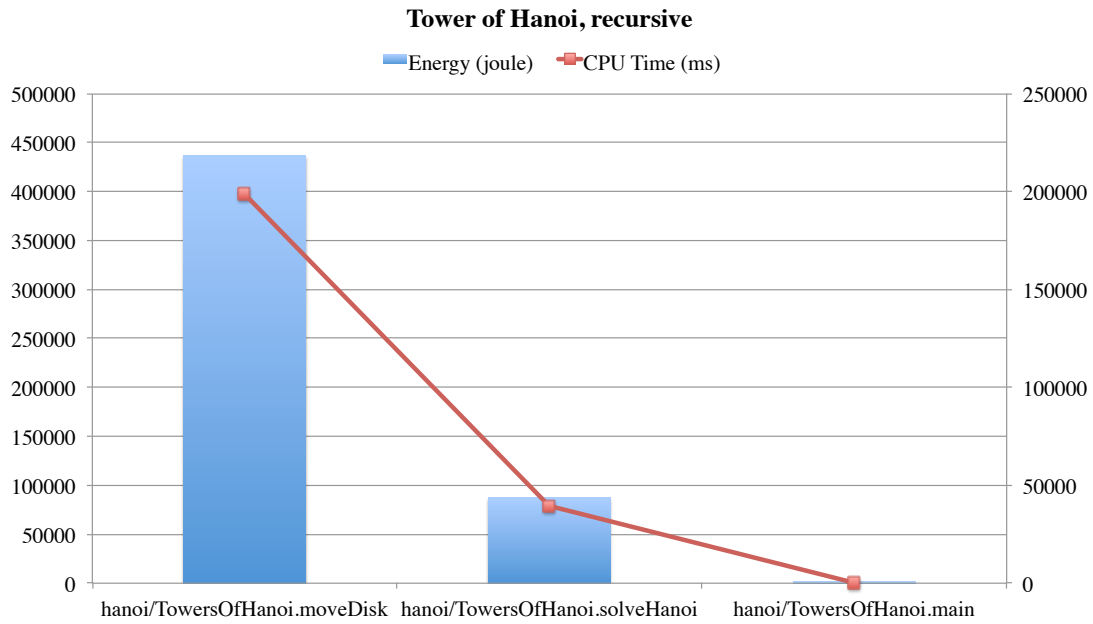


Figure 4.7: Comparison between energy consumption and CPU time of Tower of the recursive version of the Towers of Hanoi program.

Sampling version The *STS* version, on the other hand, does not use CPU time in order to estimate the energy consumption of software code. Therefore, we decide to compare it to HPROF, a software CPU profiling tool that also uses statistical sampling in estimating CPU usage of software code. The Java 2 Platform Standard Edition (J2SE) provides HPROF by default, as a command line tool. This tool estimates the CPU utilization percentage of all methods executing in the JVM. In contrast, JALEN can estimate the energy consumption of all methods, but also filter this estimation to a selection of methods (for example, limiting the estimation to the Tower of Hanoi's methods while excluding calls to the Java JDK's methods).

We compare the energy consumption provided by JALEN with the output information provided by HPROF. In HPROF, the `java.io.FileOutputStream.writeBytes` method uses 97.33% of the CPU during the execution of the program. *STS*'s version of JALEN provides an energy consumption of this method at 96.05%, thus a variation of 1.3% between JALEN and HPROF.

However, JALEN can also filter methods, therefore, when excluding JDK's methods, the results show that `hanoi.TowersOfHanoi.moveDisk` method consumes

99.92% of the energy. This is because `hanoi.TowersOfHanoi.moveDisk` calls `java.io.FileOutputStream.writeBytes` method (and other methods, such as `java.io.BufferedWriter.write` or `java.io.Writer.write`) in order to write the program's results to a file. In addition, `hanoi.TowersOfHanoi.moveDisk` itself have a net energy consumption of 0.73% (HPROF reports 0.13% for this method alone).

These numbers validate the accuracy of our statistical sampling, but they additionally outlines the impact of byte code instrumentation. The latter introduces additional energy cost due to the injected instrumentation code. We identified and discussed this impact in Section 4.3.2, and our current experiments in Figure 4.7 and in this section follow our previous observations.

Overhead

The overhead of any energy profiler, or any software profiler, for that matter, is crucial to its usability. In order to acknowledge the overhead of our agent during execution, we calculate the time per individual request in Tomcat 7.0.42 using ApacheBench 2.3. On 10,000 requests, the base mean time per request is at 4.157 ms in average. However, when using the sampling version, the mean time per request is at 4.289 ms in average. HPROF also has a similar overhead at around 4.336 ms in average. The instrumentated version has a time per request of 9.532 ms in average. The overhead of the sampling version is therefore at 3.17%, while the instrumentation version of JALEN have an overhead of 129.29% in comparison to the base Tomcat (see Figure 4.8).

Although the overhead percentage of the instrumentation version is high, it is similar to other software profilers that also use byte code instrumentation. The Java Interactive Profiler or JIP [jip] is a software profiler that uses similar byte code instrumentation of methods, however it does not produce energy related information. JIP 1.2 has a time per request of 8.241 ms in average in our experiment, thus an overhead of 98.24%. These metrics show the cost of instrumentation, and that our instrumentation version has an overhead similar to other software profilers that use byte code instrumentation.

Impact of sampling rate

Our statistical version samples data each 10 ms default. However, higher sampling periods impact the precision of the provided results. We vary the sampling rate from 10 to 50 ms with a hop of 10 ms, plus rates of 100 ms and 500 ms (500 ms being the minimal cycle recommended for the underlying application level library, POWERAPI, see Chapter 3), and reports the results in Figure 4.9.

Results show the impact of varying the sampling rate with better precision for lower rates (such as 10 ms), and more vague values for higher rates (as with 500 ms). In particular, some methods disappear in our results as higher sampling rates means quick methods are

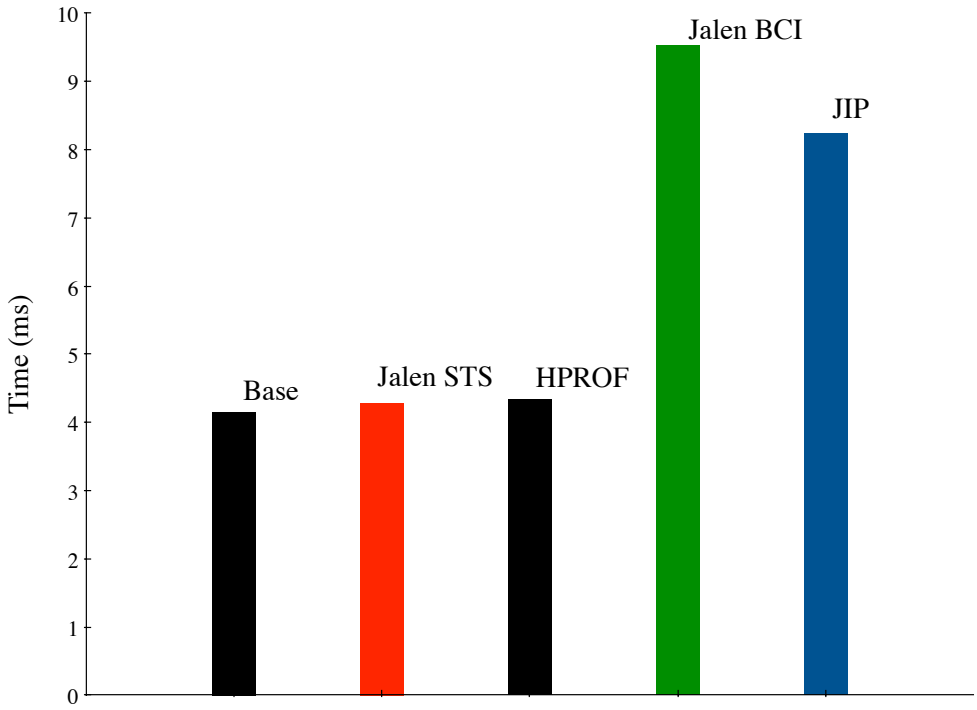
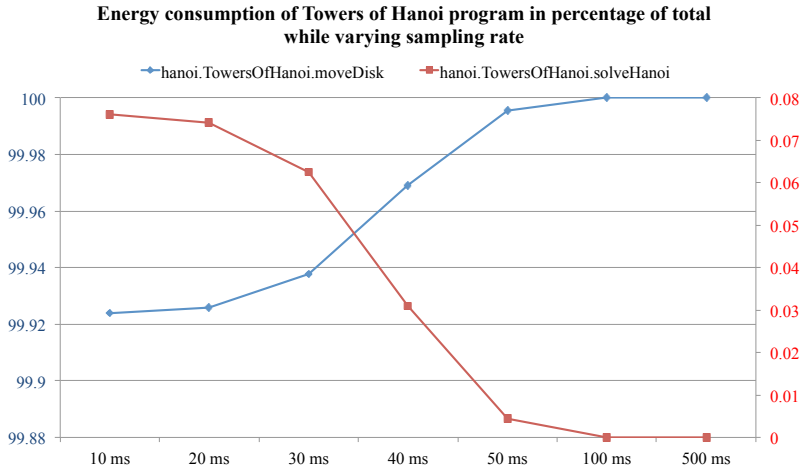


Figure 4.8: Overhead of individual Tomcat requests using ApacheBench.

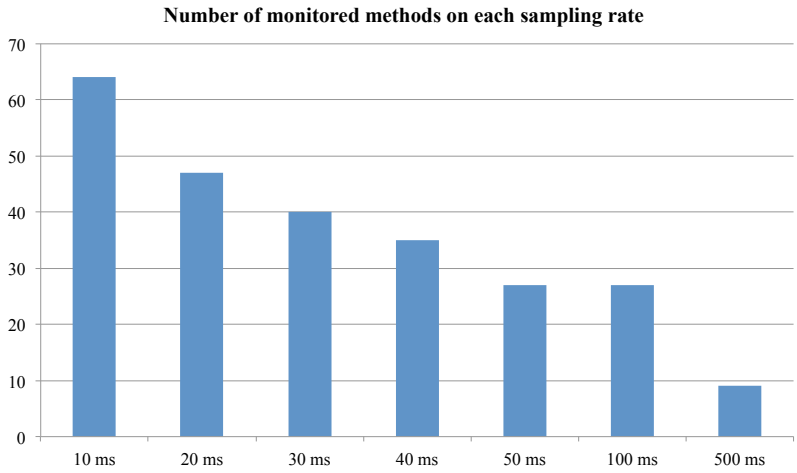
executed before being captured by Jalen *STS*. This is shown in Figure 4.9a where method `TowersOfHanoi.solveHanoi` disappears from results when sampling rate is equal or higher to 100 ms. Figure 4.9b reports on the decline of the number of captured and monitored methods when increasing the sampling rate. These numbers decline from 64 Java methods captured with a sampling rate of 10 ms down to only 9 methods when using a 500 ms sampling rate. Therefore, we argue on using the lowest possible sampling rate. In our implementation, 10 ms is the lowest rate as the monitoring and calculations done by our agent on each monitoring cycle requires between 5 and 8 ms.

Impact of different machines

Hardware components consume electrical energy. Our approach associates the energy consumption of hardware to the software code that initiated the task for hardware components. Therefore, energy consumption is highly dependent on hardware components. To illustrate the impact of changing machines to energy consumption of software, we run the Xalan benchmark in the Dacapo benchmark suite [dac] on two host configurations: a Dell OptiPlex 745 workstation with an Intel Core 2 Duo 6600 processor at 2.40 GHz and running Lubuntu Linux 13.04 64 bits; and a MacBook Pro 5,3 with an Intel Core 2 Duo T9900 processor at 3.06 GHz and running Mac OS X 10.7.5. We use version 1.6 of PowerAPI, and Java 7 on both



(a) Sampling rate using Jalen STS's Net Library energy model.



(b) Number of monitored methods on each sampling rate of Jalen STS.

Figure 4.9: Energy consumption percentage using the statistical sampling of Jalen, of the recusrive version of the Towers of Hanoi program.

configurations, with a sampling interval at 10 ms and energy data are calculated each 500 ms.

The results in Figure 4.10 of the first 10 methods show a similar energy consumption trend. Both experiments outline org.apache.xalan.templates.ElemLiteralResult.execute and org.apache.xalan.transformer.TransformerImpl.transform as the most consuming methods. The results show also similar energy percentage values for these two methods, at 22.98% and 10.37% on the Dell workstation, and 26.95% and 14.02% on the MacBook Pro, respectively. On the other hand, raw energy values in joules are different. While at the Dell workstation, templates.ElemLiteralResult.execute consumes 55.9 joules, it consumes 31.5 joules on the MacBook Pro (25.24 joules and 16.6 joules for transformer.TransformerImpl.transform,

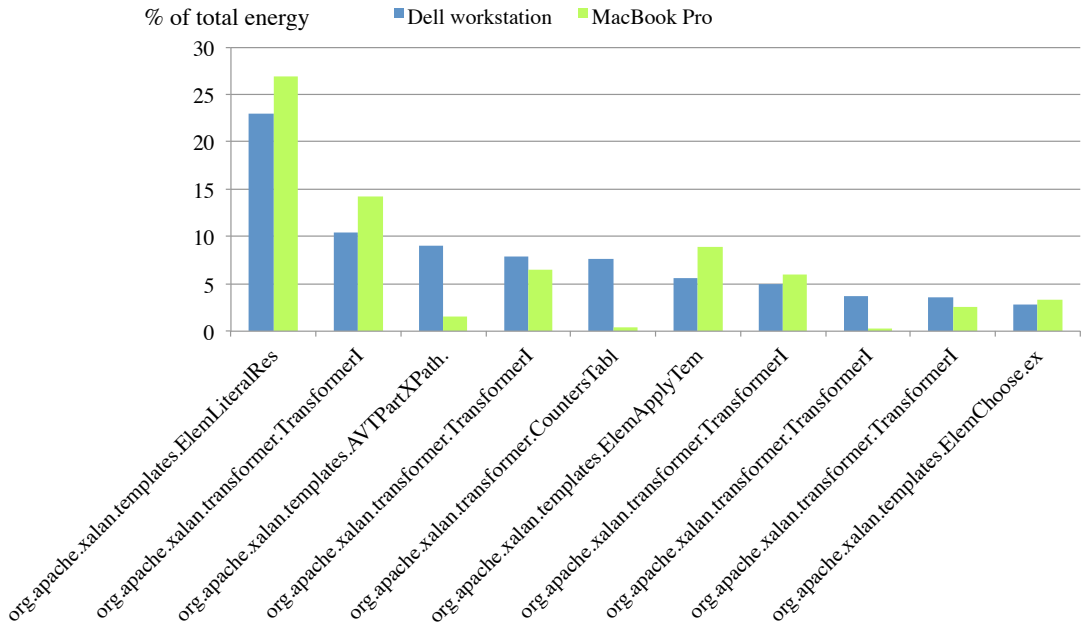


Figure 4.10: Percentage of CPU energy consumption of the top 10 most energy consuming methods of Xalan Dacapo benchmark, on a Dell workstation and on a MacBook Pro.

respectively). Therefore, the difference in percentage is 3.97% and 3.65%, respectively. On the same machine, the variance in percentage is 1.93% and 2.24%, respectively.

These results outline the importance of using percentages when comparing energy consuming of software code. This is mainly due to the different hardware that machines use, thus consuming different amount of energy while still keeping similar energy trends and distribution in software.

However, due to the high overhead of the instrumentation version of JALEN and the noise introduced by byte code instrumentation, we decide to use the statistical sampling version of JALEN in the remainder of this chapter.

4.4.2 Detecting Energy Hotspots in Software

The goal of our approach is to detect where the energy is being spent in software, or energy hotspots. This detection allows developers and other users to understand where and how the energy is consumed, and also to detect abnormal functioning in applications (*e.g.*, energy bugs).

We illustrate our approach with examples of complex applications: the Jetty web server and the Dacapo benchmark suite [BGH+06]. We use version 9.0.4.v20130625 of the Jetty

distribution, and Dacapo version 9.12. As with our previous experiences, we run our experiments on a Dell OptiPlex 745 workstation with an Intel Core 2 Duo 6600 processor at 2.40 GHz and running Lubuntu Linux 13.04 64 bits, version 1.6 of PowerAPI, and Java 7. Energy data are calculated every 500 milliseconds, and the sampling interval is at 10 ms.

Jetty Web Server

Jetty web server is a lightweight application server and javax.servlet container. It is an example of real world complex application, counting 105,156 source lines of code (SLOC) of Java in the version we use for our study. We stress Jetty's asynchronous REST web application example (async-rest) using ApacheBench. The latter uses 25 concurrent users with 100,000 requests. We run the experiment 5 times, for around 205 seconds in total execution time (the first run at 54 seconds, then the then the others run at 37 seconds in average due mainly to the Java JVM's JIT functionality).

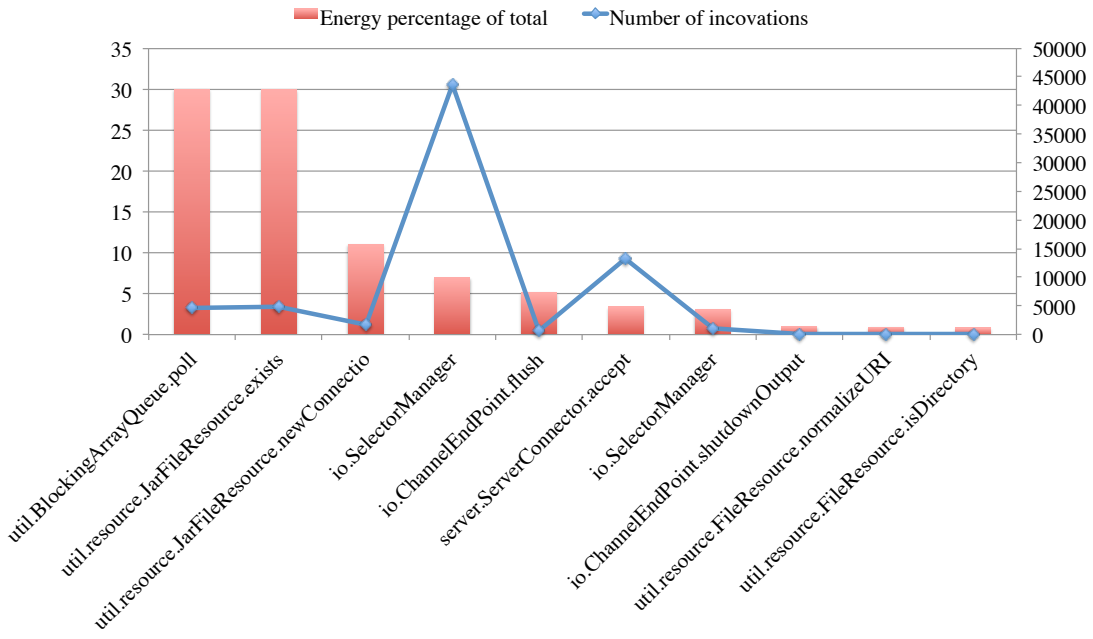


Figure 4.11: Energy consumption of the 10 most energy consuming methods of Jetty in our experiment.

Results are presented in Figure 4.11. The graph portrays the top 10 most consuming methods in term of CPU energy consumption in the X-axis. The left Y-axis (thus the bars) represents the energy consumed during the execution of the experiment in percentage of the total energy consumed at all measured Jetty methods. The right Y-axis (thus the line) represents the number of invocations of the methods. JALEN's *STS* provides the latter number.

The first observation is that the 10 most energy consuming methods of Jetty in this

experiment consume the vast majority of the energy, 92.18%. Specifically, two methods consume nearly 60% of the energy: org.eclipse.jetty.util.BlockingArrayQueue.poll (29.92%) and org.eclipse.jetty.util.resource.JarFileResource.exists (29.88%). Five other methods (util.resource.JarFileResource.newConnection, io.SelectorManager$ManagedSelector.select, io.ChannelEndPoint.flush, server.ServerConnector.accept, and io.SelectorManager$ManagedSelector.wakeup) consume between 3% and 11%, while the energy consumption of the remaining methods is negligible (less then 1%).

In contrast, the same methods are also the most invoked. Nevertheless, we observe that two methods have a high invocation number with lower energy consumption. This is the case for io.SelectorManager$ManagedSelector.select and server.ServerConnector.accept methods. The former is the most invocated, 43,624 times and consumes 7% of the total energy (or 487.34 joules on our configuration). The latter is invocated 13,250 times and consumes 3.38% of the total energy (or 236.28 joules).

In order to understand better the energy hotspots in software, we introduce energy per invocation (epi) unit, which is the energy consumed by one invocation of a method, and is calculated by dividing the energy consumption by the number of invocation. The two most invoked methods have therefore a low epi in comparison with less invocated methods (and sometimes less energy consuming methods). Figure 4.12 outlines the epi of the 10 most energy consuming methods in our experiment. We observe that in average, the epi of most methods is between 0.4 and 0.5 joule, with the notable exception of the two most invocated methods.

In addition to detecting hotspots at the methods level, our approach can detect most energy consuming classes. Figure 4.13 outlines the 6 most consuming classes of Jetty during our experimentation. These 6 methods consume together 96.02% of the total energy consumed by Jetty classes. The remaining 129 classes consume the rest, 3.97%. We observe that two classes consumes more than 70% of the energy: util.resource.JarFileResource (40.93%, 2 methods invoked) and util.BlockingArrayQueue (30.07%, 4 methods invoked). These two classes are averaged sized classes with the former counting 291 SLOC and the latter 691 SLOC.

Our benchmark stress scenario can explain these results. The former stresses Jetty's asynchronous rest web application example. This web application *uses Jetty asynchronous HTTP client and the asynchronous servlets 3.0 API, to call an eBay restful web service* as explained in [Wil]. When the initial request passes to the servlet, it is detected as the first dispatch, thus the request is suspended and a queue list (to accumulate results of requests) is added as a request attribute. This explains the energy consumption of the `util.BlockingArrayQueue` class and its methods. After the suspension, *the servlet creates and sends an asynchronous HTTP exchange for each request*, and when all responses are received, the results are retrieved and a response is generated. The calls for util.resource.JarFileResource class, and its `exists` method which checks whether a represented resource jar exists, and its `newConnection` method that is used for connecting to JAR resources, are explained by the need to access

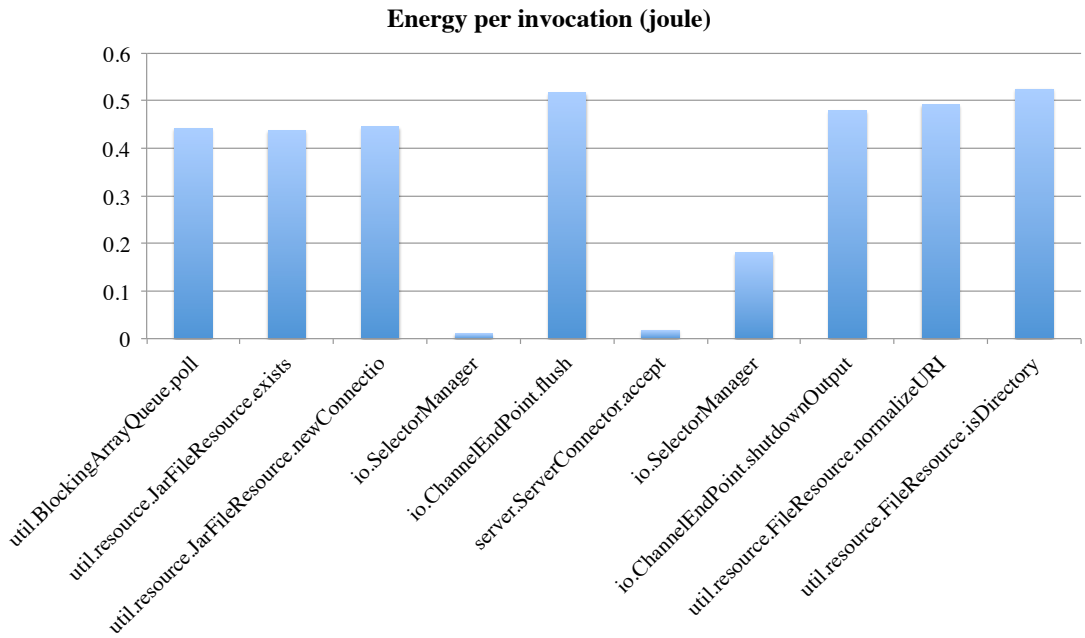


Figure 4.12: Energy per invocation (epi) of the 10 most energy consuming methods of Jetty in our experiment.

Jetty's own jar files and the web application's jar files. The experiment is run multiple times, and the asynchronous example is a relatively a small example, therefore this jar access is notable in term of total energy consumption percentage.

Dacapo Benchmark

Dacapo benchmark is a benchmark suite for Java that uses *real world* and open source applications. We use Dacapo version 9.12, and select 5 benchmarks that represents a variety of execution scenarios of Java applications, ranging from XML transformation, to Java classes analysis, SVG image manipulation and JDBC benchmark. The benchmarks and their behavior as explained in [dac] are the following:

- avrora: simulates a number of programs run on a grid of AVR microcontrollers.
- batik: produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.
- h2: executes a JDBC bench-like in-memory benchmark, executing a number of transactions against a model of a banking application.
- pmd: analyzes a set of Java classes for a range of source code problems.

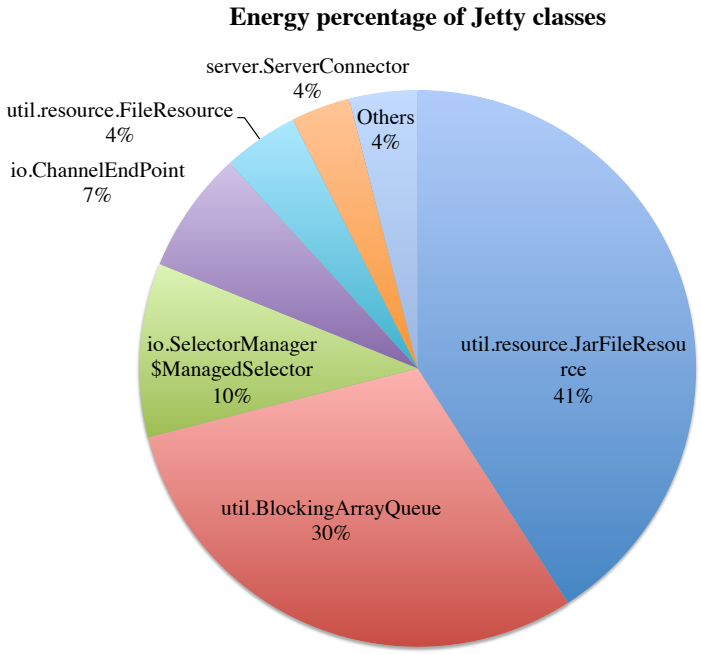
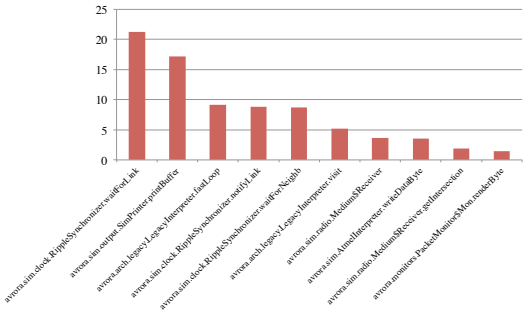


Figure 4.13: Energy consumption in percentage of the 6 most energy consuming classes of Jetty in our experiment.

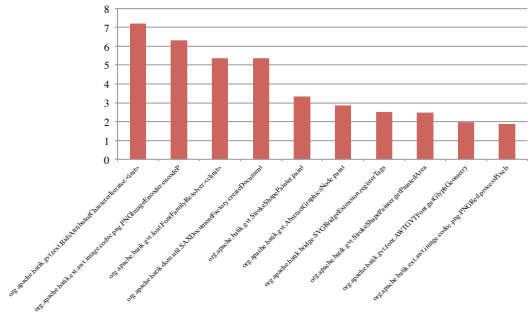
- xalan: transforms XML documents into HTML.

We report the energy consumption of methods of the benchmark tests in Figure 4.14. The results show different patterns of the energy distribution between methods of each benchmark, and are explained by the nature of each benchmark. The pmd, Xalan and Avrora benchmarks outline strong concentration of the energy consumption in one or few methods. In Xalan, org.apache.xalan.templates.ElemLiteralResult.execute method consumes nearly 23% of the energy, twice or three times more then the next most consuming methods. The same pattern is happening with pmd and Avrora benchmarks. These three benchmarks call a small number of methods frequently and for longer periods of time, thus accumulating energy consumption in few methods.

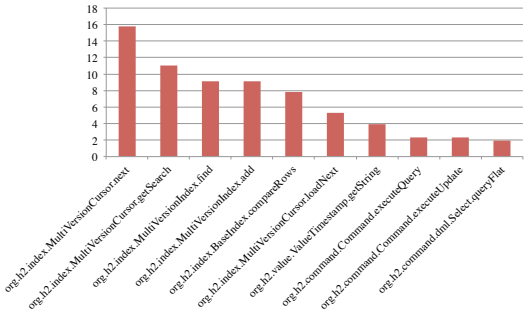
On the other hand, Batik and h2 benchmarks show an even distribution of energy between methods, where energy consumption is more evenly distributed between the top most consuming methods. These two benchmarks consist in generating SVG images and executing database transactions, therefore calling multiple methods for smaller periods of time.



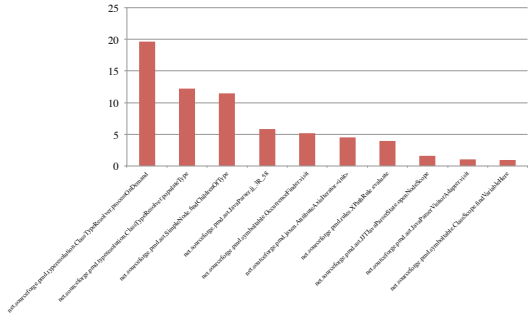
(a) Avrora benchmark.



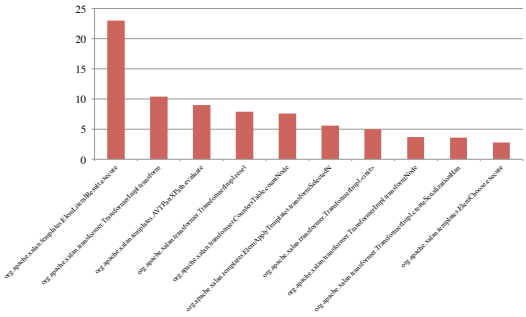
(b) Batik benchmark.



(c) h2 benchmark.



(d) pmd benchmark.



(e) Xalan benchmark.

Figure 4.14: Percentage of CPU energy consumption of the top 10 most energy consuming methods of 5 Dacapo benchmarks.

4.5 Discussions and Limitations

From our work and experiments on monitoring energy consumption at code level, we can better understand the energy consumption and distribution in software. In particular, our approach can detect energy hotspots in software and identify where the energy is spent at code level. The learning we got and limitations of our work are summarized in the next paragraphs.

4.5.1 Code level monitoring allows energy hotspot detections

The main conclusion of our work is that our approach allows developers to detect energy hotspots in software code. Experimentations (see Section 4.4.2) on both simple algorithms (*e.g.*, Towers of Hanoi), complex software (*e.g.*, the Jetty web server), and on different *real world* scenarios (*e.g.*, the Dacapo benchmarks), show the potential of our approach in energy code profiling.

We argue that the information provided by JALEN on energy consumption of software code can help developers to investigate alternative implementations of their classes and methods in order to reduce the energy footprint of their applications. By keeping track of the energy footprint of classes and methods, we think that development tools (*e.g.*, coding completion systems, documentation, debuggers, etc.) could be extended to help developers to build *greener* software. Related work in [Hin12] proposes to keep track of the energy consumption of software across its versions. However, the author does not measure energy at the code level and uses a hardware power meter for measurement. Investigating the energy consumption of software across versions and at the code level allows more accurate feedback of the energy hotspots responsible for the rise (or fall) of the energy consumption in the application.

4.5.2 Statistical sampling and instrumentation are both accurate approaches

In our experiments in Section 4.4.1, we find that using byte code instrumentation or statistical sampling offers accurate and valid energy information at code level. Similar match between CPU time and energy consumption is demonstrated in Figure 4.7 which validates the accuracy of instrumentation (this is the case because the software uses 100% of the CPU all the time, therefore rendering the energy consumption of software a function of time as demonstrated in Chapter 3). The statistical sampling version is also validated as reported in Section 4.4.1 with a variation of only 1.3% between energy values calculated by JALEN and CPU time estimations of HPROF profiler.

These results also show the validity and accuracy of our energy models (see Section 4.2). Both implementations of JALEN, byte code instrumentation and statistical sampling, uses the same energy models. Even though we use two different and radical implementations of the same models, we get similar energy consumption results (when taking into account the overhead of instrumentation as outlined in Section 4.3.2).

4.5.3 Prefer percentage over energy raw values

Energy consumption of software code is measured in joules, and is the energy consumed by hardware due to tasks initiated by software. As such, changing hardware will also change

the raw energy values even for the same software code. The same experiment run on different hardware produces different energy values due to the physical nature of hardware components (for example, running the same program on a laptop or on a server).

However, percentages values do not highly change with hardware, as software code is running the same tasks and using hardware resources accordingly. Our experiment in Section 4.4.1 and in Figure 4.10 illustrates the limited impact of changing hardware on the percentage distribution of the energy consumption in software code. The goal of our approach is to observe trends in energy consumption and profile applications to detect energy hotspots. Therefore, we argue that using percentages when comparing energy consumption of methods and classes is more useful and representative than raw values.

4.5.4 Prefer statistical sampling over byte code instrumentation

One major advantage of using the statistical sampling version of JALEN is its negligible overhead cost. Our results in Section 4.4.1 show a low overhead for sampling (at around 3%), in comparison to the high overhead of instrumentation (at around 130%). This overhead cripples any *real world* uses of instrumentation for energy measurements. However, we found that sampling provides also accurate enough values for detecting energy trends and energy hotspots. Therefore, using statistical sampling offers the best tradeoff between accuracy of results and low overhead.

Besides execution overhead, byte code instrumentation *pollutes* the energy results of each method by the energy cost of the instrumentation code itself. Without normalizing the results and removing this additional cost (that is dependent on the energy cost of the instrumentation code, and on the number of times it is executed), energy results are not correctly reported as we show in Section 4.3.2. The energy consumption of small methods is reported as higher than it is, and frequently executed methods have a high overhead due to the additional cost of multiple execution of the instrumentation code.

4.6 Summary

In this chapter, we presented energy models in order to estimate the energy consumption of software at the code level. Our approach uses energy models and offers a high accuracy as seen in Section 4.4.1. Our models are implemented by a software energy profiler named JALEN. The latter allows measuring the energy consumption of blocks of code at the granularity of methods. We implement two versions of JALEN, one that uses byte code instrumentation and another that uses statistical sampling technics. Both offer good accuracy, however the latters has less impact on the application's code and execution in comparison to instrumentation.

Based on our models and on JALEN, we conduct series of experimentations aimed to detect energy hotspots in software. Our results show that changing the configuration machine

also changes the energy consumption, but the distribution of energy between methods remains the same (see Section 4.4.1). We detect the most energy consuming methods as seen in our experiments in Section 4.4.2. We also introduce the energy per invocation (epi) measurement unit (see Section 4.4.2), which allows viewing hotspots in term of individual method execution. Finally, we outline energy consumption trends and hotspots in *real world* and diverse applications in the Dacapo benchmark suit (see Section 4.4.2).

4.7 The Need for Energy Evolution Modeling

Our experiments in Section 4.4 reports on the energy distribution and hotspots of software code in a specified context. It allows identifying the most energy consuming methods or classes when executing software with a fixed set of parameters and configuration. Our approach is similar to *screenshots*, where we take an energy snapshot of software code. Although this is useful for energy debugging, energy optimization and helps developers write more energy efficient code, it lacks execution variability.

When modifying the input parameters of an application, we expect that energy consumption will also vary. This variation may increase energy consumption, decrease it, or even not change at all. Knowing the impact of varying input parameters on the energy consumption of applications is useful for adaptive software, and for software in evolving environments. This also is important for software libraries as they are used by multiple applications. Each of these applications use the libraries using different input parameters, therefore having the energy consumption model based on input parameters is relevant. However, developers do not have any empirical results, nor tools or approaches in order to study the impact of this variation on the energy consumption. In the next chapter, we propose an approach and tools to model the energy evolution of software code based on the variability of their input parameters.

Chapter 5

Unit Testing of Energy Consumption

"It takes as much energy to wish as it does to plan."-Eleanor Roosevelt

Contents

5.1 Introduction

Measuring the energy consumption of software is the first step into producing energy efficient code. We demonstrated in the previous chapters (see Chapter 3 and 4) our approaches, methodologies and tools in order to take a snapshot of the energy consumption of software. These snapshots vary in granularity, estimating the energy consumption of a whole application down to providing detailed reports on how much each portion of code consumes.

Nevertheless, the energy information reported by our approaches is static, *e.g.*, values are related to an execution of software in one specific configuration. Changing a parameter in a method or modifying an input parameter therefore requires a new execution of the application in order to get the new energy consumption and the impact of this change. Thus, what if developers had tools to empirically measure the energy consumption of their software code, and get empirical data about the energy evolution trends in their code? And also get the impact of changing input data and parameters on the energy consumption of methods? These data can be used to diagnose the code to detect energy bugs, understand the energy distribution of the application, or establish an energy profile or classification of software.

Our contribution is therefore proposing approaches and tools in order to automatically infer the energy models of software methods based on their input parameters. This can partially be done manually by modifying a parameter then measuring the energy consumption using our approaches and tools from Chapters 3 and 4. This is time consuming, and in particular limited to the set of parameters the developer use, but most importantly it has limited additional value to the developers community.

However, one main motivation of automating these measurements is software libraries. The latter are used by other software and therefore improvement in their energy efficiency would benefit to a large pool of applications. Benchmarking libraries for their energy consumption and proposing empirical models of the evolution of their energy consumption are *win-win* situations for software developers. We already illustrated in our previous chapters that running software on different machines does not necessarly change the energy distribution between their software methods (see Section 4.4.1). Thus, proposing an empirical model of the evolution of energy consumption of software code is relevant, and can be used by the developers' community without having to benchmark again software libraries on developers' and users' configurations.

In this chapter, we propose our approach of inferring the energy evolution model of software code in Section 5.3.1. We then describe our automatic benchmarking framework, JALEN UNIT, in Section 5.3. In Section 5.4, we report the results of our experimentations using JALEN UNIT. Finally, the results and our approach are discussed in Section 5.5.

5.2 Modeling Approach

Our approach models the energy evolution trend of a software method by running benchmarks on a method while modifying its parameters. Concretely, we provide the energy evolution model of a method based on the evolution of its parameters. This provides a relational table between methods and their energy model, therefore allowing developers to choose the best energy efficient method for their software. In details, we vary the value of the input parameters of methods and measure their energy consumption using each of these values. At the end, we obtain the energy consumption of the method for each value of its parameters, therefore allowing us to have an energy evolution view of the method.

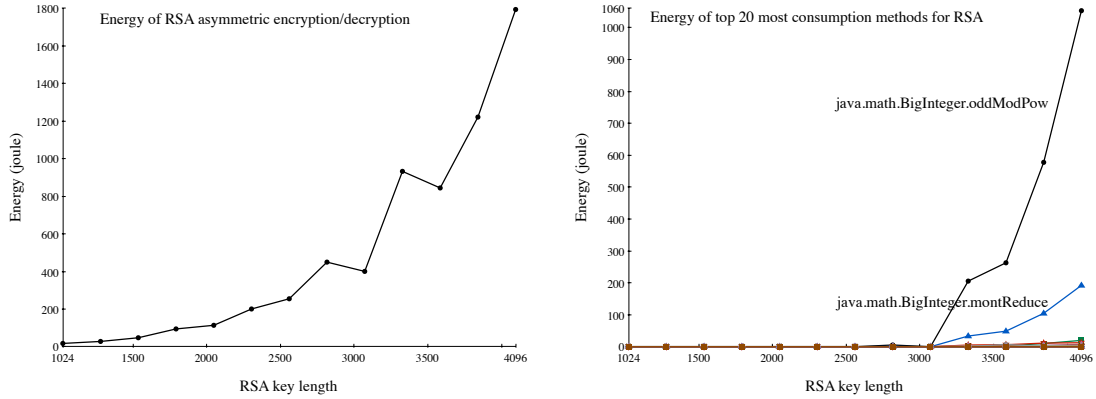
To illustrate our approach, we measure the energy consumption evolution of an RSA algorithm and outline the advantage of code level modeling. Then we use Google Guava's `Joiner.join` method to illustrate how one method can have different energy evolution models. The experimentations are done on a Dell OptiPlex 745 with an Intel Core 2 Duo 6600 processor at 2.40 GHz and running Lubuntu Linux 13.04, version 1.6 of POWERAPI, the statistical sampling version of JALEN, and Java 7. We collect energy data each 500 milliseconds and the sampling interval is at 10 ms.

5.2.1 Code level modeling

We take an RSA asymmetric encryption/decryption algorithm [RSA78] and measure its energy consumption while varying the length of the RSA public and private keys. RSA algorithm is a example of an algorithm where its input parameters (here, the RSA key) impact the functionnality of the said algorithm, *e.g.*, in term of security, robustness of encryption, and speed of encryption/decryption process. Our use case scenario generates an RSA key, then encrypts and decrypts 10 times a random BigInteger with a bit length of 10,000. We use our application level library and our code level agent, JALEN to measure the energy consumption of the RSA algorithm.

The results, in Figure 5.1a, show an exponential rise in the energy consumption of the RSA algorithm when increasing the RSA key length. Even though these numbers show the evolution of the energy consumption of the RSA algorithm, we want to understand which portion of the code is responsible for the exponential increase. Therefore, we use the statistical sampling version of JALEN to measure the energy consumption of the classes and methods of the RSA algorithm. Results, in Figure 5.1b, show that two methods are responsible for the majority of the energy consumption: `java.math.BigInteger.oddModPow`, and `java.math.BigInteger.montReduce`. From these methods, `oddModPow` has a clear exponential increase, while `montReduce` follows a logarithmic growth.

What these resutls outline is the importance of code level measurement over only application level measurement. The exponential energy evolution in Figure 5.1a is therefore explained and identified in Figure 5.1b when benchmarking at code level. It allows us to discover that `java.math.BigInteger.oddModPow` method is the culprit of the exponential



(a) Energy evolution of RSA algorithm.

(b) Energy evolution of RSA methods.

Figure 5.1: Evolution of the energy consumption of RSA asymmetric encryption/decryption according to key length.

evolution. The results also show that varying the RSA key length exponentially impacts energy consumption, and that we manage to model this evolution down to the code level. The RSA encryption/decryption algorithm is an exponential one as described in [RSA78]. Therefore, our experiment results provide additional validation to our measurement approach. In particular, the method responsible for the exponential growth in energy consumption in our implementation of RSA algorithm is the method that does the exponential calculation, `oddModPow`.

Next, we report on how a same method can have different implementations, thus different energy models.

5.2.2 One method, different models

We infer the energy evolution model of Google Guava's [gua] `Joiner.join` method. The `join` method takes two or more strings and joins them to return a bigger string. We vary the size of the strings while joining two strings, and also vary the number of strings while having a fixed string size. The join method in the Joiner class is an example of an overloaded method. It has 5 different implementations in the `Joiner` class, and 9 in total including the implementations in the `MapJoiner` inner class. `Join` method calls a method named `appendTo`, which is also overloaded and implemented 10 times in `Joiner` class, and 19 times including implementations in the inner class. When joining 2 strings, the `join` method calls 18 times other methods and constructors of the Google Guava library. Therefore, the complexity of a method with a straightforward algorithm makes `Joiner.join` method a perfect example for establishing energy consumption model.

We use version 14.0.1 of the library, and stress the `join` method of `com.google.common.base.Joiner` class by varying its parameters. For each of

these two experiments, we generate a random string that we use during the join call. We run the join stress one million times with the generated string, and repeat the stress 10 times with different strings of the same size. Finally, we record the energy consumption of the overall execution.

Varying String Size

We first vary the size of the two strings to join, from zero (empty strings) to 4,000 characters with a hop of 100. Results in Figure 5.2 show the distribution of the energy consumption of the `join` method. The method itself, and most of the methods it calls, consumes negligible energy (decreasing from 2.60% to 1.23%). On the other hand, the `common.base.Joiner.appendTo` method consumes nearly all the energy (from 97.16% up to 98.69%), and grows linearly with the increase of the size of the strings to join. These results show that the `join` method delegates practically all of its work to the `appendTo` method, which effectively performs the join (by using Java's internal methods).

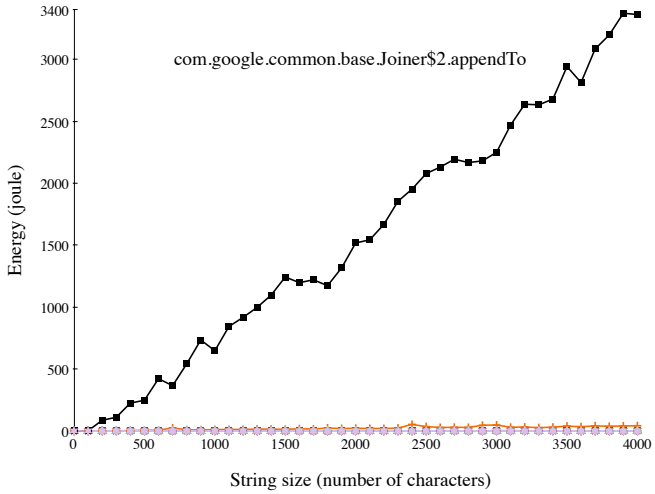


Figure 5.2: Energy consumption of Guava's join method when varying the string size.

Varying Strings Number

We then vary the number of strings to join while maintaining a fixed string size (*i.e.*, 100 characters). We vary the number of strings from 2 to 50 strings. The energy consumption results (see Figure 5.3) also show that `Joiner.appendTo` method consumes most of the energy (going from 97.14% to 99.36%). However, the energy evolution when varying the number of strings is not growing linearly as we saw when varying the string size. Instead, it alternate between phases of constant energy consumption with others of direct increase.

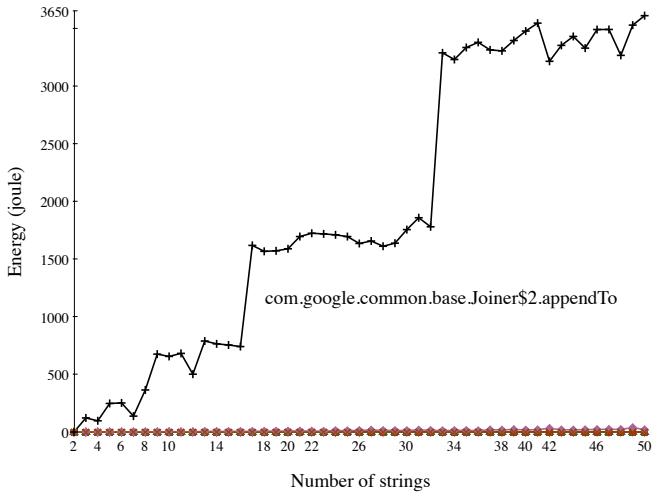


Figure 5.3: Energy consumption of Guava's join method when varying the number of strings.

These numbers are explained by the implementation of the `appendTo` method. It cycles through the strings to join (given in parameter as an iterable collection) and appends it to an *appendable* object also given in parameter. The `join` method uses a `StringBuilder` object for joining the strings, therefore calling the `append` method of the `StringBuilder` class (implementing `java.lang.Appendable.append` interface). When joining two strings while varying the strings size, the collection's length stays the same (2 elements), while the size of the strings object is growing. Therefore, the append is a linear function to the size of the strings as shown in Figure 5.2.

However, when varying the number of strings, the JVM is required to allocate memory for these strings. The strings in the string builder object are stored as an array of characters, and the JVM doubles the size of the array (until the new characters fits in the array) when appending new characters exceeding the initial size of the array [mem]. By default, the buffer size is 8192 characters in the JVM. Our experiment is run 10 times therefore reaching the limits of the buffer when joining 8 strings of 100 characters each (totaling more than 8000 characters). When the limit is reached, the JVM doubles the buffer allocation allowing more memory for joining the strings. This explains the burst of energy consumption when joining 9 strings in Figure 5.3. The joining of the strings has stable energy consumption and bursts of energy occur when the JVM needs to increase its buffer. This allocation occurs at a doubling interval, thus after joining 8, 16, 32, 64, *etc.* strings.

These results demonstrate the high potential of energy modeling in providing energy profiles of software code. We, therefore, propose a benchmarking framework called JALEN UNIT. The framework automatically stresses and benchmarks methods of an application while varying their parameters. It does so by using software injectors and empirically measuring the energy consumption of each execution. The next section describes in details our

framework, its approach and implementation.

5.3 Jalen Unit: Modeling Software Methods Energy Consumption

JALEN UNIT is an energy framework that generates energy models for software code based on empirical benchmarks. In this section, we explain our approach and architecture of JALEN UNIT, then describe its implementation.

5.3.1 Approach

Automating the generation and execution of energy benchmarks of methods invokes multiple challenges:

- Injecting valid parameters in benchmarked methods in order to have relevant execution and measurements.
- Modifying injected parameters using various strategies, therefore allowing different energy evolution models based on the software domain and context.
- Managing multiple parameters in methods, thus allowing understanding which parameter has the highest impact on energy consumption evolution in the method.
- Accessing, filtering and invoking methods with proper assessors. This is due to different encapsulation in Java software, therefore some methods and constructors require special care or filtering in benchmarking.
- And generating valid benchmarks and energy evolution models from the results.

JALEN UNIT provides benchmarks for modeling the energy consumption of software methods through automatic empirical benchmarking as illustrated in Figure 5.4. For instance, it generates individual benchmarks for each method in a software library, and for each of its parameters. These benchmarks allow stressing the method based on a set of values for its parameters. These values are determined through different injectors, and multi-parameters methods are managed through different strategies. Next, all generated benchmarks are executed. For each, we measure its energy consumption, then the results are aggregated and analyzed to produce the method's energy profile and evolution model.

Concretely, JALEN UNIT cycles through every package, class, and method in a Java library. For each method and each of its parameters, an energy benchmark is created following an evolution strategy for the benchmarked parameter. The benchmark is then executed and JALEN (see Chapter 4) is used to measure its energy consumption. Finally, energy data for the benchmark and the evolution of parameters is generated as an output file that is later plotted as a graph.

Next, we detail the implementation of our framework, then present our scenario and results obtained by JALEN UNIT framework.

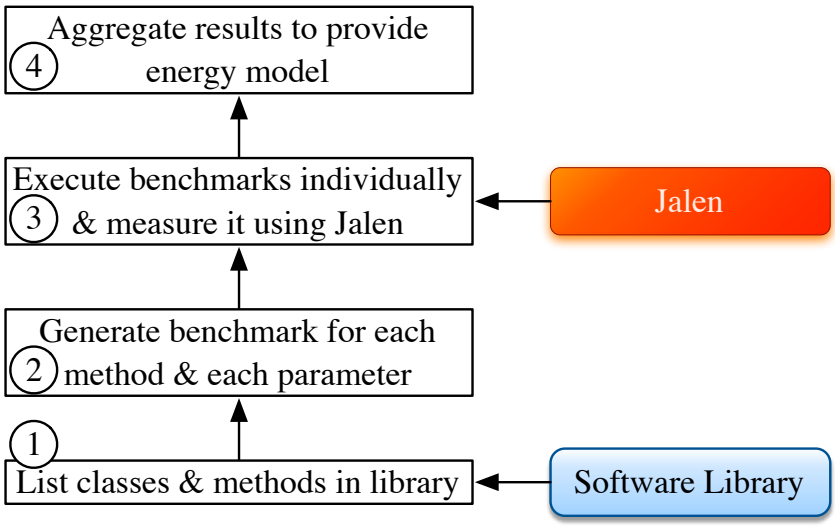


Figure 5.4: Jalen Unit approach.

5.3.2 Implementation

JALEN UNIT is built as a Java application that loops over all methods in a Java software library and generates energy benchmarks. The latter are then executed and their energy consumption is measured using JALEN automatically.

In details, JALEN UNIT generates and runs a benchmark for each method while varying its parameters. This variation of parameters is done through injectors implemented for Java primitive and object types. The framework can, therefore, be extended with user specific injectors describing alternative evolution strategies. Java objects can be benchmarked automatically if their injector model is implemented in the framework.

We implemented standard injectors for primitive types: Integer, Double, Long, Float, Boolean, and Character, in addition to the String object. We prefer to implement our own injector instead of using existing injectors, such as YETI [yet], because we want to provide different strategies for benchmarking and testing methods. YETI performs random testing. This provides a good strategy for detecting abnormal behavior in software code (such as exceptions or huge CPU load for certain values). However, it does not offer a comprehensive strategy for evaluating the energy evolution of methods by input parameters.

For example, we develop an injector for integers where the integer values evolve with an increment, from a start value to an end value (*e.g.*, integer values from 10 to 100 with a hop of 10 leads to 10 benchmarks with values of 10, 20, ... to 100). Another injector for integers evolves the integer randomly using the `Math.random` method in Java. Although integers are all of the same size, changing their value impacts the execution of methods, therefore their energy consumption. For example, an integer parameter that is used as an end value to

a for-loop may have a high impact because increasing its value implies that tasks are being executed for longer period of time and consuming more energy.

Injectors for other types also implement different evolution strategies, such as varying the length of a string object parameter randomly, or from a start value to an end value, or choosing the characters of the string from a subset of the alphabet. The evolution strategies are endless, and offer the advantage of better flexibility and extendibility of the framework. This flexibility is also useful for domain specific applications, where random testing is not representative of the *real world* workload. By providing extensible framework and providing freedom of choice in method evolution model strategies, we propose a framework that can be customized for specific needs. Therefore, better representative energy evolution models can be empirically achieved.

Concretely, an injector is a Java class implementing the `Iterator` and `JalenModel` interfaces. The latter adds additional methods to the iterator `next` and `hasNext` methods, such as a `getDefaultValue` method that returns an object of a default value of the injector. The following listing outlines the default integer injector (more examples of injectors are available in Appendix A):

```
1 package jalen.model;
2 import java.util.Iterator;
3
4 public class IntegerModel implements JalenModel, Iterator {
5         private final int start, end, increment;
6         private int current;
7
8         public IntegerModel(int start, int end, int increment) {
9                 this.start = start;
10                this.end = end;
11                this.increment = increment;
12                this.current = start;
13        }
14
15        @Override
16        public boolean hasNext() {
17                return this.current <= this.end;
18        }
19
20        @Override
21        public Object next() {
22                int result = this.current;
23                this.current += this.increment;
24                return result;
25        }
26
27        @Override
28        public void remove() {}
29
30        @Override
31        public Object getDefaultValue() {
```

```
32                  return this.start;
33          }
34 }
```

Multi-parameters methods are managed by varying one parameter alone, while the others have default fixed values. Others strategies are possible, such as varying multiple parameters while fixing the values of some, or modifying all parameters randomly. Our initial implementation, however, uses only the first strategy for multi-parameters methods.

Benchmarks are then run and their energy consumption is measured using our code level measurement tool, JALEN (see Chapter 4). Finally, the generated energy results are aggregated and the energy evolution model of method is inferred.

In the next section, we conduct experiments to infer the energy evolution model of methods in a Java software library.

5.4 Inferring Automatically the Energy Model of Software Libraries

We run our JALEN UNIT framework on the Violin Strings Java library [Sch]. This library is a collection of 138 methods (with many being overloaded) designed for manipulating strings. It extends the functionalities offered by `java.lang.String` by offering methods usually found in other programming languages such as C++.

The methods of the library use different input parameters: strings, characters, integers or booleans. We use our default injectors for these types. In particular, the strings injector model injects strings with different sizes, ranging from a start length of 100 up to 1,000 characters with a hop of 200. The integer, float, double and long injectors' models inject numbers ranging from 100 to 1000 with a hop of 200. The character injector model injects a random character of the 26 characters of the English alphabet. Finally, the boolean injector model injects both *true* and *false* values. Each benchmark is finally run multiple times and result values are reported in the next paragraphs.

The experimentation is done on a Dell OptiPlex 745 with an Intel Core 2 Duo 6600 processor at 2.40 GHz and running Lubuntu Linux 13.04, version 1.6 of POWERAPI, the statistical version of JALEN, and Java 7. Energy data are calculated each 500 milliseconds and the sampling interval is at 10 ms.

Next, we present the results of a selection of methods of the Violin Strings library benchmarked using JALEN UNIT.

5.4.1 One-parameter method: Reverse

The `reverse` method reverses the sequence of characters of the input string. The benchmark runs the `reverse` method 1,000,000 times in order to get enough execution time for estimating the energy consumption. This is due to the limitation of the underlying POWER-API monitoring cycle, where a minimum cycle of 500 milliseconds is required for accurate measurements

The implementation code of the reverse method is as follow:

```
1public static String reverse(String s) {
2   StringBuffer buf = new StringBuffer(s);
3   buf.reverse();
4   return buf.toString();
5}
```

Results in Figure 5.5a show a logarithmic evolution model of the method when varying the number of characters of the input string. This is due to the underlying call to Java's `StringBuffer.reverse` method which loops over all characters in the string (a for-loop to the size of the string) and produces a string buffer containing the reverse of the string (which is then casted as a string).

5.4.2 Two-parameters methods: Copies & Center

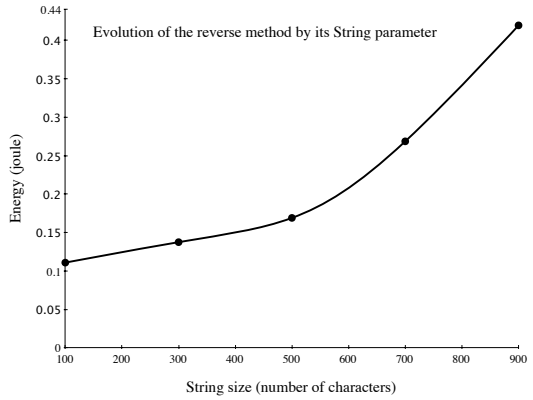
Copies

The `copies` method takes an input string and an input number, *nCopies*, and generates a string consisting of *nCopies* of the input string. The method is run 100,000 times.

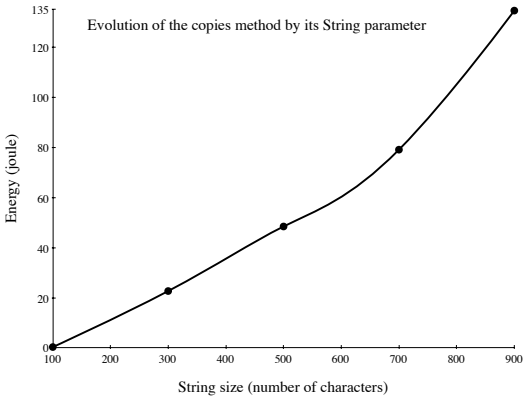
The implementation code of the copies method is as follow:

```
1public static String copies(String s, int nCopies) {
2   int ls = s.length();
3   if (nCopies < 1) {
4      return "";
5   }
6   StringBuffer buf = new StringBuffer(ls*nCopies);
7   for (int n = 0; n < nCopies; n++) {
8      buf.append(s);
9   }
10  return buf.toString();
11}
```

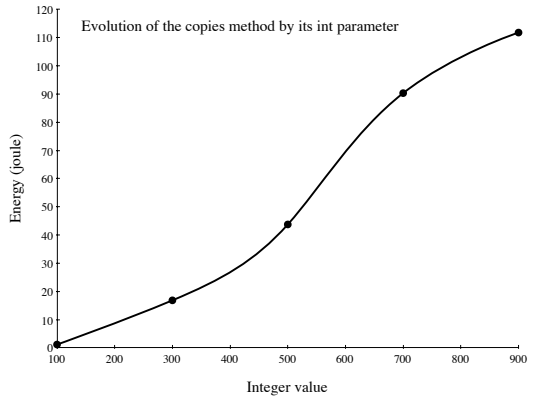
Results in Figures 5.5b and 5.5c show a clear linear evolution of the energy consumption when varying the size of the string (while fixing the number of copies to 100), and when varying the number of copies (while fixing the size of the string to 100 characters). In details, the method first creates a string buffer of a size equal to *nCopies* $\times$ size of the string. Then



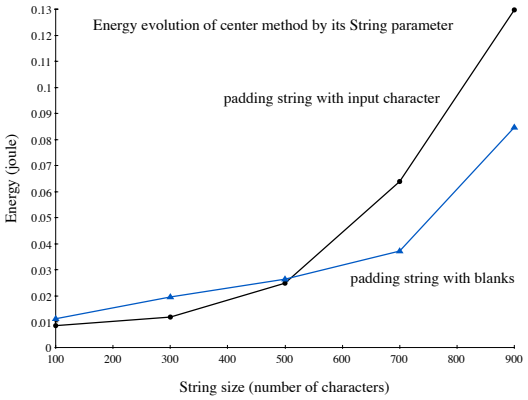
(a) Energy evolution of the reverse method by its string parameter.



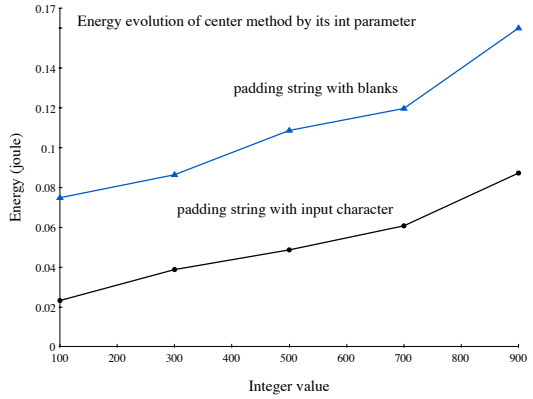
(b) Energy evolution of the copies method by its string parameter.



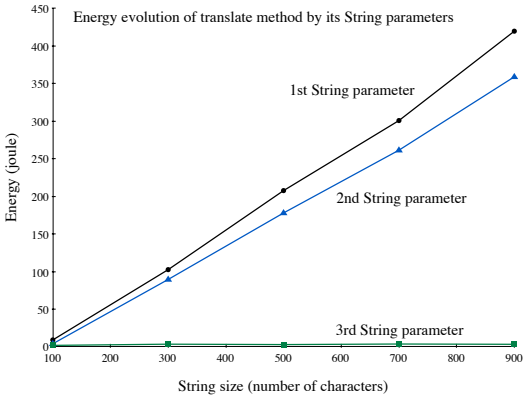
(c) Energy evolution of the copies method by its int parameter.



(d) Energy evolution of the center method by its string parameter.



(e) Energy evolution of the center method by its int parameter.



(f) Energy evolution of the translate method by its three strings parameter.

Figure 5.5: Energy evolution model of four methods from the Violin String Java library

it appends the string *nCopies* times in a for-loop using Java's `append` method. The `append` method calls the `String.getChars` method which in turn calls `System.arraycopy`. Finally, `System.arraycopy` method does the copy. The code of these methods, in particular `copies` method, explains the energy consumption while evolving the size of the string to copy. In particular, a bigger string requires more energy to append it to the `StringBuffer` object (thus a bigger loop over the characters array to copy). And a higher number of copies means the same repetitive task is executed multiple times, therefore the energy evolves linearly.

Center

There are two methods named `center`: one with two parameters (*e.g.*, a string and an integer), and one with three (*e.g.*, a string, an integer and a character). Both `center` methods center a string given in parameter within a new string of a given length (the latter is the integer parameter). The first `center` method adds blank spaces around the method, while the other takes a character in parameter and uses it for padding. Both `center` methods are run 300,000 times.

The implementation code of both methods is as follow:

```
1 public static String center(String s, int len) {
2         return center(s, len, ' ');
3 }
4
5 public static String center(String s, int len, char pad) {
6    int start;
7    int ls = s.length();
8    if (len < 1) {
9       return "";
10   }
11   char[] buf = new char[len];
12   for (int i = 0; i < len; i++) {
13      buf[i] = pad;
14   }
15   if (len > ls) {
16      start = (len-ls)/2;
17      s.getChars(0, ls, buf, start);
18   }
19   else {
20      start = (ls-len)/2;
21      s.getChars(start, start+len, buf, 0);
22   }
23   return new String(buf);
24 }
```

Results in Figures 5.5d and 5.5e show different patterns in energy evolution based on input parameters. However, the evolution is similar for both `center` methods when varying the string and the integer parameters (whether blank or random characters are used for

padding). The higher energy consumption for the `center` method with blank paddings is due to an additional method call. This is because the `center` method with two parameters is a wrapper to its overloaded method with an additional character parameter. The space character (for blank padding) is used as a parameter in the method code.

The energy models when varying the string size of the center method and when varying the integer are linear. This is explained by the use of the integer parameter as an end value in a for-loop. The evolution in this case is linear because the code executed in the loop is adding the padding character in an array cell. On the other hand, `String.getChars` method is called upon the string parameter. And as seen with `copies` method (see Section 5.4.2), `String.getChars` calls `System.arraycopy` that copies the array of characters, therefore the linear evolution.

5.4.3 Three-parameters variation: Translate

There is four `translate` methods in the Violin Strings library. The methods *convert all of the string's characters which are contained in the input set of characters to the corresponding character in the output set of characters*. The four variations of the method are due to three combinations with two additional parameters: a pad character and a boolean for ignoring the case of characters. The benchmark runs the method 1,000 times.

The signature of all four translate methods is as follow:

```
1public static String translate(String s, String setin, String setout, char pad,
2        boolean ignoreCase);
3public static String translate(String s, String setin, String setout, char pad);
4public static String translate(String s, String setin, String setout,
5        boolean ignoreCase);
6public static String translate(String s, String setin, String setout);
```

The implementation code of the translate method is as follow:

```
1public static String translate(String s, String setin, String setout, char pad,
2        boolean ignoreCase) {
3   int ls   = s.length();
4   int lout = setout.length();
5   char[] buf = new char[ls];
6   s.getChars(0, ls, buf, 0);
7   int pos = 0;
8   int n;
9   while ( (pos = indexOfAnyOf(s, setin, pos, ignoreCase)) >= 0 ) {
10      n = indexOf(setin, buf[pos], ignoreCase);
11      if (n >= lout) {
12         buf[pos] = pad;
13      }
14      else {
15         buf[pos] = setout.charAt(n);
16      }
```

```
17          pos++;
18      }
19      return new String(buf);
20  }
```

The results in Figure 5.5f show the energy evolution when varying the first three parameters of the method. We notice that the model is linear when varying the length of the input string (first parameter), and the length of the input set of characters (second parameter, *setin*). The model is constant when varying the length of the output set of characters (third parameter, *setout*).

In particular, the third parameter, *setout*, is used only twice in the method: once to get its length, and another time to get a character at a given position in an if/else condition. Both usages are relatively simple to execute, therefore consuming little energy, thus explaining the low impact of varying this parameter and the *flat* evolution of the energy consumption. On the other hand, varying the string to convert (first parameter) or the input set of characters (second parameter) has a linear impact on the energy consumption. `ViolinString.indexOfAnyOf` method is called upon the first and second parameters, and `String.getChars` method is called upon the first one too in the implementation of the `translate` method. `indexOfAnyOf` implementation also calls `String.getChars` on the *setin* parameter. The latter method uses `System.arraycopy` in order to copy an array of objects (*e.g.*, here an array of characters), and is responsible for linear energy evolution as we reported in Section 5.4.2 with the `copies` method.

5.5 Discussions

Our work on modeling the energy consumption evolution of software code based on input parameters, allows us to have an additional layer of understanding of the energy consumption and distribution in software. But also, it provides us with methodologies and tools to acknowledge the impact of input parameters on energy consumption. Therefore, providing energy efficient software cannot be restrained to coding more efficient code for a certain set of fixed parameters. Our results show higher complexity in the distribution of energy in software code, the importance of taking into account the implementation of Java's JVM, and the side effects that may happen. The learning we got from our work is summarized in the next paragraphs.

5.5.1 Model energy evolution through empirical benchmarking

The first conclusion of our work is that we can model the energy consumption evolution of software code through empirical benchmarking. In our approach, we show the validity of empirical benchmarking when studying the evolution of the RSA asymmetric encryption/decryption algorithm (see Section 5.2.1). The energy consumption evolution is

exponential and is on par with the exponential complexity of the algorithm. In addition, monitoring the evolution at software code allows detecting the methods responsible for the exponential evolution, and the implementation source code of these methods validates our approach.

We take this empirical benchmarking a step forward by automating its execution through our approach and our framework, JALEN UNIT. This automation yields challenges into providing representative models, mainly because we model energy evolution based on input parameters. Therefore, we propose different strategies for varying input parameters, and keep the choice of the strategy to the execution context. This decision allows more representative benchmarks and thus more usable energy evolution models.

5.5.2 Side effects are not negligible

The second conclusion we learn from our experiments is that side effects are not negligible. This is particularly present in software requiring a virtual machine, such as Java software. The characteristics of the Java virtual machine impacts the energy consumption evolution as we see in the `Joiner.join` experiments in Section 5.2.2. When varying the number of strings in the `Joiner.join` method, energy evolution alternates phases of constant energy consumption with direct increase phases. This is explained by the characteristics of the JVM itself, which allocates memory for appending strings in a fixed size buffer. When the number of strings is higher, the JVM allocates a bigger buffer, therefore requiring additional energy consumption on phases (see Figure 5.3).

5.5.3 Impact of Java's JVM and system calls

In addition to side effects due to the JVM's characteristics, our experiments show the impact of the JVM's core methods and implementation, and those of system calls. In particular, our results on the Violin Strings library (see Section 5.4) show how the evolution of string manipulation methods is impacted by the JVM's methods it calls. The library's `copies` method uses Java's `append` method, which in turn calls `String.getChars`. The latter finally uses `System.arraycopy` to perform characters' copies. Therefore, a bigger string to copy requires more loops over the string characters in `System.arraycopy`, leading to linear energy evolution based on the string size.

By analyzing the source code of the benchmarked methods, we see similar explanation of the energy consumption evolution. `Translate` method is another example where both the method's own implementation and JVM's methods have an impact on the energy evolution modeling. One of its parameters, the string *setout*, has little impact on the energy evolution. This is because it is used in a context where the evolution of its size has negligible impact on the performance and complexity, thus energy consumption, of the method. However, two other parameters, strings *s* and *setin*, have linear impact because the execution of the method ultimately calls `System.arraycopy` on these parameters.

Our results show the importance of the underlying Java JVM implementation and functioning in order to better understand the energy consumption, distribution and evolution in software. Providing energy efficient software is therefore dependent on this knowledge and on the lessons we learn from our approach and experimentations.

5.5.4 Limitations

JALEN UNIT framework allows modeling energy consumption evolution of software code. Our results are promising into understanding energy interaction in software, however some limitations are to be noted.

First, our framework benchmarks methods individually; therefore interactions between methods are not studied here. The impact of varying the parameters on other methods is an interesting topic we will be addressing as future work.

In addition, our model is inferred through empirical benchmarking but its mathematical analysis and notation is still manual. Automatic analysis of the empirical data and the inferring of the mathematical $\mathcal{O}$ notation, and specific evolution formulas are the next direction in our work. Ideally, a mathematical formula would provide the energy consumption of a method based on the values of its parameters. In addition, using analysis technics, such as Principal Component Analysis (PCA) or the least squares method, could help in correlating the impact of the variation of multiple parameters on the energy consumption of the method.

Finally, our framework infers energy evolution models based on CPU energy. However, more hardware components are involved in the execution of software, in particular the disk, memory and network. We are currently extending our framework into expanding the evolution model into these components. In Chapter 3, we showed that network energy is negligible compared to the CPU energy. However, disk and memory energy may account for a considerable part of energy consumption in desktop machines. Therefore, it is important to study the effects of these hardware components on the energy evolution model of methods while varying their parameters.

5.6 Summary

In this chapter, we propose an approach in order to automatically infer the energy models of software methods based on their input parameters. Our approach is implemented for Java applications in a framework called JALEN UNIT. The latter cycles through methods in an application or software library, and generates benchmarks for each method and each of its parameters. The benchmarks are then run following evolution strategies for parameters, and their energy consumption is measured using JALEN (see Chapter 4).

Our approach is motivated by the variation of the energy consumption of software when varying their input parameters (see Section 5.3.1). Our approach allows developers to understand which methods are responsible for the general energy evolution, and how each of

these methods evolves when varying their input parameters (see Section 5.2.1). In addition, we report on how varying different parameters affects the energy consumption of a method (see Section 5.2.2). Finally, we study the Violin Strings Java library and automatically infer the energy model of four of its methods, then explain the generated different energy evolution models by analyzing their source code (see Section 5.4).

Part III

Conclusion and Perspectives

Chapter 6

Conclusion and Perspectives

"We believe in the systematic understanding of the physical world through observation and experimentation, through argument and debate."-Alteran Woman, Stargate: The Ark of Truth

Contents

In this chapter, we summarize our thesis dissertation by discussing the challenges and goals addressed, and then we outline our contributions. Next, we discuss our short-term and long-term perspectives into energy measurement and management of software, middleware, and hardware devices. Finally, we present the list of our publications during this thesis.

6.1 Summary of the Dissertation

Modern software and computer configurations are increasingly distributed, use a variety of devices, and require the usage of software services, either cloud-based or local. These software and devices are constantly powered up and connected, from mobiles devices (*e.g.*,

smartphones always-on), to servers in data centers (*i.e.*, required to have high availability), and desktop computers (*e.g.*, always powered on during work hours or leisure time). Users own multiple devices, all connected to each other, such as a mobile smartphone and a laptop computer, and use Internet services or cloud-based storage and software. In these new usages, managing energy for this variety of devices and software requires new adaptation strategies. The middleware layer is an optimal candidate to manage this variability in term of software implementation and hardware characteristics.

However, managing energy cannot be efficiently done if energy itself is not measured and considered as a first class requirement. Existing energy management approaches use resources utilization information in order to provide coarse-grained, domain-limited approaches that only work under a set of conditions and configurations. A **first step** into offering energy efficient software and devices, and into managing energy consumption of these services, is to accurately measure energy. A **second step** is to understand how energy is being consumed, where it is being spent, and thhe energy distribution between software, and between software code.

In this thesis, we addressed these challenges and provided energy models, energy measurement and modeling approaches and tools. We also studied software energy and provide better knowledge of energy consumption and distribution in software based on our experimentations.

We provide energy models in order to accurately estimate the energy consumption of software (*e.g.*, applications, processes), and of software code (*e.g.*, classes and methods). These models have a low margin of error, therefore answering the criteria of *accurate measurements* we identified in our state-of-the art study for energy measurements (see Chapter 2). Our models are also fine-grained and can be tailored into providing the energy consumption of software as black box, or at the granularity of portions of code or methods. This *fine granularity* is also another criteria identified during our study, and helps energy management approaches offer more in-depth solutions for energy optimization. It also contributes to our second main step of understanding energy in software.

We also develop energy measurement and modeling approaches and tools for software. In particular, we propose a software-only approach for estimating the energy consumption of applications, *i.e.*, JALEN, therefore answering the *software-centric approaches* criteria in our study. JALEN is a software capable of measuring the energy consumption of portions of code in applications, such as classes and methods. A software framework, JALEN UNIT, is based on the measurement tool JALEN, and proposes to model the energy evolution of software methods based on their input parameters. This modeling is done through the generation and execution of benchmarks for each of the method's parameters. The implementation of both of our tools have low impact on user experience, therefore answering the *reduce user experience impact* criteria identified in the state-of-the art study.

Finally, our energy models and software measurement and modeling approaches and tools, are used to validate or recuse common belief and knowledge on software energy. In

particular, we validate that network energy cost is negligible compared to the CPU cost on a desktop computer, and that our models and implementation tools offer accurate results in comparison to hardware power meters. How software is written and the parameters of compilation impact energy consumption, such as JIT compilation of the Java virtual machine, or the optimization flags of GCC and G++ compilers. We validate that native compiled code is always more energy efficient than interpreted code and virtual machine based code, and the implementation algorithm of an application have an impact on energy consumption. This impact is even more important and complex when several factors impact energy consumption, such as choosing a programming language, compiler options, implementation algorithm, the usage of a virtual machine, and the characteristics and internal functioning of the said virtual machine. We addressed these factors and provided experimentations and analysis in order to help developers understand the energy consumption of their software, and therefore write more energy efficient software.

6.2 Contributions

The contributions of this thesis are summarized as follows:

6.2.1 Energy Models for Software

The energy models we proposed allow measuring the energy consumption of software at different granularity. We provide energy models for two levels: application level, and code level. The former provides the energy consumption of software as system processes, *i.e.*, as black boxes; while the latter measures the energy consumption of software code, *i.e.*, classes and methods. Our energy models are software-only, therefore no hardware meter is needed to provide energy measurements. The latters are also accurate with a margin of error varying between 0.5% up to 3% for complex software in comparison with a hardware power meter. Our models use hardware and software resources utilization in order to provide an estimation of the energy consumption of software and software code. In addition, we propose an approach to infer the energy evolution model of software code based on its input parameters. Our approach automatically generates energy benchmarks for software methods and models their energy evolution through empirical benchmarking.

Our contribution in energy models for software answers the challenges we identified in our study of the state-of-the art by providing:

- *Accurate measurements*: the margin of error of our energy models is low at around 0.5% and up to 3% for estimating the energy consumption at application level. Measurements at code level add in a 1.3% margin of error. Therefore, our models offer accurate estimations of the energy consumption in comparison to direct energy measurements using hardware power meters.

- *Fine-grained energy model*: our models estimate the energy consumption of software, and also software code. We propose fine-grained energy models that can estimate the energy consumption of blocks of code, such as software methods.

- *Software-centric approach*: the models we propose do not require any hardware investment. They provide accurate and fine-grained estimations using only software means such as data collected from software and operating systems.

6.2.2 Energy Measurement Approaches and Tools

We developed energy measurement approaches and tools based on our proposed energy models. We build two main tools: JALEN and JALEN UNIT, for measuring the energy consumption of software code, and for inferring the energy evolution model of software code based on empirical benchmarking, respectively.

JALEN is a software energy profiler that hooks into the Java Virtual Machine during its start and monitors resources utilization of Java applications at code level, *i.e.*, it monitors resources utilization of methods and classes. It then applies our energy models based on the collected data in order to estimate the energy consumption of software code. The statistical sampling version of JALEN is also light on the running application's resources with an execution time overhead of around 3%.

JALEN UNIT is an energy framework for modeling the energy evolution of software code based on its input parameters. It cycles through all classes and methods in a Java application or library, then generates energy evolution benchmarks for each method and each of its parameters. The variation evolution of the parameters is done through software injectors and based on different evolution strategies. Finally, the benchmarks are run and their energy consumption is measured and an energy evolution model is inferred.

Our contribution in energy measurement tools answers the following limitations (in addition to those answered by our energy models, see Section 6.2.1):

- *Software-centric approach*: both of our energy measurement tools, JALEN and JALEN UNIT, do not require any hardware power meter or other hardware investment. The tools are software programs that collect data of hardware and software resources utilizations through software means (*e.g.*, OS or JVM APIs, reading the application's files, etc.).

- *Reduce user experience impact*: our tools have a low overhead at around 3%. In particular, the statistical sampling version of JALEN does not modify the monitored software and does not interfere with its execution. The impact on user experience is minimal, and energy is measured transparently throughout the execution of the application.

6.2.3 Lessons on Software Energy Consumption

Our work is not only limited into providing energy models and measurement approaches and tools, then validating them using empirical experimentations. Our contribution is also a series of experimentations aimed to validate or recuse common belief on energy consumption of software, and to better understand this energy consumption and the energy distribution in software code.

The main lessons we learned are the following:

- Energy models can estimate the energy consumption of software, even at code level, with an accuracy similar to hardware meters. This also comes with an advantage of software-only approach that can be used easily in various configurations and deployed at large scale (see Chapters 4 and 3).
- Network Ethernet energy consumption is negligible compared to the CPU on desktop computers. Our experiments back up state-of-the art results and we found that the CPU consumes much more energy then the Ethernet network card in a network-intensive benchmark (see Chapter 3).
- Energy consumption is not linear with CPU utilization, mainly due to the energy optimizations at software and hardware layers. DVFS allows the CPU to vary its frequency and voltage in order to reduce its energy consumption, and modern CPUs are multi-cores. Therefore CPU utilization or CPU time cannot replace energy estimations or measurements (see Chapter 3).
- When monitoring energy consumption, it is always better to use non-disturbing measurement tools in order to reduce the impact of the monitored software. In this case, statistical sampling is better suited than modifying the application (for example through byte code instrumentation), and it also provides accurate results (see Chapter 4).
- Energy consumption is strongly related to the hardware, therefore raw energy values (*e.g.*, in joules or watt) are not relevant for comparing efficient software. It is preferable to use energy percentages between software and between software methods (see Chapters 4 and 3).
- The implementation algorithm, the programming language and the program parameters are factors that impact the energy consumption of software. A same software system has different energy consumption if it is implemented in a different programming language, or using a different algorithm. Compilers and compilation options also have an impact on energy consumption, and native code is always more energy efficient than interpreted code or virtual machine based code. Finally, varying the parameters values impacts the energy consumption with different evolution patterns based on which parameters are varied (see Chapters 3, 4 and 5).

- Side effects and the impact of the operating system and virtual machines are not to be neglected. They have a high impact in energy consumption and the good understanding of how system or JVM methods are implemented, allows writing better energy efficient software. The JVM, for example, uses JIT compilation in order to optimize the execution of certain portions of code (such as recursive and repetitive loops), which ultimately leads to less energy impact of the virtualization from the JVM. (see Chapters 3 and 5)

6.3 Perspectives

6.3.1 Short Term Perspectives

Our contributions to energy measurement, estimation and modeling open doors for additional contributions. In particular, more challenges still need to be tackled in order to provide even more insights on energy consumption in software.

Mathematical Analysis and Notation

A first short-term contribution we are looking at is providing mathematical analysis of the energy evolution model of software code. Concretely, we plan to propose approaches and tools for inferring the mathematical formula defining the energy evolution model. This is to be done in two steps:

- **Coarse-grained mathematical models**. In this step, we propose mathematical models similar to the complexity models (*e.g.*, the big $\mathcal{O}$ notation). Based on the energy evolution model generated by our JALEN UNIT framework, we estimate the evolution trend of the energy consumption and provide a mathematical notation of the said evolution.
- **Specific mathematical formulas**. Concretely, we plan to provide a detailed mathematical formula in order to calculate the precise energy consumption of software code based on input parameters. For example, we may have a linear formula for a method `methF` with a int parameter *param* resembling to the following: $Energy_{methF}^{param} = 23 \times param + 45$. Such specific formula would provide developers with a simple methodology to estimate the energy consumption of their methods executed using parameters not necessarily benchmarked, *e.g.*, `methF` being benchmarked for values ranging from 10 to 10,000, and the formula allows to estimate the energy consumption for any given parameter value.
  This step is trickier for several reasons:
  - It requires extensive benchmarking and a set of sufficient results for inferring the mathematical formula. This requires more benchmark time and results based on relevant benchmarking data.

– Energy consumption depends on hardware and the heterogeneous nature of modern devices and software add an additional layer of difficulty into proposing a specific formula for software code modeling. In particular, raw energy values (in watt) change greatly between devices even though percentages distribution in the same application stays the same (see Chapter 4). On the other hand, using percentages make sense when comparing the energy consumption of software code. It is not relevant for individual methods' energy consumption evolution.

This calculation for a specific mathematical formula holds challenges, not just mathematical modeling ones, but others inherent to the nature of energy consumption (*e.g.*, multiple devices, raw energy vs. percentages, etc.). Principal Component Analysis (PCA) is a first step into inferring specific mathematical formulas.

Intelligent Benchmarking

Another short-term perspective is more intelligent and efficient benchmarking. In particular, current-benchmarking strategies for energy evolution modeling involves testing values from a start point to an end point (with different variation strategies).

A more efficient approach would be benchmarking on the *edges* or *limits* of the method's parameters. For example, the edges and limits of a method manipulating an X-Y graph with boundaries can be the X and Y coordinates at the boundaries. This testing at edges can be complemented by benchmarking unit tests that are already coded by developers. Typically, unit testing checks the correct execution of methods at their boundaries and in context situations where errors, abnormal execution or specific values are expected or may occur. Benchmarking the energy consumption under these circumstances provides us with extensive data and knowledge that we may not have otherwise. In addition, random testing can be used for values in between the limits and boundaries of methods. The main advantage of this type of limited benchmarking is to accelerate the generation of energy evolution models.

Develop Energy Models for Mobile and Distributed Devices, and Virtual Machines

Our work in this thesis concentrates on desktop computers, servers, and laptops. In particular, our experimentations are done on desktop, laptop and workstation machines. However, the computing landscape is changing into being more mobile, more virtual and more distributed.

We plan to investigate the validity and accuracy of our energy models in these configurations. Then, based on the results of our investigation, we will adapt, modify or develop new energy models for these environments. Energy in mobile environments has been thoroughly studied in the last years and decades. However this environment is rapidly evolving, whether in term of hardware and software, or in term of usages. Therefore, this requires a better understanding on the hardware used in their configurations, then applying the same

approach we developed throughout this thesis in order to model energy consumption. In addition, distributed environments present the challenge of running an application on multiple and heterogeneous devices. This makes energy modeling more complex due to the additional variability of different devices. On the other hand, virtual machines add an additional layer of complexity due to the emulation of the real hardware by the virtual machine application. Some hardware features are not always supported in virtual machine guests (such as DVFS for CPUs), and the virtual machine infrastructure consumes non-negligible amounts of energy. We plan to tackle these challenges for mobile and distributed devices, and for virtual machines as short-term perspectives.

6.3.2 Long Term Perspective: Autonomous Energy Management

At long-term, we plan to use the energy models, approaches, and tools we developed throughout this thesis to manage the energy consumption of software autonomically. The lessons we learned from our work and experimentations help us shape an argument on autonomic management of energy. The inherent complexity of the energy dimension, the diversity of hardware, programming languages, and software, and the side effects and interactions in energy evolution modeling, push us into arguing to *put the human outside the loop*. Concretely, managing energy consumption, optimization and reduction, without human interaction.

Although many energy measurement and management approaches argument on the necessity of *keeping the human in the loop*, that is providing visual energy information or a form of energy feedback to users, we argue that users actually does not need, or sometime want, this type of involvement. Energy is a non-functional requirement in software and devices, and users actually use the latter for their functionality, *e.g.*, to do actual work or entertainment. Experiments and surveys on end users show that they need high-level abstractions for managing energy consumption [HSJ09]. Users also tend to stop using energy management or measurement tools after an initial *excitement* of using a novel tool, or what is called a *fad effect*. In particular, a number of energy meters were installed in several homes and researchers observed the reaction of users over a period of time [KG09]. As stated in the study, *some families reported that the impact of the meter changed over time as once interaction decreased it consequently lost its importance which made old habits reappear*. And that *the use of the meter stopped within the majority of the homes*. In a preliminary survey conducted on computer science researchers and Ph.D. students, 50% of users prefer that software reduce the energy consumption silently, without interrupting users' activities. Only 23% want the adaptation software to ask for confirmation of any adaptation or parameters modifications. We suspect the numbers for silent energy management to be higher for the general public other then computer-centric scientists. Therefore, our perspective is in proposing a software platform to manage software and hardware energy consumption autonomously and without interaction from users.

An energy-aware autonomous approach should therefore imitate the human body metabolism: the platform needs to be transparent to the user and to devices and applications, but without limiting users' high-level, non-direct energy decisions. In the human body, when energy becomes low, the system starts by using its reserves and notifying the human about the situation (*e.g.*, the human feels hunger). Therefore, the human could apply high-level decisions, such as eating (to recharge his energy and reserves), or reduce his activity, or go to sleep (low power mode). We therefore believe that energy managament approaches should take inspiration from biologic systems and provide a similar autonomous functioning for energy-awareness because the complexity of systems is rapidly increasing. This inspiration could also take the shape of studying the execution context of software and autonomously learn from it through machine learning technics. Monitoring energy consumption and detecting where and how the energy is spend, efficiently distributing energy between software and hardware, or adapting software code and components and modifying hardware parameters based on energy levels, all are to be done autonomously without human involvement. Users would only take high-level business-related decisions, while the essential of energy monitoring and management is achieved autonomously and invisible to users' sight.

6.4 Publications

Our work in this thesis has been published in recognized journals, conferences, and workshops. Here is the list of our most important publications:

Journals

- **Adel Noureddine**, Romain Rouvoy, and Lionel Seinturier. *Monitoring Energy Hotspots in Software*. In Automated Software Engineering journal. *Accepted with major revision on February 2014*.
- **Adel Noureddine**, Romain Rouvoy, and Lionel Seinturier. *A Review of Energy Measurement Approaches*. In ACM SIGOPS Operating Systems Review journal (OSR). Volume 47, Issue 3, pages 42-49, December 2013.
- **Adel Noureddine**, Romain Rouvoy, and Lionel Seinturier. *A Review of Middleware Approaches for Energy Management in Distributed Environments*. In Software: Practice and Experience journal (SPE). Volume 43, Issue 9, pages 1071-1100, September 2013. Impact factor: 1.008.

Conferences and Workshops

- **Adel Noureddine**, Romain Rouvoy, and Lionel Seinturier. *Unit Testing of Energy Consumption of Software Libraries*. In Software Engineering Aspects of Green Comput-

ing track of the 29th Annual ACM Symposium on Applied Computing (SAC'14). Gyeongju, South Korea, March 2014. Acceptance rate: 24%.

- **Adel Noureddine**, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. *Runtime Monitoring of Software Energy Hotspots*. In the 27th International Conference on Automated Software Engineering (ASE'12). Pages 160-169. Essen, Germany, September 2012. Acceptance rate: 15%.
- **Adel Noureddine**, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. *A Preliminary Study of the Impact of Software Engineering on GreenIT*. In the First International Workshop on Green and Sustainable Software (GREENS'12/ICSE'12). Zurich, Switzerland, June 2012. Acceptance rate: 41%.
- Aurelien Bourdon, **Adel Noureddine**, Romain Rouvoy, and Lionel Seinturier. *Linux: Understanding Process-Level Power Consumption*. Invited speaker at the 2nd International Workshop on Green Computing Middleware (GCM'11/Middleware'11). Lisbon, Portugal, December 2011.
- **Adel Noureddine**, Romain Rouvoy, and Lionel Seinturier. *Supporting energy-driven adaptations in distributed environments*. In the First Workshop on Middleware and Architectures for Autonomic and Sustainable Computing (MAASC'11/NOTERE'11). Paris, France, May 2011.

Presentations and Talks

- **Adel Noureddine**, Romain Rouvoy, and Lionel Seinturier. *Mesure et modélisation de l'énergie logicielle*. In 4th Green Days @ Lille. Lille, France, November 2013.
- **Adel Noureddine**, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. *Runtime Monitoring of Software Energy Hotspots*. In Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS'13). Grenoble, France, January 2013.
- **Adel Noureddine**. *"Why Humans Can't Green Computers", An Autonomous Green Approach for Distributed Environments*. Talk at the 10th Belgian-Netherlands software evolution seminar (BENEVOL'11). Brussels, Belgium, December 2011.

Other Publications

- Aurelien Bourdon, **Adel Noureddine**, Romain Rouvoy, and Lionel Seinturier. *PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level*. In ERCIM News magazine, No. 92. January 2013.
- **Adel Noureddine**, Aurelien Bourdon, Romain Rouvoy, and Lionel Seinturier. *e-Surgeon: Diagnosing Energy Leaks of Application Servers*. INRIA Research Report. January 2012.

Bibliography

[Ale] AlertMe. http://www.alertme.com/smart_energy. 28, 36

[ant] ANTS Performance profiler. http://www.red-gate.com/products/dotnet-development/ants-performance-profiler/. 36

[aqt] AQtime Pro. http://smartbear.com/products/development-tools/performance-profiling/. 36, 37

[ASG10] Yuvraj Agarwal, Stefan Savage, and Rajesh Gupta. Sleepserver: a software-only approach for reducing the energy consumption of pcs within enterprise environments. In *Proceedings of the 2010 conference on USENIX annual technical conference*, pages 22–22, Berkeley, CA, USA, 2010. USENIX Association. 12, 16, 25, 26

[asm] Asm. http://asm.ow2.org/. 73

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press. 85

[BNRS13] Aurelien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Powerapi: A software library to monitor the energy consumed at the process-level. *ERCIM News*, 2013, 2013. 50

[BS09] Walter Binder and Niranjan Suri. Green Computing: Energy Consumption Optimized Service Hosting. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science*, pages 117–128, Berlin, Heidelberg, 2009. Springer-Verlag. 12, 15, 20, 21, 22, 25, 26

[BSGR03] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. Base - a micro-broker-based middleware for pervasive computing. In *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications*, pages 443–451, march 2003. 15

[CH10] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association. 14, 36

[cpr] C Profiler. http://www.semdesigns.com/Products/ Profilers/CProfiler.html. 36, 37

[dac] List of benchmarks of dacapo benchmark suite. http://dacapobench.org/benchmarks.html. 83, 88

[DPHu+08] Isabelle Demeure, Guilhem Paroux, Javier Hernando-ureta, Amir R Khakpour, and Julien Nowalczyk. An energy-aware middleware for collaboration on small scale manets. In *Proceedings of the Autonomous and Spontaneous Networks Symposium*, Paris, France, November 2008. 12, 25

[DRS09] Thanh Do, Suhib Rawshdeh, and Weisong Shi. pTop: A Process-level Power Profiling Tool. In *Proceedings of the 2nd Workshop on Power Aware Computing and Systems*, Big Sky, MT, USA, october 2009. 33, 36, 37

[EG01] W. Keith Edwards and Rebecca E. Grinter. At home with ubiquitous computing: Seven challenges. In *Proceedings of the 3rd international conference on Ubiquitous Computing*, UbiComp '01, pages 256–272, London, UK, UK, 2001. Springer-Verlag. 26

[ej-] ej-techonologies. JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html. 36, 37

[ene] Intel Energy Checker SDK. http://software.intel.com/en-us/articles/intel-energy-checker-sdk. 35, 36

[FN01] Laura Marie Feeney and Martin Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *In IEEE Infocom*, pages 1548–1557, 2001. 29

[fre] Openmoko Neo Freerunner. http://wiki.openmoko.org/wiki/Neo_FreeRunner. 36

[FS99] Jason Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*, page 2, Washington, DC, USA, 1999. IEEE Computer Society. 32, 36, 37

[Gar07] Gartner. Green IT: The New Industry Shockwave. In *Gartner*, Presentation at Symposium/ITXPO Conference, 2007. 3

[gcc] Gnu compiler collection manual. http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. 59

[gcu] The Green Challenge for USI 2010. http://sites.google.com/a/octo.com/green-challenge. 68

[Git] Vivek Gite. How do I Find Out Linux CPU Utilization? http://www.cyberciti.biz/tips/how-do-i-find-out-linux-cpu-utilization.html. 35

[Goo] Google Powermeter. http://www.google.com/powermeter. 28

[gpr] GNU gprof. http://www.gnu.org/software/binutils/. 36, 37

[gua] Google guava library. https://code.google.com/p/guava-libraries/. 78, 98

[Hin12] Abram Hindle. Green mining: investigating power consumption across versions. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE 2012, pages 1301–1304, Piscataway, NJ, USA, 2012. IEEE Press. 91

[hpr] Hprof: A heap/cpu profiling tool. http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html. 80

[HSJ09] Seth Holloway, Drew Stovall, and Christine Julien. What users want from smart environments. Technical report, The University of Texas at Austin, 2009. 122

[Ing] Giorgio Ingargiola. Data transmission. http://www.cis.temple.edu/ giorgio/-cis307/readings/datatrans.html. 72

[ipe] Iperf command. http://linux.die.net/man/1/iperf/. 54

[ipm] IPMItool. http://ipmitool.sourceforge.net/. 32

[jal] Jalen. https://github.com/adelnoureddine/jalen. 72

[jet] Jetty web server. http://www.eclipse.org/jetty/. 52

[jip] Jip - the java interactive profiler. http://jiprof.sourceforge.net/. 36, 37, 82

[jme] Apache jmeter. https://jmeter.apache.org/. 52

[jol] Jolinar. https://github.com/adelnoureddine/jolinar. 51

[jou] Joulemeter. http://research.microsoft.com/en-us/projects/joulemeter/. 35, 36

[Kan09] K. Kant. Toward a science of power management. *Computer*, 42(9):99 –101, september 2009. 28

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003. 22, 23, 27

[KG09] C. Kelsey and V. Gonzalez. Understanding the use and adoption of home energy meters. In *Proceedings of El Congreso Latinoamericano de la Interaccion Humano-Computadora*, pages 64–71, 2009. 39, 44, 122

[Kra07] Sacha Krakowiak. *Middleware Architecture with Patterns and Frameworks*. Sacha Krakowiak, http://sardes.inrialpes.fr/ krakowia/MW-Book/, 2007. 11

[Kul07] Eugene Kuleshov. Using the ASM framework to implement common java bytecode transformation patterns. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, Vancouver, Canada, mar 2007. 73

[KZ08] Aman Kansal and Feng Zhao. Fine-grained energy profiling for power-aware application design. In *Proceedings of the 1st Workshop on Hot Topics in Measurement and Modeling of Computer Systems at ACM Sigmetrics*, pages 26–31, june 2008. 28, 29, 36

[LGT08] Adam Lewis, Soumik Ghosh, and N.-F. Tzeng. Run-time energy consumption estimation based on workload in server systems. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower'08, pages 4–4, Berkeley, CA, USA, 2008. USENIX Association. 31, 36

[lina] Linux stress command. http://linux.die.net/man/1/stress/. 52

[Linb] Linux User's Manual. iostat. http://linux.die.net/man/1/iostat. 35

[Linc] Linux User's Manual. top. http://linux.die.net/man/1/top. 33

[LQBC11] Yung-Hsiang Lu, Qinru Qiu, Ali R. Butt, and Kirk W. Cameron. End-to-end energy management. *Computer*, 44:75–77, 2011. 11, 28

[LV99] Sheng Liang and Deepa Viswanathan. Comprehensive profiling support in the javatm virtual machine. In *Proceedings of the 5th conference on USENIX Conference on Object-Oriented Technologies & Systems*, pages 17–17, Berkeley, CA, USA, 1999. USENIX Association. 37

[MDNV07] S. Mohapatra, N. Dutt, A. Nicolau, and N. Venkatasubramanian. Dynamo: A cross-layer framework for end-to-end qos and energy optimization in mobile handheld devices. *Selected Areas in Communications, IEEE Journal on*, 25(4):722–737, may 2007. 12, 14, 25

[mem] Memory usage of Java Strings and string-related objects. http://www.javamex.com/tutorials/memory/string_buffer_memory_usage.shtml. 100

[Mou01] Eric Mouw. Linux kernel procfs guide. http://www3.big.or.jp/ sian/linux/DocBook/procfs-guide/index.html, June 2001. 33

[mpl] Mplayer. http://www.mplayerhq.hu/. 52

[MSK07] Dustin McIntire, Thanos Stathopoulos, and William Kaiser. ETOP: sensor network application energy profiling on the LEAP2 platform. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 576–577, New York, NY, USA, 2007. ACM. 28, 36

[MV03] Shivajit Mohapatra and Nalini Venkatasubramanian. Parm: Power aware reconfigurable middleware. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 312, Washington, DC, USA, 2003. IEEE Computer Society. 12, 14, 25

[oca] The ocaml system, documentation and user's manual. http://caml.inria.fr/pub/docs/manual-ocaml/manual022.html. 59

[OKT] OKTECH-Info Kft. OKTECH Profiler. http://code.google.com/p/oktech-profiler. 36, 37

[Ora] Oracle. The java hotspot performance engine architecture. http://www.oracle.com/technetwork/java/whitepaper-135217.html. 58

[pen04] Enhanced intel speedstep technology for the intel pentium m processor. White Paper, march 2004. 47, 48

[Pet08] Luigia Petre. Energy-Aware Middleware. In *Proceedings of the 15th Annual International Conference and Workshop on the Engineering of Computer Based Systems*, pages 326–334. IEEE, 2008. 30, 36

[PlDR08] Guilhem Paroux, lsabelle Demeure, and Laurent Reynaud. A Power-aware Middleware for Mobile Ad-hoc Networks. In *Proceedings of the 8th International Conference on New Technologies in Distributed Systems*, pages 1–7. ACM, 2008. 12, 25

[PMND07] Guilhem Paroux, Ludovic Martin, Julien Nowalczyk, and Isabelle Demeure. Transhumance: A power sensitive middleware for data sharing on mobile ad hoc networks. In *Proceedings of the 7th international Workshop on Applications and Services in Wireless Networks*, Spain, 2007. 12, 25

[powa] Powerapi. https://github.com/rouvoy/powerapi/. 50

[powb] Powerspy 2. http://www.alciom.com/en/products/powerspy2.html. 51

[powc] PowerTop. https://01.org/powertop/. 35, 36

[pre] Predictive commoning. http://gcc.gnu.org/wiki/PredictiveCommoning/. 60

[Pro] OKTECH Profiler. Sampling VS Instrumentation. http://code.google.com/p/oktech-profiler/wiki/SamplingVsInstrumentation. 37

[PSW00] Luigia Petre, Kaisa Sere, and Marina Waldén. A topological approach to distributed computing. *Electronic Notes in Theoretical Computer Science*, 28:59–80, 2000. WDS'99, Workshop on Distributed Systems (A satellite workshop to FCT'99). 30

[PSW06] Luigia Petre, Kaisa Sere, and Marina Waldén. A language for modeling network availability. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 639–659. Springer Berlin, Heidelberg, 2006. 30

[RGKP10] Joshua Reich, Michel Goraczko, Aman Kansal, and Jitendra Padhye. Sleepless in seattle no longer. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association. 12, 17, 21, 22, 23, 25, 26, 35

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. 97, 98

[RSRK07] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. Joulesort: a balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 365–376, New York, NY, USA, 2007. ACM. 48, 55, 63

[SB07] Gregor Schiele and Christian Becker. SANDMAN: an Energy-Efficient Middleware for Pervasive Computing. Technical report, University of Mannheim, Karlsruhe, Germany, october 2007. 12, 15, 25

[Sch] Michael Schmeling. Violin Strings. http://www.schmeling-consulting.de/violinstrings.html. 104

[SHB08] Gregor Schiele, Marcus Handte, and Christian Becker. Experiences in designing an energy-aware middleware for pervasive computing. In *Proceedings of the 6th Annual IEEE International Conference on Pervasive Computing and Communications*, pages 504–508, Washington, DC, USA, 2008. IEEE Computer Society. 12, 15, 25

[Sin] Amit Singh. Hanoimania! http://www.kernelthread.com/projects/hanoi/. 56

[SLH+12] Siddhartha Sen Sen, Jacob R. Lorch, Richard Hughes, Carlos Garcia Jurado Suarez, Brian Zill, Weverton Cordeiro, and Jitendra Padhye. Don't lose sleep over availability: The greenup decentralized wakeup service. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, 2012. 12, 17, 25, 26

[sli] SlimTune. http://code.google.com/p/slimtune/. 36, 37

[SMM07] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. An energy consumption framework for distributed java-based systems. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 421–424, New York, NY, USA, 2007. ACM. 28, 29, 36

[SMM08] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. Estimating the energy consumption in pervasive java-based systems. In *Proceedings of the 6th Annual IEEE International Conference on Pervasive Computing and Communications*, pages 243–247, Washington, DC, USA, 2008. IEEE Computer Society. 29, 36

[soc] Socketimpl java class. http://docs.oracle.com/javase/7/docs/api/ java/net/-SocketImpl.html. 74

[SYH+04] Daniel Grobe Sachs, Wanghong Yuan, Christopher J. Hughes, Albert Harris, Sarita V. Adve, Douglas L. Jones, Robin H. Kravets, and Klara Nahrstedt. Grace: A hierarchical adaptation framework for saving energy, 2004. 12, 25

[The] The Linux Kernel. cpufrequtils. http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufrequtils.html. 35

[tom] Apache tomcat. https://tomcat.apache.org/. 52

[TT13] Anne E. Trefethen and Jeyarajan Thiyagalingam. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science*, (0), 2013. 30, 36

[Ven99] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill Professional, 1st edition, 1999. 74

[Vis] VisualVM. http://visualvm.java.net. 36, 37

[VVHC+10] W. Vereecken, W. Van Heddeghem, D. Colle, M. Pickavet, and P. Demeester. Overall ict footprint and green communication technologies. In *Proceedings of the 4th International Symposium on Communications, Control and Signal Processing*, pages 1–6, 2010. 3

[VYI+09] Vibhore Vardhan, Wanghong Yuan, Albert F. Harris III, Sarita V. Adve, Robin Kravets, Klara Nahrstedt, Daniel Grobe Sachs, and Douglas L. Jones. Grace-2: integrating fine-grained application adaptation with global adaptation for

saving energy. *International Journal of Engineering Science*, 4(2):152–169, 2009. 12, 25

[wat] Watts Up Prp. http://www.wattsupmeters.com. 34

[Web08] Molly Webb. *SMART 2020: enabling the low carbon economy in the information age, a report by The Climate Group on behalf of the Global eSustainability Initiative (GeSI)*. GeSI, 2008. 3

[Wil] Greg Wilkins. Asynchronous rest with jetty-9. http://webtide.intalio.com/2013/04/async-rest-jetty-9/. 87

[XHSYJ11] Yu Xiao, Pan Hui, Petri Savolainen, and Antti Yla-Jääski. Cascap: cloud-assisted context-aware power management for mobile devices. In *Proceedings of the 2nd International Workshop on Mobile Cloud Computing and Services*, MCS '11, pages 13–18, New York, NY, USA, 2011. ACM. 12, 13, 22, 25

[XKYJ09] Yu Xiao, Ramya Sri Kalyanaraman, and Antti Yla-Jääski. Middleware for energy-awareness in mobile devices. In *Proceedings of the 4th International ICST Conference on Communication System software and middleware*, pages 1–6, New York, NY, USA, 2009. ACM. 12, 13, 20, 21, 22, 23, 25, 26

[XLWN03] Rong Xu, Zhiyuan Li, Cheng Wang, and Peifeng Ni. Impact of data compression on energy consumption of wireless-networked handheld devices. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 302–, Washington, DC, USA, 2003. IEEE Computer Society. 29

[yet] York Extensible Testing Infrastructure (. https://code.google.com/p/yeti-test/. 102

[Zad65] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338 – 353, 1965. 25

[ZEL05] Heng Zeng, Carla S. Ellis, and Alvin R. Lebeck. Experiences in managing energy with ecosystem. *IEEE Pervasive Computing*, 4:62–68, January 2005. 12, 15, 25

Appendices

Appendix A

Jalen Unit Injectors

We report here the code for the default JALEN UNIT injectors: integer, character, string and boolean.

The default integer injector:

```
1 package jalen.model;
2 import java.util.Iterator;
3
4 public class IntegerModel implements JalenModel, Iterator {
5         private final int start, end, increment;
6         private int current;
7
8         public IntegerModel(int start, int end, int increment) {
9                 this.start = start;
10                this.end = end;
11                this.increment = increment;
12                this.current = start;
13        }
14
15        @Override
16        public boolean hasNext() {
17                return this.current <= this.end;
18        }
19
20        @Override
21        public Object next() {
22                int result = this.current;
23                this.current += this.increment;
24                return result;
25        }
26
27        @Override
28        public void remove() {}
29
30        @Override
```

```
31          public Object getDefaultValue() {
32                  return this.start;
33          }
34 }
```

The default character injector:

```
1 package jalen.model;
2 import java.util.Iterator;
3 import java.util.Random;
4
5 public class CharModel implements JalenModel, Iterator {
6           private final int end, current;
7           private char theChar;
8
9           public CharModel(int end) {
10                  this.end = end;
11                  this.current = 0;
12                  this.theChar = this.generateChar();
13          }
14
15          public char generateChar() {
16                  Random r = new Random();
17                  String alphabet = "abcdefghijklmnopqrstuvwxyz";
18                  char c = alphabet.charAt(r.nextInt(alphabet.length()));
19                  this.current++;
20                  return c;
21          }
22
23          @Override
24          public boolean hasNext() {
25                  return his.current <= this.end;
26          }
27
28          @Override
29          public Object next() {
30                  return this.generateChar();
31          }
32
33          @Override
34          public void remove() {}
35
36          @Override
37          public Object getDefaultValue() {
38                  return this.generateChar();
39          }
40 }
```

The default string injector:

```
1 package jalen.model;
2 import java.util.Iterator;
```

```
3import java.util.Random;
4
5public class StringSizeModel implements JalenModel, Iterator {
6        private final int start, end, increment;
7        private int current;
8        private String theString;
9
10       public StringSizeModel(int start, int end, int increment) {
11               this.start = start;
12               this.end = end;
13               this.increment = increment;
14               this.current = start;
15               this.theString = this.generateString(this.start);
16       }
17
18       public String generateString(int size) {
19               if (size == 0)
20                       return "";
21               Random r = new Random();
22               String s = "", alphabet = "abcdefghijklmnopqrstuvwxyz";
23               for (int i=0; i<size; i++) {
24                       s += alphabet.charAt(r.nextInt(alphabet.length()));
25               }
26               return s;
27       }
28
29       @Override
30       public boolean hasNext() {
31               return this.theString.length() <= this.end;
32       }
33
34       @Override
35       public Object next() {
36               String result = this.theString;
37               this.theString += this.generateString(this.increment);
38               return result;
39       }
40
41       @Override
42       public void remove() {}
43
44       @Override
45       public Object getDefaultValue() {
46               return this.generateString(this.start);
47       }
48}
```

The default boolean injector:

```
1package jalen.model;
2import java.util.Iterator;
3
```

```
4public class BooleanModel implements JalenModel, Iterator {
5        private boolean theBoolean;
6        private boolean hasnext;
7        private int counterBoolean;
8
9        public BooleanModel() {
10               this.theBoolean = true;
11               this.hasnext = true;
12               int counterBoolean = 0;
13       }
14
15       @Override
16       public boolean hasNext() {
17               return this.counterBoolean <= 1;
18       }
19
20       @Override
21       public Object next() {
22               boolean result;
23               if (this.counterBoolean == 0)
24                       result = true;
25               else
26                       result = false;
27               this.counterBoolean++;
28               return result;
29       }
30
31       @Override
32       public void remove() {}
33
34       @Override
35       public Object getDefaultValue() {
36               return true;
37       }
38}
```