Software Development Lifecycle for Energy Efficiency: Techniques and Tools

STEFANOS GEORGIOU, Athens University of Economics and Business, Singular Logic S.A.
STAMATIA RIZOU, Singular Logic S.A.
DIOMIDIS SPINELLIS, Athens University of Economics and Business

Motivation: In modern IT systems, the increasing demand for computational power is tightly coupled with ever higher energy consumption. Traditionally, energy efficiency research has focused on reducing energy consumption at the hardware level. Nevertheless, the software itself provides numerous opportunities for improving energy efficiency.
Goal: Given that energy efficiency for IT systems is a rising concern, we investigate existing work in the area of energy-aware software development and identify open research challenges. Our goal is to reveal limitations, features, and tradeoffs regarding energy-performance for software development and provide insights on existing approaches, tools, and techniques for energy-efficient programming.
Method: We analyze and categorize research work mostly extracted from top-tier conferences and journals concerning energy efficiency across the software development lifecycle phases.
Results: Our analysis shows that related work in this area has focused mainly on the implementation and verification phases of the software development lifecycle. Existing work shows that the use of parallel and approximate programming, source code analyzers, efficient data structures, coding practices, and specific programming languages can significantly increase energy efficiency. Moreover, the utilization of energy monitoring tools and benchmarks can provide insights for the software practitioners and raise energy-awareness during the development phase.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; **Requirements analysis**; **Software design engineering**; **Software verification and validation**; *Extra-functional properties*; *Data types and structures*; *Classes and objects*; *Software design tradeoffs*; Concurrent programming structures; Patterns; Software implementation planning; • **Computing methodologies** → *Parallel computing methodologies*;

Additional Key Words and Phrases: GreenIT, energy efficiency, energy optimization, energy profiling, design patterns, parallel programming, code refactoring, source code analysis, coding practices, approximate programming, software development lifecycle

ACM Reference format:
Stefanos Georgiou, Stamatia Rizou, and Diomidis Spinellis. 2019. Software Development Lifecycle for Energy Efficiency: Techniques and Tools. *ACM Comput. Surv.* 52, 4, Article 81 (August 2019), 33 pages.
https://doi.org/10.1145/3337773

This work is supported and funded by the European Union within the H2020 MSCA-ITN-2014-EID Project "SENECA".
Authors' addresses: S. Georgiou and D. Spinellis, Department of Management Science and Technology, Athens University of Economics and Business, Patision 76, Athina 10434; emails: {sgeorgiou, dds}@aueb.gr; S. Rizou, European R&D Department, Singular Logic, Achaias 3 & Trizinias, N. Kifissia 14564, Greece; email: srizou@singularlogic.eu.

0360-0300/2019/08-ART81 $15.00
https://doi.org/10.1145/3337773

1 INTRODUCTION

ICT-related products and services contribute to high energy consumption.[1] As indicated by Gelenbe and Caseau (2015) and Van Heddeghem et al. (2014), the energy consumption of the IT sector is rising faster than initially predicted and is expected to reach 15% of the world's total energy consumption by 2020. Besides, recent studies presented by climate scientists in the SMARTer2030[2] report, outline that GreenHouse Gases (GHGs) due to IT, currently estimated at around 2.3% of the global GHGs, are growing much faster than initially predicted.

A proposed method to counter ICT's growing energy hunger is through *GreenIT*: the practice of designing, implementing, using, and disposing of IT-related products in an eco-friendly way (Murugesan 2008). The concept of GreenIT has seen widespread adoption and acceptance from research communities and organizations (Hernandez and Ona 2015; Deng and Ji 2015). Initially, GreenIT was mostly adopted and considered at the hardware level. For example, Bacha and Teodorescu (2013, 2014) and Papadimitriou et al. (2017) proposed firmware to reduce the voltage margin and supply voltage without degrading the operating frequency of the CPU to save energy. Leng et al. (2015) showed the energy benefits obtained by reducing the GPU's voltage margin. Through a survey study, Mittal and Vetter (2016) illustrated the energy optimization opportunities that non-volatile memories can offer.

Although hardware design and utilization are undoubtedly key factors affecting energy consumption, there is solid evidence that software design can also significantly alter the energy consumption of IT products (Capra et al. 2012; Ferreira et al. 2013; Eder 2013). To this end, dedicated conference tracks (e.g., GREENS,[3] eEnergy[4]) have identified energy efficiency as an emerging research area for reducing software energy consumption through software development practices. Related research has investigated the field of energy-efficient placement of processing tasks on computing nodes and heterogeneous computer systems (CPU and GPU), operating systems, virtual machines, real-time systems, and so on (Beloglazov et al. 2010; Kong and Liu 2014; Mastelic et al. 2014; Chen and Kuo 2007; Mittal and Vetter 2015). In this study, we aim at investigating the energy efficiency gains that can be extracted at each phase of the software development process.

Existing research works in the area have tried to address some of the challenges for reducing energy consumption in software development by defining appropriate metrics, employing energy measuring tools, and proposing best practices. For example, Bozzelli et al. (2013) presented some energy consumption metrics and classified them under various environments and purposes. In the context of energy monitoring tools, Noureddine et al. (2013) performed a study to point out the current state-of-the-art by contextualizing existing approaches regarding energy measuring tools for workstations/servers and smart-phones. An initial study by Procaccianti et al. (2016) shows 34 best practices[5] that can improve the energy efficiency of software.

Overall, current research provides a fragmented view of the energy-efficient techniques associated with the Software Development Life Cycle (SDLC) and examines only particular phases of it. Our work intends to fill this gap by presenting works in the development of energy-efficient software under the holistic scheme of SDLC. For the presented works, we focus on eliciting implications at each phase of the SDLC, not only concerning energy consumption but also for runtime

[1] Although in the physical sense energy cannot be consumed, we will use the terms energy "consumption," "requirement," and "usage" to refer to the conversion of electrical energy by ICT equipment into thermal energy dissipation to the environment. Correspondingly, we will use the terms energy "savings," "reduction," "efficiency," and "optimization" to refer to reduced consumption.

[2] http://smarter2030.gesi.org/downloads.php.

[3] http://greens.cs.vu.nl/.

[4] http://conferences.sigcomm.org/eenergy/2017/cfp.php.

[5] wiki.cs.vu.nl/green_software/.

performance, where it is relevant. The goal is to guide researchers and software practitioners on existing methods that can be beneficial and practical at each phase of software development. Additionally, we aim to raise awareness regarding current difficulties and limitations.

1.1 Contribution of This Study

This survey contributes to the field of energy efficiency for software development as follows.

- —It provides an overall view, analysis, and taxonomy of existing technologies, tools, and techniques for each phase of the SDLC, i.e., *Requirements* (Section 2), *Design* (Section 3), *Implementation* (Section 4), *Verification* (Section 5), and *Maintenance* (Section 6), for energy efficiency.
- —It identifies the state-of-the-art on energy-efficient design and development, presents a critical review on different parameters that may affect energy efficiency at each phase of the SDLC, and discusses limitations and future challenges (Section 7).

In the rest of this section, we explain our method for compiling related studies (Subsection 1.2), and describe our approach for classifying them (Subsection 1.3).

1.2 Methodology

Our study mainly focuses on research that has been conducted within the period spanning from 2010 to 2017. We chose this interval based on the observation that energy efficiency in software development gained further acceptance and publicity from significant organizations and conferences during these years. Nevertheless, we widened this scope by including work published before 2010 in cases where noteworthy studies exist.

This study aims to investigate dimensions that affect energy consumption at different phases of the SDLC. To retrieve related studies, we composed search queries from the keywords "energy" and "power," combined with relevant words, i.e., "software development lifecycle," "software requirements," "design patterns," "parallel programming/computing," "approximate programming/computing," "coding practices," "data structures," "programming languages," "code analysis," "benchmarks," "monitoring tools," "evaluation tools," "maintenance," and "refactoring."

We searched for relevant publications by querying the following digital libraries, journals, and magazines: IEEExplore, ACM, ACM Computing Surveys, Springer, ScienceDirect, and IEEE Software Magazine. Initially, we had restricted our search spectrum only to the main tracks of the top software engineering conferences as proposed by Pinto et al. (2015). However, the field of energy efficiency in software development is relatively new; hence related work published in these venues is still sparse. Therefore, we extended our search to other energy- and software-related conferences and workshops to enrich our dataset. Additionally, when a retrieved paper was on a topic close to our interests, we used *back reference searching* process to track down supplementary relevant publications.

1.3 Energy Efficiency in the Context of SDLC

In this article, we limit our scope of interest in existing work related to the energy efficiency focused at each phase of SDLC, i.e., *Requirements, Design, Implementation, Verification, and Maintenance*, following the *waterfall model* (Royce 1987). Although the SDLC waterfall model is an outdated software development approach, it is still useful as a reference model for categorizing the area's research. In Figure 1, we present the mapping of the energy efficiency techniques and tools under the SDLC process. Following this classification, we structure this article accordingly to provide a complete view of existing tools and techniques for energy-aware software development.

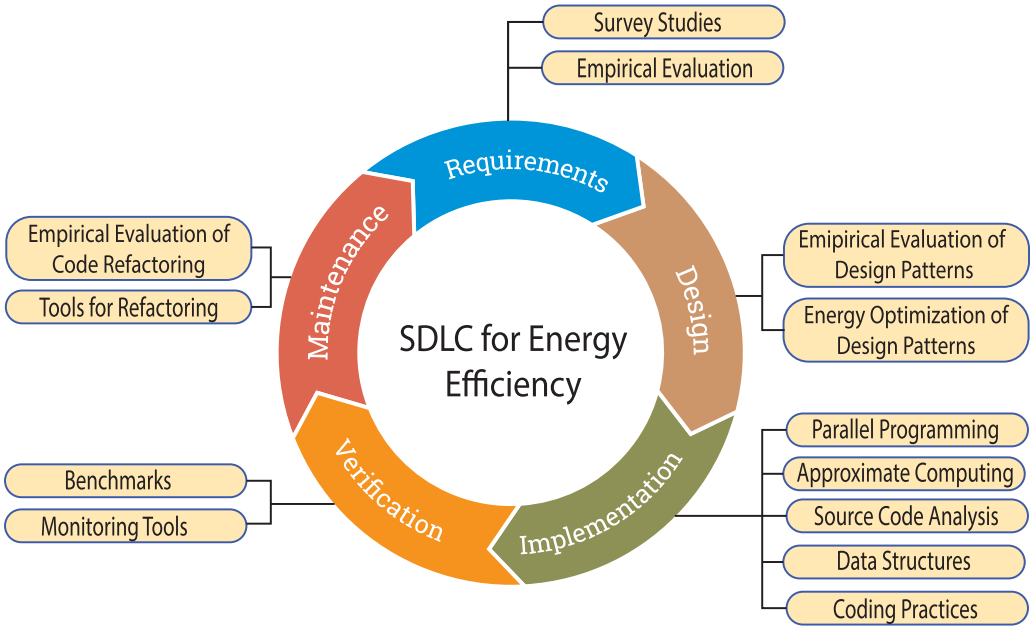

Fig. 1. SDLC for energy efficiency taxonomy.

The *Requirements* phase describes aspects and needs relevant to the development of software projects/systems when the energy efficiency is compulsory. During the *Design* phase, software design patterns document best practices used for solving common ground problems or poor design decisions taken during the development of an application that can impact the energy consumption. *Implementation* phase clusters methods/practices, namely *Parallel Programming, Approximate Computing, Source Code Analysis, Programming Languages, Data Structures,* and *Coding Practices*, which practitioners can adopt to reduce the software's energy consumption when developing. The *Verification* phase considers metrics and tooling support aiming to evaluate the software's energy consumption before its deployment. *Maintenance* is the phase that aims to apply refactoring patterns/techniques on a deployed application to reduce its energy consumption.

2 REQUIREMENTS

The classification of energy efficiency for the SDLC falls under the non-functional requirements such as runtime performance and security. We present relevant work on requirements concentrating on suggestions gathered by surveying practitioners and on results based on empirical evaluation. Accordingly, we divide the *Requirements* section into two subsections: survey studies and empirical evaluation requirements.

2.1 Survey Studies

Mobile devices are more energy dependent than servers or workstations; therefore, it is crucial to adopt an energy-conscious approach and consider the battery life, as a limitation, before developing a mobile application. A qualitative study conducted by Manotas et al. (2016) examined green, energy-efficient software engineering perspectives of 464 practitioners from ABB, Google, IBM, and Microsoft software development departments. Although the study concerned different computing systems, specifically, mobile and embedded systems and data centers, the practitioners expressed ideas, mostly, on how to reduce energy consumption on mobile applications. Some of the

practitioners also provided examples regarding the sustainability of smart-phones energy consumption, such as *"turn-by-turn guided navigation should not drain more battery than the car can charge"* and *"under normal usage, a device with an X W h battery should last for Y hours"* (Manotas et al. 2016). Additionally, practitioners suggest that particular tasks be executed without disturbing the users about battery drain.

Likewise, Pang et al. (2015) performed a study surveying on-line 122 programmers and concluded in line with Manotas et al. that software practitioners mostly consider energy consumption as a requirement for mobile application development. To this end, the authors argued that developers could extract energy-efficiency requirements by correlating applications functional requirements with the corresponding software components' energy consumption. To accomplish this, a consistent body of knowledge and understanding of software's and hardware's interaction is necessary. The authors listed the following relevant instructions that developers can take into account:

—Bulk Operations, to keep I/O calls minimum.
—Hardware Coordination, such as minimizing memory access.
—Concurrent Programming, such as selecting appropriate thread construct.
—Efficient Data Structures, selecting less energy-greedy data structures.
—Loop Transformation, such as loop fusion to reduce control operations.
—Data Compression, to reduce file sizes before transmitting them.
—Offloading Methods, by calculating heavy operations remotely, i.e., cloud.
—Approximate Programming, to reduce unnecessary precision of computations.

2.2 Empirical Evaluation Studies

An approach to identifying non-functional requirements to reduce energy consumption in software development has been proposed by Beghoura et al. (2015). The authors focused their study on desktops and smart-phones, where they argued that, to meet energy-efficient software development, it is necessary to identify the characteristics and requirements of software. Along these lines, a practitioner may consider four characteristics for providing energy efficient requirements for different types of systems, namely:

—Computations, to optimize energy consumption through less expensive computations (CPU-bound).
—Data Management, to reducing the amount of I/O operations, because they are slow and expensive for a system (storage-bound).
—Data Communication, to mind the amount of data sent or received through a network channel (network-bound operations).
—Energy Consumption Awareness, to provide energy-related information for individual software layers (e.g., through energy profiling tools) and for the software as a whole.

To this end, Beghoura et al. developed a tool for testing application requirements by estimating the energy consumption of CPU, RAM, and NIC. In contrast to Pang et al. and Manotas et al., Beghoura et al. empirically evaluate their proposed requirements. The work of Pang et al. and Manotas et al. are survey oriented and do not provide an assessment of the proposed suggestions. However, Manotas et al. obtained their requirements from experienced and well-known software development companies. Overall, there is a lack of research work concerning practical guidelines for the *Requirements* phase regarding the energy efficiency of software development. Most of the works were limited to developers experience. As illustrated in Table 1, all the researchers have

Table 1. Collected Requirements and Identified Limitations

Platform	Type of Study	Identified Limitations	Source
Smart-phones	Survey	In programmers education	(Pang et al. 2015)
Workstations and Smart-phones	Empirical Evaluation	On sustainability model for software development	(Beghoura et al. 2015)
Smart-phones	Survey	On guidelines, support infrastructure, and reasonable cost of a given task	(Manotas et al. 2016)

identified some limitations and challenges in the phase of *Requirements* which can serve as future research guidelines.

3 DESIGN

Proper design decisions are crucial when it comes to energy-efficient software development, as the software components and their interactions can alter significantly energy consumption. In the subsequent subsections we present studies that evaluate the energy implications of design patterns (Gamma et al. 1995) or recommend adjustments for optimizing them in terms of energy consumption. We also indicate cases where inappropriate design decisions lead to increased energy consumption.

3.1 Empirical Evaluation of Design Patterns

Design patterns are general and reusable solutions to commonly appearing software design problems. In the context of design patterns, Sahin et al. (2012) and Bunse et al. (2013) performed empirical studies where they compared the energy consumption of selected patterns. Both works evaluated design patterns from the *creational, structural,* and *behavioral* categories introduced by Gamma et al. (1995).

Specifically, Sahin et al. examined the energy consumption of different applications[6] running on an embedded system, with and without the application of design patterns. The authors illustrated a method of correlating software design and energy consumption that helps software developers understand the tradeoffs of their design decisions with energy consumption. For their experiment, they used 15 of the 23 design patterns. As an outcome, 3 of 15 used patterns resulted in substantial energy savings, while the remaining resulted in mini-scale changes or negatively affected energy consumption. Particularly the *Flyweight, Mediator,* and *Proxy* patterns resulted in energy savings when applied on selected applications, while the *Decorator* pattern tremendously increased energy consumption.

Bunse et al. focused on evaluating the energy consumption and runtime performance impact of design patterns on Android applications. The authors observed an increase in both energy consumption and execution time after applying 6 of the 23 design patterns (*Facade, Abstract Factory, Observer, Decorator, Prototype*, and *Template Method*) on selected applications.

The results summarized in Table 2 show that even in between instances of applying the same design patterns, large variations in energy consumption exist. For instance, according to the results of Bunse et al. the *Decorator* pattern increased energy consumption by 133%, while Sahin et al. found a significantly higher energy consumption, i.e., 712%; this might occur, because Bunse et al. employed a smart-phone and Java-based code snippets to experiment with, while Sahin et al. used an embedded system and code snippets written in C++.

[6] https://sourcemaking.com/.

Table 2. Design Patterns Empirical Evaluation on Energy-Performance Impact

Type of Study	Implications (avg. in %)			Platform's Type	Source
	Pattern	Energy	Performance		
Empirical Evaluation	Flyweight	58	–	Embedded System	(Sahin et al. 2012)
	Proxy	36	–		
	Mediator	9.56	–		
	Composite	– 5.14	–		
	Abstract Factory	– 21.55	–		
	Observer	– 62.20	–		
	Decorator	– 712.89	–		
Empirical Evaluation	Decorator	– 133.60	– 132.40	Smart-phone	(Bunse et al. 2013)
	Prototype	– 33.20	– 33		
	Abstract Factory	– 14.20	– 14.20		
Patterns Optimization	Observer	4.32	–	Workstation	(Noureddine and Rajan 2015)
	Decorator	25.47	–		

In conclusion, both Sahin et al. and Bunse et al. observed the negative impact of some design patterns in embedded and smart-phone devices regarding energy consumption. Moreover, Bunse et al. showed that particular design patterns causing high energy consumption were also contributing to lower runtime performance. As stated in both studies, the number of objects and communications among the software components comprise primary factors for increasing energy consumption. Both studies identified the *Decorator* and *Abstract Factory* as the most energy-inefficient design patterns. Additional work using specific benchmarks may help understand in depth the effects of design patterns in diverse applications and platforms.

3.2 Energy Optimization of Design Patterns

Structural changes in the design patterns can lead to significant energy optimizations as presented by Noureddine and Rajan (2015). Initially, the authors performed an experimental study where they manually applied 21 different design patterns on 11 applications written in both C++ and Java. Their experiment revealed the *Observer* and *Decorator* as the most energy-greedy design patterns. The authors transformed the existing source code, by reducing the number of created objects and function calls, and accomplished important energy savings. Specifically, after optimizing the *Observer* and *Decorator* design patterns, the authors reduced the applications' energy consumption by 10%, on average.

The work of Noureddine and Rajan (like Sahin et al.) lacks runtime performance measurements. Consequently, it is uncertain how the design pattern optimizations affected applications execution time. However, such design pattern optimizations seem to be a promising area for further investigation. To this end, a tool that indicates structural changes for design patterns which lead to optimized applications' energy consumption could benefit software practitioners.

4 IMPLEMENTATION

During the *Implementation* phase, there are several tools, techniques, and strategies that software practitioners can exploit to improve the energy consumption and runtime performance of their

Table 3. Parallel Programming Energy-Performance Impact

Thread Management Strategies	Adjustments	Implications (in %) Energy	Performance	Application Type	Source
Effective Parallelization[c]	From 1 to 16 Threads	55 avg.	69 avg.	CPU & GPU bound[d]	(Kambadur and Kim 2014)
Effective Parallelization & Thread Management Constructs	Explicit Threading	–	–	I/O-bound[e]	(Pinto et al. 2014)
	Work Stealing	(−50) – 30	(−23) – 10	Embarrassingly Parallel[f]	
Pack & Cap[a]	Criticality level DVFS	56 avg.	Unaffected	CPU-bound[b]	(Cai et al. 2011)
Work Stealing[g]	Load-base DVFS	11–12	(−3)–(−4)	CPU & GPU bound[h]	(Ribic and Liu 2014a)

[a] Collecting threads under the same core and reducing its frequency.
[b] Recognition-Mining-Synthesis (RMS), an Intel's application (Chen et al. 2008).
[c] Statically selecting the effective number of threads.
[d] Parsec 3.0, SPLASH-2X, SPEC CPU 2006, DaCapo 9.12, SPEC JBB 2013.
[e] Application such as *Largestimage*.
[f] Applications such as *Sunflow, Spectralnorm, N-Queens, Tomcat.*
[g] *"Underutilized processors take the initiative: they attempt to steal threads from other processors"* (Blumofe and Leiserson 1999).
[h] Applications from Problem-based benchmark suite (Shun et al. 2012).

applications. To this line, we present research results associated with programming techniques, source code analyzers, and programming languages.

4.1 Parallel Programming

Parallel programming is the process of breaking a large problem into smaller ones to solve them simultaneously. In this section, we discuss works that empirically evaluate applications and algorithms that utilize parallel computing. Table 3 summarizes implications of related thread management strategies and applied adjustments on energy consumption and runtime performance for different application types. Fields with negative values indicate an increase in energy consumption or execution time for the related source.

4.1.1 Experimental Studies. By performing an empirical study, Kambadur and Kim (2014) identified configurations and parameters that can reduce the energy consumption of parallel applications. The authors compared nine existing energy management strategies by using a standardized system architecture, operating system (OS), measuring tool, and five benchmark suites. The strategies (e.g., `Processor Frequency Tuning`, `Overclocking`, `Parallelism`, and `Compiler Optimization`) were run on 220 experimental configurations and tested on 41 applications totaling in more than 200,000 executions. The obtained results highlight the importance of effective source code parallelization by employing the appropriate number of threads (see Table 3).

To evaluate the energy efficiency of different thread management constructs, Pinto et al. (2014) performed an empirical study comparing three constructs (i.e., *explicit thread creation, fixed-size thread pooling*, and *work stealing*) on nine benchmarks by adjusting the number of threads, the task's granularity level, the data size, and the nature of the data access. The authors observed that the constructs' energy consumption vary in different situations. For example, *explicit thread creation* exhibits the lowest energy consumption when it comes to I/O-bound applications. For highly parallelized benchmarks, *work stealing* outperforms *explicit thread creation* and *fixed-size thread pooling* by being 30% more energy efficient; however, for more serialized benchmarks, *work stealing* is under-performing. Also, the authors noticed that energy consumption increases as the

number of threads increases. This occurs until the number of threads reaches the number of CPU cores. Afterward, the authors detected a reduction in energy consumption for most of the tested applications.

As we can see from Table 3, the gains and losses concerning energy consumption and runtime performance are feasible by selecting the appropriate number of threads. For instance, both Pinto et al. and Kambadur and Kim tuned the number of threads to evaluate their applications. By using different thread management constructs and various number of threads, Pinto et al. altered energy consumption from −50% to 30%, while Kambadur and Kim decreased it by 55%, on average. A major distinction between the two works, that can explain the discrepancies in results, is that Pinto et al. used Java applications, while Kambadur and Kim utilized Java and, also, C programs with compiler runtime optimization flags.

As shown above, performing experiments with a varying number of threads or thread management constructs can help practitioners identify the configurations most likely to reduce the energy consumption and execution time of their applications. However, users interaction is mandatory to point out the best configurations and parameters through experimentation—a fact that makes the selection process time-consuming and cumbersome.

4.1.2 Algorithms. To take advantage of parallel processing, Cai et al. (2011) and Ribic and Liu (2014b) developed algorithms to minimize the energy consumption by efficiently managing thread workloads. According to Cai et al., most of the Dynamic Voltage Frequency Scaling (DVFS) techniques—a mechanism that tunes CPU voltage to adjust its frequency based on the current workload—are not built to run on multi-threaded processors; therefore, they are unable to save a considerably large amount of energy. To this end, the authors suggested *thread shuffling*, a way of combining techniques such as thread migration and DVFS to reduce the energy consumption of an application without compromising its runtime performance. The basic idea behind *thread shuffling* is to identify threads with the same *thread critical degree* (slow threads execution) by using a prediction algorithm and map them under the same core via thread migration. Afterward, the algorithm applies voltage dynamically to scale the cores' frequency for the cores housing non-critical threads (fast threads execution).

Likewise, Ribic and Liu introduced HERMES, a strategy for work-stealing that employs a *thief-victim* approach. The authors refer to *thief* as the thread which finishes its tasks and *steals* work from other threads, the so-called *victims*. HERMES is composed of two main algorithms, the *workpath-sensitive* and *workload-sensitive*. The *workpath-sensitive* algorithm defines a thread's tempo (execution speed) based on the *thief-victim*'s relationship; that means, when a thief steals from a victim worker, its tempo is set lower (because it always steals insignificant tasks) and it is raised once the victim runs out of work. The *workload-sensitive* algorithm is a work-flow based tuning mechanism for the workers execution tempo that, if necessary, adjusts the core's frequency via DVFS to reduce energy consumption. For instance, if a worker's queue is empty, it tries to steal work from other workers; afterward, it adjusts its core's frequency according to its workload.

In the above works, both *thread shuffling* and HERMES utilize DVFS techniques with workload migration to accomplish energy savings. *Thread shuffling* packs all the threads with a similar criticality level under specific cores and then adjusts the cores' clock frequency accordingly, while HERMES alters the clock frequency of each core based on the threads' current workload. Particularly, the *work-stealing* approach of HERMES resulted in minor energy savings viz-a-viz the *thread shuffling* technique which resulted in 56% of energy savings, on average, as illustrated in Table 3. In conclusion, HERMES provided energy savings with a minor runtime performance penalty (i.e., 3–4%), while *thread shuffling* achieved compelling energy savings without compromising runtime performance at all (see Table 3).

Table 4. Approximate Computing Energy-Performance Impact

Tool Name	Approximate Optimization Focus	Implications (in %) Energy	Performance	Precision Loss	Source
GREEN	Computations and loop termination	14 avg.	21 avg.	0.27	(Baek and Chilimbi 2010)
–	Function memoization	74 avg.	79 avg.	<3	(Agosta et al. 2011)
Parrot	Optimizes regions of imperative code	66 avg.	56 avg.	10 avg.	(Esmaeilzadeh et al. 2012)
Chisel	Computational kernel operations	9–20	–	<3	(Misailovic et al. 2014)
EnerJ	Computations, Data Storage, and Algorithmic[a]	10–50	–	–	(Sampson et al. 2011)
Axilog	Source code parts	54 avg.	–	10 avg.	(Yazdanbakhsh et al. 2015)
–	Selected group of tasks[b]	–	–	–	(Vassiliadis et al. 2016a)
DCO SCORPIO	Selected group of tasks[b]	56 avg.	–	–	(Vassiliadis et al. 2016b)

[a]Programmer can write two different implementations, one is invoked when the data are precise and the other when they are approximate.
[b]Allows the developer to set which computation tasks, from a group, are going to be executed approximately/precise.

4.2 Approximate Computing

Approximate computing is an approach for sacrificing computation accuracy—when an application tolerates it—to increase runtime performance or energy savings (Mitra et al. 2017). For instance, techniques such as `loop perforation` that allow users to manage runtime performance and accuracy tradeoffs based on the desired output quality (Sidiroglou-Douskos et al. 2011). In this section, we discuss studies divided into programming frameworks, memoization, annotation-based, and directive-based extensions. In Table 4, we summarize the corresponding works.

4.2.1 Programming Frameworks. For their research, both Misailovic et al. (2014) and Baek and Chilimbi (2010) introduced energy-conscious programming frameworks to achieve energy savings through approximate computations. Specifically, Misailovic et al. proposed *Chisel*, an optimization framework that acts in an automated manner by selecting approximate kernel operations that result in energy, reliability, and accuracy optimizations. Baek and Chilimbi suggested GREEN, a framework aiming to optimize expensive loops and functions by considering user-defined Quality of Service (QoS) requirements. GREEN achieves energy savings through approximate computations for functions and early loop termination. Moreover, it addresses the QoS and energy consumption tradeoffs by applying approximate programming techniques only when the QoS requirements are fulfilled. A common element of both GREEN and *Chisel* is the comparison between precise and approximate instances for calculating the reliability of the results. However, what differentiates GREEN from *Chisel* is the periodical runtime QoS sampling to adjust its approximation techniques and QoS model to meet the target requirements. In contrast to GREEN, *Chisel* is utilizing the approximate memory of its running platform to increase energy savings by sacrificing some of its quality output (see Table 4).

4.2.2 Annotation-based Extensions. Sampson et al. (2011), Esmaeilzadeh et al. (2012), and Yazdanbakhsh et al. (2015) proposed *EnerJ*, *Parrot* transformation, and *Axilog*, respectively; all are extensions that achieve energy savings by executing approximately annotated source code

portions. Specifically, *EnerJ* is a Java-based extension, furnished with a manual annotation functionality for defining approximate or precise data selection for an application. By declaring variables and objects as approximate, *EnerJ* maps them to approximate memory[7] and generates low-cost energy code by using approximate operations and algorithms. *Parrot* transformation is a neural network model that identifies imperative code regions and offloads them to neural processing unit, instead of CPU, to increase energy and runtime performance. *Axilog* is a Verilog extension composing brief and high-level annotations for full control and governance of approximate hardware. In contrast to the manual annotating approach of *EnerJ* and *Parrot*, *Axilog* employs a Relaxability Interface Analysis algorithm to automate the approximation processes based on the designer's choices. All three annotation systems offer a safety mechanism for isolating approximate from precise portions of code, thus guaranteeing the main functionality of an application.

In contrast to GREEN, *Chisel*, and *Axilog*, with *EnerJ* and *Parrot* the developers are the wheelholders of the applied approximation techniques, by selecting which code portions to be executed as precise. Therefore, *EnerJ* and *Parrot* can help practitioners better understand the energy-approximation tradeoffs of their design choices.

4.2.3 Directive-based Extensions. Another approach for applying approximation techniques on computational tasks is to depend on their significance level (level of importance or criticality). To this line, Vassiliadis et al. (2016a, 2016b) proposed directive-based approaches by extending the OpenMP that use approximation techniques to reduce applications energy consumption. In the Vassiliadis et al. (2016a) work, the authors proposed a programming model aiming to elicit the highest level of accuracy for an application according to a user-defined energy budget. Therefore, the authors introduced a runtime system that is responsible for choosing the appropriate configurations, (i.e., number of cores, clock frequency, and accuracy ration) for a specific input size. The appropriate configurations are inferred by a model that is trained to identify the configurations that provide the highest possible output accuracy for a given energy budget. Another approach is DCO/SCORPIO, a framework suggested by Vassiliadis et al. (2016b) that supports automated analysis to identify the code's significance level. DCO/SCORPIO accomplishes that by employing *interval arithmetic* (Rall 1984) and *algorithmic differentiation* (Naumann 2012) to quantify the significance of particular computations for a specific input. This output is in turn used by an OpenMP-like model to classify the computations in task groups according to their significance level. Thus, it provides approximate methods based on each group's significance level.

DCO/SCORPION and the work of Vassiliadis et al. (2016a) differ in that the former reduces the energy consumption of an application, while the latter achieves the highest possible output quality within a specific energy consumption threshold. Also, DCO/SCORPIO is releasing the hands of a user by selecting the computation tasks significance, while Vassiliadis et al. (2016a) require the programmer's involvement to input directives for critically important parts of the code that do not tolerate imprecision.

4.2.4 Memoization. Another approach of saving energy through approximate computing is by using memoization to store expensive function call results. Agosta et al. (2011) developed a performance model to select and memoize computationally intensive function from Financial applications and JavaGrande benchmark. Compared to the above approaches, memoization seems to have the highest energy savings and runtime performance (see Table 4). However, the authors did not applied their method on real-world applications.

[7] Memory parts with reduced voltage or refresh rate such as cache, registers, functional units, and main memory.

Table 5. Source Code Analysis Related-Work Information

Tool Name	Target Platform	Energy Measurement Correlation	Error Margin (in %)	Source
Eprof	Android & Windows OS	Process, Threads, System calls, Routines	6<	(Pathak et al. 2012)
eLens	Android	Full Source Code Granularity	10	(Hao et al. 2013)
GreenAdvisor	Android	Routines, System-calls	–	(Aggarwal et al. 2015)
PEEK	Embedded	Function Level Systems	–	(Honig et al. 2014)
SEEDS	Any Java-based Platform	User can set which portion to analyze	–	(Manotas et al. 2014)
–	Smart-phones	Function Level	–	(Banerjee et al. 2014)

4.3 Source Code Analysis

Source code analysis is a testing process that focuses on revealing defects and vulnerabilities in a computer program before its deployment phase. In this section we discuss works on dynamic source code analysis that aim to identify energy-related bugs and hot-spots by testing a computer program at real-time. In the context of source code analysis, we did not find available tools for static code analysis that provide rules for analyzing source code before execution. Table 5 shows works on source code analysis and provides information on the target platform, energy measurement correlation at various software granularities, and the error margin rate.

4.3.1 System Calls. *GreenAdvisor* is a profiler system calls that predicts behavior, runtime performance, and energy-related modifications of an application (Aggarwal et al. 2015). To predict energy-related changes, *GreenAdvisor* compares the number of system calls on a current version of a software and its previous one. If energy consumption increases, then it pinpoints the energy hot-spots that caused the changes, thus helping developers in analyzing and understanding the implications of their decisions.

To diagnose energy bottlenecks at the source code level, Pathak et al. (2012) presented *Eprof*, a system-calls based power modeling tool for smart-phone applications. To achieve high accuracy in energy consumption measurements, *Eprof* incorporates two subsequent components. First, it uses finite-state machines to model different power states and transitions for individual hardware components and the smart-phone as a whole. Then, for each hardware component, it runs a benchmark suite that consists of applications for collecting the different system calls and their power transitions. Afterward, it generates rules for the finite-state machines by integrating the collected knowledge from the executed applications. To estimate energy consumption at source code level, *Eprof* cross-references routines (blocks of code) with system call traces. As an outcome, the authors found that applications using third-party processes tend to have a 65–75% increase in energy consumption.

Overall, the above-discussed tools are utilizing distinct energy estimation models to present energy measurements. For example, *Eprof* employs a model that calculates energy consumption of an application at the routine level and various hardware components, while *GreenAdvisor* compares the energy consumption of different versions of a product and points out the system calls that caused the change.

4.3.2 Optimization Tools. Honig et al. (2014) and Manotas et al. (2014) recommended tools for dynamic source code analysis. Both Honig et al. and Manotas et al. suggested energy-aware programming approaches to guide developers during the implementation phase by providing

energy-related hints. Honig et al. presented the Proactive Energy-awarE development Kit (PEEK), while Manotas et al. promoted the Software Engineer's Energy-optimization Decision Support (SEEDS) framework. Periodically, both approaches analyze the source code under development, and seamlessly create many different instances from the current source code to identify optimal code modifications. However, a significant difference between the two approaches is that PEEK's energy-associated hints are related to power management mechanisms (e.g., sleep state, DVFS, idle state, program-code logic, libraries, and compiler runtime optimization flags) while SEEDS' suggestions are limited to Java Collection Libraries (JCL), algorithms, or refactoring parts of the source code. Besides, SEEDS offers to developers the possibility to set a code block range for analysis while PEEK analyzes source code at function granularity.

4.3.3 Tests Generation Framework. Another way for identifying energy-related bugs is through an automated test generation framework, introduced by Banerjee et al. (2014) The aforementioned research points out energy-related hot-spots in four categories of smart-phone applications that are (1) *hardware resources*, (2) *sleep-state transitions*, (3) *background services*, and (4) *defective functionality*. First, a detection process is invoked to search for possible user interactions through event flow graphs. Then, the advocated framework generates test cases to capture interaction scenarios and, subsequently, to identify energy hot-spots. Alongside the energy hot-spots identification, the tool issues test reports for the developers. However, compared to the work of Honig et al. and Manotas et al., the work of Banerjee et al. does not provide hints for energy optimization, but only finds the energy-wasteful parts of an application.

4.3.4 Line-by-Line. To measure applications energy consumption at different levels of software granularity and raise energy-awareness during the development phase, Hao et al. suggested *eLens*. *eLens* estimates energy consumption via program analysis and per-instruction power modeling. The authors use program analysis to obtain execution-related information such as bytecode or API calls from various smart-phone components (e.g., CPU, RAM, GPS, 3G). Then, the collected information is passed to the per-instruction power modeling component to estimate an application's energy consumption. Thus, *eLens* provides energy measurements at various levels of granularity: application, method, class, path, and source code lines. Compared to all presented tools, *eLens* is the only one that provides energy measurements at all levels of source code granularity. Thus, raising energy-awareness by providing fine-grained information on application consumption.

4.4 Programming Languages

Programming languages offer a set of instructions that allow users to utilize system resources to solve a problem. Languages differ in features and the way they allocate computing resources. In this section, we examine empirical studies investigating the energy consumption and runtime performance implications of programming languages. Table 6 lists related works in terms of energy consumption, runtime performance, employed optimization flag, and test cases. We summarize related results in a consolidated list, found in Table 6, of average values calculated by us.[8]

The studies presented in this section are based on different experimental platforms. Particularly, Abdulsalam et al. (2014) performed their tests on a workstation, Rashid et al. (2015) used an embedded system, and Chen and Zong (2016) conducted their experiments on a smart-phone. However, Abdulsalam et al. and Chen and Zong used similar testing parameters, as depicted in Table 7. Furthermore, Abdulsalam et al. also compared the energy implications of four memory allocation methods (i.e., `malloc, new, array,` and `vector`) where they presented `malloc` as the most energy and runtime-performance efficient.

[8] https://github.com/stefanos1316/Proof_for_Survey/blob/master/Programming_Languages_Average_Values.txt.

Table 6. Programming Languages Energy-Performance Impact

Comparing X to Y	Implications (avg. in %) Energy	Performance	Opt. Flag	Run-Time Env.	Selected Application	Source
C++ to C	8.4	8.67	−O3	–	All Apps[a]	(Abdulsalam et al. 2014)
C++ to Java	47.4	38.12				
C++ to Python	166.28	195.17				
C to Java	38.39	29.08				
C to Python	106.4	195.5				
Java to Python	195.87	194.77				
C++ to Java	166.14	159.13	−O3	Dalvik[b]	Quick Sort	(Chen and Zong 2016)
C++ to Java	Identical	Identical		ART[c]		
C to C++	Identical	3.38		–	Fibonacci	
ARM-assembly to C	15.38	10.52	–	–	Counting Sort	(Rashid et al. 2015)
ARM-assembly to Java	58.84	90.90			Counting Sort	

[a] Fast Fourier, Quick Sort, Linked List.
[b] https://source.android.com/devices/tech/dalvik/.
[c] Android Run-Time (ART), https://source.android.com/devices/tech/dalvik/.

Table 7. Programming Languages Configurations

Programming Languages	Optimization Flags	Target Platforms	Test Cases	Source
C, C++, Java, and Python	−O{1,2,3}	Server system	Fast Fourier, Linked List, Quick Sort	(Abdulsalam et al. 2014)
ARM assembly, C, and Java	–	Embedded system	Bubble, Merge Quick, Counting	(Rashid et al. 2015)
C, C++, and Java	−O{1,2,3}	Android devices	Fibonacci, Tower of Hanoi, Pi calculation	(Chen and Zong 2016)

In their experiment, Chen and Zong utilized the Native Development Kit[9] tool-set for executing native code such as C and C++ inside Android applications. The derived output, by both Abdulsalam et al. and Chen and Zong, is that C and C++ achieved significant energy savings and, also, reduced execution time against the other programming languages. Also, both works showed that the runtime compiler optimization flag −O3, had the most significant energy savings and increased runtime performance for workstation and smart-phone applications. Moreover, for Java applications, Chen and Zong showed that the use of Android Run-Time instead of Dalvik runtime environment, contributed to energy and runtime performance results similar to the C and C++ implementations.

In the context of embedded systems, Rashid et al. performed an experiment to compare the energy and runtime performance implications of four sorting algorithms written in ARM-assembly, C, and Java. By performing their experiments on a Raspberry Pi,[10] the authors showed that the implementations of ARM-assembly achieved the most energy-efficient results viz-a-viz the C and Java implementations. Likewise, Rashid et al. presented Java as the most energy-hungry among the selected programming languages.

[9] https://developer.android.com/ndk/index.html.
[10] https://www.raspberrypi.org/.

Table 8. Data Structures Energy-Performance Impact

Collection Interface	Collection Library	Data Structure	Avg. Energy Impact (%)	Selected Apps	Source
	C5	HashedLinkedList	23.27	Apps[a]	(Michanan et al. 2016)
	JCF	AttributeList	24.88	CEB[b]	(Pereira et al. 2016)
List	JCF	ArrayList	38	Gson	
		LinkedList	−309	SETS[c]	(Hasan et al. 2016)
	JCF	LinkedHashMap	50.16	CEB[b]	(Pereira et al. 2016)
Map	JCF	ConcurrentHashMapV8	17.8	XALAN	
	JCF	ConcurrentHashMapV8	9.32	TOMCAT	(Pinto et al. 2016)
	C5	HashSet	31.44	Apps[a]	(Michanan et al. 2016)
Set	JCF	LinkedHashSet	12.5	CEB[b]	(Pereira et al. 2016)
Queue	JCF	PriorityQueue, LinkedTransferQueue, ConcurrentLinkedDeque, LinkedBlockingDeque, ConcurrentLinkedQueue	7.5	Apps[d]	(Manotas et al. 2014)
Bag	C5	HashBag	16.93	Apps[a]	(Michanan et al. 2016)

[a] A* Path Finder, Huffman Encoder, Genetic Algorithm.
[b] https://github.com/greensoftwarelab/Collections-Energy-Benchmark.
[c] Stock Exchange Trading Simulator.
[d] Barbecue, Jdepend, Apache-xml-security, Joda-Time, Commons Lang, Commons CLI.

Table 6 illustrates the superiority of compiled programming languages against the interpreted and semi-compiled,[11] in terms of energy consumption and runtime performance. Java and Python suffer the most from high energy consumption and low runtime performance. The use of an interpreter makes Python slower and less energy efficient, because it has to interpret source code for each execution. Moreover, the dynamic compiling, library linking, and interpretation of bytecode in JVM are additional burdens on the execution of Java programs. However, Chen and Zong showed that the use of the Android Run Time environment could reduce the energy consumption and increase runtime performance of Android applications.

A limitation that we observed, for all the discussed works, is the lack of information regarding the versions of the employed compilers, interpreters, modules, and libraries used in the experiments. Additionally, we did not find any research study that compares the energy consumption and runtime performance implications of the same application across different versions of compiler, runtime engine, or interpreter.

4.5 Data Structures

A data structure is a way to organize, manage, and store data for further process or analysis. This section consists of *Empirical Studies* and *Tooling Support* for data structures. In the *Empirical Studies* part, we discuss works trying to identify which data structures are the most energy efficient for particular cases. For the *Tooling Support*, we show tools that inform a practitioner which data structure to select to reduce energy consumption. Table 8 summarizes works on the data structures collection interface and library, energy consumption, and tested applications for the relevant source.

[11] Semi-compiled languages compile source code into intermediate code and execute it on a VM.

4.5.1 Experimental Studies. By conducting experimental studies, Pereira et al. (2016), Hasan et al. (2016), and Pinto et al. (2016) identified some positive and negative cases concerning the energy consumption of various data structures. The authors evaluated the energy consumption of data structures from different interfaces, but mostly from the Java Collection Framework (JCF).

Pereira et al. performed an experimental study to evaluate the energy efficiency of different JCF interfaces methods such as search, iteration, removal, and insertion. By manually replacing data structures in applications, the authors obtained significant energy savings as illustrated in Table 8. Because the authors of the above work report an extensive list of results, we compared the energy consumption of the most and least energy-efficient interface,[12] and we present in Table 8 the data structure with the most energy-efficient methods for each interface.

By examining the memory usage and analyzing byte-code traces of Android applications, Hasan et al. evaluated the energy consumption of different collection types. Apart from JCF, the authors used data structures from Apache Commons Collections (ACC) and Trove.[13] As an outcome, the authors showed that energy consumption starts to diverge among the data structures only when the number of elements they contain is above 500. Additionally, the authors noted for the data structures with primitive data types (found in Trove collection) that, while consuming less memory than objects, they are more energy-inefficient in most of the cases.

Similarly, Pinto et al. used 13 thread-safe and three non-thread-safe implementations of JCF. The authors tested the above data structures by utilizing distinct configurations such as the number of threads, initial capacity, and load factor. As a result, the authors observed that the proper data structure selection and the number of threads (for most of the thread- and non-thread-safe implementations) can decrease applications energy consumption. For example, when the authors replaced the `HashTable` instances with `ConcurrentHashTableV8`, in real-world benchmarks, they achieved significant energy savings.

Overall, Pereira et al. showed that the same kind of method implementations (e.g., add, remove, search) affect the energy consumption of data structures in a different way. In Table 8, we can see that Pereira et al. achieved the highest energy savings in their experiments. However, compared to Hasan et al. and Pinto et al., Pereira et al. used micro-benchmarks and not real-world applications. Pinto et al. examined the energy impact of real-world applications by (i) replacing non-thread-safe with thread-safe data structures and (ii) changing the number of threads. This helped them to achieve substantial energy savings as illustrated in Table 8. Furthermore, the results by Hasan et al. reveal the negative impact of unwise data structure selection which can significantly affect energy consumption (see Table 8).

4.5.2 Tooling Support. Predicting the most energy-efficient data structures for a given problem can decrease energy consumption. To this line, Michanan et al. (2016) introduced GreenC5, a tool that can predict which data structures among the Copenhagen Comprehensive Collection Classes for C# (C5) collection can reduce application energy consumption based on the system's workload. GreenC5 is composed of a predictive algorithm based on machine learning and neural network models. As shown in Table 8, Michanan et al. achieved energy savings for real-world applications.

Manotas et al. (2014) introduced SEEDS (also discussed in Subsection 4.3.2), a decision support framework that can dynamically evaluate Java collection types and modify them to reduce the energy consumption of an application. To do that, SEEDS creates instances of an application under development using different queue data structures, to find the most energy efficient ones. This

[12] https://github.com/stefanos1316/Proof_for_Survey/blob/master/Pereira_Data_Structure.txt.
[13] http://trove.starlight-systems.com/.

Table 9. Code Practices Energy-Performance Impact

Coding Practice	Practice Implication (in %) Practices	Energy	Performance	Operating System	Source
	For loop with length	36–52	33–38	Android [a]	(Tonini et al. 2013)
		10 avg.	–	Android 4.2	(Li and Halfond 2014)
Good	Efficient query usage	25.1 avg.	24.9 avg.	Linux distro	(Procaccianti et al. 2016)
	Put application to sleep	8.48 avg.	6 avg.		
	Change macros to function calls, loop unrolling, reducing lookUp tables sizes[b]	179.3 avg.	–	–	(Grossschadl et al. 2007)
	Avoid setters & setters	24–27	24–30	Android [a]	(Tonini et al. 2013)
		30–35	–	Android 4.2	
	Invoke static methods	15 avg.	–	Android 4.2	(Li and Halfond 2014)
Bad	Use of relational database				
	Use of unnecessary views and widgets	–	–	Android 4.2	(Linares-Vásquez et al. 2014)

[a]Used Android 2.3.6 for a Samsung Galaxy Y Pro Duo, Android 4.0.3 for an Asus Tablet, and Android 4.0.4 for a Motorola Tablet.
[b]Comparing RC6 against Twofish, also runtime performance was calculated in clock cycles instead of time, therefore, we did not adding it in our results.

helped Manotas et al. gain energy savings, in real-world applications, while choosing the proper queue type data structures.

For the tooling support, we observe several shortcomings such as limited number of collection types (SEEDS is available only for queue) or focus on specific collections (GreenC5 uses only the C5 collection). However, these tools can assist a developer for manually selecting and experimenting with various data structures to obtain sufficient energy savings. Moreover, further support on various data structure types and different collections can make such tools more beneficial and attractive to the developers.

4.6 Coding Practices

Best coding practices are sets of rules, formally or informally, established from various coding communities that help software practitioners to improve software quality. In this section, we discuss works on empirical evaluation that examine the energy consumption of coding practices. Table 9 summarizes results identified from the related works as "Good" and "Bad" coding practices. By the terms "Good" and "Bad," we refer to coding practices that may impact a program's readability, maintainability, efficiency, and usability positively or negatively, respectively.

In terms of embedded systems, Grossschadl et al. (2007) evaluated the runtime performance, energy consumption, memory usage, and code size of five block ciphers (i.e., RC6, Rijndael, Serpent, Twofish, and XTEA) on a StrongARM SA-1100 processor. The authors modified the existing source code of the cipher algorithms (to reduce their lines of code) by:

—Replacing `macros` with `function calls`.
—Using `loop unrolling` in the encryption and decryption functions.
—Replacing `T-lookup` with forward and inverse `S-box` tables and reducing their sizes.

Their results show that the block ciphers XTEA and RC6 (which had the smallest code size) were the most energy efficient, offered the best runtime performance, and utilized less main memory for the encryption and decryption tasks.

In their experiment, Tonini et al. investigated the energy efficiency of best practices for Android development. Their results indicate that the proper use of `for loop` and `getters/setters` can improve energy consumption. Initially, the authors performed experiments by using different variations of `for loop`, i.e., `for-each`, when the loop's termination condition is i) calculated at each iteration, and ii) when it is passed as a variable. In addition, the author evaluated scenarios with and without `getters/setters` to access the class fields. Their results show significant energy savings by using a variable as the loop termination condition and accessing class variables without using `getter/setter` functions (see Table 9).

Likewise, Li and Halfond checked practices such as HTTP request bundling with specific size and memory usage, and performance tips. Particularly, for the performance tips, the authors inspected coding practices, obtained from the Android developer forum.[14] The application of coding practices helped Li and Halfond to obtain notable energy savings as illustrated in Table 9. The practice of avoiding calculating a data structure's length in a loop proved beneficial, since, having the loop's termination condition in a variable saves energy by bypassing the calculation of the length at each iteration. The practice of direct field access also proved beneficial, because no additional function call is required from the system, unlike the case where a field value is retrieved through method invocation. Finally, the practice of static invocation proved energy efficient, because calling a method statically saves energy as it avoids the lookup overhead for calling methods through an existing object.

Android offers a variety of Application Programming Interface (API) calls and if not used efficiently they can contribute in increased energy consumption (Linares-Vásquez et al. 2014). In their study, Linares-Vásquez et al. performed an analysis on 55 Android applications from various domains and they listed the most energy-inefficient API methods. Moreover, the authors suggested a list of practices that can yield energy savings by effectively using API calls. To obtain their results, the authors correlated `timestamps` of method execution traces with energy consumption measurements. After analyzing their results, they identified 133 energy-greedy APIs of the total of 807. From the energy-greedy APIs, 61% are related to graphical user interface and image manipulation, while the remaining 39% fall under the category of database. In conclusion, the authors highlighted that the unnecessary refreshing of `views` (e.g., redrawing a `view` upon receiving new data) and widgets can consume a significant amount of energy. In addition, they highly recommend users to avoid relational databases, such as SQLite,[15] when it is not of paramount importance.

Two best practices were recommended and evaluated by Procaccianti et al. (2016). The selected practices were *put the application to sleep*, i.e., put processes or threads on sleep state if they are waiting for I/O operations or they are no longer active, and *use efficient queries*, i.e., avoid the use of expensive energy operations such as ordering or indexing when not needed. As an outcome, the authors achieved energy efficiency by using both practices and, also, increased runtime performance (see Table 9).

The results in Table 9 show that the usage of `for` with a given length size contributes to energy savings from 36% up to 52% according to Tonini et al. and Li and Halfond, respectively. Furthermore, in both works the authors avoided the use of `getters/setters` and saved energy equal to 24–27% (Tonini et al. 2013) and 30–35% (Li and Halfond 2014). Although both studies used micro-benchmarks, their results offer a different scale of energy savings. This may have occurred because

[14] https://developer.android.com/training/articles/perf-tips.html.
[15] https://www.sqlite.org/about.html.

Table 10. Benchmarks Related-Work Information

Benchmark Suite	Target Platform	Energy Correlcation With	Optimization Level	Source
GBench	Linux	Hardware Components	Data memory movement, block size, number of cores	(Subramaniam and Feng 2012)
ALEA	Linux	Basic code blocks	Power capping, DVFS, compiler optimization, thread throttling	(Mukhanov et al. 2015)
AxBench	Linux	Hardware components	Source code via approximation	(Yazdanbakhsh et al. 2017)
PowerBench	TinyOS	Each node of the test-bed	–	(Haratcherev et al. 2008)

they employed different hardware devices, distinct Android versions, and different tools to obtain their energy measurements. From Table 9, we can also observe that Grossschadl et al. achieved significant energy savings (i.e., 179%) by using the corresponding coding practices. However, it should be noted that the authors compared the block ciphers with one another instead of the original and optimizated versions.

To sum up, there are opportunities for energy consumption and runtime performance improvements even by applying minor code changes such as passing a variable in a loop's termination condition or by invoking a method statically (Li and Halfond 2014; Tonini et al. 2013). Moreover, the proper API selection that demands fewer system resources can also reduce energy consumption, according to Linares-Vásquez et al. Additionally, Procaccianti et al. improved energy consumption and runtime performance by reducing the use of expensive database operations (when these were not mandatory) and putting applications to sleep (when they were not performing any action).

5 VERIFICATION

Here, we discuss a range of tools to measure and test software's energy and power consumption after the development of an application. We divide the collected works into *Benchmarks* and *Monitoring Tools* in Sections 5.1 and 5.2, respectively.

5.1 Benchmarks

Benchmarks are tools consisting of two main components: (i) a profiling tool, responsible for obtaining instructions used from a specific execution, and (ii) a performance benchmark that generates workloads for a system. The above components combined perform energy consumption measurements. Table 10 summarizes collected information in terms of target platform, energy measurements correlation with hardware or software components, optimizations applied, and the related resources.

PowerBench is a scalable test-bed infrastructure that benchmarks and retrieves power consumption traces, in parallel, from various wireless sensor nodes of a cluster (Haratcherev et al. 2008). To do that, *PowerBench* utilizes particular hardware and software components to offer an off-line processing, analysis, debugging, and visualization of the elicited power measurements. The features of such an approach can aid developers detect power consumption fluctuations and anomalies in clusters and complex IOT environments.

Subramaniam and Feng (2012) proposed Green Benchmark (*GBench*), an approach that employs the Load Varying-LINPACK[16] that produces a variety of workloads to evaluate the energy consumption of a system. The authors tested *GBench* with different configurations such as block size, work-loads, number of cores, and memory access rate. As an effect, they detected a correlation between energy consumption and runtime performance for the second level (L2) of cache misses on specific workloads.

Mukhanov et al. (2015) suggested Abstract-Level Energy Accounting (ALEA), a highly accurate (1.4% of error mean) portable tool that retrieves energy measurements from any micro-processor architecture. At its core, ALEA has a fine-grained energy profiling tool that retrieves measurements from basic code blocks. For evaluation, the authors employed well-known benchmark suites such as SPEC 2000,[17] SPEC,[18] OMP,[19] and so on, and analyzed the relation between basic code blocks' energy consumption and cache accesses to identify energy hot-spots. As a result, by using ALEA, the authors achieved 37% of energy savings, on some of the mentioned benchmarks, through different kinds of energy-efficient strategies and adjustments, e.g., concurrency throttling, thread packing.[20] Moreover, they revealed a strong correlation between energy consumption and cache access rate.

AxBench, proposed by Yazdanbakhsh et al. (2017), is a benchmark suite that supports tests in diverse domains such as finance, signal processing, image processing, machine learning, and so on, aiming to evaluate systems' runtime and energy performance by exploiting approximation techniques. Specifically, the approximation techniques supported by *AxBench* consist of `loop perforation` and `neural processing units`. *AxBench* obtains energy measurements of the CPU, GPU, and Axilog hardware (Yazdanbakhsh et al. 2015). *AxBench* offers (1) the feature to test various levels of the computing stack (software and hardware), (2) various test inputs, and (3) application-specific quality metrics, i.e., average relative error for numeric output, miss rate for `boolean` result, and image difference. However, to perform benchmarking, a user has to identify and manually annotate regions of code that can tolerate imprecision.

The major difference between *GBench* and ALEA is that ALEA correlates energy measurements with basic code blocks, while *GBench* maps the energy consumption of the entire application to hardware components. *AxBench* is equipped with benchmarks for CPU and GPU aiming to offer a fine-grained understanding of their energy and runtime performance implications. In contrast to the above benchmarks, *PowerBench* is the only one to evaluate the energy consumption of an IOT-like infrastructure and cluster.

5.2 Monitoring Tools

To derive the energy consumption of a computer system, two approaches currently exist: (1) indirect energy measurements through estimation models or performance counters or (2) direct measurements, through hardware energy analyzers and sensors. In this section, we discuss tools that are using indirect and direct approaches to perform measurements, some of the tools we present, analyze running applications or system calls to estimate energy consumption. However, compared to the source code analysis tools presented in Section 4.3, the energy monitoring tools only report the energy consumption of an application without pointing out energy hot-spots or providing hints for improving the spotted deficiencies. Tables 11 and 12 present the discussed software and

[16] https://www.top500.org/project/linpack/.
[17] http://www.spec2000.com/.
[18] https://www.spec.org/.
[19] https://www.spec.org/omp2012/.
[20] Collecting threads under a specific number of cores.

Table 11. Software-based Monitoring Tools Related-Work Information

Tool's Name	Target Platform	Measurement Type	Sampling Rate (msec)	Median Error Rate	Source
Jalen	Linux	Energy	500	–	(Noureddine et al. 2012a)
PowerAPI	Linux	Energy	500	0.5–3	(Bourdon et al. 2012)
jRAPL	Linux	Energy	1	1.13	(Liu et al. 2015)
Jolinar	Linux	Energy	500	3	(Noureddine and Rajan 2015)
RAPL	Linux	Energy	1	3	(David et al. 2010)
SoCWatch	Windows, Linux, & Android	Power	1–1000	–	(Pantels 2015)
PowerGadget	Windows & Linux	Power	1–1000	–	(Pantels et al. 2014)
JouleMeter	VMs & Windows	Power	1000	5	(Liu et al. 2010)
VMeter	VMs	Power	–	6	(Bohra and Chaudhary 2010)
BitWatts	VMs	Energy	500	2	(Colmant et al. 2015)
PowerBooter	Android	Power	–	0.8	(Zhang et al. 2010)
GreenOracle	Android	Energy	–	10	(Chowdhury and Hindle 2016)
AEP	Android	Power	–	–	(Chen and Zong 2016)
PETrA	Android	Energy	1000	0.04	(Di Nucci et al. 2017)

Table 12. Hardware-based Monitoring Tools Related-Work Information

Test-bed's Name	Target Platform	Measurement Type	Sampling Rate (in sec.)	Median Error Rate	Source
Atom LEAP	Linux	Energy	1	–	(Peterson et al. 2011)
SEFLab	Windows	Power, Energy	1	1%	(Ferreira et al. 2013)
GreenMiner	Android	Power	1	insignificant	(Hindle et al. 2014)

hardware energy monitoring tools. The tables depict a variety of information such as tool names, target platform, measurement types, sampling intervals, and median error rates.

5.2.1 Software Energy Monitoring Tools. We use the term "software energy monitoring tools" for software-based analyzers that utilize performance counters or estimation models to measure the energy consumption of running applications. We analyze the monitoring tools concerning their features, limitations, architecture (energy estimation model), and supported OS. Moreover, we further classify the software energy monitoring tools according to the platform they target into three categories: (1) workstations and servers, (2) VMs, and (3) smart-phones.

Workstations and Servers. Running Average Power Limit (RAPL) monitors and controls energy consumption via performance counters (Pandruvada 2014). By utilizing the Linux kernel pseudo file system (`sysfs`), RAPL exposes kernel subsystems, hardware devices, and device driver information from the kernel to userspace allowing the estimation of the software's energy consumption. Liu et al. (2015) introduced jRAPL,[21] a framework that combines RAPL and the Java Native Interface

[21] https://github.com/kliu20/jRAPL/.

to measure the energy consumption of CPU, RAM, and Package[22] components for a Java application. Apart from jRAPL, Pantels et al. (2014) and Pantels (2015) introduced the tools PowerGadget and SoCWatch, respectively. Both tools are using RAPL to elicit power consumption from the CPU's performance counters. PowerGadget retrieves package power metrics exposed by the CPU and GPU and can be integrated within a user's application through a C++ API. SoCWatch provides power-related information for the CPU's and GPU's `C-` and `P-state` residencies. All the discussed RAPL-based tools have a high sampling rate and can retrieve a substantial number of samples per millisecond as shown in Table 11. However, tools incorporating RAPL work under specific hardware limitations such as particular microprocessor architecture (Pandruvada 2014). For instance, PowerGadget is compatible only with Intel's second generation CPUs, and it is not yet supported for architectures such as *Skylake, Broadwell,* and *Haswell.*

To estimate application energy consumption, Noureddine et al. (2012a), Bourdon et al. (2012), and Noureddine and Rajan (2015), proposed a number of tools. Specifically, Noureddine et al. proposed Jalen,[23] a Java agent attached to an application, that gathers energy measurements after the initialization of the JVM. Jalen measures selected methods and classes. Additionally, it provides measurements on explicit hardware components (e.g., CPU and HDD) and estimates the energy consumption of software by analyzing executed Java instructions. Jolinar[24] is a Java-based tool for monitoring energy consumption at the process level. According to its developers, Noureddine and Rajan, the tool measures the energy consumption of specific hardware components such as CPU, RAM, and HDD. However, both Jalen and Jolinar use an old Intel's energy module that is not supported by the Linux kernel versions 3.10 and above unless Intel `p_state` is disabled.[25]

A coarse-grained tool for monitoring process-level energy consumption is PowerAPI.[26] PowerAPI is a Scala-based middleware that implements an API for monitoring applications at real time (Bourdon et al. 2012). It estimates the energy consumption of various hardware components (CPU, RAM, HDD, etc.). Noureddine et al. (2012b) evaluated its accuracy against the powerspy2 power analyzer and showed a low median error rate (i.e., 0.5–3%). A weakness of PowerAPI is that it expects time duration from a user to collect energy measurements. Likely worst-case scenarios here are (1) over-collection of measurements (when an application elapses and the tool still measures the idle time) and (2) the incomplete collection of measurements (when an application is still running but the tool's given duration time is too short).

A noteworthy fact is that both Jalen and Jolinar have PowerAPI as their core component for extracting energy measurements. However, PowerAPI exposes energy measurements of an application at the system process-level, whereas Jalen and Jolinar correlate the collected energy-related information with Java applications. Compared to the tools proposed by Pandruvada, Liu et al., Pantels et al., and Pantels, the tools above are estimating energy and power consumption through instrumentation profiling, while RAPL utilizes performance counters. Moreover, as shown in this section, most of the tools lack interoperability which can be a hardware or software limitation.

Virtual Machines. Joulemeter, a tool introduced by Liu et al. (2010), fetches energy measurements from VMs, servers, desktops, laptops, and specific applications by tracking resource usage from various hardware components such as CPU, RAM, HDD, and screen. Its model estimates energy consumption by utilizing the VM's resource tracing, which in turn obtains information through the performance counters. Joulemeter is not limited to energy consumption measurements;

[22] the core (all CPU cores) and the un-core (GPU, LLC, etc.).
[23] https://github.com/adelnoureddine/jalen.
[24] https://github.com/adelnoureddine/jolinar.
[25] https://github.com /stefanos1316/Proof_for_Survey/blob /master/Emails%20from%20Nourredine%20Adel.txt.
[26] https://github.com/Spirals-Team/powerapi.

additionally, it offers the feature of per-VM power capping, sleep, and remote wake-up control management procedures that significantly lessen the energy consumption of a server. However, Joulemeter is not supported by an OS newer than Windows 7.[27]

Bohra and Chaudhary (2010) presented VMeter, a VM power modeling method for estimating energy consumption of various components such as CPU, cache, DRAM, and HDD. The tool monitors regularly the system's resource usage and calculates energy consumption by employing a power model, which has minimal overhead on the system's total energy consumption (i.e., 0.012%). To extract resource usage information from a VM, VMeter utilizes the performance counters and a disk monitoring tool.

BitWatts[28] is a middleware solution that calculates the energy consumption of an application inside a VM via an energy estimation model (Colmant et al. 2015). It is an extension of the PowerAPI toolkit (Bourdon et al. 2012) and is designed to collect energy measurements from modern and complex microprocessors that support multi-cores, hyper-threading, DVFS, and dynamic overclocking. A strong feature of BitWatts is the collection and aggregation of energy measurements from multiple processes that are located in a distributed environment.

BitWatts energy model was compared viz-a-viz PowerSpy[29] Bluetooth energy meter and RAPL. As an outcome, BitWatts measurements were in the scale of 2% median error rate, which is the lowest against VMeter and Joulemeter. Also, compared to VMeter and Joulemeter, BitWatts can be used to analyze the energy consumption of more complex environments such as IoT infrastructure and data centers.

Smart-Phones. Measuring the energy consumption of mobile devices is done in various ways. Zhang et al. (2010) introduced PowerBooter, an on-line energy estimation model, which calculates energy consumption by combining measurements from battery's voltage sensors and discharge rate. Thus, PowerBooter estimates energy consumption without utilizing external hardware power meter. By combining PowerBooter and PowerTutor,[30] the authors obtained energy measurements based on the activity of various hardware components such as CPU, LCD/OLED, GPS, Wi-Fi, and cellular network components.

Similarly, Chen and Zong (2016) developed the Android Energy Profiler (AEP), a tool that correlates process resource usage activities with voltage and current information. The voltage and current information is generated through a smart-phone's built-in voltage sensor and is used by AEP to estimate energy consumption. AEP is not limited to energy consumption measurements; it can also provide runtime performance results. However, in contrast to PowerTutor, AEP provides energy measurements only for the CPU and main memory. A pitfall of AEP is a runtime performance degradation of around 25% that incurs from the data collection process.

Another tool to obtain energy measurements from smart-phones is GreenOracle (Chowdhury and Hindle 2016). GreenOralce is an energy estimation model that after being trained with a variety of Android applications, it can obtain energy measurements for any mobile application. The introduced model, once trained, is usable by application developers to retrieve energy measurements without the need of an external instrumentation tool. To estimate energy consumption, GreenOracle uses dynamic tracing on system calls (i.e., strace) and CPU utilization.

Profiling Energy Tool for Android (PETrA) is a software-based energy profiling tool (Di Nucci et al. 2017). A feature of the tool is the fine-grained energy measurements that are obtained at the method level. Additionally, compared to prior work, PETrA does not require calibration. PETrA's

[27] https://social.microsoft.com/Forums/en-US/home?forum=joulemeter.
[28] https://github.com/mcolmant/powerapi/tree/BitWatts.
[29] http://www.alciom.com/en/products/ powerspy2-en-gb-2.html.
[30] https://github.com/msg555/PowerTutor.

precision was validated on 54 Android applications and compared against a hardware-based energy consumption toolkit, the Monsoon. The outcome shows that PETrA's energy measurements deviate little from Monsoon's (see Table 11).

In contrast to all the preceding tools, PowerBooter offers energy measurements for a large number of smart-phone components. However, in contrast to PowerBooter and PETrA, GreenOralce suffers from high median error rate (see Table 11). Compare to the above, GreenOracle has a sophisticated energy estimation model that, once trained, offers energy measurements for various Android applications.

5.2.2 Hardware Energy Monitoring Tools. Hardware energy analyzers or sensors can also retrieve energy consumption measurements from a computer system. However, the disaggregation of the coarse-grained energy measurements into software or hardware components is a demanding task. The hardware energy monitoring tools are typically no-stand-alone tools that require additional hardware components such as an external device to obtain power or energy measurements. In this section, we describe some hardware energy monitoring tools for workstations, servers, and smart-phones.

Workstation and Server Monitoring Tools. Software Energy Footprint Lab (SEFLab) aims to capture the energy consumption measurements of a computer system and map them to its hardware components (i.e., CPU, RAM, HDD) (Ferreira et al. 2013). Atom Low-Energy Aware Platform (LEAP) is a test-bed that is capable of measuring the energy consumption of small code segments in the kernel and userspace (Peterson et al. 2011). Both tools make use of the Data AcQuisition device (DAQ), an external energy profiling tool that obtains coarse-grained measurements from a computer system. After obtaining the energy consumption of an application, both tools are trying to correlate the retrieve measurements with time-stamps and system resource usage. Moreover, SEFLab uses Joulemeter (Liu et al. 2010) to compare its accuracy against the DAQ energy profiler. The major distinction between the two approaches is that Atom LEAP tries to map the collected energy consumption of an application with software components, while SEFLab maps them with various hardware components. However, to utilize one of the above approaches, a number of tools and set up is required.

Smart-phone Monitoring Tools. GreenMiner is an experimental platform introduced by Hindle et al. (2014), which retrieves energy consumption of a smart-phone through scheduled tests. All the tests are scheduled by a web service, while a Raspberry Pi[31] is responsible for launching test scripts on a smart-phone by using the Android Debug Bridge interface. Once the tests are launched, an INA219[32] chip is used to record the smart-phone's current and voltage measurements. Afterward, an Arduino[33] device is responsible for fetching, calculating, and storing all power measurements from the INA219 chip. Subsequently, the Raspberry Pi collects the energy consumption measurements and meta-data from the Arduino device and forward them to a server where a web service aggregates, analyzes, and stores the experiment's data. Additionally, *GreenMiner* acts as a continuous integration tool for running tests and comparing the energy consumption of different repositories. Through data aggregation and analysis, it provides insights that can guide developers in determining energy consumption change through the evolution of source code at particular product versions. Similarly to the workstation and server monitoring tools presented in the previous paragraph, *GreenMiner* also requires a number of tools and a set up to estimate applications energy consumption.

[31] https://www.raspberrypi.org/.
[32] http://www.ti.com/product/INA219.
[33] https://www.arduino.cc/.

Table 13. Refactoring Energy-Performance Impact

Refactoring Techniques	Implications		Test Cases	Source
	Energy (%)	Performance		
Dead Local Variable, Non Short Circuit,[a] Parameter By Value, Repeated Conditionals, Self Assignment Variable	< 1 avg.	Energy Code Smells are not affecting execution time	Authors' Apps[e]	(Morisio et al. 2013)
Convert Local Variable to Field,[b] Extract Local Variable,[c] Extract Method, Introduce Indirection,[d] In line Method,[g] Introduce Parameter Object[h]	(−7.5)–4.54	There is no rough correlation between runt time performance and energy consumption for JVM 6 and 7	Selected Apps[f]	(Sahin et al. 2014)
Replace Method with Method Object and Encapsulate Collection	−7.91–6.99	–	M.Fowler's Code Samples[i]	(Park et al. 2014)
Loop Unrolling, Loop Unswitching, Method Inline	6.4–50.21	Observed boost on runtime performance	Cocos2d game engine	(Li and Gallagher 2016)

[a] Using "&" and "||" cause all statements evaluation, in contrast to "&&" and "|| ||".
[b] Refactoring local variables to public class fields.
[c] Occurrences of the same expression can be replaced from a variable.
[d] Redirecting all the method invocations to a newly created static method.
[e] Use of mirco-benchmarks.
[f] Commons-{Beanutils, CLI, Collections, IO, Lang, Math}, Joda-Convert, Joda-Time, Sudoku.
[g] In place of the method's invocation its source code is added instead.
[h] Created a new class at the top level (super class).
[i] Code Samples by Fowler et al. (1999).

6 MAINTENANCE

Maintenance is the process of enhancing or fixing errors in a software after its deployment. For the maintenance phase, we discuss techniques and tools for refactoring source code aiming to demonstrate energy savings. In the context of SDLC for energy efficiency, refactoring is the practice that aims to optimize energy consumption of applications through code-level modifications without altering the underlying code structure. In this line, this is also used for source code analysis during software development to detect energy hot-spots or bugs (as discussed in Section 4.3). In this section, we present refactoring techniques applied after the deployment phase (i.e., during the maintenance phase).

6.1 Empirical Evaluation of Code Refactoring

Here, we consider empirical evaluations of refactoring techniques and patterns for energy-performance optimization that software practitioners may employ to reduce applications energy consumption. Table 13 depicts some of the authors' collected results[34] concerning the relevant refactoring method's energy and runtime performance implications, and their test cases.

Fowler et al. (1999) introduced the concept of code refactoring to improve understandability, maintainability, and extensibility of existing source code. To this end, Sahin et al. (2014) and Park

[34] Concerning consistency for the research of Park et al. (2014), we added only the results of the highest and lowest energy consumption, since they investigated over 63 refactoring patterns.

et al. (2014) used the described patterns to perform empirical studies that examine how certain code refactoring patterns affect an application's energy consumption. Both authors used distinct embedded systems, parameters, and a number of smells for their empirical studies. Specifically, Sahin et al. evaluated six widely known refactoring techniques on two Jave Virtual Machine (JVM) versions (i.e., 6 and 7) over nine applications. Park et al. evaluated 63 of the 68 refactoring techniques over code samples proposed by Fowler et al.; Sahin et al. concluded that every refactoring method may increase or decrease the application's energy usage; apart from *Extend Local Variables* which always reduces energy consumption. Similarly, Park et al. found that some of the refactoring code smells may alter positively or negatively the energy consumption. Particularly, from the obtained results Park et al. illustrated that 33 of the refactoring techniques lead to energy savings while the remaining 30 do not. Besides, the authors shared that the energy consumption between Java versions, in the context of the refactoring techniques, is not consistent.

Energy Code Smells is a term stated by Morisio et al. (2013) relating to inefficient implementation choices that increase energy consumption. Through their experiential study, Morisio et al. aimed to determine a number of *code smells* (Fowler et al. 1999) that can alter the energy, runtime performance, or even both of them. Hence, they performed their experiment on existing *code smells* found in CppCheck[35] and FindBugs[36] tools. To minimize the software noise from threads that may run in parallel and subsequently affect energy measurements, the authors performed their experiment on embedded system (Waspmote V1.1).[37] By testing nine different refactoring patterns, Morisio et al. inferred that only five of those reduce the energy consumption by less than 1%, on average, (see Table 13) while the remaining increase the energy consumption or leave it intact. Nevertheless, the authors claim that no correlation exists between *Energy Code Smells* and *Performance Smells* apart from a single case (i.e., *Mutual Exclusion* OR).

The results depicted in Table 13 show that Morisio et al., Sahin et al., and Park et al. elicited both gains and losses in energy consumption such as 1%, −7% to 5%, and −8% to 7, respectively. We can observe that the results of Sahin et al. and Park et al. used quite similar refactoring patterns, do not differ much even though they used different languages to develop them (i.e., Java and C++, respectively). Morisio et al. also employed the refactoring patterns introduced by Fowler et al. (1999); however, they demonstrated only minor energy savings (less than 1%). Moreover, Morisio et al. showed that the *Energy Code Smells* are not affecting *Performance Smells* or vice versa. The above results suggest that the energy savings, achieved by refactoring techniques, are very low. This fact highlights that the applied refactoring techniques are mainly focused on improving code maintainability, extensibility, and understandability. Thus, they cannot offer any substantial gains regarding energy efficiency.

6.2 Tools for Refactoring

Work in this area aims provide maintenance by reducing energy consumption through code refactoring patterns embedded in tools.

An energy optimization framework for Android applications introduced by Li and Gallagher (2016) which focuses on lending energy optimizations through a set of refactoring strategies on the deployed source code. The proposed framework takes source code as an input and analyzes it to retrieve energy-related data to correlate them with basic code blocks. Afterward, the framework spots energy hot-spots of the source code and applies, either autonomously or through manual involvement, refactoring strategies (e.g., *Loop Unrolling, Loop Unswitching, Method In-line*).

[35] https://sourceforge.net/p/cppcheck/wiki/Home/.
[36] http://findbugs.sourceforge.net/.
[37] http://www.libelium.com/development-v11/.

Employing such a refactoring tool can help a user to modify existing or legacy systems source code, so as to reduce their energy consumption and increase runtime performance. The authors evaluated their tool on real-world applications and obtained energy savings ranging from 6% to 50%.

7 CONCLUSIONS

Overall, our analysis reveals that techniques and tools to provide energy efficiency exist for each phase of the SDLC. However, for software practitioners to adopt and use existing tools and techniques, interoperability, usability, and adequate support are crucial factors. To this end, we point out a number of possible research challenges that we identified from our study. In the following paragraphs, we analyze each of these research challenges and we provide future research directions.

RC1. *Selection of configurations and parameters.*

Parallel programming is proven beneficial regarding energy consumption by applying the appropriate configurations and parameters such as the number of threads, data size, and data locality. As expressed by Kambadur and Kim, it is possible to gain energy savings of 55% and runtime performance of 69% if application configurations and parameters are tuned effectively. Likewise, as shown in Section 4, the utilization of approximate techniques can bring an energy reduction of 10–50% for imprecision tolerant applications. However, an open challenge for both parallel and approximate computing is (i) to locate portions of source code that can be optimized and (ii) choose suitable parameters and configurations to reduce energy consumption. A possible approach could be a source code analyzer that can label possible energy hot-spots and suggest energy optimizations such as the selection of adequate number of threads or identify portions that can benefit from approximate computing.

RC2. *Limited investigation on diverse programming languages.*

The same application developed in distinct programming languages varies concerning energy usage and runtime performance (see Section 4). Programming languages such as C and C++ might be challenging when it comes to memory management safety and reliability; nevertheless, they do pay back the developers with lower energy consumption and, at the same time, better runtime performance. Still, researchers have investigated only a small portion of the available programming languages. Others may benefit applications running on diverse computer systems and domains such as (HPC and IOT).

RC3. *Appropriate data structure selection.*

As presented in Section 4, selecting energy-efficient data structures is crucial, because it can significantly affect the energy dissipation of an application. In some instances, selecting the most efficient data structure in real-world applications, such as Google Gson, Xalan, Tomcat, and Huffman Encoder resulted in energy savings of 38%. However, knowing when to select particular data structures without the need of experimenting is quite an ambitious and challenging task. Existing tools such as SEEDS (Manotas et al. 2014) can suggest data structures for Java applications; but, SEEDS is limited to queue collection interface selection and it is not yet available for public use. A possible research direction here is to examine more data structures types, identify which are more energy efficient for selected cases, and compose this knowledge in the form of a refactoring tool that can suggest the replacement of collection implementations.

RC4. *Interoperability, usability, and precision for tooling support.*

In Section 5, we found that most of the energy monitoring tools are oriented toward specific CPU architectures or OSs making them difficult to use across diverse computing environments.

Additionally, to configure these tools it often takes a great deal of time and effort. For instance, to accurately retrieve energy measurements the user has to be aware of configurations such as CPU voltage at a specific frequency. Another challenging task is software energy monitoring tool evaluation. To do so, many researchers compare energy measurements fetched from hardware power analyzers with the results of their proposed tools. However, it is hard to compare or correlate these measurements, because many hardware power analyzers have low sampling frequency and collect energy measurements in a coarse-grained manner. To this end, accurate and versatile software-based energy monitoring tools are of paramount importance. This will promote a wider take-up from the software developers.

Future work should also consider the evolution of software development processes by designing tools that are appropriate for agile software development. In this line, energy-efficient software development approaches should be tightly integrated with energy monitoring tools that continuously assess real energy consumption at the hardware level and provide feedback to the development process which can be used to fine-tune the source code for energy savings. Such a knowledge is crucial and mandatory to allow software practitioners to pinpoint energy gains in ever more complex computer systems and domains such as cloud environments, data centers, and large IOT infrastructures.

ACKNOWLEDGMENTS

The authors thank Maria Kechagia, Tushar Sharma, and Vasiliki Efstathiou from the Athens University of Economics and Business and Iqbal Ahmend from the Saga University for reviewing and providing their knowledge and suggestions in earlier versions of this article. In addition, the authors thank Spyros Alevertis from AllCanCode Inc. for drawing the figures included in this article.

REFERENCES

S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong. 2014. Program energy efficiency: The impact of language, compiler and implementation choices. In *Proceedings of the 2014 International Green Computing Conference (IGCC'14)*. 1–6. DOI: https://doi.org/10.1109/IGCC.2014.7039169

K. Aggarwal, A. Hindle, and E. Stroulia. 2015. GreenAdvisor: A tool for analyzing the impact of software evolution on energy consumption. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*. 311–320. DOI: https://doi.org/10.1109/ICSM.2015.7332477

G. Agosta, M. Bessi, E. Capra, and C. Francalanci. 2011. Dynamic memoization for energy efficiency in financial applications. In *Proceedings of the 2011 International Green Computing Conference and Workshops*. 1–8. DOI: https://doi.org/10.1109/IGCC.2011.6008559

Anys Bacha and Radu Teodorescu. 2013. Dynamic reduction of voltage margins by leveraging on-chip ECC in itanium II processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 297–307. DOI: https://doi.org/10.1145/2485922.2485948

Anys Bacha and Radu Teodorescu. 2014. Using ECC feedback to guide voltage speculation in low-voltage processors. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Los Alamitos, CA, 306–318. DOI: https://doi.org/10.1109/MICRO.2014.54

Woongki Baek and Trishul M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, NY, 198–209. DOI: https://doi.org/10.1145/1806596.1806620

Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, 588–598. DOI: https://doi.org/10.1145/2635868.2635871

Mohamed Amine Beghoura, Abdelhak Boubetra, and Abdallah Boukerram. 2015. Green software requirements and measurement: Random decision forests-based software energy consumption profiling. *Requir. Eng.* 22, 1 (Jul. 2015), 1–14. DOI: https://doi.org/10.1007/s00766-015-0234-2

Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, and Albert Zomaya. 2010. A taxonomy and survey of energy-efficient data centers and cloud computing systems. In *Advances in Computers*, V. Marvin Zelkowitz (Ed.). Elsevier, 47–111. http://www.sciencedirect.com/science/article/pii/B9780123855121000037

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (Sep. 1999), 720–748. DOI: https://doi.org/10.1145/324133.324234

A. E. Husain Bohra and V. Chaudhary. 2010. VMeter: Power modelling for virtualized clouds. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW'10)*. 1–8. DOI: https://doi.org/10.1109/IPDPSW.2010.5470907

A. Bourdon, A. Noureddine, R. Rouvoy, and L. Seinturier. 2012. PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level. Retrieved June 13, 2016 from http://ercim-news.ercim.eu/en92/special/powerapi-a-software-library-to-monitorthe-energy-consumed-at-the-process-level.

Paolo Bozzelli, Qing Gu, and Patricia Lago. 2013. A Systematic Literature Review on Green Software Metrics. Retrieved from https://pdfs.semanticscholar.org/7f7d/7e7d53febd451e263784b59c1c9038474499.pdf.

Christian Bunse, Zur Schwedenschanze, and Sebastian Stiemer. 2013. On the energy consumption of design patterns. In *Proceedings of the 2nd Workshop EASED@ BUIS Energy Aware Software-Engineering and Development*. Citeseer, 7–8.

Q. Cai, J. González, G. Magklis, P. Chaparro, and A. González. 2011. Thread shuffling: Combining DVFS and thread migration to reduce energy consumptions for multi-core systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'11)*. 379–384. DOI: https://doi.org/10.1109/ISLPED.2011.5993670

Eugenio Capra, Chiara Francalanci, and Sandra A. Slaughter. 2012. Is software "Green"? Application development environments and energy efficiency in open source applications. *Inf. Softw. Technol.* 54, 1 (Jan. 2012), 60–71.

J. Chen and C. Kuo. 2007. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*. 28–38. DOI: https://doi.org/10.1109/RTCSA.2007.37

X. Chen and Z. Zong. 2016. Android app energy efficiency: The impact of language, runtime, compiler, and implementation. In *Proceedings of the 2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud'16), Social Computing and Networking (SocialCom'16), Sustainable Computing and Communications (SustainCom'16) (BDCloud-SocialCom-SustainCom'16)*. 485–492. DOI: https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.77

Y. K. Chen, J. Chhugani, P. Dubey, C. J. Hughes, D. Kim, S. Kumar, V. W. Lee, A. D. Nguyen, and M. Smelyanskiy. 2008. Convergence of recognition, mining, and synthesis workloads and its implications. *Proc. IEEE* 96, 5 (May 2008), 790–807. DOI: https://doi.org/10.1109/JPROC.2008.917729

Shaiful Alam Chowdhury and Abram Hindle. 2016. GreenOracle: Estimating software energy consumption with energy measurement corpora. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR'16)*. ACM, New York, NY, 49–60. DOI: https://doi.org/10.1145/2901739.2901763

Maxime Colmant, Mascha Kurpicz, Pascal Felber, Loïc Huertas, Romain Rouvoy, and Anita Sobe. 2015. Process-level power estimation in VM-based systems. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*. ACM, New York, NY, 14:1–14:14. DOI: https://doi.org/10.1145/2741948.2741971

Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'10)*. ACM, New York, NY, 189–194. DOI: https://doi.org/10.1145/1840845.1840883

Qi Deng and Shaobo Ji. 2015. Organizational green IT adoption: Concept and evidence. *Sustainability* 7, 12 (Dec. 2015), 16737–16755. DOI: https://doi.org/10.3390/su71215843

Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2017. Software-based energy profiling of Android apps: Simple, efficient and reliable? In *Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17)*. 103–114.

K. Eder. 2013. Energy transparency from hardware to software. In *Proceedings of the 2013 3rd Berkeley Symposium on Energy Efficient Electronic Systems (E3S'13)*. 1–2. DOI: https://doi.org/10.1109/E3S.2013.6705855

H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. 2012. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 449–460. DOI: https://doi.org/10.1109/MICRO.2012.48

M. A. Ferreira, E. Hoekstra, B. Merkus, B. Visser, and J. Visser. 2013. Seflab: A lab for measuring software energy footprints. In *Proceedings of the 2013 2nd International Workshop on Green and Sustainable Software (GREENS'13)*. 30–37.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Erol Gelenbe and Yves Caseau. 2015. The impact of information technology on energy consumption and carbon emissions. *Ubiquity* 2015, June (Jun. 2015), 1:1–1:15. DOI: https://doi.org/10.1145/2755977

Johann Grossschadl, Stefan Tillich, Christian Rechberger, Michael Hofmann, and Marcel Medwed. 2007. Energy evaluation of software implementations of block ciphers under memory constraints. In *Proceedings of the 2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 1–6. DOI: https://doi.org/10.1109/DATE.2007.364443

S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. 2013. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 35th International Conference on Software Engineering (ICSE'13)*. 92–101. DOI: https://doi.org/10.1109/ICSE.2013.6606555

I. J. Haratcherev, G. P. Halkes, T. E. V. Parker, O. W. Visser, and K. G. Langendoen. 2008. *PowerBench: A Scalable Testbed Infrastructure for Benchmarking Power Consumption*. 37–44.

Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy profiles of Java collections classes. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, 225–236.

Alexander A. Hernandez and Sherwin E. Ona. 2015. A qualitative study of green IT adoption within the philippines business process outsourcing industry: A multi-theory perspective. *Int. J. Enterp. Inf. Syst.* 11, 4 (Oct. 2015), 28–62. DOI: https://doi.org/10.4018/IJEIS.2015100102

Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. 2014. GreenMiner: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*. ACM, New York, NY, 12–21. DOI: https://doi.org/10.1145/2597073.2597097

Timo Honig, Heiko Janker, Christopher Eibel, Oliver Mihelic, and Rüdiger Kapitza. 2014. Proactive energy-aware programming with PEEK. In *Proceedings of the 2014 International Conference on Timely Results in Operating Systems*. USENIX Association, 6 pages. http://dl.acm.org/citation.cfm?id=2750315.2750321.

Melanie Kambadur and Martha A. Kim. 2014. An experimental survey of energy management across the stack. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*. ACM, New York, NY, 329–344.

Aman Kansal, Feng Zhao, Jie Liu, Nupur Kothari, and Arka A. Bhattacharya. 2010. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 39–50. http://doi.acm.org/10.1145/1807128.1807136

Fanxin Kong and Xue Liu. 2014. A survey on green-energy-aware power management for datacenters. *ACM Comput. Surv.* 47, 2 (Nov. 2014), 30:1–30:38. DOI: https://doi.org/10.1145/2642708

J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi. 2015. Safe limits on voltage reduction efficiency in GPUs: A direct measurement approach. In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 294–307. DOI: https://doi.org/10.1145/2830772.2830811

Ding Li and William G. J. Halfond. 2014. An investigation into energy-saving programming practices for Android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software (GREENS'14)*. ACM, New York, NY, 46–53. DOI: https://doi.org/10.1145/2593743.2593750

Xueliang Li and John P. Gallagher. 2016. A source-level energy optimization framework for mobile applications. In *Proceedings of the 16th International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, Los Alamitos, CA.

Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining energy-greedy API usage patterns in Android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*. ACM, New York, NY, 2–11. DOI: https://doi.org/10.1145/2597073.2597085

Kenan Liu, Gustavo Pinto, and Yu David Liu. 2015. Data-oriented characterization of application-level energy optimization. In *Fundamental Approaches to Software Engineering*, Alexander Egyed and Ina Schaefer (Eds.), Vol. 9033 in Lecture Notes in Computer Science. Springer, Berlin, 316–331. DOI: 10.1007/978-3-662-46675-9_21.

Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. 2016. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, 237–248.

Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 503–514.

Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V. Vasilakos. 2014. Cloud computing: Survey on energy efficiency. *ACM Comput. Surv.* 47, 2 (Dec. 2014), 33:1–33:36. DOI: https://doi.org/10.1145/2656204

Junya Michanan, Rinku Dewri, and Matthew J. Rutherford. 2016. GreenC5: An adaptive, energy-aware collection for green software development. *Sust. Comput. Inf. Syst.* 12 (Nov. 2016), 42–60. DOI: https://doi.org/10.1016/j.suscom.2016.11.004

Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*. ACM, New York, NY, 309–328.

Subrata Mitra, Manish K. Gupta, Sasa Misailovic, and Saurabh Bagchi. 2017. Phase-aware optimization in approximate computing. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO'17)*. IEEE Press, Los Alamitos, CA, 185–196. http://dl.acm.org/citation.cfm?id=3049832.3049853

Sparsh Mittal and Jeffrey S. Vetter. 2015. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.* 47 (2015), 69–69. DOI: https://doi.org/10.1145/2788396

Sparsh Mittal and Jeffrey S. Vetter. 2016. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. Parallel Distrib. Syst.* 27, 5 (May 2016), 1537–1550. DOI: https://doi.org/10.1109/TPDS.2015.2442980

Maurizio Morisio, Luca Ardito, Antonio Vetro', and Giuseppe Procaccianti. 2013. Definition, implementation and validation of energy code smells: An exploratory study on an embedded system. In *Proceedings of the Third International Conference on Smart Grid, Green Communications and IT Energy-aware Technologies (Energy'13)*. 34–39.

Lev Mukhanov, Dimitrios S. Nikolopoulos, and Bronis R. de Supinski. 2015. ALEA: Fine-grain energy profiling with basic block sampling. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT'15)*. IEEE Computer Society, Los Alamitos, CA, 87–98. DOI: https://doi.org/10.1109/PACT.2015.16

S. Murugesan. 2008. Harnessing green IT: Principles and practices. *IT Profess.* 10, 1 (Jan. 2008), 24–33. DOI: https://doi.org/10.1109/MITP.2008.10

Uwe Naumann. 2012. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Adel Noureddine, Aurélien Bourdon, Romain Rouvoy, and Lionel Seinturier. 2012b. A preliminary study of the impact of software engineering on GreenIT. In *Proceedings of the First International Workshop on Green and Sustainable Software*. IEEE Press, 21–27. http://dl.acm.org/citation.cfm?id=2663779.2663783.

A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. 2012a. Runtime monitoring of software energy hotspots. In *Proceedings of the 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*. 160–169. DOI: https://doi.org/10.1145/2351676.2351699

A. Noureddine and A. Rajan. 2015. Optimising energy consumption of design patterns. In *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15)*, Vol. 2. 623–626. DOI: https://doi.org/10.1109/ICSE.2015.208

Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. 2013. A review of energy measurement approaches. *SIGOPS Oper. Syst. Rev.* 47, 3 (Nov. 2013), 42–49. DOI: https://doi.org/10.1145/2553070.2553077

S. Pandruvada. 2014. Running Average Power Limit—RAPL textbar 01.org. Retrieved June 28, 2016 from https://01.org/blogs/tlcounts/2014/running-average-power-limit---rapl.

C. Pang, A. Hindle, B. Adams, and A. Hassan. 2015. What do programmers know about software energy consumption? *IEEE Softw.* 33, 3 (2015), 83–89.

Thomas Pantels. 2015. Optimizing Power for Interactions between Virus Scanners and Pre-bundled Software. Retrieved from https://software.intel.com/en-us/articles/optimizing-power-for-interactions-between-virus-scanners-and-pre-bundled-software.

Thomas Pantels, Sheng Guo, and Rajshree Chabukswar. 2014. Touch Response Measurement, Analysis, and Optimization for Windows* Applications. Retrieved from https://software.intel.com/en-us/articles/touch-response-measurement-analysis-and-optimization-for-windows-applications.

George Papadimitriou, Manolis Kaliorakis, Athanasios Chatzidimitriou, Dimitris Gizopoulos, Peter Lawthers, and Shidhartha Das. 2017. Harnessing voltage margins for energy efficiency in multicore CPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. ACM, New York, NY, 503–516. DOI: https://doi.org/10.1145/3123939.3124537

Jae Jin Park, Jang-Eui Hong, and Sang-Ho Lee. 2014. Investigation for software power consumption of code refactoring techniques. In *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering*, Marek Reformat (Ed.). Knowledge Systems Institute Graduate School, 717–722.

Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. 2012. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, 29–42. DOI: https://doi.org/10.1145/2168836.2168841

Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. 2016. The influence of the Java collection framework on overall energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software (GREENS'16)*. ACM, New York, NY, 15–21. DOI: https://doi.org/10.1145/2896967.2896968

Peter A. H. Peterson, Digvijay Singh, William J. Kaiser, and Peter L. Reiher. 2011. Investigating energy and security trade-offs in the classroom with the atom LEAP testbed. In *Proceedings of the 4th Conference on Cyber Security Experimentation and Test (CSET'11)*. USENIX Association, Berkeley, CA, 11–11.

Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)*. ACM, New York, NY, 345–360. DOI: https://doi.org/10.1145/2660193.2660235

Gustavo Pinto, Kenan Liu, and Fernando Castor. 2016. A comprehensive study on the energy efficiency of Java thread-safe collections. In *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, Los Alamitos, CA.

G. Pinto, F. Soares-Neto, and F. Castor. 2015. Refactoring for energy efficiency: A reflection on the state of the art. In *Proceedings of the 2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software (GREENS'15)*. 29–35. DOI: https://doi.org/10.1109/GREENS.2015.12

Giuseppe Procaccianti, Héctor Fernández, and Patricia Lago. 2016. Empirical evaluation of two best practices for energy-efficient software development. *J. Syst. Softw.* 117, C (July 2016), 185–198. DOI: https://doi.org/10.1016/j.jss.2016.02.035

L. B. Rall. 1984. *The Arithmetic of Differentiation*. Mathematics Research Center, University of Wisconsin—Madison.

M. Rashid, L. Ardito, and M. Torchiano. 2015. Energy consumption analysis of algorithms implementations. In *Proceedings of the 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'15)*. 1–4. DOI: https://doi.org/10.1109/ESEM.2015.7321198

Haris Ribic and Yu David Liu. 2014a. Energy-efficient work-stealing language runtimes. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 513–528.

Haris Ribic and Yu David Liu. 2014b. Energy-efficient work-stealing language runtimes. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 513–528. DOI: https://doi.org/10.1145/2541940.2541971

W. W. Royce. 1987. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering (ICSE'87)*. IEEE Computer Society Press, Los Alamitos, CA, 328–338. http://dl.acm.org/citation.cfm?id=41765.41801.

C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. 2012. Initial explorations on design pattern energy usage. In *Proceedings of the 2012 1st International Workshop on Green and Sustainable Software (GREENS)*. 55–61. DOI: https://doi.org/10.1109/GREENS.2012.6224257

Cagri Sahin, Lori Pollock, and James Clause. 2014. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*. ACM, New York, NY, 36:1–36:10. DOI: https://doi.org/10.1145/2652524.2652538

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, 164–174. DOI: https://doi.org/10.1145/1993498.1993518

Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: The problem based benchmark suite. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12)*. ACM, New York, NY, 68–70. DOI: https://doi.org/10.1145/2312005.2312018

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM, New York, NY, 124–134. DOI: https://doi.org/10.1145/2025113.2025133

Balaji Subramaniam and Wu-chun Feng. 2012. GBench: Benchmarking methodology for evaluating the energy efficiency of supercomputers. *Comput. Sci. Res. De.* 28, 2–3 (May 2012), 221–230. DOI: https://doi.org/10.1007/s00450-012-0218-0.

A. R. Tonini, L. M. Fischer, J. C. B. d Mattos, and L. B. d Brisolara. 2013. Analysis and evaluation of the Android best practices impact on the efficiency of mobile applications. In *Proceedings of the 2013 III Brazilian Symposium on Computing Systems Engineering*. 157–158. DOI: https://doi.org/10.1109/SBESC.2013.39

Ward Van Heddeghem, Sofie Lambert, Bart Lannoo, Didier Colle, Mario Pickavet, and Piet Demeester. 2014. Trends in worldwide ICT electricity consumption from 2007 to 2012. *Comput. Commun.* 50 (Sep. 2014), 64–76. DOI: https://doi.org/10.1016/j.comcom.2014.02.008

Vassilis Vassiliadis, Charalampos Chalios, Konstantinos Parasyris, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2016a. Exploiting significance of computations for energy-constrained approximate computing. *Int. Parallel Program.* 44, 5 (Oct. 2016), 1078–1098. DOI: https://doi.org/10.1007/s10766-016-0409-6

Vassilis Vassiliadis, Jan Riehme, Jens Deussen, Konstantinos Parasyris, Christos D. Antonopoulos, Nikolaos Bellas, Spyros Lalis, and Uwe Naumann. 2016b. Towards automatic significance analysis for approximate computing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO'16)*. ACM, New York, NY, 182–193. DOI: https://doi.org/10.1145/2854038.2854058

A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. 2017. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Des. Test* 34, 2 (Apr. 2017), 60–68. DOI: https://doi.org/10.1109/MDAT.2016.2630270

A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan. 2015. Axilog: Language support for approximate hardware design. In *Proceedings of the 2015 Design, Automation Test in Europe Conference Exhibition (DATE'15)*. 812–817. DOI: https://doi.org/10.7873/DATE.2015.0513

Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. 2010. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS'10)*. ACM, New York, NY, 105–114. DOI: https://doi.org/10.1145/1878961.1878982

Received May 2017; revised March 2019; accepted May 2019