Pós-Graduação em Ciência da Computação

"A Refactoring Approach to Improve Energy Consumption of Parallel Software Systems"

By

Gustavo Henrique Lima Pinto

Ph.D. Thesis

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, 2015

Universidade Federal de Pernambuco
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Gustavo Henrique Lima Pinto

"A REFACTORING APPROACH TO IMPROVE ENERGY CONSUMPTION OF PARALLEL SOFTWARE SYSTEMS"

A Ph.D. Thesis presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science.

ADVISOR: ***Prof Fernando José Castor de Lima Filho***

RECIFE
2015

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

P659r Pinto, Gustavo Henrique Lima
A refactoring approach to improve energy consumption of parallel software systems / Gustavo Henrique Lima Pinto. – Recife: O Autor, 2015.
155 f.: il., fig., tab.

Orientador: Fernando José Castor de Lima Filho.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn, Ciência da computação, 2015.
Inclui referências.

1. Engenharia de software. 2. Evolução de software. 3. Software - Refatoração. I. Lima Filho, Fernando José Castor de (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2015-161

Tese de Doutorado apresentada por **Gustavo Henrique Lima Pinto** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**A Refactoring Approach to Improve Energy Consumption of Parallel Software Systems**" orientada pelo **Prof. Fernando José Castor de Lima Filho** e aprovada pela Banca Examinadora formada pelos professores:

_____
Prof. Paulo Henrique Monteiro Borba
Centro de Informática/UFPE

_____
Prof. Ricardo Massa Ferreira Lima
Centro de Informática / UFPE

_____
Prof. Marcelo Bezerra D'Amorim
Centro de Informática / UFPE

_____
Prof. . Fernando Magno Quintão Pereira
Departamento de Ciência da Computação / UFMG

_____
Prof. Marco Tulio Valente
Departamento de Ciência da Computação / UFMG

Visto e permitida a impressão.
Recife, 24 de fevereiro de 2015.

_____
Profa. Edna Natividade da Silva Barros
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

To Pedro & Joana, João & Julieta, and Francisca.

Acknowledgements

I hesitated for so long
To write such thing
Because in my heart
There were countless people
Who I have to acknowledge

But my poor English
Would not be enough
To write here down
The much that I owe
For *all* of those
Who have shaped me
Into the person that I am

So the unfair way that I found
To not leave this blank page
Is to write what I feel
Without writing a single name

I hope you will forgive me
When you find yourself
In one of the sentences
In the paragraphs that come

First, I would like to say thanks to Him
The Father of creation
Who has different names and prayers
But the relief He brings
Comes from the same place
Thank you for the gift of life, family and friends

I owe a huge debt of gratitude to
Those who not only taught me the very basics
How to talk and walk
How to be polite and respectful

How to love and be loved
But also gave me the freedom
To explore my own path
To make my own decisions
To see the world in my own eyes
Thank you

I am also indebted in
Every respect to
The great family that I happen to have
Constantly teaching me
How to be humble
How to serve others selflessly
And how to deeply care about those around us
Thank you

As I grew up
I have learned that
Friends come in and out of our lives
But fortunate I was
To find the best ones
Who teach me most powerful life lessons
That only the science of friendship can teach
Thank you

From the old friends
You have taught me that
Physical distance is only a theoretical concept
Since our friendship remained tight and solid
Even though we do not see each other
Day after day, year after year
Thank you

From the new ones
I would like to let you know that
Even if we do not know much of each other
I always think about you
How have you been? Are you doing well?
But thanks to Facebook

I have answers for most of my questions
Thank you

In particular
I would like to thank those
Who had the chance of
Introducing me to
Their friends and families
Making me feel like one of them
Thank you

From those that are not here anymore
I would like you to know
That you made a hole in my heart
But you also made me think
That I should do my best while here
Since life is a breath of fresh air
"*Time flies, remember death*"
Thank you

I am also grateful to
The masters that I happen to choose
Who did a paramount job
In educating and empowering me
To overcome any challenge or barrier that I face
Always providing enlightening guidance and advices
Thank you

I am also indebted with those
Who I never met in person
But have a direct impact on my professional life
The reviewers that rejected my papers
The authors of other papers
The funding research agency
SVs, PCs, Chairs, Professors, Directors, Deans
Song writers and movie directors
And many, many, many others. Thank you

Last but not least

I need to thank she
Who left everything to join me
Who tirelessly supported me
Morally, spiritually and lovingly
Telling me repeated times
That I can become everything
I always wanted to be
Thank you

I finally want to apologize to readers
Who may feel that this acknowledgment is too personal
Or unnecessary in a Ph.D. thesis
That should be about research

However, without such acknowledgment
There will be no Ph.D.
There will be no research

Thank you

Try not.
Do.
Or do not.
There is no try.

—YODA (Star Wars: Episode V – The Empire Strikes Back)

Resumo

Fornecer meios para que desenvolvedores de software tomem decisões energéticamente eficientes é uma dimensão crítica para se melhorar o consumo de energia de sistemas computacionais. Apesar do crescente interesse em processos de desenvolvimento de software, arcabouços, e modelos de programação de forma a facilitar o gerenciamento de energia no nível da aplicação, pouco se sabe sobre como arquitetar sistemas concorrentes energéticamente eficientes que rodem em arquiteturas paralelas. Isso é inoportuno por pelo menos duas razões: (1) graças a proliferação de CPUs multicore, programação concorrente se tornou uma prática padrão na engenharia de software moderna; (2) uma CPU com várias unidades de processamento (por exemplo, 32) geralmente dissipa mais potência do que uma com um número menor (por exemplo, 1 ou 2).

No entanto, desenvolvedores ainda não entendem como suas modificações de código impactam no consumo de energia de uma aplicação paralela. Uma análise do StackOverflow mostrou evidências que esse é um problema real; mesmo embora exista um crescente interesse em questões relacionadas ao consumo de energia, desenvolvedores ainda cometem equívocos e mantêm suposições que não são sempre verdadeiras. Essa falta de conhecimento é primariamente devido a falta de ferramentas apropriadas para medir/identificar/refatorar hotspots de consumo de energia. Essa tese então começa a pavimentar o abismo do primeiro problema — a falta de conhecimento — através de uma extensa exploração experimental de dois dos pilares fundamentais da programação concorrente: (1) coleções thread-safe e (2) construções para o gerenciamento de threads. Através de uma lista de achados que não são sempre óbvios, esta tese ilumina o relacionamento entre escolhas de design de código paralelo com seu consumo de energia.

Esta tese começa então a pavimentar a lacuna do segundo problema — a falta de ferramentas. Lições aprendidas em um dos estudos anteriores mostraram que várias tarefas do arcabouço `ForkJoin` operam em estrutura de dados indexáveis, com sub-tarefas operando somente em parte dessa estrutura de dados. Uma solução ingênua é de *copiar* parta da estrutura de dados e utiliza-la na computação sub-sequente. Em um arcabouço recursivo como o `ForkJoin`, dado uma representação baseada em arrays, cada chamada recursiva criará n novos arrays, onde n é a profundidade do fork. Como solução, esta tese apresenta uma refatoração que, ao invés de *copiar* parte da estrutura de dados, ela *compartilha-a*, possibilitando que sub-tarefas operem em partições contíguas da estrutura de dados.

Essa refatoração foi avaliada em 15 projetos de código aberto, a qual foi capaz de economizar energia em todos os casos (média de 12% de economia). A versão refatorada foi enviada aos mantenedores do projeto original e, durante um período de 40 dias, 7 dos 9 mantenedores que responderam aos patches enviados já haviam aceitado-os e integrado-os. Discussões durante o processo de integração revelaram que desenvolvedores não estão cientes desta otimização. Esta tese então implementou essa refatoração como um plug-in da IDE Eclipse

de forma que outros desenvolvedores possam (1) detectar usos de cópia em cenários o quais seriam beneficiais o uso do modelo de compartilhamento and (2) refatorar o código de forma automática.

Palavras-chave: Eficiência Energética, programação concorrente, refatoração

Abstract

Empowering application programmers to make energy-aware decisions is a critical dimension in improving energy efficiency of computer systems. Despite the growing interest in designing software development processes, frameworks, and programming models to facilitate application-level energy management, little is known on how to design application-level energy-efficient solutions for concurrent software running on parallel architectures. This is unfortunate for at least two reasons: (1) thanks to the proliferation of multicore CPUs, concurrent programming is a standard practice in modern software engineering; (2) a CPU with more cores (say 32) often consumes more power than one with fewer cores (say 1 or 2).

However, application developers still do not understand how their code modifications impact energy consumption in a parallel system. Analyzing StackOverflow showed evidence that this is a real problem; Even though the interest in energy consumption issues is increasing over the years, developers still hold misconceptions and assumptions that are not always true. This lack of knowledge is primarily due to a lack of appropriate tools to measure/identify/refactor energy consumption hotspots. This thesis begins to bridge the chasm of the first problem — the lack of knowledge — by presenting an extensive experimental space exploration over two concurrent programming building blocks: (1) thread-safe collections and (2) thread management constructs. Through a list of findings that are not always obvious, we illuminate the relationship between the choices and settings of design decisions and energy consumption of parallel systems.

This thesis then starts to bridge the gap of the second problem — the lack of tools. Lessons learned in our previous studies showed that `ForkJoin` tasks often operate on an indexable data structure, with subtasks operating only on part of this data structure. One naive solution is to *copy* part of the data structure and use it in the next computation. In a recursive framework such as `ForkJoin`, given an array-based representation, each recursive call will create *n* new arrays, where *n* is the width of forking. To address this, we derive a refactoring that, instead of *copy* part of the data structure, it *shares* it, allowing subtasks to operate on contiguous partitions of the data structure.

We manually applied this refactoring into 15 open source projects. Our refactoring succeed in saving energy for each one of them (12% average saving). We sent the refactored versions to the project owner and, during a timeframe of 40 days, 7 out of 9 projects that replied to our patches have already accepted and merged them. Discussions during the merge process revealed that developers were not aware of this optimization. We then implemented this refactoring as an Eclipse plug-in so that other developers can (1) detect uses of copy where it would be beneficial to use sharing and (2) refactor the code in an automated way.

Keywords: Energy efficiency, concurrent programming, refactoring

List of Figures

List of Tables

Contents

1

Introduction

Um passo a frente e você não está mais no mesmo lugar
One step forward and you are not in the same place
—CHICO SCIENCE (Um Passeio No Mundo Livre, Afrociberdelia)

Modern computing platforms are experiencing an unprecedented diversification. Beneath the popularity of the Internet of Things, Android phones, Apple iWatch and Unmanned Aerial Vehicles, a critical looming concern is *energy consumption*. This concern pertains not only for unwired devices as data centers have limited scalability as they struggle with soaring energy costs. Since large companies rely on reliable and fast computing services, cooling such hardware turned out to be a new concern for them.

This issue, if not properly addressed, not only can impact negatively on revenue, but can also emit dozens of thousands of tons of carbon dioxide in the atmosphere. As a consequence, large companies are struggling to find an optimal solution for this case. Google, on the one hand, claims to save millions of dollars per year by following a set of recommendations on how to improve energy efficiency in data centers[1]. Facebook, on the other hand, is planing a more aggressive approach: move part of its data center to just 100km south of the Arctic Circle, because of "*its access to renewable energy and the cold climate that is crucial for keeping the servers cool*"[2]. Since not all companies can afford such investments, a better way to decrease energy costs is by promoting energy efficient systems.

For many years, research that connects computing and energy efficiency has concentrated on the hardware layer. One of the reasons is due to, for instance, while a lightbulb can consume 40 watts, a CPU with interconnects and DRAM can consume 60 watts (see Chapter 4). However, there are studies motivated by the assumption that only hardware dissipates power, not software. However, recent studies have showed strong evidences that this assumption does not capture the whole picture (BELLOSA, 2000; LI; TRAN; HALFOND, 2014). That would be analogous

[1] http://www.google.com/about/datacenters/efficiency/external/
[2] http://www.theguardian.com/environment/2011/dec/15/facebook-coal-clean-power-energy-greenpeace

to postulating that only automobiles are responsible for burning gasoline, not the people who drive them and the way they are used. In any computer system, it is software that directs much of the activity of the hardware. Consequently, software can have a substantial impact on power consumption.

In spite of advances in many areas, IT energy consumption keeps rising steeply (ASAFU-ADJAYE, 2000), which indicates that rising demand is outpacing efficiency improvement. This is not surprising, given that software engineering has, in the past decades, focused on developer efficiency, not on minimization of resource consumption. Software solutions for improving energy efficiency of computer systems can work at different levels, ranging from machine code level to end-user applications. But, generally speaking, concerns about energy usage were left for low-level developers. And they made it right. Energy efficient solutions on hardware/architecture (HOROWITZ; INDERMAUR; GONZALEZ, 1994; TIWARI; MALIK; WOLFE, 1994; IYER; MARCULESCU, 2002; ISCI et al., 2006; KUMAR et al., 2003; SOLERNOU et al., 2013), operating systems (GE et al., 2007; YUAN; NAHRSTEDT, 2003; MERKEL; BELLOSA, 2006; RANGAN; WEI; BROOKS, 2009), and runtime systems (VIJAYKRISHNAN et al., 2001; FARKAS et al., 2000; RIBIC; LIU, 2014) are more established.

While the strategy of leaving the energy consumption optimization problem to the lower-level layers has been successful, recent work showed that even better results can be achieved by empowering and encouraging software developers to participate in the process (KWON; TILEVICH, 2013; LI; TRAN; HALFOND, 2014; CARBIN et al., 2012). As a result, in recent years, a number of solutions from higher levels of the computer stack — such as program analysis (HAO et al., 2013; BARTENSTEIN; LIU, 2013), programming models (SORBER et al., 2007; BAEK; CHILIMBI, 2010; SAMPSON et al., 2011; COHEN et al., 2012; KANSAL et al., 2013), and applications (PINTO; CASTOR; LIU, 2014a; ZHANG et al., 2012) — were proposed.

These application-level energy management strategies complement lower-level strategies with an expanded optimization space, yielding distinctive advantages: first, applications are viewed as a white box, whose structural features may be considered for energy optimization; second, the knowledge of programmers and their design choices can influence energy efficiency; third, application programmers can take the usage context into account to make more aggressive optimizations whereas low level ones are inherently conservative.

Concomitantly, in the last decade, the impact of multicore architectures has clearly been felt by computer users. Multicore systems offer the potential for cheap, scalable, high-performance computing. To achieve this potential, it is essential to take advantage of new heterogeneous architectures comprising collections of multiple processing elements. To leverage multicore technology, applications must be concurrent, which poses a challenge, since it is well-known that concurrent programming is hard (SUTTER, 2005). However, both academia and industry believe that mutli-core technology will remain prevalent for the years to come (ES-MAEILZADEH et al., 2013).

In order to adapt to the new hardware, the software community has been very active in developing novel software techniques to address a wide range of properties of multithreaded programs, such as correctness, programmability, and performance. However, despite their promise, few language-level or application-level energy-efficient solutions address concurrent software running on parallel architectures (BARTENSTEIN; LIU, 2013; GAUTHAM et al., 2012; TREFETHEN; THIYAGALINGAM, 2013; RIBIC; LIU, 2014). This is unfortunate for at least two reasons: (1) thanks to the proliferation of multicore CPUs, concurrent programming is a standard practice in modern software engineering (TORRES et al., 2011); (2) a CPU with more cores (say 32) often consumes more power than one with fewer cores (say 1 or 2). Energy optimization over programs on such platforms has the potential to yield larger savings, but may also face more challenges (IYER; MARCULESCU, 2002; ISCI et al., 2006; PINTO; CASTOR; LIU, 2014b).

Ultimately, the energy consumption of a multithreaded program is not an easy task to reason about. For instance, if a multi-threaded program receives a 2x speed-up but, at the same time, yields a fivefold increase in power consumption (as compared with a single core execution), energy consumption — the product of power consumption times execution time — and thus energy efficiency — the amount of work that can be achieved by consuming a certain amount of energy — degrades as the user embraces multi-core CPUs. This challenge, if not addressed properly, may have a severe negative impact on the future of multicore technology, since one of the main reasons for the popularization of multi-core architectures was the high-energy consumption of high-end single core processors.

In this thesis, we believe that educating and empowering software developers with useful tools can play a prominent role in reducing the energy consumption of the applications they write. On large-scale long-running applications deployed on computing clusters, even a small drop in power consumption can implicate in large savings. To achieve this goal, it is important to get a better understanding of high-level design and implementation choices and the associated implications for software energy consumption. However, understanding and redesigning an application to consume less energy is easier said than done.

1.1 The Problem

Developing an energy efficient concurrent system is a daunting task. One of the most important problems in this task is to understand where energy is being consumed and, then, understand how the code can be changed to reduce the energy consumed. The software development process is usually supported by a great diverse set of tools, and programmers are used to rely on them. For measure and analyze software energy consumption, it would not be different. Energy consumption estimation tools do exist (LIU; PINTO; LIU, 2015; HAO et al., 2013; LI et al., 2013; SEO; MALEK; MEDVIDOVIC, 2008a), but they do not fully support this activity, due to at least three reasons:

1. They require an in-depth knowledge of low-level implementation details and programmers under time pressure have little change to learn how to use them;

2. They do not provide direct guidance on energy optimization, i.e., bridging the gap between understanding where energy is consumed and understanding how the code can be modified in order to reduce energy consumption;

3. They do not take into consideration the impact of parallel programming techniques into the code.

Without usable and useful tools, software developers can only rely on their conventional wisdom, or search for energy saving best practices in software development forums and blogs. Unfortunately, many of these guidelines are not supported by empirical evidence or, at the worst case, can be even incorrect (PINTO; CASTOR; LIU, 2014b).

In this thesis we tackle two important problems found in the development of concurrent applications that run on parallel architectures with focus on software energy consumption optimization. The first one is the *lack of knowledge*. Nowadays, developers do not fully understand how their code modification can impact in energy consumption in a parallel software system. Moreover, since there are several threading implementations available in a mainstream programming language such as Java, practitioners should carefully choose which one implementation is more appropriated to which scenario. These implementations differ in several ways, such as performance (GU; LEE; CAI, 1999) and programmer effort, satisfaction and error-proneness (PANKRATIUS; SCHMIDT; GARRETON, 2012), but their trade-off are reasonably well-understood. However, little is known about their energy efficiency.

This lack of knowledge is primarily due to our second problem: the *lack of tools*. Developers are in need of appropriated tools to measure/identify/refactor energy consumption hotspots. Even though there are some preexisting tools that can be used to fill part of this gap, most of them falls in one of the following categories: (1) they do not provide direct guidance, (2) they are platform dependent, (3) they are difficult to use or (4) they do not take parallel programming constructs into consideration. This fact indicates an opportunity to build a better, simpler, more applicable tools.

1.2 The Goal

The goal of this study is to mitigate these two problems aforementioned: the lack of knowledge, and the lack of tools for developing parallel energy efficient applications. To achieve this goal, this thesis investigates the following key research questions regarding software energy consumption:

- **RQ1.** What are the common problems faced by developers building energy-efficient applications?

- **RQ2.** Do different thread-safe collections have different impacts on energy consumption?
- **RQ3.** Do different thread management techniques have different impacts on energy consumption?
- **RQ4.** Can refactoring play a role in reducing the energy consumption of a parallel system?

To answer these questions, we conducted several investigations. To gain an understanding of this problem and answer **RQ1**, we first quantitatively and qualitatively analyzed STACKOVERFLOW, one of the most used Q&A website in the software development world, in order to observe how practitioners are dealing with energy consumption issues. Second, we answer **RQ2** by investigating the energy consumption characteristics of 16 Java collection implementations grouped by 3 well-known interfaces: `List`, `Set`, and `Map`. Third, in order to answer **RQ3**, we conducted an empirical study aiming to illuminate the relationship between the choices and settings of three thread management constructs, the `Thread`, `Executors` and `ForkJoin` styles, and energy consumption. We then conducted an extensive experimental space exploration over both micro-benchmarks and real-world Java programs, which were manually modified to use three important thread management constructs in concurrent programming. Ultimately, we provide answers for **RQ4** by presenting a refactoring approach that can be used to mitigate one common misuse present in `ForkJoin` data-parallel computations. With such tool, we mitigate the burden faced by developers when writing energy efficient parallel programs.

1.3 The Contributions

This thesis makes several contributions. Some contributions bring significant improvements over the state-of-the-art tools for improving software energy consumption, while others break new ground in areas never before explored.

- **Insights about what developers think about software energy consumption.** To the best of our knowledge, ours is the first quantitative and qualitative study that offers insights into what developers *think* about software energy consumption.
- **The categorization of the most common energy problems, causes and solutions.** Following a qualitative research approach, we categorize 5 common problems, 7 possible causes and 8 possible solutions for those problems. We also compare if the solutions suggested by practitioners are supported by researchers.
- **An understanding of software energy consumption in thread-safe collections.** We present an empirical study evaluating the performance and energy consumption

characteristics of 16 Java collection implementations grouped by 3 well-known interfaces: `List`, `Set`, and `Map`.

- **An understanding of software energy consumption in parallel systems.** We conduct an extensive experimental space exploration illuminating the relationship between the choices and settings of thread management constructs, and performance and energy consumption over real-world Java programs;
- **The Λ curve**. We introduce the Λ curve, which is an observation that energy consumption typically increases as the number of threads increases, and then gradually decreases as the number of threads approaches the number of CPU core.
- **Race to idle does not hold.** Race to idle is the assumption that faster programs will consume less energy because they will have the machine idle fast. However, we provide evidences that being "faster" has little correlation with being "greener" for CPU computations only, running in multi-core architectures.
- **A discussion of when to use the Java threading constructs.** We observe that different thread management constructs have different impacts on energy consumption. For instance, for I/O-bound programs, the `Thread` style exhibits the best energy consumption, whereas the `ForkJoin` style has the worst. For embarrassingly parallel benchmarks, the opposite holds.
- **A refactoring approach.** To the best of our knowledge, we derived the first refactoring approach aiming to improve software energy consumption of a parallel software system.
- **A refactoring engine.** We implemented the first refactoring engine as an Eclipse plug-in. This plugin can be used to (1) *help* a programmer detect a kind of `ForkJoin` misuses; and (2) *refactor* the misused code into a lightweight form.

1.4 Organization

The remainder of this work is organized as follows.

- Chapter 2 reviews essential concepts used throughout this work;
- Chapter 3 presents the motivation to study software energy consumption. In this chapter, we present the results of an empirical analysis of STACKOVERFLOW, the most important forum in the software development world. This study reveals that energy consumption is a real problem observed in practice, and illustrates some problems, causes and solutions for energy-related problems (Section 3.3.2, 3.3.3 and 3.3.4, respectively);

- Chapter 4 shows the impact on energy consumption of different thread-safe collection implementations. We consider three important collection interfaces: `List`, `Set`, and `Map`. We also consider different "tuning knobs" of these constructs: the number of threads, the initial capacity, and the load factor. To gain confidence in our results in the presence of platform variations and measurement environments, we employ two machines with different architectures (a 32-core AMD vs. a 16-core Intel). We further use two distinct energy measurement strategies: an external energy meter, and Machine-Specific Registers (MSRs).

- Chapter 5 shows how different threading techniques impact energy consumption. We consider three important thread management constructs in concurrent programming: explicit thread creation, fixed-size thread pooling, and work stealing. We further shed light on the energy/performance trade-off of three "tuning knobs" of these constructs: the number of threads, the task division strategy, and the characteristics of processed data. Through an extensive experimental space exploration over real-world Java programs, we produce a list of findings about the energy behaviors of concurrent programs, which are not always obvious.

- Chapter 6 presents a catalog of bottlenecks present in ForkJoin applications, and it discusses how one can overcome them. Also, it presents our proposal of a refactoring tool. Our refactoring is based in one of the patterns identified in the previous chapter. We then present the design and evaluation our approach for detecting and refactoring a kind of `ForkJoin` misuse. We have applied our approach in 15 open-source projects, and we observed an energy saving of up to 23% (12% on average). Also, our evaluation reveals that our approach is *useful* and requires little programming *effort*.

- Chapter 7 surveys the related work; and

- Chapter 8 presents our final considerations and schedule for the remaining activities;

The main chapters of this thesis have been published at premiere software engineering conferences and journals. In particular, Chapter 3 is a MSR-2014 paper (PINTO; CASTOR; LIU, 2014b). Chapter 4 is a JSS-2015 paper (PINTO et al., 2015). A previous version of this chapter has also appeared in the SEPS-2014 workshop (PINTO; CASTOR, 2014). Chapter 5 appeared in an OOPSLA-2014 paper (PINTO; CASTOR; LIU, 2014a). Chapter 6 is currently under review (PINTO et al., 2015) at a premiere software engineering conference. These chapters have been extended and revised when writing this thesis.

2

Background

When a history of the last 5,000 years of computing is written in the future, shared memory will be viewed as an aberration.

—JOE ARMSTRONG

In this chapter we review and introduce some essential concepts used in this work. First, we discuss what concurrent programming is in Section 2.1. In this context, we discuss the benefits and the challenges that still make concurrent programming hard. Section 2.2 discusses important concepts of software energy consumption and how it can be measured. Finally, Section 2.3 introduces the benefits and challenges related to software refactoring.

2.1 A Sip of the History of Concurrency

Nowadays, computers are capable of doing several things at the same time, such as playing games, movies, and music. This is due to the innate capacity of sophisticated operating systems in performing multi-tasking, which is achieved primarily with *time-sharing* techniques. In this model, the scheduler has to share the processor time between all other processes that want to use the processor. One of the motivating factors led to the development of operating systems that support time sharing is because it is more efficient to use the wait time of a program to let another program use the CPU in his place.

This model emerged as the prominent model of computing during the 1970s, representing one of the major technological shifts in the history of computing (LEA, 1999). Using this model, a larger number of clients can interact concurrently with a single computer, dramatically lowering the cost of using computers. Some time-shared computers consist of a single CPU, while others consist of a set of identical CPUs. With only one CPU, programs can be executed *concurrently*. However, with more than one CPU, programs can be executed in *parallel*. With a single CPU, no real parallel execution is possible, but that one CPU can be shared in such a way that many programs seem to be executing at the same time and that CPU is used more efficiently (*e.g.*, wait time is decreased)

Although concurrency and parallelism are similar concepts, they are not the same. Concurrency consists in simulating several execution units, taking advantage of the fact that some of these operations may take longer and do not require them to use the CPU (*e.g.* I/O operations). On the other hand, the term parallelism refers to techniques used to make the program faster. This is achieved by performing several computations in parallel. Parallelism, however, requires hardware with multiple CPUs. Depending on the number of CPUs available, the execution of program can be literally parallel, entirely time-shared, or a combination of both.

Sequential programs are easier to write because they are more intuitive to reason about, since they only have a single control flow. However, the new reasoning model is not the unique problem in the transitioning from sequential to concurrent systems. Several difficulties and challenges are encountered in order to write correct and efficient concurrent programs. Here after we briefly present some historical facts (Section 2.1.1), some basic concepts for the construction of concurrent programs (Section 2.1.2), and some well-known problems that programmers might stumble upon when writing concurrent programs (Section 2.1.3). In Section 2.1.4, we turn our focus on writing concurrent programs using the Java programming language.

2.1.1 The Free Lunch is Over

Gordon Moore, co-founder of Intel, observed that the number of transistors on a chip would double in approximately every two years with no additional cost. This prediction has proven to be accurate and became known as the Moore's Law (SCHALLER, 1997), which was valid for years. In part, the reason is because, as processor technology progressed, the size of transistors has decreased significantly. This exponential improvement has dramatically enhanced the impact of digital electronics in general, and this law has been constantly employed to describe the growth of performance metrics in technology, and has also applied in other fields such as economy. Developers got used to the "free lunch" of computer performance, which allowed them to have faster applications simply by using faster computers, without the need of any additional source code modification.

However, exponential growth can not last forever. For instance, one hard physical limit can never be overcome: light will not get faster. Around 2004, it has become difficult to increase the clock rate due to not just one but at least two issues such as (1) too much heat and (2) too high power consumption. Cooling processors with clock rates higher than 3.4GHz became prohibitively expensive. In this scenario, new solutions to improve performance became necessary.

One approach to address this problem is to use more than one processor in a single chip (also known as multi-core processors). Thus, each processor does not need to work on a high frequency. However, this architectural change places a burden on application developers since to leverage parallel architectures, applications must be built with parallelism in mind. One possible solution is through automatic program parallelization of sequential programs (BANERJEE et al.,

1993). However, many researchers agree that automatic techniques are capable of exploiting only modest parallelism and have been pushed as far as they will go (LEE, 2006). A natural conclusion is that programs themselves must become more concurrent. If developers hope to continue to get performance gains in computing, now they should rethink about software development — the lunch is not free anymore.

Multi-core processors are mainstream nowadays, and both academia and industry believe that multi-core technology will remain prevalent for the years to come (ESMAEILZADEH et al., 2013). Nevertheless, multi-core processors are only beneficial for concurrent applications and have little value for most existing mainstream software. Moreover, processor manufacturer are talking about someday producing 1,000-core chips on mainstreams computers. A single-threaded application running in those chips can exploit, at most, 1/1,000 of potential throughput. This is a near future, though. In a quick search for the top 10 supercomputers in the world[1], we observed that, on average, the systems contain 182,035 processors. Since the average clock frequency of these machines is fairly low (in our search it was 2.5 GHz), the full potential of these systems must be exploited through efficient use of the parallelism that is provided by the thousands of processors they contain.

Another issue that parallel programming researchers have do deal with is, as showed in a recent study (ESMAEILZADEH et al., 2011), regardless of chip organization and topology, multicore scaling is power limited and, at the best-case, researchers believe on an average speedup of only 7.9x between 2011 and 2024. This calls for better energy-parallel efficient solutions.

On the practitioners side, the main issue remains on redesign one application to fully explore parallelism. However, redesign single-threaded applications to better leverage parallelism is easier said than done. Several reasons explain this problem. One reason is because not all problems can be fully parallelizable. Another one is because concurrent programming is still hard. In the next section we will describe the common abstractions used to write concurrent programs.

2.1.2 Processes and Threads

In concurrent programming, a program execution is called a process (TANENBAUM, 2007). For example, when the web browser program is started, a process is created. Processes are *heavyweight*: each process consumes a non-trivial amount of memory. To run, the process needs an address space and a list of memory locations in which the process can read and write. The address space contains the executable program, the data of the program, and its stack. Also, associated with each process there is a set of resources, such as registers, a list of open files, a list of related processes, and all the other information needed to run the program. A process is fundamentally a container that holds all the information needed to run a program.

[1] http://www.top500.org

Unlike processes, threads are *lightweight*. The implementation of threads and processes differs from one operating system to another but, in most cases, a thread is contained inside a process. Multiple threads can exist within the same process sharing memory, while multiple processes usually do not share memory. Threads share the code and the context — that is, the values that their variables reference at any given moment — of a process, but threads do not share registers and stack.

In multi-threaded applications, multiple threads can be created, each representing a call stack of methods separately. Multiple procedure calls are thus running simultaneously. On a single processor, multi-threading generally occurs by time-division multiplexing: the processor switches between different threads. This context switching generally happens faster than the user perceives that different threads are running at the same time. On a multi-core system, threads can actually run at the same time: each processor running a particular thread. The number of cores in the processor is then the maximum number of runnable threads simultaneously, without the need of time division multiplexing and context switches.

High-level programming languages favor the use of threads instead of process for the development of concurrent software. The main reason to choose threads is because they are easier and faster to manage: creating a thread is one hundred times faster than creating a process (BUTENHOF, 1997). Nevertheless, dealing with threads is not straightforward as it would seem (LEE, 2006). Next we describe the most common errors that could be raised in a multi-core environment.

2.1.3 Concurrent Programming Errors

Previously, we mentioned that threads are one of the most common ways to write concurrent programs. We also mentioned that threads share state. However, when threads are being executed within the same process performing read and write operations to the same memory address, a new class of errors arise. The most common concurrent errors are described next.

Race conditions. Race condition arise when two or more threads are trying to change the same shared memory at the same time. Since the thread scheduling algorithm can swap between threads at any time, the programmer does not have any guarantee about the order in which the threads will attempt to access the shared memory. Therefore, the final result can be inconsistent. To avoid this problem, the programmer has to ensure that the shared memory is accessed by only one thread at a time. Race conditions have a reputation of being difficult to reproduce and debug, since the end result is nondeterministic and depends on the relative timing between interfering threads. Problems occurring in production systems can therefore disappear when running in debug mode, when additional logging is added. Despite early efforts, race condition detection and fixing is still a hot topic of research and debate.

Deadlocks. Deadlock happens when a thread enters in a waiting state because a resource

requested by it is being held by another waiting thread, which in turn is waiting for another resource, resulting in both threads ceasing to function. For instance, Thread X holds lock A and tries to acquire lock B, but at the same time Thread Y holds B and tries to acquire A. Both threads will wait forever. Figure 2.1 illustrates this case.

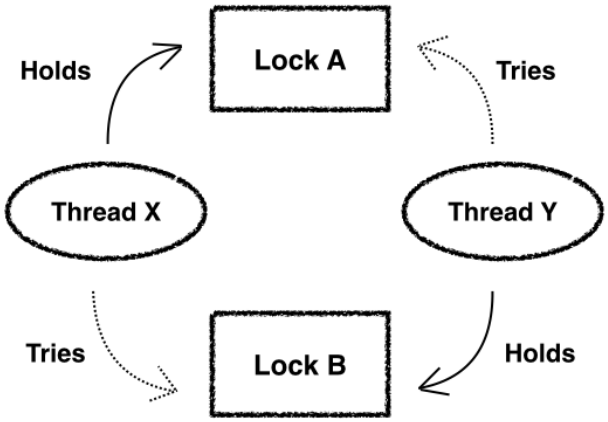


Figure 2.1: A simple example of a deadlock.

One attempt to mitigate this problem is, after several attempts to acquire both resource, some thread can unlock its resource in order to repeat and in next try to succeeded in acquire both resource. However, this approach do not guarantee that deadlock will not occurs again. Despite early efforts, thread deadlock is also a topic of research concern.

Livelocks. A livelock is similar to a deadlock, except that the states of the threads involved constantly change with regard to one another, none progressing. However, it is not evident that this is happening. A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time. Livelock is a risk with some algorithms that detect and recover from deadlock. If more than one process takes action, the deadlock detection algorithm can be repeatedly triggered. This can be avoided by ensuring that only one process (chosen randomly or by priority) takes action.

Starvation. Starvation happens when one thread waits for an event that might take much longer, or forever to happen. For example, wait for an input from a user who is not present at the time, can result in starvation. When a thread waits indefinitely for a condition to unlock a lock may also result in starvation. Starvation is similar to deadlock in the sense that it causes a process to freeze. Two or more processes become deadlocked when each of them is doing nothing while waiting for a resource occupied by another program in the same set. This problem can be avoided by setting a maximum timeout for the waiting thread.

2.1.4 Concurrent Programming in Java

Along this thesis, we will focus on the Java programming language, since it is widespread among concurrent programmers. In the Java programming language, concurrent programming is mostly concerned with threads. Multi-threaded execution is an essential feature of the Java platform. A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously. Every Java program starts with a main thread. This thread has the ability to create additional threads. In Java, every thread is created and controlled through the `java.lang.Thread` class, although programmers can also use the `java.lang.Runnable` interface, which is the most often used approach that Java developers use to create threads (TORRES et al., 2011).

The Java programming language also has support for synchronization techniques. The Java memory model guarantees that synchronized code will only be executed by one thread at a time. Synchronization is required for objects that are shared among multiple threads to avoid corruption caused by a race condition. Synchronization in Java will only be needed if a shared object is mutable. Read-only (objects that are protected from accidental changes to its content) and immutable (objects that are object whose content can not be modified after it is created) objects do not require synchronization. The Java programming language has additional support for a number of concurrent abstractions, which are not covered by this thesis proposal. To a deeper understanding of these concepts, we recommend the book of PEIERLS et al. (2005).

2.1.5 High-Level Concurrent Programming in Java

Regardless of the model of concurrency, many researchers argue that high level concurrency libraries can improve software quality (DIG; MARRERO; ERNST, 2009; ISHIZAKI; DAIJAVAD; NAKATANI, 2011; OKUR; DIG, 2012). Also, recent study have pointed out that, when using a high-level concurrent library, non-experts concurrent programmers achieved similar results (in terms of code size, execution time, and speedup) as expert concurrent programmers (NANZ; WEST; SILVEIRA, 2013). Thus using a high-level concurrent library, even a less experienced programmer can write working concurrent applications.

In particular, the Java programming language has included a set of high-level concurrency APIs in its version 1.5. This library, the `java.util.concurrent`, aims to simplify development of concurrent applications in the Java platform. With this library, the complexity to develop concurrent applications decreased significantly. The `java.util.concurrent` library offers several features to make the task of concurrent programing easier. In addition, the library is optimized for performance. For instance, a `ConcurrentHashMap` yields a 4.72x performance improvement when compare to its predecessor, `Hashtable` (PINTO et al., 2015). This thesis is inspired in the use of such library. Below we discuss some of the most well-known constructs.

Locks: Implementations of the `Lock` interface, such as `ReentrantLock`, support more flexible locking than can be performed using `synchronized` methods and blocks. They promote more versatile structuring, may have different properties depending on how threads access data, and may support multiple associated `Condition` (an interface defining condition variables associated with a lock) objects. A lock is a tool for controlling access to a shared resource by multiple threads. Commonly, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and every access to the shared resource requires that the lock be acquired first. However, some locks may allow concurrent access to a shared resource, such as the read lock of a `ReadWriteLock`. Its implementation provide additional functionality over the use of `synchronized` methods and blocks by supporting non-blocking attempts to acquire a lock (`tryLock()`), and attempts to acquire lock that can be interrupted.

Atomic Data Types: These data types are provided by a small toolkit of classes that support lock-free, thread-safe programming on single variables. In essence, the classes in the `java.util.concurrent.atomic` package extend the notion of `volatile` values, fields, and array elements, providing an atomic conditional operation using the `compareAndSet()` method. This method atomically sets a variable if its current value equals that of the method's first argument, returning `true` on success. The classes in this package also contain methods to get and unconditionally set values, and to increment and decrement the value of the variable. Examples of classes in this package are `AtomicBoolean`, `AtomicInteger` and `AtomicIntegerArray`.

Concurrent Collections: A group of collections specialized for multithreading, such as `ConcurrentHashMap`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet`. The *Concurrent* prefix used in some of these classes is a shorthand to indicate the difference from their relatives, the *synchronized* ones, which usually employ a single lock for the entire collection. For example, `Collections.synchronizedMap()` is single-lock-based, whereas `ConcurrentHashMap` is "concurrent", but not governed by a single lock. In a recent effort, we characterized the energy consumption and performance behavior of some of the concurrent collections (PINTO; CASTOR, 2014; PINTO et al., 2015).

Condition-based synchronization: `java.util.concurrent` provides some classes that can replace the `wait()` and `notify()` methods. `CountDownLatch` is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads have all been completed. A `CountDownLatch` waits for *N* threads to finish before allowing all of them to proceed. `CyclicBarrier` is another synchronization aid. It allows a set of threads to all wait for each other to reach a common barrier point.

Executors: The `Executors` framework separates thread management and creation from the rest of the application. Using this framework, instead of creating a new thread each time, programmers create a pool of threads — often fixed in size — and further submit logically independent units of work to the thread pool. The main advantage of using this pool is that

threads can be scheduled, stoped and reused. The relationship between threads and the units of work is often *1:n*. Threads select and execute submitted units of work from a centralized buffer managed by the language runtime. Executors also support multiple approaches for managing thread execution. They provide an asynchronous task execution framework. An `ExecutorService` manages queuing, scheduling of tasks, and allows controlled shutdown. The `ExecutorService` interface provides methods to asynchronously execute any function expressed as a `Callable`, the result-bearing analog of `Runnable`. A `Future` returns the results of a function, allows determining whether the execution has completed, and provides the means to cancel execution. Its implementations provide tunable, flexible thread pools. The `Executors` class provides factory methods for the most common kinds and configurations of `Executors`, as well as a few utility methods for using them.

ForkJoin: The `ForkJoin`, similarly to the `Executors` framework, also allows programmers to create a pool of operating system threads and submit logically independent units of work to the pool. However, the `ForkJoin` model (LEA, 2000) makes use of the work stealing approach. Work stealing was first introduced in the Cilk language (FRIGO; LEISERSON; RANDALL, 1998), a C-like language designed for parallel programming. Nowadays it is widely available in industry-strength C/C++/C#-based language frameworks. The core idea of work stealing has also made its way into mainstream languages such as Java, X10, Haskell, and Scala.

Generally speaking, the program runtime consists of multiple *workers*, each executing on a host CPU core (or hardware parallel unit in general). Each worker maintains a queue-like data structure — called *deque* — each item of which is a task to be processed by the worker. When a worker completes the processing of a task, it picks up one more task from the deque and continues the execution for that item. When the deque is empty (we say the worker or its host core is idle), the worker *steals* a task from the deque of another worker. In this case we call the stealing worker a *thief* whereas the worker whose item was stolen a *victim*. The selection of victims follows the principles observed by load balancing and may vary in different implementations of work stealing.

The `ForkJoinPool` class is the entry point of this framework. It provides management and monitoring operations, and also the work-stealing implementation. In the Java `ForkJoin` framework, parallel computations are modeled as subclasses of `ForkJoinTask` or one of its subclasses: `RecursiveTask` and `RecursiveAction`. Both `RecursiveTask` and `RecursiveAction` inherit from `ForkJoinTask`, which is a thread-like entity (it implements the `Future` interface), but it is much lighter weight than a normal thread. Large numbers of tasks and subtasks can be hosted in a small number of actual threads in a `ForkJoinPool`. The `RecursiveTask` and `RecursiveAction` classes are similar to the `Thread` one in the sense that the programer should inherited from one of these classes, and override the `compute()` method (the `run()` method in case of the `Thread` class) with the computation that the programer wants to run in parallel. The difference between `RecursiveTask` and `RecursiveAction` is that the former can return the result of a computation while the latter

does not.

Once the `ForkJoinTask` is started, it will usually in turn starts other subtasks. As indicated by the name of this class, many programs using `ForkJoinTask` employ only the methods `fork()` and `join()`. The `fork()` method creates a new task that will be added in the deque of workers, waiting for execution. The `join()` method should be called after the `fork()` one, and it introduces a point of synchronization, that halts the parent computation until the child computation has returned. The framework also provides additional utility methods such as the `invokeAll()`, which is a syntatic-sugar for the `fork()-join()` methods, and `isDone()`, which verifies the status of the computation (returns `true` if the task is completed).

In this work, we devote special attention to the thread-safe data structures and the `Thread`, `Executors` and `ForkJoin` concurrent programming constructs. In Chapter 5, we caracterize the performance and energy consumption of these constructs in several configurations and workloads.

2.2 Software Energy Consumption

For many years, research that connects computing and energy efficiency has concentrated on the hardware layer. These studies are motivated by the assumption that only hardware dissipates power, not software. However, there are studies that show that this assumption does not capture the whole picture (COHEN et al., 2012; SAMPSON et al., 2011). That would be analogous to postulating that only automobiles are responsible for burning gasoline, not the people who drive them and the way they are used.

In any computer system, it is software that directs much of the activity of the hardware. Consequently, software can have a substantial impact on energy consumption. Software solutions for improving energy efficiency of computer systems can work at different levels, ranging from machine code level to end-user applications. In spite of advances in many areas (ASAFU-ADJAYE, 2000), IT energy consumption keeps rising steeply, which indicates that rising demand is outpacing efficiency improvement. This is not surprising, given that software engineering has, in the past decades, focused on developer efficiency, not on minimization of resource consumption. Concerns about energy usage were left for compiler writers, operating system designers and hardware engineers.

Thanks to the rapid proliferation of mobile phones, tablets, and unwired devices in general, energy efficiency is becoming a key software design consideration where the energy consumption is closely related to battery lifetime. It is also of increasing interest in the non-mobile arena, such as data centers and desktop environments. Energy-efficient solutions are highly sought after across the computer stack, with more established results through innovations in hardware/architecture (BIRCHER; JOHN, 2008; IYER; MARCULESCU, 2002; TIWARI; MALIK; WOLFE, 1994), operating systems (GE et al., 2007; MERKEL; BELLOSA, 2006; RANGAN; WEI; BROOKS, 2009), and runtime systems (FARKAS et al., 2000; RIBIC; LIU,

2014; VIJAYKRISHNAN et al., 2001).

While the strategy of leaving concerns about energy consumption to the lower-level systems and architecture layers has been successful, recent empirical studies have provided initial evidence that software engineers can play an effective role in reducing energy usage through their high-level design and implementation decisions (COHEN et al., 2012; HINDLE, 2012; HINDLE et al., 2014; KIRBAS et al., 2014; KWON; TILEVICH, 2013; LI; HALFOND, 2014; MANOTAS; POLLOCK; CLAUSE, 2014; PINTO; CASTOR; LIU, 2014a; SAHIN et al., 2012; SAHIN; POLLOCK; CLAUSE, 2014; TREFETHEN; THIYAGALINGAM, 2013; ZHANG et al., 2012). These approaches complement prior hardware/OS-centric solutions, so that improvements at the hardware/OS level are not cancelled out at the application level, *e.g.,* due to misuses of language/library/application features. Due to satisfactory initial results, energy efficiency for higher layers of the software stack, in particular at the application level, is now an emerging research topic. However, a major obstacle to developers fulfilling their role in reducing energy consumption is a lack of information about how high-level decisions impact energy consumption. Developers currently do not understand how the choices and tradeoffs they make on a daily basis impact the energy consumption of their software (PINTO; CASTOR; LIU, 2014b). In a preliminary report, we observed that programmers still rely in the old fashion frequency scaling approach to save energy (MOURA et al., 2015). However, according to a recent studies (LIU; PINTO; LIU, 2015; KAMBADUR; KIM, 2014), blindly downscaling CPU frequency often leads to increased energy consumption and performance loss.

Dynamic Voltage and Frequency Scaling (DVFS) (PERING; BURD; BRODERSEN, 1998) is a common CPU feature where the operational frequency and the supply voltage of the CPU can be dynamically adjusted. It is one of the most effective power management strategy used in architecture research (LEBRETON; VIVET, 2008; TSENG; CHANG, 2008). Most of the CPUs being used today support DVFS. In the era of multi-core CPUs, the frequency of individual cores can also be adjusted separately, a feature known as multiple frequency domain support. In addition to small portions related to static leakage, the vast majority of a CPU's power consumption P results from its dynamic operation, which can be computed as $P = C * V^2 \times F$, where V is the voltage, F is the frequency, and C is the capacitance. The energy consumption E is an accumulation of power consumption over time, $E = P \times t$, where t is the operating time.

Due to the innate nature of CPU design, voltage and frequency are often scaled together. Scaling down the CPU frequency is thus effective in saving power. Saving energy however is slightly more complex, because a reduction of frequency may increase the execution time, t. DVFS-based energy management thus often deals with the trade-off between energy consumption and performance. Even though DVFS features are still used these days (BARTENSTEIN; LIU, 2013; RIBIC; LIU, 2014), most of the existing research is based on understanding the different trade-offs between two or more programming techniques, and provide an energy consumption point of view based on such empirical evidence. One mandatory step towards this goal is to accurately measure the energy consumption of a given software system.

2.2.1 Energy Consumption Measurement

Power measurement and energy estimation are a broad area of research that encompasses several sub-fields, including architecture, operating systems, and software engineering. The first group of techniques, power measurement, makes use of power measurement hardware to obtain power samples and then uses software based techniques to attribute the power to implementation structures. Due to the wide spread presence of manufacturers, different power meters are currently available in the market. Different power meters have different characteristics. One of the most important among these characteristics, however, is the sampling rate, that is, the number of samples obtained per second. The sample is often measured in watts, P (power), and the conversion to joules, E (energy), can be done by $E = P \times time$. Depending on the power meter used, the sampling rate can vary from 1 sample per second, to more than 1,000 samples per second. The higher the sampling rate, the more accurate the power curve will be. In Chapter 5 we describe a study that focus on the energy efficiency of Java threading implementations. In this study, we have used a power measurement plugged directly on the CPU cable. Figure 2.2 shows the physical setup needed. Details about the hardware configurations are provided in the same chapter.

Figure 2.2: An infrastructure for power measurement.

The second area, energy estimation, assumes that developers do not have access to power measurement hardware and uses software-based techniques to predict how much energy an

application will consume at runtime. One example of this approach is the powertop[2] utility. This tool takes one sample per second, and generates a log with these measurements. This tool analyzes the programs, device drivers, and kernel options running on a computer based on the Linux and Solaris operating systems, and estimates the power consumption resulting from their use. Powertop can also instrument laptop battery features in order to estimate power usage (in Watts) and battery life.

Another example is the Running Average Power Limit (RAPL) interface (DAVID et al., 2010). Originally designed by Intel to enable chip-level power management, RAPL is widely supported in today's Intel architectures, including Xeon server-level CPUs and the popular `i5` and `i7`. RAPL-enabled architectures monitors the energy consumption information and stores them in Machine-Specific Registers (MSRs). Such MSRs can be accessed by OS, such as the `msr` kernel module in Linux. RAPL is an appealing design, particularly because it allows energy/power consumption to be reported at a fine-grained manner, *e.g.*, monitoring CPU core, CPU uncore (caches, on-chip GPUs, and interconnects), and DRAM separately. One drawback of this approach is the fact that programmers need a deep knowledge on how to use these low-level registers, which is not straight-forward. In a cooperative work (LIU; PINTO; LIU, 2015), we have developed a set of APIs for profiling Java programs running on CPUs with RAPL support, called `jRAPL`. The library can be viewed as a software wrapper to access the MSRs. Since the user interface for `jRAPL` is simple, the programmer can focus her efforts in the high-level application design. For any block of code in the application whose energy/performance information is to the interest of the user, she simply needs to enclose the code block with a pair of `statCheck` invocations. For example, the following code snippet attempts to measure the energy consumption of the `doWork()` method, whose value is the difference between `beginning` and `end`:

```
double beginning = EnergyCheck.statCheck();
doWork();
double end = EnergyCheck.statCheck();
```

Figure 2.3: An example on how to measure energy consumption using the `jRAPL` library

Compared with traditional approaches based on physical energy meters, the `jRAPL`-based approach comes with several unique advantages:

- *Refined Energy Analysis*: thanks to RAPL, our library can not only report the overall energy consumption of the program, but also the breakdown (1) among hardware components and (2) among program components (such as methods and code blocks). As we shall see, refined hardware-based analysis allows us to understand the relative activeness of different hardware components, ultimately playing an important role in analyzing the energy behavior of programs. In meter-based approaches, hardware

[2] https://01.org/powertop

design constraints often make it impossible to measure a particular hardware component (such as CPU cores only, or even DRAMs because they often share the power supply cables with the motherboard).

- *Synchronization-Free Measurement*: in meter-based measurements, a some- what thorny issue is to synchronize the beginning/end of measurement with the beginning/end of the execution of interest. This problem would be magnified if one considers fine-grained code-block based measurement, where the problem de facto becomes the synchronization of measurement and the program counter. With `jRAPL`, the demarcation of measurement coincides with that of execution; no synchronization is needed.

In this thesis, we rely on both power measurement hardware and energy estimation tools (with `jRAPL`). As we shall see, Chapter 4, which uses both strategies, shows that similar results can be achieved using hardware and software-based techniques. Experiments in Chapter 5 uses only a power measurement hardware. Chapter 6 uses only `jRAPL`.

2.3 Software Refactoring

Programs evolve. They might change because of changing in a requirement or because the context in which they exist changes. In order to update their programs, developers usually do so incrementally, by changing one single piece at a time. Each step can be seen as a behavior-preserving transformation, *i.e.*, a refactoring. Refactorings (FOWLER et al., 1999) are program transformations that change the structure of a program but not its behavior. For instance, a programmer can change the name of a method, remove duplicate code, or change one interface to make it more reusable. The term refactoring was coined by OPDYKE (1992) in his PhD thesis two decades ago, and it was later popularized by FOWLER et al. (1999) influential textbook.

Programmers have been refactoring for decades, and nowadays refactoring is well-known as one of the key steps during a project life cycle. Refactoring occurs at all levels of the software development process — from testing to maintenance. Refactoring is also used to overcome technical issues related to software evolution, such as limited understanding and maintainability. Along the years, a wide range of refactorings have been introduced, some of them from the research community (DIG et al., 2009; LIN; DIG, 2013; KJOLSTAD et al., 2011). Refactoring has several benefits, such as improving developer productivity by making it easier to maintain and understand a software. Agile advocates go further and claim that a lack of refactoring incurs technical debt (BECK; ANDRES, 2004). Some of these benefits include (FOWLER et al., 1999; OPDYKE, 1992): make it easier to add new code and make it easier to change existing code. However, refactoring benefits are likely to go beyond understandability, covering different requirements such as extensibility, reusability, and testability. Also, recent research

has succeed in applying refactoring in areas such as performance (OVERBEY et al., 2005) and correctness (DIG; MARRERO; ERNST, 2009).

However, perform a refactoring is *tedious*, because it requires changing and reasoning about several lines of source code, *error-prone*, because programmers can update the code to use a wrong or deprecated API, and *omission-prone*, because the programmer can miss an opportunity to use a more efficient language construct. Also, when naively applied, refactoring can break the code of an working system. For instance, when not updating the old references of a renamed variable. In order to point out the challenges involved to sucessfully apply a refactoring, we will introduce and briefly describe the purpose of a well-known refactoring: EXTRACT METHOD.

2.3.1 Example of Refactoring: Extract Method

As our example, let us consider the EXTRACT METHOD refactoring. This refactoring has garnered some attention in the literature (TSANTALIS; CHATZIGEORGIOU, 2009; SCHAEFER; MOOR, 2010) as a typical example of a non-trivial refactoring that requires a certain amount of analysis to be performed correctly. According to Fowler, an implementation of EXTRACT METHOD is the hallmark of a "serious" refactoring tool. This refactoring can be described as follows:

> *Given a sequence of statements, extract these statements into the body of a newly created method, and replace the original statements by a call to that method.*

As a simple example, consider method `m` in Figure 2.4 on the left, and assume we want to extract the body of the for loop into a new method `process`. Such task is relatively simple; all the programmer needs to do is to provide `i` and `total` as parameters to the new method, and return the value of `total` to update the original variable after the method returns. Thus the resulting program should look like the one showed on the right of the same code snippet.

```
class A {

  void m() {
    int total = 0;
    for (Item i: getItems()){
      total += i.getValue();
    }
  }
}
```

⇒

```
class A {
  void m() {
    int total = 0;
    for (Item i: getItems()) {
      total = process(i, total);
    }
  }
  int process(Item i, int total)
    return total += i.getValue();
}
```

Figure 2.4: A simple example of the EXTRACT METHOD refactoring.

However, the simplicity of this example is misleading. Some questions that are easily raised are not easly answered, such as: What if the code to be extracted is in several lines instead

of just one? What if the code changes the control flow using `return`, `break` or `continue` keyword? What if it throws/catch exceptions? How to determine which parameters do we need to pass to the new method? How to decide which values do we need to return to the calling method? What if more than one value need to be returned?

Although EXTRACT METHOD is a fairly well-studied and well-applied refactoring (PINTO; KAMEI, 2013a), guarantee behavior preservation is a difficult task. In the case of EXTRACT METHOD it would be beneficial if we could split the transformation into several steps that each address a subset of the problematic issues in turn.

2.3.2 Refactoring Tools

In its early days, refactoring has typically been done manually, or with the help of primitive tools (MENS; TOURWÉ, 2004). However, manually performing these tasks is fairly time-consuming and programmers under time pressure have little chance to do it. Refactoring tools automate refactorings that you would otherwise perform with an Integrated development environment (IDE). Many popular development environments for a variety of languages — such as Eclipse[3], Microsoft Visual Studio[4], Xcode[5], and NetBeans[6] — now include refactoring tools. Researchers have then devoted efforts in the creation of refactoring tools (OKUR et al., 2014; OKUR; DIG, 2012; WLOKA; SRIDHARAN; TIP, 2009; KJOLSTAD et al., 2011; ISHIZAKI; DAIJAVAD; NAKATANI, 2011; DIG et al., 2009; ZHANG et al., 2012; SCHäFER et al., 2011).

Modern refactoring tools provide a rich set of functionalities, which basically promise two benefits: First, refactoring tools promise to preserve functionality. Second, refactoring tools promise to refactor faster than a programmer can refactor by hand (MURPHY-HILL; BLACK, 2008). In that sense, the use of refactoring tools should increase the programmer productivity.

2.4 Summary

This chapter introduces important concepts used through this work. It first gives a gentle introduction to some concurrent programming concepts. Then it presents the importance of software energy consumption, and explain how we can measure it. We conclude by presenting refactoring as program transformations, showing how a simple extract method refactoring could be challenging.

[3] www.eclipse.com
[4] www.visualstudio.com
[5] www.developer.apple.com/xcode/
[6] www.netbeasns.org

3

Software Energy Consumption in Practice

Reason, observation, and experience; the holy trinity of science.

—ROBERT G. INGERSOLL

In last chapter, we discussed some of the basic concepts related to concurrent programming, software energy consumption and refactoring. In this chapter we present an empirical investigation used to asses whether software energy consumption is a real issue in the software development world. Through a categorization of questions and answers posted on STACKOVERFLOW, we first analyze the distinctive characteristics of energy consumption questions (Section 3.3.1), the most common energy-related problems (Section 3.3.2), causes (Section 3.3.3) and solutions (Section 3.3.4). After that, we discuss some recurring problems identified in the answers of these questions (Section 3.4.2). Finally, we compare the solutions proposed by the STACKOVERFLOW users with the state of the art, in order to verify whether they are supported (Section 3.4.3).

3.1 Overview

We believe a critical dimension to further improve energy efficiency of software systems is to understand *how software developers think*. The needs of developers and the challenges they face may help energy-efficiency researchers stay focused on the real-world problems. The collective wisdom shared by developers may serve as a practical guide for future energy-aware and energy-efficient software development. The conceptually incorrect views they hold may inspire educators to develop more state-of-the-art curricula.

The goal of this work is to obtain a deeper understanding of (i) whether application programmers are interested in software energy consumption, and, if so, (ii) how they are dealing with energy consumption issues. Specifically, the questions we are trying to answer are:

RQ1 What are the distinctive characteristics of energy-related questions?

RQ2 What are the most common energy-related problems faced by software developers?

RQ3 According to developers, what are the main causes for software energy consumption?

RQ4 What solutions do developers employ or recommend to save energy?

Our study is based on data from STACKOVERFLOW, a collaborative development questions and answers (Q&A) website. As one of the most popular forums in the software development world, STACKOVERFLOW is often used for software engineering studies (MORRISON; MURPHY-HILL, 2013; PINTO; KAMEI, 2013a; WANG; LO; JIANG, 2013). It contains over 2 million users, 5.5 million questions and 10 million answers. STACKOVERFLOW data can be easily accessed through an open backup[1]. In this chapter, we are looking for questions related to "energy efficiency", "energy consumption", and related terms. We found a total of 325 questions and 558 answers from more than 800 software developers. We employ a thematic analysis (FEREDAY, 2006) to examine the data and identify recurring topics in the questions and associated answers. The main findings of this study are the following:

- Energy-related questions have distinct characteristics relative to the average STACKOVERFLOW questions. On the average, they have 2.6 times more answers, are marked as favorites 3.89 times more often, have 68% more views, 10% more "up-votes", and 11% more comments. Yet, the answers to them are *less* frequently "up-voted." Albeit *interesting*, we believe these questions are also more *challenging* for the software development community.

- We identify 5 main themes regarding energy consumption questions, namely: *Measurement*, *General Knowledge*, *Code Design*, *Context Specific* and *Noise*. Questions that pertain to the *Code Design* theme receive more attention.

- Energy consumption questions sustain a near-linear growth in the last 5 years, with contributors from 9 geographic regions. This suggests that energy consumption is an *emerging* topic.

- We identify 7 major causes for energy consumption problems according to developers. Factors such as unnecessary resource usage, background activities, and excessive synchronization are considered to be the common causes of energy consumption problems.

- We summarize 8 common solutions suggested by developers to improve energy efficiency. We discuss their perceived effectiveness, compare them with the state of the art of software energy consumption research, and highlight some of the mistakes made by STACKOVERFLOW users.

[1] http://blog.stackoverflow.com/category/cc-wiki-dump/

3.2 Research Methodology

In this section we describe our data collection procedure and the qualitative research approach we employ.

Our data collection follows a mixed-method approach, collecting both quantitative and qualitative data. Using the STACKOVERFLOW dump, we extracted questions, answers, tags, and other metadata. STACKOVERFLOW was officialy launched on July 31, 2008. Thus, the data reported in this study are based on questions that were asked from its creation to September 06, 2013. We filter through more than 5 million questions, 10 million answers and 2 million users.

STACKOVERFLOW allows users to register, post questions, and answer posted questions. Since users are registered, one could track the questions/answers on a per-user basis. For each posted question, a user can include a **title** and textual description of the problem in the **body**. The user can also include code snippets. Code snippets are often separated from regular text. **Tags** are used to organize questions. Users have to attach at least one tag and can attach up to five tags when asking a question. For each question, multiple answers can be given by different users. The original user who ask the question can then either post a comment or indicate one of the answers as correct. Other people can also rate whether they like either the questions and/or the answers. The community itself is responsible for assuring the quality of the questions and answers: if a question or answer is considered relevant, thorough, or correct, users "up-vote" it; if not, they "down-vote" it. Users who posted those up-voted questions and answers receive "reputation points". That is, building reputation within the community is a key motivator for contributing to STACKOVERFLOW. A snapshot of the STACKOVEFLOW page showing a question and its corresponding answer is shown in Figure 3.1.

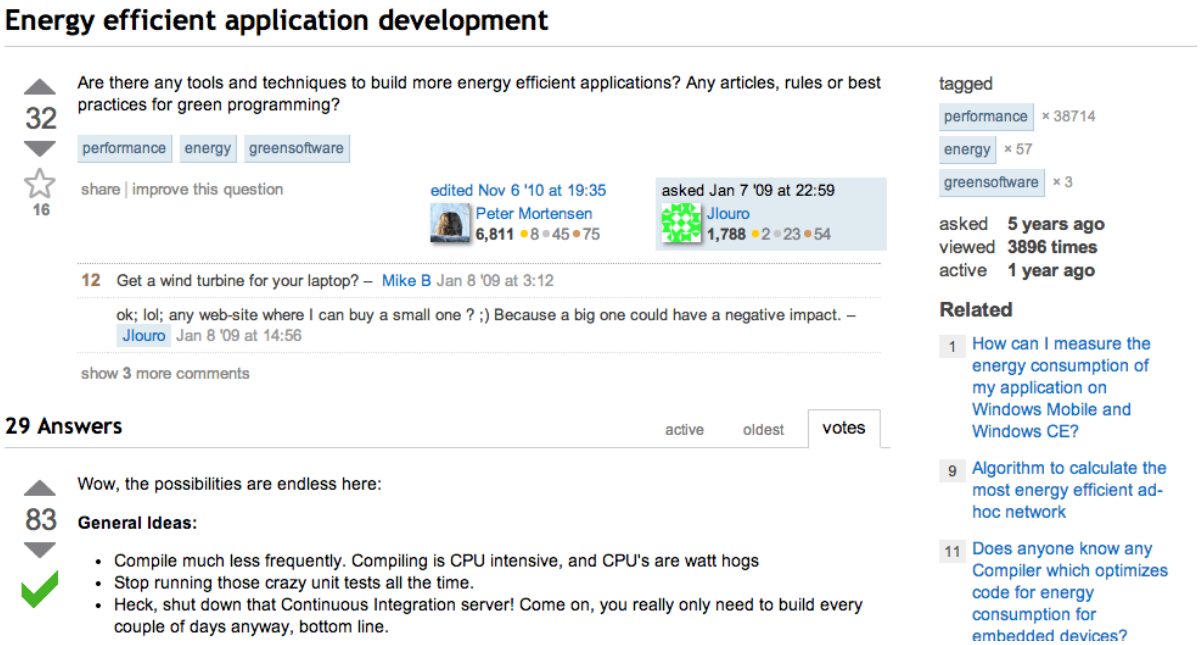


Figure 3.1: STACKOVERFLOW environment.

We follow a two-phase approach to filter out the questions related to energy consumption. We first load all data in a relational database. Then we query the table with the following terms: *energy consum*, *energy efficien*, *energy sav*, *save energy*, *power consum*, *power efficien*, *power sa*, *save power*. The character '*' in each term works as a wildcard: the query will select questions that match at least one of these keywords, regardless of the beginning or the end of the content. The searched keywords are matched against the subject, body or tags associated with each question. After this automatic process, we found 615 questions and 1.197 answers.

Next, we manually eliminated the false positive questions. For instance, a false positive question is when a programmer has an application that shows the energy consumption levels for a given scenario, and the programmer wants to add a feature to this application[2]. After this extraction phase, a total of 325 questions and 558 answers were selected. We refer this group throughout this chapter as our **base group**. The amount of data we extracted comprises 123,075 words. The remaining questions on STACKOVERFLOW are called as STACKOVERFLOW group.

Once the data is collected, we extract reliable information through a thematic analysis approach (FEREDAY, 2006). Thematic analysis involves examining, identifying and recording patterns (or "themes") within data. Themes are patterns across data sets important to the description of a phenomenon and are associated with a specific research question. Themes are identified by bringing together components or fragments of ideas or experiences, which are often meaningless when analyzed in isolation. These themes become the categories for analysis. Thematic analysis is performed through the process of coding in six phases to create established, meaningful patterns. These phases are:

1. *Familiarization with data:* At this stage, we analyzed the STACKOVERFLOW question and their answers. When a STACKOVERFLOW user mentioned very specific construct or library, we have searched on internet forums and in mailing lists in order to better familiarize with them.

2. *Generating initial codes:* Here we gave a code for each question. This code tries to summarize the core of the question. For instance, a question that ask for examples on how to improve the energy efficiency of a given algorithm was coded as "general knowledge". In this step, we refine codes by combining and splitting potential codes.

3. *Searching for themes:* In this step, we already had a list of initial themes (*e.g.*, threading and synchronization), but we begin to focus on broader patterns in the data, combining coded data with proposed themes.

4. *Reviewing themes:* At this stage, we have a potential set of themes. In this phase, we searched for data that supports or refutes our theme. For instance, we initially themed a question that asked "I want to prevent the monitor from going to sleep. What call

[2] www.stackoverflow.com/questions/413227

do I make?" as "Noise". However, we later realized that this question would fit better as "Context-specific".

5. *Defining and naming themes:* Here we refined existing themes. At this time, most of the themes had already a name. However, we have renamed some of them in order to cover codes with small number of questions, otherwise we would discard those codes, since we established 5 as a threshold for the minimum number of repetitions of a code required for it to be considered a theme (PINTO; CASTOR; LIU, 2014b; SINGER; FILHO; STOREY, 2014). This heuristic is based on the assumption that if more users are supporting a specific theme then it is likely to be stronger and more useful.

6. *Producing the final report:* Finally, we chose only themes that make meaningful contributions to answering our research questions. This resulted in 5 main themes. We discuss the individual themes in Section 3.3.2. After each one, we list representative examples of questions that were grouped into that theme.

3.3 Empirical Evaluation

In this section we describe our results grouped by research questions.

3.3.1 RQ1: What are the distinctive characteristics of energy-related questions?

To answer our first research question, we analyzed several quantitative aspects regarding our base group of questions.

First, we have analyzed whether the interest in energy consumption is increasing or decreasing over the years. To that end, Figure 3.2 shows the distribution between questions and answers created during the analyzed period.

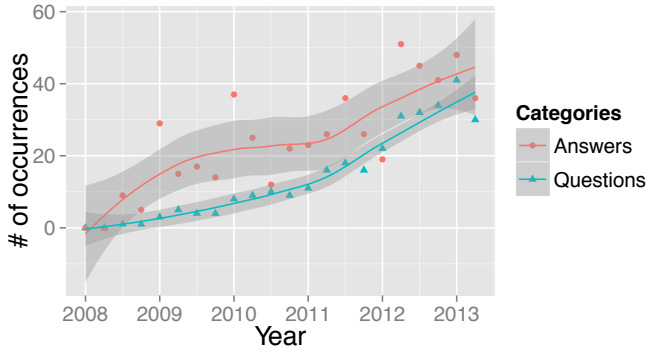


Figure 3.2: Questions and answers, from the base group, per year.

The figure presents a chart indicating what trend is suggested by the data. The shaded areas surrounding the lines indicate the confidence interval calculated by smoothing. Each point in the figure represents quarterly data. We can observe a couple of interesting findings from this figure. First, it suggests an overall growth in the number of questions throughout the years. We observed a great increase in the number of questions until the first quarter of 2012, where the number of questions had increased 100% when compared to the same period of 2011. The number of questions kept increasing until it reached the highest part of the graph, located on the first quarter of 2013. At this point, we observed an increase of 183% in the number of questions when compared to the same period of 2012. The last point of the graph was not provided because it does not cover the entire quarter: the data is available until September 6, 2013. Second, the figure shows a great number of answers followed by a large standard deviation (overall SD: 13.89 for answers and 12.36 for questions). We observed that most of questions have less than 2 answers (3rd quartile: 2, median: 1, mean: 1.71), and only 15 questions (4% of them) have received more than 5 answers. However, the number of answers regarding the latter group is about 25% of the overall number of answers.

Second, on STACKOVERFLOW, the user who asks a question can mark at most one answer per question as **accepted**. We use this feature to define the *success* of a question as follows: A **successful** question has an accepted answer, an **ordinary** question has answers, but none of them was accepted, and an **unsuccessful** question has no answer. Following these definitions, Table 3.1 shows the number of questions comparing our base group of questions with the overall questions in STACKOVERFLOW (excluding the base group).

As we can see in the Table 3.1, the majority of the questions have answers – 85.85% and 90.80% for our base group of questions, and the remaining group of questions on STACKOVERFLOW, respectively. However, only part of them are successful. For the base group of questions, more than a half of these questions have successful answers. On the other hand, about half of them are successful for the remaining questions on STACKOVERFLOW. Notwithstanding, the base group also has the highest percentage of unsuccessful answers. As we will discuss later (Section 3.3.2), some categories of questions are more unsuccessful than others.

We also observed that there are no obvious "energy consumption experts". Only 1 user has answered 3 different questions, and 27 users (4.83% of the total) have answered 2 different questions. Also, no user asked more than one energy consumption-related question. Still, we observed that the questions in the base group come from a greatly diverse group of developers. They are from 9 geographic regions: North America, Central America, South America; Europe;

Table 3.1: The success status of questions on STACKOVERFLOW.

Source	Successful	Ordinary	Unsuccessful
Base Group	45.85%	40.00%	14.15%
STACKOVERFLOW	39.99%	50.81%	9.20%

Africa; West Asia, Central Asia, East Asia; and Australia/New Zealand. The average age, reputation and views per user page are, respectively, 28.58, 1,798 and 218.1.

Table 3.2 summarizes the characteristics of the users who created the base group of questions, and the remaining users on STACKOVERFLOW.

Table 3.2: The summary of users who asked energy consumption-related questions. We removed outliers from the histograms, but not from the numerical results.

	Mean	Median	Standard Dev.	Histogram	STACKOVERFLOW
Reputation	1,798	165	6,607		134.9
Age	28.58	27	7.75		29.60
Views	218.1	32	594.8		15.87

Interestingly, some STACKOVERFLOW users have much more visibility than other STACKOVERFLOW users. This is observed in the standard deviation of the Views variable. We have also compared the popularity of energy consumption questions with the average STACKOVERFLOW questions. To measure if a given question is popular, we derived the following formula – which we believe provides a summarized view that aggregates a number of different popularity measures:

$$\mathbb{P} = \mathbb{S} + \mathbb{A} + \mathbb{C} + \mathbb{F} + \mathbb{V}$$

where $\mathbb{S}$ is the score of the question. The user can "up-vote" a question if she thinks that the question is good enough, or "down-vote" it otherwise. The score is the result of this votation process. The variables $\mathbb{A}$, $\mathbb{C}$, and $\mathbb{F}$ are, respectively, the number of answers, comments, and favorizations per question. A *favorite* question is one that developers want to know more about. By indicating a question as favorite, a developer receives notifications whenever that questions is updated (modified, answered, etc). The $\mathbb{V}$ variable corresponds to the number of views. We normalize each variable to avoid distortions caused by the very large absolute values. Taking the $\mathbb{V}$ variable as an example, it is normalized by the average of the number of views of the overall STACKOVERFLOW questions (which is 380.06), as follows: $\mathbb{V} = questionsViews \Big/ stackOverflowViews$. The other variables follow the same rule. We chose this normalization approach because it can give us a fair way to compare the two groups of questions. Table 3.3 shows the normalized results of each variable of our base group of questions.

Since the value for each metric for the STACKOVERFLOW group is normalized to be 1 (then its $\mathbb{P}$ value is 5) we can observe that questions from our base group are two times more popular when compared to the average questions on STACKOVERFLOW. It not means, however, that energy consumption is the most popular topic on STACKOVERFLOW. In particular, only the $\mathbb{S}$ and the $\mathbb{C}$ variables have similar values. For the $\mathbb{S}$ variable, we observed that 3.69% (12 out of 325) of the questions have negative scores. Analyzing those questions and the associated

Table 3.3: The popularity of questions in each group of data. We removed outliers from the histograms, but not from the numerical results.

	Base Group	Median	Standard Dev.	Histogram
$\mathbb{S}$	1.10	1.00	4.73	
$\mathbb{A}$	2.67	1.00	2.35	
$\mathbb{C}$	1.11	1.00	2.46	
$\mathbb{F}$	3.89	0.00	1.79	
$\mathbb{V}$	1.68	1.00	3.37	
$\mathbb{P}$	10.45	–	–	–

comments, we have found that they were negative because they are (i) poorly elaborated[3] or (ii) already answered[4]. The values for $\mathbb{A}$ and $\mathbb{F}$ present the most significant differences.

Table 3.4 shows the results for $\mathbb{S}$ and $\mathbb{C}$ variables, but now regarding the answers of those questions. We did not compute the other variables because answers do not have such information. The popularity of answers is calculated as $\mathbb{PA} = \mathbb{S} + \mathbb{C}$. Here, comments and scores for the answers are similar although the results for the base group were lower. Although it is easy to create and favorite (and then, follow) such questions, as we will discuss later, providing reliable answers to them is not straightforward.

Table 3.4: The popularity of answers from each group. We removed outliers from the histograms, but not from the numerical results

	Base Group	Median	Standard Dev.	Histogram
$\mathbb{S}$	0.96	1.00	4.52	
$\mathbb{C}$	0.86	0.00	1.80	
$\mathbb{PA}$	1.82	–	–	–

Finally, we investigated how tags are used in StackOverflow. A StackOverflow user employs tags to categorize a question and group it with other similar questions. We found a total of 970 tags, but only 360 different keywords were used to tag questions. Each energy consumption question has between one and five tags (median: 3, mean: 2.99, 3rd quartile: 4, SD: 1.28). The five most common tags are: (i) “android” with 172 occurrences, (ii) “power-management” with 33 occurrences, (iii) “Java” with 28 occurrences, (iv) “iOS” with 24 occurrences, and (v) “iPhone” with 24 occurrences. This result indicates that energy consumption questions are strongly related to mobile development – 26.28% of the tags are mobile-related. In particular, the Android platform has 17.78% of overall tag usage. This is probably due to its strong adoption, and the ease of developing Android applications. Another indicator is the Java

[3] www.stackoverflow.com/questions/4946600
[4] www.stackoverflow.com/questions/9187303

programming language appears to be of the most often used tags. Since Android applications are written in Java, developers are also interested in saving energy in this particular language.

3.3.2 RQ2: What are the most common energy-related problems faced by software developers?

To further understand the characteristics of software energy consumption-related questions on STACKOVERFLOW, we manually analyzed the titles and bodies of the questions from the base group in order to identify what recurring categories these questions define. The categories are our themes (Section 6.2). We identified a total of 5 themes.

Measurements. Questions about measurement tools or techniques, e.g., "*I want to measure the energy consumption of my own application (which I can modify) [. . .] on Windows CE 5.0 and Windows Mobile 5/6. Is there some kind of API for this?*"[5].

General Knowledge. Questions that do not have a concrete use case, e.g., "*Can a code optimized for least MCPS be guaranteed to have least power consumption as well?*"[6].

Code Design. Questions about programming techniques that can help in saving energy, e.g., "*Are there any s/w high level design considerations [. . .] to make the code as power efficient as possible?*"[7].

Context-specific. The authors of such questions need to provide more details in order for other users to better understand the problem, and to facilitate replication, e.g., "*I want to prevent the monitor from going to sleep. [. . .] What call do I make?*"[8].

Noise. These questions are not directly associated with an energy consumption issue. Usually, the user first wants to improve one aspect of her application and, as a secondary goal, also improve energy consumption, e.g., "*What are the good features of a processor should have which help in carrying out multimedia(Video/Image)?. [. . .] PS: It has to be low power as it is for portable applications.*"[9].

Table 3.5 shows the distribution of questions and answers per theme as well as the number of successful, ordinary and unsuccessful questions in each category. The table also shows the value of each normalized popularity variable. We use **boldface** to highlight the highest value for each case.

From this table we can make some interesting observations. First, the number of questions and answers per category can vary greatly. For instance, *Code Design* is the category with the smallest number of questions (36 questions; 113 answers) and *Noise* is the one with the greatest number (107 questions; 134 answers). On the one hand, questions that lie in the *Code Design* category are usually focused on how programmer decisions may improve energy consumption.

[5] www.stackoverflow.com/questions/724349
[6] www.stackoverflow.com/questions/506452
[7] www.stackoverflow.com/questions/506452
[8] www.stackoverflow.com/questions/1003394
[9] www.stackoverflow.com/questions/3625568

Table 3.5: Common energy-consumption themes with their status and popularity. Q means questions, A means answers. All the values for variables $\mathbb{S}$, $\mathbb{A}$, $\mathbb{C}$, $\mathbb{F}$ and $\mathbb{V}$ are normalized.

#	Measurements	Context Specific	Code Design	G. Knowledge	Noise
Questions	59	83	36	40	**107**
Answers	97	110	133	84	**134**
A/Q	1.64	1.33	**3.69**	2.10	1.25
Successful	32.2%	42.18%	**72.22%**	50.00%	45.65%
Ordinary	47.45%	**50.60%**	25.01%	40.00%	33.00%
Unsuccessful	20.33%	7.22%	2.77%	10.00%	**21.35%**
$\mathbb{S}$	1.12	0.64	**2.80**	1.87	0.76
$\mathbb{A}$	2.56	2.06	**5.76**	3.28	1.95
$\mathbb{C}$	1.04	0.81	**1.57**	1.32	1.19
$\mathbb{F}$	4.42	2.26	**10.21**	4.73	2.94
$\mathbb{V}$	**1.97**	1.87	1.63	1.58	1.24
$\mathbb{P}$	11.11	7.66	**21.97**	12.78	8.08

We believe that this category has a low number of questions because energy consumption, as an emerging concern, has not yet firmly established itself in modern software engineering practices outside of specific domains such as embedded systems. On the other hand, the high number of questions in *Noise* category shows that, although energy consumption is not the primary software requirement, programmers usually refer to it as a desirable requirement. Usually, programmers are interested in improving energy consumption's closest relative: performance. Another point regarding the *Code Design* category is that it has the highest number of answers per question (A/Q ratio: 3.96), which suggests that developers are strongly interested in this kind of questions. However, we found 3 outlier questions with a high number of answers. Together, these three questions have 43% of the overall answers in this category. Analyzing those questions, we have found that they have a high number of answers because they are (i) challenging[10][11] and (ii) polemic[12]. Despite the low number of questions, we believe that the *Code Design* category is the most important one for future energy-aware software development. Thus, we dedicated Section 3.3.4 to describe this category in greater detail.

Second, we observed that the success rate can vary significantly. For instance, *Code Design* is the category that has more accepted answers (72.22%), and *Measurements* is the worst (32.22%). Analyzing the answers to those questions, we observed that, for the *Code Design* category, an accepted answer usually provides a number suggestions, and one of them may work for the programmer who asked the question. In contrast, energy consumption measurement is not always straightforward due of the lack of specialized tools. Programmers often want to measure energy consumption at different levels of granularity. We observed a number of

[10] www.stackoverflow.com/questions/61882
[11] www.stackoverflow.com/questions/422539
[12] www.stackoverflow.com/questions/1318851

granularities, including hardware devices (USB, Camera, GPS), network sensors (wifi, 3G, bluetooth), operating systems, kernel, virtual machine, process, thread, and application (line, method, and whole program). Although it is easy to create such questions, answering them is not. For most of these granularities, there is no standard mainstream hardware device, tool or technique available. In fact, only recent studies propose methods to estimate energy consumption at the source line of code level (HAO et al., 2013; LI et al., 2013). At the same time, STACKOVERFLOW users are clearly interested in "Measurement" questions. It is the category with the highest number of views per question (1.97 times the average STACKOVERFLOW question).

Third, the rate for unanswered questions (the % of unsuccessful questions) is often less than 20%, with a small variation (SD: 8.18). Only for the *Measurement* and *Noise* categories, the rate for unanswered questions is greater than 20%. We have explained why the *Measurement* has a high percentage of unsuccessful questions. Since questions in the *Noise* category are not mainly about energy consumption, the answers are not as well. Thus, the low answer rate is not directly related to energy consumption issues. Finally, the popularity (Section 3.3.1) of the questions in each theme varies from 7.66 to 21.97 (SD: 5.79). Again, the *Code Design* category has the highest values for score, answers, comments, and favorites (variables $\mathbb{S}$, $\mathbb{A}$, $\mathbb{C}$ and $\mathbb{F}$, respectively). In fact, the value of the $\mathbb{F}$ variable for *Code Design* is more than twice higher than the value of $\mathbb{F}$ for the *General Knowledge* category, which has the second highest value. This result might suggest that, although a small number of users are asking these questions, they are more interesting than any other kind of energy consumption questions.

3.3.3 RQ3: According to developers, what are the main causes for software energy consumption?

Analyzing the answers from the base group, we identified seven common themes regarding the sources of energy consumption. Table 3.6 summarizes these themes with their occurrences.

Table 3.6: The causes that STACKOVERFLOW users believe to impact on energy consumption.

Cause	# Occurrences
Unnecessary resource usage	49
Faulty GPS behavior	42
Background activities	40
Excessive synchronization	32
Background wallpapers	17
Advertisement	11
High GPU usage	8

Unnecessary resource usage. When one resource is not being used, the programmer should power it down (or put it to sleep). We found 49 occurrences of this theme in our data. A resource running at full power but sitting unused can represent a powerful source of energy drain. Indeed, a user suggested, "*to have a background application that monitors device usage, identifies unused/idle resources, and acts appropriately*"[13]. However, this suggestion might consume extra energy by polling the device, and by creating new background activities.

Faulty GPS behavior. STACKOVERFLOW users (42 occurrences) suggest that application programmers should take special care when dealing with GPS features. As a user pointed out "*[..] Make sure that the GPS is only on for the bare minimum of time. Of course, when there are bugs that are introduced that keep the GPS turned on too long they go to the top of the list to get fixed*"[14]. Another user also suggested to "*avoid using fine grain location where a coarse location would do (GPS vs cellular location)*"[15]. In fact, GPS sensors are known for their high impact on energy consumption (ZHUANG; KIM; SINGH, 2010).

Background activities. Some users (40 occurrences) highlight unnecessary background activities, such as performing HTTP connections or changing some device features, as another important source of energy consumption problems. When combined, such activities might represent an important energy drain. The application programmer could avoid such background activities by debugging her application, or by providing features to turn on/off such activities. As a STACKOVERFLOW user has pointed, "*Inefficient background activity has a dramatic impact on system performance, power consumption, responsiveness, and memory footprint*"[16].

Excessive synchronization. STACKOVERFLOW users (32 occurrences) suggest that synchronizing states between thread/process might increase energy consumption. For example "*I need to maintain the data updated, according to the modify that may happens server-side. [...] If new data are available, other web service is called for obtaining them. This solution works fine for my [...], but this solution is too expensive in term of energy consumption [...].*"[17].

Background wallpapers. STACKOVERFLOW users (17 occurrences) have pointed out that wallpapers in mobile applications, may incur significant energy consumption. It could be even worse when the wallpaper uses animations at a high frame rate. Also, colors play an important role in energy consumption. STACKOVERFLOW users agree that the white color is the most energy inefficient, whereas black is the most energy efficient. As a STACKOVERFLOW user said "*To give you an idea, a static blue wallpaper (for instance a jellyfish in an aquarium) consumes more battery than the 3D galaxy live wallpaper*"[18].

Advertisement. According to STACKOVERFLOW users (11 occurrences), advertisement can increase energy consumption in mobile applications. "*to send advertisements, just turns on*

[13] www.stackoverflow.com/questions/3092498
[14] www.stackoverflow.com/questions/4361967
[15] www.stackoverflow.com/questions/4361967
[16] www.stackoverflow.com/questions/422539
[17] www.stackoverflow.com/questions/14997997
[18] www.stackoverflow.com/questions/2902382

briefly to do a quick Tx/Rx. When scanning, the Rx is turned on for a relatively long time because you don't know at what time or on which frequency advertisements will arrive"[19]. A recent study has discussed the impact of advertisement on different mobile applications (WILKE et al., 2013). The authors showed savings of up to 75% in applications that do not use advertisement.

High GPU usage. STACKOVERFLOW users (8 occurrences) believe that GPUs waste more energy than CPUs. The rationale behind this intuition is that GPUs usually have more cores (and more cooling devices as well), as stated by a user: "*[. . .] The higher the clock rate, the more power used. GPUs tend to have high clock rates so they can do lots of work;* "[20].

3.3.4 RQ4: What solutions do developers employ or recommend to save energy

We identified that only 11.07% (36 out of 325) of the questions ask about solutions to improve the energy consumption by doing modifications at the code level. Table 3.7 shows the themes and occurrences. However, it is interesting to observe that some respondents seem to know that a particular piece of code might have a substantial impact on energy consumption, which is not straightforward. Only recent studies have focused on understanding the relationship between code design and energy consumption in high-level applications (KWON; TILEVICH, 2013; PINTO; CASTOR, 2013; SAHIN et al., 2012). We hypothesize that those STACKOVERFLOW users are experiencing significant battery drain while introducing new features.

We found a total of 8 themes in our qualitative analysis. We identified these themes by analyzing the answers about reducing the energy consumption through modifications at the code design level. We chose these themes due to the frequency with which they appear in our data. We briefly describe each theme next:

Keep IO to a minimum. STACKOVERFLOW users (29 occurrences) believe that accessing external devices increases energy consumption. In particular, this could be even more detrimental to mobile devices, which perform IO operations frequently via bluetooth, wifi,

[19] www.stackoverflow.com/questions/13584367
[20] www.stackoverflow.com/questions/12332609

Table 3.7: The solutions that STACKOVERFLOW users suggest in order to save energy.

Solution	# Occurrences
Keep IO to a minimum	29
Bulk operations	24
Avoid Polling	17
Hardware Coordination	11
Concurrent programming	9
Lazy Initialization	7
Efficient Data structure	5

camera and GPS, in the same application, concurrently. One common suggestion is "*to keep IO to a minimum*"[21].

Bulk operations. A number of users (24 occurrences) argued that, if one needs to send a sequence of commands to a hardware device, it is preferable to buffer them up and send them out all at once. In this manner, the programmer may avoid multiple wakeup → send → sleep cycles. For instance, as a user pointed out "*Don't transfer say 1 file, and then wait for a bit to do another transfer. Instead, transfer right after the other. This reduces the amount of time the radios need to be active for, and hence conserves battery life*"[22]. It also includes other IO operations, such as read/write in files, wifi transmissions, GPS usage, synchronizing data through the network, among many others.

Avoid Polling. Several users (17 occurrences) have argued that polling is not a good design choice for energy consumption. With polling, the application is continuously busy, polling the hardware to check if the desired value is available, which may become a source of wasted CPU cycles. As one user said, "*Polling is imprecise by its nature. The higher your target precision gets, the more wasteful the polling becomes*"[23]. A common suggestion is to use asynchronous communication instead. Since the recipient of asynchronous communication can remain dormant until an event occurs, an interrupt-driven approach can make efficient use of existing resources. Programmers should consider polling only if they cannot achieve the goal using interruptions, "*The best way I can think of to do this would to make an application entirely interrupt-driven*"[24].

Hardware Coordination. Some users (11 occurrences) claim that the programmer should know better the underlying hardware to save energy. Then, the programmer can (i) "*execute [the code] entirely in the processor cache, you'll have less bus activity and save power*"; (ii) "*If the processor has multiple levels of cache, try to fit in the lowest level of instruction or data cache possible.*"; (iii) "*if the code needs to use external memory, try to use it as little as possible*"; (iv) "*don't use [the] floating point unit or any instructions that may power up any other optional functional units unless you can make a good case that use of these instructions significantly shortens the time that the CPU is out of sleep mode*", and (v)"*Set unused memory or flash to 0xFF not 0x00*"[25].

Concurrent programming. Appropriate use of multi-threading can greatly improve application performance. Some users (9 occurrences) have pointed out that concurrency could be used to improve energy efficiency as well. For example, "*using multiple threads can save energy when you have I/O waits. One thread can wait while other threads can perform other computations; instead of having your application idle*"[26].

[21] www.stackoverflow.com/questions/2905958
[22] www.stackoverflow.com/questions/12120629
[23] www.stackoverflow.com/questions/10929875
[24] www.stackoverflow.com/questions/3866746
[25] All suggestions stated in this paragraph are in the same question: www.stackoverflow.com/questions/61882
[26] www.stackoverflow.com/questions/6925572

Lazy Initialization. Some users (7 occurrences) pointed out that lazy resource initialization could save energy, *e.g.* the programmer needs to "*draw to the screen only when necessary rather than endlessly*"[27]. The application should initialize resources only when they are strictly necessary.

Race to Idle. Some of the STACKOVERFLOW users (7 occurrences) suggest that, to save energy, the programmer should "*do the work as quickly as possible, and then go to some idle state*"[28]. The rationale behind this is that faster programs will theoretically consume less energy because they will have the machine idle faster, waiting for events (or interrupts) to happen. Moreover, modern CPUs are very energy-efficient when idle (BIRCHER; JOHN, 2008).

Efficient Data structure. Finally, some users (5 occurrences) agree that the use of high level data structures, such as maps or concurrent implementations, should be avoided when unnecessary, and simple data structures should be preferred instead. As a STACKOVERFLOW user suggested, "*Use plain C data structures (instead of Foundation objects) and pack them*"[29]. The programmer saves energy by avoiding creating new internal objects.

3.4 Discussion

In this section, we summarize our findings, and provide additional discussion on the data presented in the previous section.

3.4.1 Overall Assessment

Our study reveals distinct characteristics of energy consumption questions and their answers.

Compared with the average of the questions in STACKOVERFLOW, discussions related to energy consumption can be considered more *interesting* and *challenging*:

- *interesting*: Questions from the base group are up-voted 1.1 times more frequently than the average STACKOVERFLOW question. In the same vein, they are answered 2.67 more frequently than the average STACKOVERFLOW question, viewed 1.68 times more frequently, and marked as favorites 3.89 more frequently.

- *challenging*: even though the questions related to energy consumption are more likely to be up-voted, their answers are not (the $\mathbb{S}$ value for the answers is 1.05 times lower than STACKOVERFLOW control set). These answers are also entailed by fewer comments (the $\mathbb{C}$ value is 1.16 times lower). We have examined these answers and noticed that some of them have evident problems. We discuss them in Section 3.4.2.

[27] www.stackoverflow.com/questions/4361967
[28] www.stackoverflow.com/questions/61882
[29] www.stackoverflow.com/questions/4361967

3.4.2 Misconceptions, panaceas, and lack of tools

We identified three recurring problems in the answers in our base group of questions: (i) misconceptions about software energy consumption and how it can be reduced, (ii) solutions that are applicable in certain contexts being presented as universal, and (iii) lack of tools, in particular, measurement tools. We examine each one in turn.

First, we find 37 users holding *misconceptions*, of which we describe three. (1) Some users confuse "power" and "energy." Scaling down CPU frequencies is often suggested as a direct solution to reduce *energy* consumption, which in reality has more direct impact on *power*. Scaling down the CPU frequency is effective in saving *power*. Saving *energy* however is more complex. Since energy consumption "E" is an accumulation of power consumption "P" over time "t", that is $E = P \times t$, a reduction of frequency may increase the execution time, keeping energy constant or even increasing it. (2) StackOverflow users often use performance as the primary indicator to estimate energy consumption, such as "*It may reduce execution time since HW accel is there, therefore power consumption may be lower*"[30]. According to numerous studies (CAO et al., 2012; ESMAEILZADEH et al., 2012; TREFETHEN; THIYAGALINGAM, 2013; PINTO; CASTOR, 2013; RANGAN; WEI; BROOKS, 2009), power and performance are not always correlated. (3) Some StackOverflow users believe that switching to a managed language runtime, such as Java or .NET, might improve energy consumption, such as "*My take is that the best way is to mix languages, use existing Java-based infrastructural tools [...] to build the performance critical parts and something more nimble to build complex but not that heavy business and presentation logic.*"[31]. However, recent studies show that energy consumption is heavily workload-dependent (ESMAEILZADEH et al., 2012). Also, managed languages have to deal with the energy consumption cost of their underlying services. Another recent study points out that the total energy consumption in VM services ranges from 9% to 82% of the overall energy consumption of an application (CAO et al., 2012). Curiously, none of these answers were "down-voted."

Second, we identify some generic *panaceas* offered (by 23 users) as solutions to energy problems. They are not necessarily incorrect, but the generic folklore nature of such answers hardly qualifies them as problem solvers. For instance, one user asked "*[...] given this how can I write the code 'power' efficiently so as to consume minimum watts?*"[32]. Some suggested answers include:

- *Interrupts are your friends; Polling / wait() aren't your friends*
- *Do as little as possible*
- *Make your code as small/efficient as possible*

[30] www.stackoverflow.com/questions/9924255
[31] www.stackoverflow.com/questions/1318851
[32] www.stackoverflow.com/questions/61882

- *Turn off as many modules, pins, peripherals as possible in the micro*
- *Run as slowly as possible*

Part of the vagueness here is innate to online forums. Nonetheless, many other common questions frequently asked by STACKOVERFLOW users, such as *How to avoid memory leaks?* or *How to fix null pointer bugs?*, are often entailed by much more concrete and useful answers.

Third, we believe the *lack of tool support* poses significant hurdles to energy-aware software development. We observe that STACKOVERFLOW users are keenly aware of the multi-granularity multi-level nature of energy optimizations with diverse discussions on hardware devices, operating systems, and applications. From the 59 questions under the *Measurement* theme in Section 3.3.2, 38 of them are directly related with a lack of tool support. These programmers are faced with insufficient support of established tools, from lower-level ones for energy profiling, measurement, and estimation, to higher-level ones for energy management frameworks, software architectures, and refactoring tools. One example of it is the following question created by a STACKOVERFLOW user: "*JouleMeter is a tool from Microsoft for measuring power consumption by different processes on a windows machine. Please tell me if there is any similar tool on linux for getting information of energy consumption by different processes and applications on linux machine. Also I am looking for an open-source solution.*"[33], which has no answer.

3.4.3 Do researchers agree with the code design suggestions?

In Section 3.3.4 we categorized 8 strategies that STACKOVERFLOW users suggested as a way to save software energy consumption. In this section, we compare these strategies with the state of the art, in order to verify whether they are supported.

Keep IO to a minimum. A recent study showed that an IO-intensive benchmark could produce high energy consumption (PINTO; CASTOR, 2013). Similarly, network communication is a major consumer of energy (PENTIKOUSIS, 2010), consuming between 10% and 50% of the total energy budget of a typical mobile application. Thus, reducing IO might save energy.

Bulk operations. This strategy could save energy by reducing IO overhead and network communication (KWON; TILEVICH, 2013). However, the degree of batching should be determined by the network conditions in place. For example, for a network with limited bandwidth (e.g., 3G or 4G), batching too many individual remote communications can saturate the network, which in turn can increase the aggregate latency of remote interactions, thus incurring high energy costs.

Hardware Coordination. Several authors support the idea of using memory optimization techniques in order to reduce the bus switching activity (BRANDOLESE et al., 2002; TIWARI; MALIK; WOLFE, 1994; GE et al., 2007; HAO et al., 2013). Other software-oriented

[33] www.stackoverflow.com/questions/13286662

proposals focus on instruction scheduling and code generation, possibly minimizing memory access cost (FARKAS et al., 2000).

Concurrent programming. Several authors (DAYLIGHT et al., 2002; LIU, 2012a; PINTO; CASTOR, 2013; RANGAN; WEI; BROOKS, 2009; SOLERNOU et al., 2013) have been studying the relationship between concurrent programming, performance and energy, but no consensus has emerged from it. For example, LIU (2012a) conducted an initial study on the energy consumption of a number of synchronization approaches. Furthermore, PINTO; CASTOR (2013) investigated a group of high level concurrent constructs in different scenarios in order to identify when it is possible to switch from one construct to another. In both studies the authors suggest that concurrency cannot be seen as a one-size-fits-all solution.

Race to Idle. Race to idle is the assumption that, the faster the program executes, the more energy efficient it will be. Most of this belief comes from the use performance as a proxy for energy consumption. However, this is not an accurate approximation. For instance, mobile devices and modern CPUs, scale their voltage dynamically, which makes it more difficult to say beforehand which computation will consume less energy (LI et al., 2013). Also, hardware components have varied energy consumption patterns (HAO et al., 2013). Concurrent software plays yet another role: it has more complex behavior and overheads. Recent studies have shown that the race to idle principle does not always hold for multi-threaded applications. (PINTO; CASTOR, 2013; SOLERNOU et al., 2013).

Efficient Data structures. The idea of energy efficient data structures is not new. DAYLIGHT et al. (2002) discussed some energy consumption patterns and trade-offs faced during the implementation of non-trivial energy efficient data structures. The authors showed gains in energy consumption with a relatively small overhead.

As for the remaining solutions proposed by the respondents (Section 3.3.4), we did not find studies in the research literature that directly support them, although there are related ones. For example, SAHIN et al. (2012) analyzed the impact of some well-known design patterns on energy consumption, and MENARINI et al. (2013) provided important clues on how different e-services impact on energy consumption. We describe four more themes, from the researchers perspective, that we consider valuable for the development community.

Loop transformations. In BRANDOLESE et al. (2002), the authors suggests 8 loop transformations to save energy. Some of those include (i) *loop fusion*: merges different loops to reduce control operations, and (ii) *loop interchange*: modifies the nesting ordering of the loops to change array accesses. Even though some of these can be achieved by automatic compiler optimization, programmers can greatly help in this area to compensate for the fundamentally conservative nature of compiler optimizations.

Data compression. WILKE et al. (2013) showed that compressed image formats may save energy (*e.g.*, when transferring images through networks). With compressed image formats, programmers decrease the overall image size, whereas maintaining reasonably image quality.

Offloading methods. Few answers (4 occurrences) mention offloading techniques

to save energy. When developing a mobile application, sending heavy calculations to a server (KWON; TILEVICH, 2013) has the potential to save energy. However, a threshold should be defined when using this approach: if the calculation is too fast, it might be better to run it locally, rather than incurring the cost of network communication.

Approximated programming. Many software components can tolerate occasional "soft errors", *i.e.*, errors that may reduce the accuracy of the results. This technique might bring the benefits of fast and energy-efficient execution at the price of some controlled approximation (CARBIN et al., 2012).

3.5 Threats to Validity

We divide our threats to validity discussion in terms of internal ones and external ones.

Internal: First, our study is restricted by the size of our dataset. Despite having mined the entire STACKOVERFLOW site, we only found 325 questions. Moreover, a number of studies have been conducted using STACKOVERFLOW in recent years (MORRISON; MURPHY-HILL, 2013; WANG; LO; JIANG, 2013). In fact, STACKOVERFLOW is the most widely used Q&A website in the software development world. Studies show that STACKOVERFLOW users are greatly diverse with relation to their skills and age (MORRISON; MURPHY-HILL, 2013). Second, not all questions that we find through querying the database using energy consumption keywords are related to energy consumption. We minimize the false-positive rate by investigating all questions and answers manually. Still, in order to reduce the number of false-negatives, we experimented a number of energy consumption-related keywords. Most of the keywords used were based on the study of WILKE et al. (2013). We examined each one of them and kept only the keywords that retrieved relevant questions. Third, we choose Thematic Analysis as our research method. While we achieved saturation (*i.e.*, no more themes could be absorbed) regarding the topics we focused on in our research, there may be additional themes that might add new insights. We also share our raw data (extracted questions and answers) to provide a means for replication and verification.

External: Our results only apply to application programmers interested in energy consumption on STACKOVERFLOW. It does not cover systems programmers, nor application programmers that use other Q&A websites. However, even though the results of this work might not be generalizable, we believe that our data source is reliable. Since no user has created more than one question and no user has produced more than three answers, our data demonstrates diversity. Finally, we are only considering English questions and answers.

3.6 Summary

In this chapter, we investigated how STACKOVERFLOW users are interested in software energy consumption. We observed that energy consumption questions are more *interesting* and

also more *challenging* than the average STACKOVERFLOW question. We identify 5 main themes regarding energy consumption questions, namely: *Measurement*, *General Knowledge*, *Code Design*, *Context Specific* and *Noise*. Questions that focus on source code modifications receive more attention (popularity and answers). We identify 7 major causes for energy consumption problems, varying from background activities to synchronization. We also discuss how to address each one of them and, when possible, compare with the state-of-the-art software energy consumption research.

4

The Energy Efficiency of Java Thread-Safe Collections

Bad programmers worry about the code.
Good programmers worry about data structures and their relationships.
—LINUS TORVALDS

In this chapter, we present an empirical study evaluating the performance and energy consumption characteristics of 16 Java collection implementations grouped by 3 well-known interfaces: `List`, `Set`, and `Map`. We start by providing a brief overview of the problem (Section 4.1). Section 4.2 describes our methodology, describing both benchmarks and environment. Finally, Section 4.3 provides our results.

4.1 Overview

A question that often arises in software development forums is: "since Java has so many collection implementations, which one is more suitable to my problem?"[1]. Answers to this question come in different flavors: these collections serve for different purposes and have different characteristics in terms of performance, scalability and thread-safety. Developers should consider these characteristics in order to make judicious design decisions about which implementation best fits their problems. In this chapter, we turn our focus to *energy efficiency*. Energy consumption estimation tools do exist (LI et al., 2013; SEO; MALEK; MEDVIDOVIC, 2008b; LIU; PINTO; LIU, 2015), but they do not provide direct guidance on *energy optimization*, *i.e.*, bridging the gap between understanding where energy is consumed and understanding how the code can be modified in order to reduce energy consumption. With no other option, developers have to rely on conventional wisdom, consult software development forums and blogs, or simply search online for "tips and tricks". However, many of these guidelines are often

[1] `http://stackoverflow.com/search?q=which+data+structure+use+java+is:question`

anecdotal or even incorrect (PINTO; CASTOR; LIU, 2014b).

In this chapter, we elucidate one important area of the application-level optimization space, focusing on understanding the energy consumption of different Java collections running on parallel architectures. This is a critical direction at the junction of data-intensive computing and parallel computing, which deserves more investigation due to at least three reasons:

- Collections are one of the most important building blocks of computer programming. Multiplicity — a collection may hold many pieces of data items — is the norm of their use, and it often contributes to significant memory pressure — and performance problems in general — of modern applications where data are often intensive (XU, 2013; BU et al., 2013).
- Not only high-end servers but also desktop machines, smartphones and tablets need concurrent programs to make best use of their multi-core hardware. A CPU with more cores (say 32) often consumes more power than one with fewer cores (say 1 or 2) (LI; MARTÍNEZ, 2005).
- Mainstream programming languages often provide a number of implementations for the same collection and these implementations have potentially different characteristics in terms of energy efficiency.

To gain confidence in our results in the presence of platform variations and measurement environments, we employ two machines with different architectures (a 32-core AMD vs. a 16-core Intel). We further use two distinct energy measurement strategies: an external energy meter, and Machine-Specific Registers (MSRs). Our research is motivated by the following questions:

RQ1. Do different implementations of the same collection have different impacts on energy consumption?

RQ2. Do different operations in the same implementation of a collection consume energy differently?

RQ3. Do collections scale, from an energy consumption perspective, with an increasing number of concurrent threads?

RQ4. Do different collection configurations and usages have different impacts on energy consumption?

The goal of this study is to answer these research questions. In order to answer **RQ1** and **RQ2**, we select and analyze the behaviors of three common operations — traversal, insertion and removal — for each collection implementation. To answer **RQ3**, we analyze how different implementations scale in the presence of multiple threads. In this experiment, we cover the

spectrum including both under-provisioning (the number of threads exceeds the number of CPU cores) and over-provisioning (the number of CPU cores exceeds the number of threads). In **RQ4**, we analyze how different configurations — such as the load factor and the initial capacity of the collection — impact energy consumption.

Our study produces a list of interesting findings, some of which are not obvious. We summarize them in Section 5.4. To highlight one of them, our experiments show that execution time is not always a reliable indicator for energy consumption. This is particularly true for various `Map` implementations. In other words, the consumption of power — the rate of energy consumption — is not a constant across different collection implementations.

4.2 Study Setup

In this section we describe the benchmarks that we analyzed, the infrastructure and the methodology that we used to perform the experiments.

4.2.1 Benchmarks

The benchmarks used in this study consist of 16 commonly used collections available in the Java programming language. Our focus is on the thread-safe implementations of the collection. Hence, for each collection, we selected a single non-thread-safe implementation to serve as a baseline. For each implementation, we analyzed insertion, removal and traversal operations. We grouped these implementations by the logical collection they represent, into three categories:

`Lists (java.util.List)`: `Lists` are ordered collections that allow duplicate elements. Using this collection, programmers can have precise control over where an element is inserted in the list. The programmer can access an element using its index, or traverse the elements using an `Iterator`. Several implementations of this collection are available in the Java language. We used `ArrayList`, which is not thread-safe, as our baseline. We studied the following thread-safe `List` implementations: `CopyOnWriteArrayList`, `Vector`, and `Collections.synchronizedList()`. The main difference between the latter two is their usage pattern in programming. With `Collections.synchronizedList()`, the programmer creates a wrapper around the current `List` implementation, and the data stored in the original `List` object does not need to be copied into the wrapper object. It is appropriate in cases where the programmer intends to hold data in a non-thread-safe `List` object, but wishes to add synchronization support. With `Vector`, on the other hand, the data container and the synchronization support are unified so it is not possible to keep an underlying structure (such as `LinkedList`) separate from the object managing the synchronization. `CopyOnWriteArrayList` creates a copy of the underlying `ArrayList` whenever a mutation operation (*e.g.*, using the `add` or `set` methods) is invoked.

`Maps (java.util.Map)`: `Maps` are objects that map keys to values. Logically, the keys of a map cannot be duplicated. Each key is uniquely associated with a value. An insertion of a (key, value) pair where the key is already associated with a value in the map results in the old value being replaced by the new one. Our baseline thread-unsafe choice is `LinkedHashMap`, instead of the more commonly used `HashMap`. This is because the latter sometimes caused non-termination during our experiments[2]. Our choice of thread-safe `Map` implementations includes `Hashtable`, `Collections.synchronizedMap()`, `ConcurrentSkipListMap`, `ConcurrentHashMap`, and `ConcurrentHashMapV8`. `ConcurrentHashMap` and `ConcurrentHashMapV8` are different in the sense that the latter is an optimized version released in Java 1.8, while the former is the version present in the JDK until Java 1.7. While all `Map` implementations share similar functionalities and operate on a common interface, they are particularly known to differ in the order of element access at iteration time. For instance, while `LinkedHashMap` iterates in the order in which the elements were inserted into the map, a `Hashtable` makes no guarantees about the iteration order.

`Sets (java.util.Set)`: As its name suggests, the `Set` collection models the mathematical set abstraction. Unlike `Lists`, `Sets` do not count duplicate elements, and are not ordered. Thus, the elements of a set cannot be accessed by their indices, and traversals are only possible using an `Iterator`. Among the available implementations, we used `LinkedHashSet`, which is not thread-safe, as our baseline. Our selection of thread-safe `Set` implementations includes `ConcurrentSkipListSet`, `ConcurrentHashSet`, `CopyOnWriteArraySet`, `ConcurrentHashSetV8`, and `Collections.synchronizedSet()`. It should be noted that both `ConcurrentHashSet` and `ConcurrentHashSetV8` are not top-level classes readily available in the JDK library. Instead, method `Collections.newSetFromMap()`, which is available in the JDK, provides its implementation. The returned `Set` object observes the same ordering as the underlying map.

4.2.2 Experimental Environment

To gain confidence in our results in the presence of platform variations, we run each experiment on two significantly different platforms:

- `System#1`: A 2×16-core AMD Opteron 6378 processor (Piledriver microarchitecture), 2.4GHz, with 64GB of DDR3 1600 memory. It has three cache levels (L1, L2, L3): L1 with 32KB per core, L2 with 256KB per core, and L3 20480 (Smart cache). It is running Debian 3.2.46-1 x86-64 Linux (kernel 3.2.0-4-amd64), and Oracle HotSpot 64-Bit Server VM (build 21) JDK version 1.7.0_11.

[2]A possible explanation can be found here: `http://mailinator.blogspot.com/2009/06/beautiful-race-condition.html`

- `System#2`: A 2×8-core (32-cores when hyper-threading is enabled) Intel(R) Xeon(R) E5-2670 processor, 2.60GHz, with 64GB of DDR3 1600 memory. It has three cache levels (L1, L2, L3): L1 with 48KB per core, L2 with 1024KB per core, and L3 20480 (Smart cache). It is running Debian 6 (kernel 3.0.0-1-amd64) and Oracle HotSpot 64-Bit Server VM (build 14), JDK version 1.7.0_71.

When we performed the experiments with `Sets` and `Maps`, we employed the jsr166e library[3], which contains the `ConcurrentHashMapV8` implementation. Thus, these experiments do not need to be executed under Java 1.8.

We also used two different energy consumption measurement approaches. For `System#1`, energy consumption is measured through current meters over power supply lines to the CPU module. Data is converted through an NI DAQ and collected by NI LabVIEW SignalExpress with 100 samples per second and the unit of the current sample is *deca-ampere* (10 ampere). Since the supply voltage is stable at 12V, energy consumption is computed as the sum of current samples (100 samples per second, or 0.01) multiplied by 10, that is $12 \times 0.01 \times 10$. We measured the "base" power consumption of the OS when there is no JVM (or other application) running. The reported results are the measured results *modulo* the "base" energy consumption.

For `System#2`, we have used `jRAPL` (LIU; PINTO; LIU, 2015), which is a framework that contains a set of APIs for profiling Java programs running on CPUs with Running Average Power Limit (RAPL) (DAVID et al., 2010) support. Due to architecture design, the RAPL support for `System#2` can access CPU core, CPU uncore data (*i.e.* caches and interconnects), and in addition DRAM energy consumption data. As we shall see in the experiments, DRAM power consumption is nearly constant. In other words, even though our meter-based measurement strategy only considers the CPU energy consumption, it is still indicative of the relative energy consumptions of different collection implementations. It should be noted that the stability of DRAM power consumption through RAPL-based experiments does not contradict the established fact that the energy consumption of memory systems is highly dynamic (BENINI; BOGLIOLO; DE MICHELI, 2000). In that context, memory systems subsume the entire memory hierarchy, and most of the variations are caused by caches (PAPAMARCOS; PATEL, 1984) — part of the "CPU uncore data" in our experiments.

All experiments were performed with no other load on the OS. We conform to the default settings of both the OS and the JVM. Several default settings are relevant to this context: (1) the power management of Linux is the default *ondemand* governor, which dynamically adjusts CPU core frequencies based on system workloads. (2) For the JVM, the parallel garbage collector is used, and just-in-time (JIT) compilation is enabled. The initial heap size and maximum heap size are set to be 1GB and 16GB respectively. We run each benchmark 10 times within the same JVM; this is implemented by a top-level 10-iteration loop over each benchmark. The reported data is the average of the last 3 runs. We chose the last three runs because, according to a recent

[3]Source code available at: `http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/jsr166e/`

study, JIT execution tends to stabilize in the latter runs (PINTO; CASTOR; LIU, 2014a). The standard deviation also supports this decision. We experienced differences in standard deviation of over 30% when comparing the warmup run (first 3 executions) and later runs, but less than 5% when comparing the last 3 runs. Hyper-threading is enabled and Turbo Boost feature is disabled on `System#2`.

4.3 Study Results

In this section, we report the results of our experiments. Results for **RQ1** and **RQ2** are presented in Section 4.3.1, describing the impact of different implementations and operations on energy consumption. In Section 4.3.2 we answer **RQ3** by investigating the impact of accessing data collections with different numbers of threads. Finally, in Section 4.3.3 we answer **RQ4** by exploring different “tuning knobs" of data collections.

4.3.1 Energy Behaviors of Different Collection Implementations and Operations

For **RQ1** and **RQ2**, we set the number of threads to 32 and, for each group of collections, we performed and measured insertion, traversal and removal operations.

- For the insertion operation, we start with an empty data collection, and have each thread insert 100,000 elements. Hence, at the end of the insertion operation, the total number of elements inside the collection is 3,200,000. To avoid duplicate elements, each insertion operation adds a `String` object with value *thread-id* + *“-"* + *current-index*.

- For the traversal operation, each thread traverses the entire collection generated by the insertion operation, *i.e.*, over 3,200,000 elements. On `Sets` and `Maps`, we first get the list of keys inserted, and then we iterate over these keys in order to get their values. On `Lists`, the traversal operation is performed using a top-level loop over the collection, accessing each element by its index using the `E get(int i)` method of each collection class, where `E` is the generic type (instantiated to `String` in our experiments).

- For the removal operation, we start with the collection with 3,200,000 elements, and remove the elements one by one. For `Maps` and `Sets`, the removals are based on keys, and we remove until the collection becomes empty. On `Lists`, however, the removal operation is based on indexes, and occurs *in-place* — that is, we do not traverse the collection to look up for a particular element before removal.

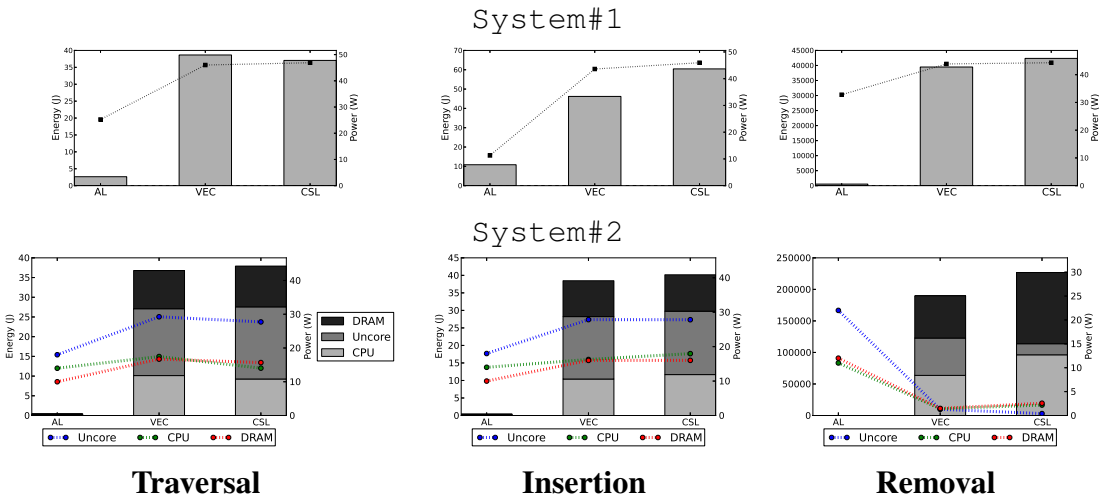


Figure 4.1: Energy and power results for traversal, insertion and removal operations for different `List` implementations. Bars denote energy consumption and lines denote power consumption.

> Here, according to the `List` documentation, the `E remove(int i)` method "*removes the element at the specified position in this list (optional operation). Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.*"[4]. As we shall see, removal operations on `Lists` are excessively expensive. In order to make it feasible to perform all experiments, we chose to remove only half of the elements.

Lists. Figure 4.1 shows the energy consumption (bars) and power consumption (lines) results of our `List` experiments. Each bar represents one `List` implementation. `AL` means `ArrayList`, `VEC` means `Vector`, and `CSL` means `Collections.synchronizedList()`. The three graphs at top of the figure are collected from `System#1`, whereas the three graphs in the bottom are from `System#2`. We do not show the figures for `CopyOnWriteArrayList` because the results for insertion and removal are an outlier and would otherwise skew the proportion of the figures.

First, we can observe that synchronization does play an important role here. As we can see, `ArrayList`, the non-thread-safe implementation, consumes much less energy than the other ones, thanks to its lack of synchronization. `Collection.synchronizedList()` and `Vector` are similar in energy behaviors. The greatest difference is seen on insertion, on `System#1`, in which `Collection.synchronizedList()` consumed about 24.21% less energy than `Vector`. `Vector` and `Collection.synchronizedList()` are strongly correlated in their implementations, with some differences. While both of them are thread-safe on insertion and removal operations, `Collection.synchronizedList()` is not thread-safe on traversals, when performing through an `Iterator`, whereas `Vector` is thread-safe on

[4]Documentation available at `http://docs.oracle.com/javase/7/docs/api/java/util/List.html#remove(int)`

the `Iterator`. In contrast, the `CopyOnWriteArrayList` implementation is thread-safe in all operations. However, it does not need synchronization on traversal operations, which makes this implementation more efficient than the thread-safe ones (it consumes 46.38x less energy than `Vector` on traversal).

Furthermore, different operations can have different impacts. As we can see on traversal, the `Vector` implementation presents the worst result among the benchmarks: it consumes 14.58x more energy and 7.9x more time than the baseline on `System#1`(84.65x and 57.99x on `System#2`, respectively). This is due both `Collection.synchronizedList()` and `Vector` implementations need to synchronize in traversal operations. As mentioned contrast, the `CopyOnWriteArrayList` implementation is more efficient than the thread-safe implementation.

For insertion, `ArrayList` consumes the least energy for `System#1` and `System#2`. `Collections.synchronizedList()`, on the thread-safe side, consumes 1.30x more energy than `Vector` (1.24x for execution time) on `System#1`. On `System#2`, however, they consume barely the same amount of energy (`Vector` consumes 1.01x less energy than `Collections.synchronizedList()`). `CopyOnWriteArrayList`, on the other hand, consumes a total of 6,843.21 J, about 152x more energy than `Vector` on `System#1`. This happens because, for each new element added to the list, the `CopyOnWriteArrayList` implementation needs to synchronize and create a fresh copy of the underlying array using the `System.arraycopy()` method. As discussed elsewhere (PINTO; CASTOR; LIU, 2014a; DE WAEL; MARR; VAN CUTSEM, 2014), even though the `System.arraycopy()` behavior can be observed in sequential applications, it is more evident in highly parallel applications, when several processors are busy making copies of the collection, preventing them from doing important work. Although this behavior makes this implementation thread-safe, it is ordinarily too costly to maintain the collection in a highly concurrent environment where insertions are not very rare events.

Moreover, removals usually consumes much more energy than the other operations. For instance, removal on `Vector` consumes about 778.88x more energy than insertion on `System#1`. Execution time increases similarly, for instance, it took about 92 seconds to complete a removal operation on `Vector`. By way of contrast, insertions on a `Vector` takes about 1.2 seconds. We believe that several reasons can explain this behavior. First, the removal operations need to compute the size of the collection in each iteration of the for loop and, as we shall see in Section 4.3.4, such naive modification can greatly impact both performance and energy consumption. The second reason is that each call to the `List.remove()` method leads to a call to the `System.arrayCopy()` method in order to resize the `List`, since all these implementations of `List` are built upon arrays. In comparison, insertion operations only lead to a `System.arrayCopy()` call when the maximum number of elements is reached.

Power consumption also deserves attention. Since `System.arrayCopy()` is a memory intensive operation, power consumption decreases, and thus, execution time increase. More-

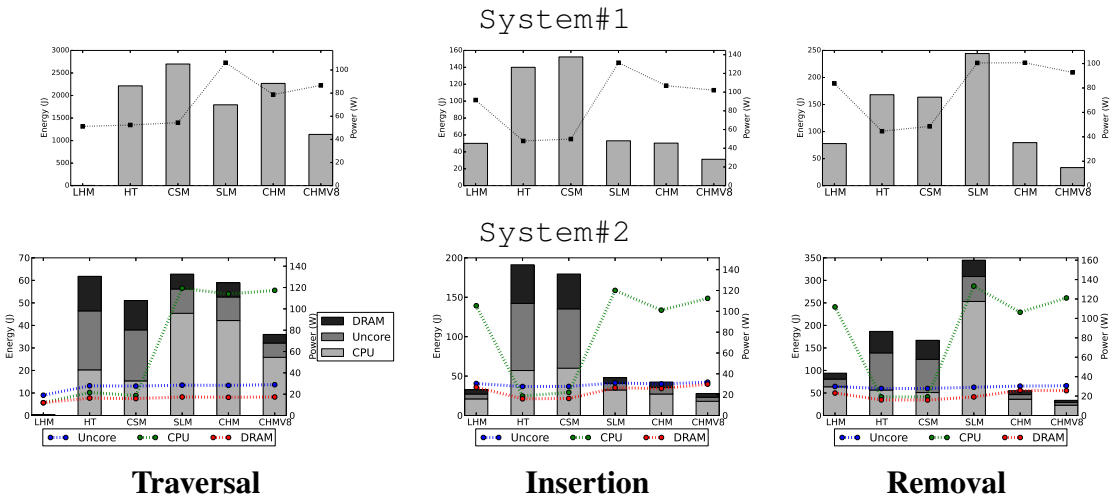


Figure 4.2: Energy and power results for traversal, insertion and removal operations for different `Map` implementations. Bars mean energy consumption and line means power consumption.

over, for most cases, power consumption follows the same shape of energy. Since energy consumption is the product of power consumption and time, when power consumption decreases and energy increase, execution time tends to increase. This is what happens on removal on `System#2`. The excessive memory operations on removals, also observed on DRAM energy consumption (the black top-most part of the bar), prevents the CPU to do useful work, which increases the execution time. Also, since the removal operation on `System#2`takes about twice of the time needed by `System#1`, power consumption drops.

We also observed that the baseline benchmark on `System#2` consumes the least energy when compared to the baseline on `System#1`. We atribute that to our energy measurement approaches. While RAPL-based measurement can be efficient in retrieving only the necessary information (for instance, package energy consumption), our hardware-based measurement gathers energy consumption information pertaining to everything that happens in the CPU. Such noise can be particularly substantial when the execution time is small.

For all aforementioned cases, we observed that energy follows the same shape as time. At the first impression, this finding might seem to be “boring". However, recent studies have observed that energy and time are often not correlated (LI et al., 2013; PINTO; CASTOR; LIU, 2014a; TREFETHEN; THIYAGALINGAM, 2013), which is particularly true for concurrent applications. For this set of benchmarks, however, we believe that developers can safely use time as a proxy for energy, which can be a great help when refactoring an application to consume less energy. Ultimately, although we have found some differences in the results, both `System#1` and `System#2` presented a compelling uniformity.

Map. Figure 4.2 presents the results with `Map` implementations. `LSM` means `LinkedHashMap`, `HT` means `Hashtable`, `CSM` means `Collections.synchronizedMap()`, `SLM` means `ConcurrentSkipListMap`, `CHM` means `ConcurrentHashMap`, and `CHMV8` means

`ConcurrentHashMapV8`. The energy behavior for `LinkedHashMap`, `Hashtable`, and `Collections.synchronizedMap()` follows the same curve as time, for both traversal and insertion operations, on both `System#1` and `System#2`. Surprisingly, however, the same cannot be said for the removal operations. Removal operations on `Hashtable` and `Collections.synchronizedMap()` exhibited energy consumption that is proportionally smaller than their execution time for both systems. Such behavior is due to a drop on power consumption. Since such collections are single-lock based, for each removal operation, the other threads need to wait until the underling structure is rebuilt. This synchronization prevents the collection to speed-up, and also decreases power usage.

On the other hand, for the `ConcurrentSkipListMap`, `ConcurrentHashMap` and `ConcurrentHashMapV8` implementations, more power is being consumed behind the scenes. As we mentioned that energy consumption is the product of power consumption and time, if the benchmark receives a 1.5x speed-up but, at the same time, yields a three-fold increase in power consumption, energy consumption will increase twofold. This scenario is roughly what happens in traversal operations, when transitioning from `Hashtable` to `ConcurrentHashMap`. Even though `ConcurrentHashMap` produces a speedup of 1.46x over the `Hashtable` implementation on `System#1`, it achieves that by consuming 1.51x more power. As a result, `ConcurrentHashMap` consumed slightly more energy than `Hashtable` (2.38%). On `System#2`, energy consumption for `Hastable` and `ConcurrentHashMap` are roughly the same. This result is relevant mainly because several textbooks (PEIERLS et al., 2005), research papers (DIG; MARRERO; ERNST, 2009) and internet blog posts (GOETZ, 2003) suggest `ConcurrentHashMap` as the *de facto* replacement for the old associative `Hashtable` implementation. Our result suggests that the decision on whether or not to use `ConcurrentHashMap` should be made with care, in particular, in scenarios where the energy consumption is more important than performance. However, the newest `ConcurrentHashMapV8` implementation, released in the version 1.8 of the Java programming language, handles large maps or maps that have many keys with colliding hash codes more gracefully. On `System#1`, `ConcurrentHashMapV8` provides performance savings of 2.19x when compared to `ConcurrentHashMap`, and energy savings of 1.99x in traversal operations (these savings are, respectively, 1.57x and 1.61x in insertion operations, and 2.19x and 2.38x in removal operations). In addition, for insertions and removals operations on both systems, `ConcurrentHashMapV8` has performance similar or even better than the not thread-safe implementation.

`ConcurrentHashMapV8` is a completely rewritten version of its predecessor. The primary design goal of this implementation is to maintain concurrent readability (typically on the `get()` method, but also on `Iterators`) while minimizing update contention. This map acts as a binned hash table. Internally, it uses tree-map-like structures to maintain bins containing more nodes than would be expected under ideal random key distributions over ideal numbers of bins. This tree also requires an additional locking mechanism. While list traversal is always

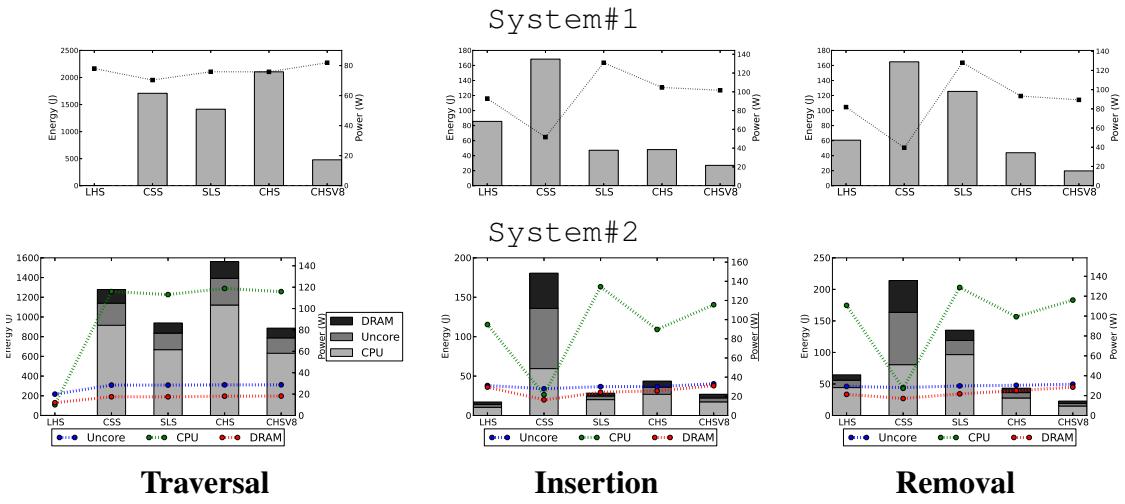


Figure 4.3: Energy and power results for traversal, insertion and removal operations for different `Set` implementations. Bars mean energy consumption and lines mean power consumption.

possible by readers even during updates, tree traversal is not, mainly because of tree-rotations that may change the root node and its links. Insertion of the first node in an empty bin is performed with a Compare-And-Set operation. Other update operations (insertional, removal, and replace) require locks. Locking support for these locks relies on builtin “synchronized" monitors.

Sets. Figure 4.3 shows the results with `Set`. `LSH` means `LinkedHashSet`, `CSS` means `Collections.synchronizedSet()`, `SLS` means `ConcurrentSkipListSet`, `CHS` means `ConcurrentHashSet`, and `CHSV8` means `ConcurrentHashSetV8`. We did not present the results for `CopyOnWriteHashSet` in this figure because it exhibited a much higher energy consumption, which made the figure difficult to read. First, for all of the implementations of `Set`, we can observe that energy consumption follows the same behavior of power on traversal operations for both `System#1` and `System#2`. However, for insertion and removal operations, they are not always proportional. Notwithstanding, an interesting trade-off can be observed when performing traversal operations. As expected, the non-thread-safe implementation, `LinkedHashSet`, achieved the least energy consumption and execution time results, followed by the `CopyOnWriteArraySet` implementation. We believe that the same recommendation for `CopyOnWriteArrayList` fits here: this collection should only be used in scenarios where reads are much more frequent than insertions. For all other implementations, the `ConcurrentHashSetV8` presents the best results among the thread-safe ones. Interestingly, for traversals, `ConcurrentHashSet` presented the worst results, consuming 1.23x more energy and 1.14x more time than `Collections.synchronizedSet()` on `System#1` (1.31x more energy and 1.19x more time on `System#2`).

Another interesting result is observed with `ConcurrentSkipListSet`, which consumes only 1.31x less energy than a `Collections.synchronizedList()` on removal operations on `System#1`, although it saves 4.25x in execution time. Such energy consumption

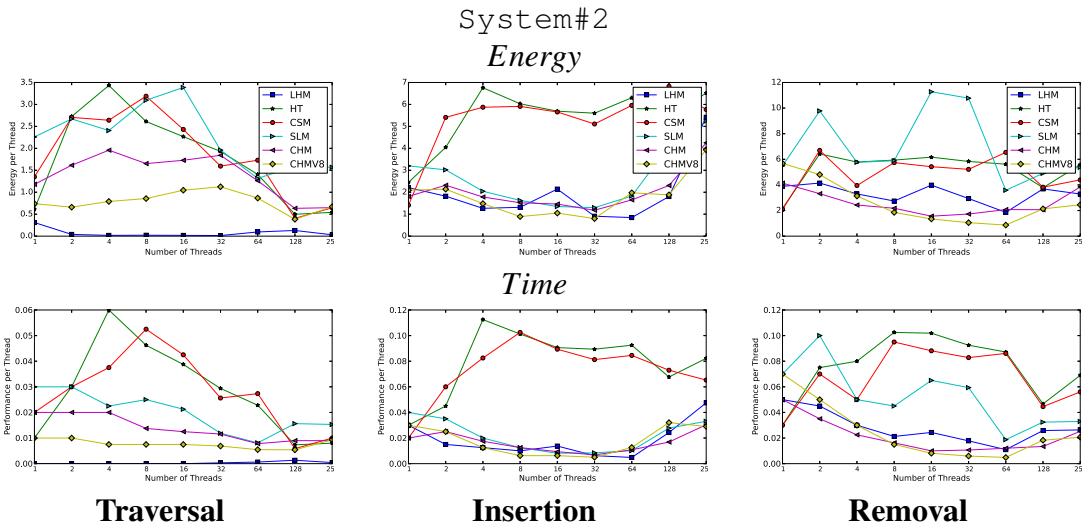


Figure 4.4: Energy consumption and execution time in the presence of concurrent threads (X axis: the number of threads, Y axis: energy consumption normalized against the number of element accesses, in joules per 100,000 elements)

overhead is also observed on `System#2`. Internally, `ConcurrentSkipListSet` relies on a `ConcurrentSkipListMap`, which is non-blocking, linearizable, and based on the compare-and-swap (CAS) operation. During traversal, this collection marks the “next" pointer to keep track of triples (predecessor, node, successor) in order to detect when and how to unlink deleted nodes. Also, because of the asynchronous nature of these maps, determining the current number of elements (used in the `Iterator`) requires a traversal of all elements. These behaviors are susceptible to create the energy consumption overhead observed in Figure 4.3.

4.3.2 Energy Behaviors with Different Number of Threads

In this group of experiments, we aim to answer **RQ3**. For this experiment, we chose `Map` implementations only, due to presence of both single-lock and high-performatic implementations. We vary the number of threads (1, 2, 4, 8, 16, 32, 64, 128, and 256 concurrent threads) and study how such variations impact energy consumption. An increment in the number of threads also increments the total number of elements inside the collection. Since each thread inserts 100,000 elements, when performing with one thread, the total number of elements is also 100,000. When performing with 2 threads, the final number of elements is 200,000, and so on. To give an impression on how `Map` implementations scale in the presence of multiple threads, Figure 4.4 demonstrates the effect of different thread accesses on benchmarks.

In this figure, each data point is normalized by the number of threads, so it represents the energy consumption per thread, per configuration. Generally speaking, `Hashtable` and `Collections.synchronizedMap()` scale up well. For instance, we observed a great increment of energy consumption when using `Collections.synchronizedMap()` when we move from 32 to 64 threads performing traversals, but this trend can also be observed

for insertions and removals. Still on traversals, all `Map` implementations greatly increase the energy consumed as we add more threads. Also, all thread-safe implementations have their own “15 minutes of fame". Despite the highly complex landscape, some patterns do seem to recur. For instance, even though `ConcurrentHashMapV8` provides the best scalability among the thread-safe collection implementations, it still consumes about 11.6x more energy than the non-thread-safe implementation. However, the most interesting fact is the peak of `ConcurrentSkipListMap`, when performing with 128 and 256 threads. As discussed earlier, during traversal, `ConcurrentSkipListMap` marks or unlinks a node with `null` value from its predecessor (the map uses the nullness of value fields to indicate deletion). Such mark is a compare-and-set operation, and happens every time it finds a `null` node. When this operation fails, it forces a re-traversal from caller.

For insertions, we observed a great disparity. On the one hand, `Hashtable` and `Collections.synchronizedMap()` do not scale well. `ConcurrentSkipListMap`, `ConcurrentHashMap` and `ConcurrentHashMapV8`, on the other hand, scale up very well. `ConcurrentHashMapV8` has one particular characteristic about: insertions of the first element in an empty map employs compare-and-set operations. Other update operations (insert, delete, and replace) require locks. Locking support for these locks relies on builtin “synchronized" monitors. When performing using from 1 to 32 threads, they have energy and performance behaviors similar to the non-thread-safe implementation. Such behavior was previously discussed in Figure 4.2.

Finally, taking into consideration removal operations, both `ConcurrentHashMap` and `ConcurrentHashMapV8` scale better than all other implementations, even the non-thread-safe implementation, `LinkedHashMap`. Interestingly, `ConcurrentSkipListMap` presents the worst scenario, in particular with 16, 32 and 128 threads, even when compared to the single-lock implementations, such as `Hashtable` and `Collections.synchronizedMap()`.

4.3.3 Collection configurations and usages

We now focus on **RQ4**, studying the impact of different collection configurations and usage patterns on program energy behaviors. The `Map` implementations have two important “tuning knobs”: the *initial capacity* and *load factor*. The capacity is the total number of elements inside a `Map` and the initial capacity is the capacity at the time the `Map` is created. The default initial capacity of the `Map` implementations is only 16 locations. We report a set of experiments where we configured the initial capacity to be 32, 320, 3,200, 32,000, 320,000, and 3,200,000 elements — the last one is the total number of elements that we insert in a collection. Figure 4.5 shows how energy consumption behaves using these different initial capacity configurations.

As we can observe from this figure, the results can vary greatly when using different initial capacities, in terms of both energy consumption and execution time. The most evident cases are when performing with a high initial capacity in `Hashtable` and `ConcurrentHashMap`.

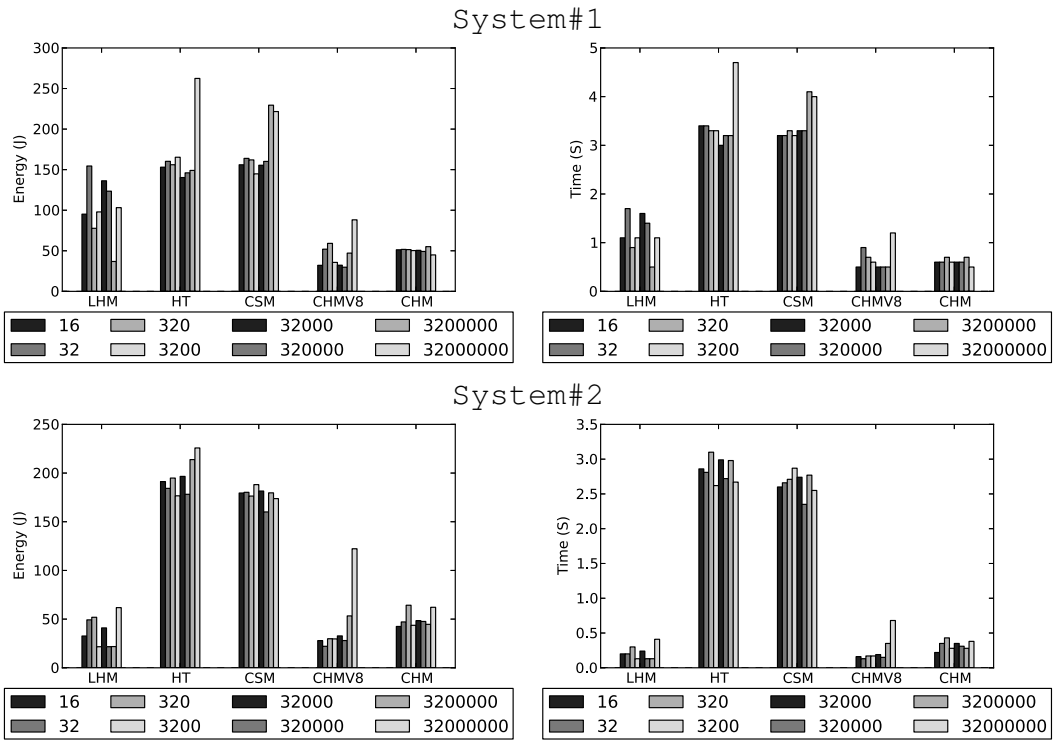


Figure 4.5: Energy consumption and performance variations with different initial capacities.

`ConcurrentHashMapV8`, on the other hand, presents the least variation on energy consumption.

The other tuning knob is the load factor. It is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of elements inside a `Map` exceeds the product of the load factor and the current capacity, the hash table is rehashed; that is, its internal structure is rebuilt. The default load factor value in most `Map` implementation is 0.75. It means that, using initial capacity as 16, and the load factor as 0.75, the product of capacity is 12 (16 * 0.75 = 12). Thus, after inserting the 12th key, the new map capacity after rehashing will be 32. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur. Figure 4.6 shows how energy consumption behaves using different load factors configurations. We perform these experiments only with insertion operations.[5].

From this figure we can observe that, albeit small, the load factor also influences both energy consumption and time. For instance, when using a load factor of 0.25, we observed the most energy inefficient results on `System#1`, except in one case (the energy consumption of `LinkedHashMap`). On `System#2`, the 0.25 configuration was the worst in three out of 5 of the benchmarks. We believe they is due to the successive rehashing operations that must occur.

[5]We did not performed experiments with `ConcurrentSkipListMap` because it does not provide access to initial capacity and load factor variables.

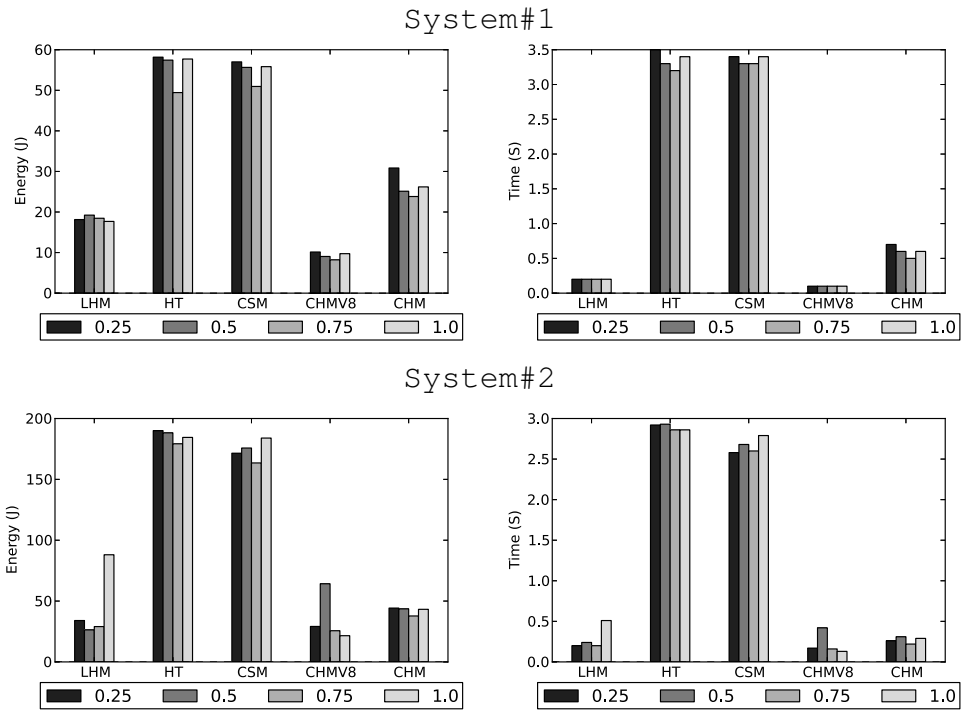


Figure 4.6: Energy consumption and performance variations with different load factors.

Generally speaking, the default load factor (.75) offers a good tradeoff between performance, energy, and space costs. Higher values decrease the space overhead but increase the time cost to look up an entry, which can reflect in most of the `Map` operations, including `get()` and `put())`. It is possible to observe this cost when using a load factor of 1.0, which means that the map will be only rehashed when the number of current elements reaches the current maximum size. The maximum variation was found when performing operations on a `Hastable` on `System#1`, in the default load factor, achieving 1.17x better energy consumption over the 0.25 configuration, and 1.09x in execution time.

4.3.4 The Devil is in the Details

In this section we further analyze some implementation details that can greatly affect energy consumption.

Upper bound limit.

We also observed that, on traversal and insertion operations, when the upper bound limit needs to be computed in each iteration, for instance, when using

```
for(int i=0; i < list.size(); i++) {
// do stuf...
}
```

the `Vector` implementation consumed about twice as much as it consumed when this

limit is computed only once on (1.98x more energy and 1.96x more time), for instance, when using

```
int size = list.size();
for(int i=0; i < size; i++) {
// do stuf...
}
```

When this limit is computed beforehand, energy consumption and time drop by half. Such behavior is observed on both `System#1` and `System#2`. We believe it happens mainly because for each loop iteration, the current thread needs to fetch the `list.size()` variable from memory, which would incur in some cache misses. When initializing a `size` variable close to the loop statement, we believe that such variable will be stored in a near memory location, and thus, can be fetched all together with the remaining data. Using this finding, well-known IDEs, such as Eclipse and IntelliJ, can take advantage of it and implement refactoring suggestions for developers. Currently, the Eclipse IDE does not provide such feature. Also, recent studies have shown that programmers are more likely to follow IDE tips (MURPHY-HILL; JIRESAL; MURPHY, 2012). One concern, however, is related to removal operations. Since removal on `Lists` shift any subsequent elements to the left, if the limit is computed beforehand, the `i++` operation will skip one element.

Enhanced for loop.

We also analyzed traversal operations when the programmer iterates using an *enhanced for loop*, for instance, when using

```
for (String e: list) { ... }
```

which is translated to an `Iterator` at compile time. Figure 4.7 shows the results. In this configuration, `Vector` needs to synchronize in two different moments: during the creation of the `Iterator` object, and in every call of the `next()` method. By contrast, the `Collections.synchronizedList()` does not synchronize on the `Iterator`, and thus has similar performance and energy usage when compared to our baseline, `ArrayList`. On `System#1`, energy decreased from 37.07J to 2.65J, whereas time decreased from 0.81 to 0.10. According to the `Collections.synchronizedList()` documentation, the programmer must ensure external synchronization when using `Iterator`.

Removal on objects. When using `Lists`, instead of perform removals based on the indexes, one can perform removals based on object instances, for instance, when using

```
for (int i = 0; i < threads; i++) {
  for (int j = 0; j < list.size(); j++) {
    boolean b = list.remove(i + "-" + j);
  }
}
```

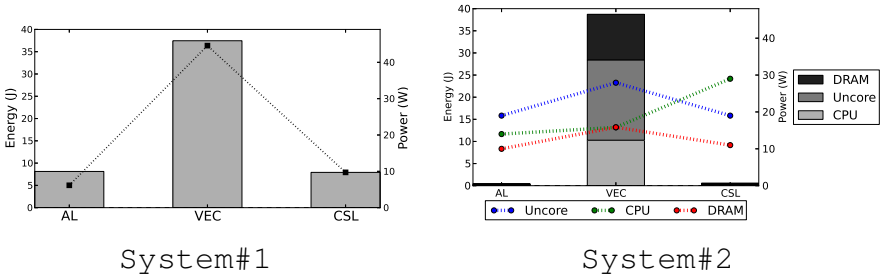


Figure 4.7: traversal operations using the `get()` method. We use the same abbreviations of Figure 4.1.

When using this operation, we observed an increment on energy consumption of 39.21% on `System#1` (32.28% on execution time). This additional overhead is due to the traversal needed for this operations. Since the collection does not know in which position the given object is placed, it needs to traversal and compare each element until it finds the object – or until the collection ends.

4.4 Threats to Validity

We divide our discussion on threats to validity into internal factors and external factors.

Internal factors: First, the elements which we used are not randomly generated. We chose to not use random number generators because they can greatly impact the performance and energy consumption of our benchmarks. We observed standard deviation of over 70% between two executions when using the random number generators. We mitigate this problem by combining the index of the for loop plus the thread id that inserted the element. This approach also prevents compiler optimizations that may happen when using only the index of the for loop as the element to be inserted in the collection.

External factors: First, our results are limited by our selection of benchmarks. Nonetheless, our corpus spans a wide spectrum of collections, ranging from lists, sets, and maps. Second, there are other possible collections implementations beyond the scope of this paper. With our methodology, we expect similar analysis can be conducted by others. Third, our results are reported with the assumption that JIT is enabled. This stems from our observation that later runs of JIT-enabled executions do stabilize in terms of energy consumption and performance PINTO; CASTOR; LIU (2014a). We experienced differences in standard deviation of over 30% when comparing the warmup run (first 3 executions) and later runs, but less than 5% when comparing the last 3 runs.

4.5 Summary

In this chapter, we presented an empirical study that investigates the impacts of using different collections on energy usage. As subjects for the study, we analyzed the main methods of 16 types of commonly used collection in the Java language. Some of the findings of this study include: (1) just by switching to a newer implementation of the `ConcurrentHashMap` can yield a 2.19x energy savings when compared to the old associative implementation. (2) Execution time is not always a reliable indicator for energy consumption; this is particularly true for various `Map` implementations. In other words, the consumption of power — the rate of energy consumption — is not a constant across different collection implementations.

5

The Energy Efficiency of Java Threading Constructs

The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency

—HERB SUTTER

This chapter presents an empirical study to illuminate and understand energy behaviors of concurrent programs on multi-core architectures. In particular, our study is unique in its focus on how programmer decisions — the choices and settings of thread management constructs — may impact energy consumption and its close relative, performance. We first provide an overview (Section 5.1) of these concurrent programming constructs, and state the research questions. Section 5.2 shows how code is modified to use these three techniques to solve the same problem. Section 5.3 presents our methodology, describing both benchmarks and environment. Section 5.4 shows the results of a first round of experiment, grouped by the research questions. Finally, Section 5.5 presents a second round of experiments aiming to understand how different platform variations impact the first round of experiments.

5.1 Overview

Despite their promise, few language-level or application-level energy-efficient solutions address concurrent software running on parallel architectures (BARTENSTEIN; LIU, 2013; GAUTHAM et al., 2012; TREFETHEN; THIYAGALINGAM, 2013; RIBIC; LIU, 2014). This is unfortunate for at least two reasons: (1) thanks to the proliferation of multicore CPUs, concurrent programming is a standard practice in modern software engineering (TORRES et al., 2011); (2) a CPU with more cores (say 32) often consumes more power than one with fewer cores (say 1 or 2) (LI; MARTÍNEZ, 2005). Energy optimization over programs on such platforms has the potential to yield larger savings, but may also face more challenges (IYER; MARCULESCU,

2002; ISCI et al., 2006).

We believe a first step to optimize energy consumption of concurrent programs is to gain a comprehensive understanding of their energy behaviors. This chapter presents an empirical study to illuminate and understand energy behaviors of Java concurrent programs on multicore architectures. In particular, our study is unique in its focus on how programmer decisions — the choices and settings of thread management constructs — may impact energy consumption and its close relative, performance. Our research is motivated by the following questions:

- **RQ1.** Do alternative *thread management constructs* have different impacts on energy consumption?
- **RQ2.** What is the relationship between *the number of threads* and energy consumption?
- **RQ3.** What is the relationship between *task division strategies* and energy consumption?
- **RQ4.** What is the relationship between *data* volume/access and energy consumption?

To answer **RQ1**, we select three thread management constructs influential in concurrent language design:

- *Explicit threading* ("the Thread style"): programmers manually map logically independent units of work to threads, *i.e.,* the scheduling unit of the virtual machine and/or the underlying operating system. Explicit threading is the most widely used approach in Java multi-threaded programming (TORRES et al., 2011).
- *Thread pooling* ("the Executor style"): programmers create a pool of threads — often fixed in size — and further submit logically independent units of work to the thread pool. The relationship between threads and the units of work is often 1:n. Threads select and execute submitted units of work from a centralized buffer managed by the language runtime. In Java, this mechanism is known as *executors* and is part of the `java.util.concurrent` library.
- *Work stealing* ("the ForkJoin style"): similar to thread pooling, programmers also create a pool of threads and submit logically independent units of work to the pool. What is unique to work stealing is that each thread maintains its own buffer of units of work. When one such buffer becomes empty, its maintaining thread may "steal" work from other threads. Its incarnation in Java is the ForkJoin framework (LEA, 2000).

Given these constructs, our investigation is further aimed at understanding how their settings —"tuning knobs" of concurrent programming for programmers — may impact energy

consumption. Among them, the number of threads and the size of data are two classic knobs, addressing the dual control vs. data aspects of concurrency. Their respective impacts on energy consumption are focuses of our study. *Tasks*, *i.e.*, logically independent units of work, have an intimate relationship with both. Just as the Executor and ForkJoin styles indicate, the ratio between the number of tasks and the number of threads is a design consideration of concurrent programmers. When the number of tasks increases while the size of data remains the same, each task will process a smaller "slice" of data, *de facto* tuning *task granularity*. We call the programmer job of dividing work to achieve desirable task granularity *task division*. The impact of task division strategies on energy consumption is another focus of our study.

Our study produces a list of findings, many of which are not obvious. We summarize them in Section 5.4, at the end of each **RQ**'s discussion. We now highlight two of them.

First, our study reveals the context-dependent nature of the energy behaviors of thread management constructs. Each thread management construct has its own "15 minutes of fame." Despite the highly complex landscape, some patterns do seem to recur. For example, as the number of threads for running a concurrent program continues to increase, we observe its energy consumption often increases first, and then decreases later, a phenomenon we term the Λ *curve*. The shape of the curve differs significantly from the one that describes performance (execution time).

Second, taking into consideration only the CPU usage, our experiments further demonstrate that "faster" is not a synonym with "greener" for concurrent programs, and performance as an indicator to estimate energy consumption is unreliable at best — incorrect in most cases — for multi-threaded Java programs. We observed that a (faster) multi-threaded program execution generally does not consume less energy. In fact, the opposite is often true: the sequential variants of the benchmarks (*i.e.*, executing a multi-threaded program with one thread) often consume the lowest energy consumption. That being said, the (effective) use of multi-threading does have its benefit in promoting energy efficiency: except for some embarrassingly serial benchmarks, multi-threading often achieves the best trade-off between energy consumption and performance. For example, one benchmark achieved a speedup of 9.5x when running with 32 threads, while its energy consumption only grew 1.97x.

Throughout our exploration, a recurring theme is to illuminate the intricate relationship between energy consumption and performance. There exists a rich literature on this topic (IYER; MARCULESCU, 2002; ISCI et al., 2006; SAMPSON et al., 2011; COHEN et al., 2012; BARTENSTEIN; LIU, 2013; RIBIC; LIU, 2014). We enrich existing work by offering a programming-level perspective.

This chapter makes the following contributions:

1. It describes an empirical study — the first of its kind to the best of our knowledge — to correlate energy behaviors of concurrent programs with thread management constructs and their knobs.

2. It conducts an extensive experimental exploration that involves a combination of factors, ranging from thread management constructs, the number of threads, task division strategies, task granularity choices, data sizes, and data access characteristics. The exploration carves out a landscape that involves thousands of distinct points in the experiment space. In addition, the chapter describes a preliminary study on the stability and portability of our results under different settings of heap size, garbage collection, just-in-time compilation, and platforms.

3. It offers insights into energy behaviors of real-world concurrent Java programs, with a detailed list of often non-obvious findings.

5.2 Programming Patterns for Thread Management

We use an (overly) simplified version of the `sunflow` benchmark (BLACKBURN et al., 2006) to illustrate the distinct programming patterns of the three thread management constructs. Figure 5.1 and Figure 5.2 demonstrate the Thread style and the Executor style, respectively. Figure 5.3 and Figure 5.4 both demonstrate the ForkJoin style, with a difference we will explain shortly. The three parameters related to **RQ2**-**RQ4** are `THREADN` for the number of threads (**RQ2**), and `TASKN` for the number of tasks (**RQ3**), and `DATAN` for the data size (**RQ4**), respectively.

The `sunflow` benchmark centers around a rendering algorithm (ray tracing) where coordinates are stored in array `coords` and method `render` takes one coordinate to render. The rendering logic is encompassed in a method called `dowork`. The coordinates to be processed by a `dowork` invocation are a range of `size` number of consecutive elements beginning at index `start`. For brevity, the code snippets here omit the body of the `render` method, and further omit program logic unrelated to our discussion here, such as post-rendering processing (typically performed through placing a barrier at the end of the `main` function).

In the Thread style, the program explicitly bootstraps `THREADN` threads, through messaging the `start` method of a `Bucket` object, whose class is a subclass of the JDK `Thread` class. The `run` method of the `Bucket` class (an inner class of `Main` in the example) is executed by each bootstrapped thread. Here, each thread continuously processes tasks through a busy `while` loop, and each task is defined as executing an instance of `dowork`. Since there are `TASKN` tasks, each task will work on a "slice" of coordinates of size `DATAN/TASKN`. A global counter `d` is used to track the size of data that has been processed, and the counter is accessed from within a `synchronized` block.

In the Executor style, `THREADN` threads are created in a fixed-size thread pool, managed by an instance of the `ExecutorService` class of the JDK. The inner class `Bucket` now only encompasses a task and its `run` method only executes the `dowork` method (definition identical to that in Figure 5.1) once. Each task is identified by a counter `t`. In the `main` method, `TASKN`

```
class Main {
  int coords[DATAN];
  void main() {
     for(int i=0; i<THREADN; i++)
         (new Bucket()).start();
  }
  class Bucket extends Thread {
    static int d = 0;
    public void run() {
      int start;
      while(d<DATAN){
         synchronized (this) {
            if (d >= DATAN) return;
         start=d; d+=DATAN/TASKN;
       }
    dowork(start, DATAN/TASKN);
       }
    }
    public void dowork(int start, int size) {
      for (int j=start; j<DATAN && size>0; j++, size--)
         render(coords[j]);
    }
  }
}
```

Figure 5.1: Concurrent Programming in Thread Style

```
class Main {
  int coords[DATAN];
  void main() {
     ExecutorService es = Executors.newFixedThreadPool(THREADN);
     for (int i = 0; i < TASKN; i++)
       es.execute(new Bucket(i));
  }
  class Bucket extends Thread {
    ...
    int t;
    Bucket(int t) {this.t = t;}
    public void run()
      {dowork(t * DATAN/TASKN, DATAN/TASKN); }
    }
}
```

Figure 5.2: Concurrent Programming in Executor Style

```
class Main {
  int coords[DATAN];
  void main() {
     (new ForkJoinPool(THREADN)).submit(new Bucket(0));
  }
  class Bucket extends RecursiveAction {
    ...
    int t;
    Bucket(int t) {this.t = t;}
    public void compute() {
      if (t < TASKN) {
         (new Bucket(t+1)).fork();
         dowork(t * DATAN/TASKN, DATAN/TASKN);
      }
    }
  }
}
```

Figure 5.3: Concurrent Programming in Task-Centric ForkJoin Style

```
class Main {
  int coords[DATAN];
  void main() {
     (new ForkJoinPool(THREADN))
          .submit(new Bucket(0, DATAN));
  }
  class Bucket extends RecursiveAction {
    ...
    int start, size;
    Bucket(int start, int size) {
      this.start = start; this.size = size;
    }
    public void compute() {
      if(size < SEQUENTIAL_CUTOFF) dowork(start, size);
      else {
        int half = size / 2;
        new Bucket(start, half).fork();
        new Bucket(start + half, size - half).fork();
      }
    }
  }
}
```

Figure 5.4: Concurrent Programming in Data-Centric ForkJoin Style

tasks will be managed by the pool of `THREADN` threads. The submission for management is achieved through the use of the `execute` method of the `ExecutorService` object.

The ForkJoin style is similar to the Executor style in that a fix-sized pool – the `ForkJoinPool` object – will manage `THREADN` threads. Unlike Executor however, ForkJoin adopts a work stealing algorithm to manage threads. Instead of submitting all tasks to a centralized service such as in Executor, each thread under a work-stealing scheduler maintains its own localized queue-like structure, called a *deque*, for tasks. A thread running out of tasks (a *thief*) will "steal" a task from the deque of another randomly selected thread (a *victim*). The runtime behavior of work stealing is defined through a classic yet sophisticated algorithm, with subtleties detailed in prior work (FRIGO; LEISERSON; RANDALL, 1998; LEA, 2000).

From a programming perspective, the thread pool for work stealing is initially only `submit`'ed with one task, an object subclassed from the JDK class `RecursiveAction`. A thread in the pool will pick up the task, *i.e.,* run its `compute` method. The `compute` method may further `fork` new tasks "on the go," where forking can be viewed as placing the task on the thread's own deque. Such a task in turn may either be picked up by the current thread, or be stolen and picked up by other threads in the pool. Both Figure 5.3 and Figure 5.4 follow this common pattern.

Recursively dividing work into smaller tasks is a distinct programming pattern for programs written in work-stealing languages or language frameworks. As it turns out, different task division strategies exist, with Fig. 5.3 highlighting a *task-centric* task division strategy, and Fig. 5.4 demonstrating a *data-centric* task division strategy. In the task-centric approach, we directly fix the number of tasks (through `TASKN`), and keep a counter to track how many tasks have been forked so far. In contrast, the data-centric approach sets a *sequential cutoff threshold* to data, *i.e.*, the size of data a task will work on, instead of explicitly setting and tracking the number of tasks. The two strategies lead to different programming patterns, and the choice is largely dependent on what is considered more natural to specific programs. It however should be pointed out that they are indeed two sides of the same coin for task granularity: given the overall data, fixing the number of tasks will implicitly set the data size per task, whereas fixing (sequential cutoff) data size will implicitly determine the number of tasks.

In the rest of the chapter, we manually refactor each benchmark into the four programming patterns. Figures 5.1, 5.2, 5.3, and 5.4 serve as examples of what we view as "comparable" programs in our benchmarking process. We routinely fix two of the three parameters — `THREADN`, `DATAN`, `TASKN` (or its counterpart of sequential cutoff threshold) — and observe the impact on energy/performance when the 3rd parameter varies. For example, when `THREADN` and `DATAN` remain the same but `TASKN` increases, it is aligned with our intuition that tasks become more "fine-grained."

5.3 Experiment Setup

In this section we describe the benchmarks that we analyzed, the infrastructure and the methodology that we used to perform the experiments.

5.3.1 Benchmarks

We use a variety of benchmarks for evaluation, listed as follows. Benchmarks 1-3 are from a Debian-based language benchmark suite[1]. Benchmark 4 was developed by us. Benchmark 5 is a modification of a program originally developed for a work-stealing language system (KUMAR et al., 2012). The rest of the benchmarks are from the well-known DaCapo suite (BLACKBURN et al., 2006).

1. `knucleotide`: This benchmark takes a DNA sequence, and counts the occurrences and their frequencies of nucleotide patterns. The memory-intensive benchmark employs string manipulation intensively. There is no synchronization point in the program, but one atomic variable is used.

2. `mandelbrot`: A mandelbrot is a mathematical set of points whose boundary is a distinctive and easily recognizable two-dimensional fractal shape. Mandelbrot set images are created by sampling complex numbers and determining for each one whether the result tends toward infinity when a particular mathematical operation is iterated on it. According to its website, this benchmark spends 99% of the time using CPU, and uses I/O only to print the results. There is no synchronization point in the program, but one atomic variable is used.

3. `spectralnorm`: The spectral norm is the maximum singular value of a matrix. The benchmark is CPU-intensive, and scales up well in multicore machines. This benchmark synchronizes threads using a barrier, and uses one atomic variable.

4. `largestimage`: This I/O-intensive benchmark performs a recursive search into the file system, looking for image files. During traversal, it keeps track of the number of image files it encountered and the largest among them. This benchmark has two synchronization points and is strongly I/O-bound.

5. `n-queens`: This benchmark is the classic N-queens chessboard game, placing N chess queens on an NxN chessboard so that no two queens attack each other. It is a computationally intensive, CPU-bound problem. This benchmark does not have synchronization points, but uses one atomic variable.

[1] `http://benchmarksgame.alioth.debian.org`

6. `sunflow`: renders a set of images using ray tracing[2].

7. `xalan`: transforms XML documents into HTML.

8. `h2`: executes a number of transactions against a model of a banking application, in a style similar to JDBCbench.

9. `tomcat`: runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages.

We selected the benchmarks based on their diverse characteristics. For instance, according to a recent study (KALIBERA et al., 2012), `sunflow` scales well when the number of CPU cores increases, `h2` scales rather poorly, and `xalan` is the middle-of-the-road benchmark in terms of scalability. Benchmark `largestimage` is I/O-intensive, `knucleotide` is memory-intensive, and benchmarks `mandelbrot`, `n-queens`, and `spectralnorm` are CPU-intensive.

For the benchmarks, `DATAN` represents the number of patterns for `knucleotide`, the size of the vector for both `mandelbrot` and `spectralnorm`, the size of a matrix for `n-queens`, the number of directories for `largestimage`, the size of the image for `sunflow`, the number of converted files for `xalan`, and the number of database transactions for `h2`.

5.3.2 Experimental Environment

Unless noted otherwise, all experiments were conducted on a machine with 2×16-core AMD Opteron 6378 processor (Piledriver microarchitecture), 2.4GHz, with 64GB of DDR3 1600 memory. It has three cache levels (L1, L2, L3): L1 with 32KB per core, L2 with 256KB per core, and L3 with 20480KB (Smart cache). It is running Debian 3.2.46-1 x86-64 Linux (kernel 3.2.0-4-amd64), and Oracle HotSpot 64-Bit Server VM (build 21) JDK version 1.7.0_11.

All experiments were performed with no other load on the OS. We conform to the default settings of both the OS and the JVM. Several default settings are relevant to this context: (1) the power management of Linux is the default *ondemand* governor, which dynamically adjusts CPU core frequencies based on system workloads. (2) For the JVM, the parallel garbage collector is used, and just-in-time (JIT) compilation is enabled. The initial heap size and maximum heap size are set to be 1GB and 16GB respectively. We run each benchmark 10 times within the same JVM; this is implemented by a top-level 10-iteration loop over each benchmark. The reported data is the average of the last 3 runs. We justify this decision in Section 5.5.

Energy consumption is measured through current meters over power supply lines to the CPU module. Data is converted through an NI DAQ and collected by NI LabVIEW SignalExpress with 100 samples per second and the unit of the current sample is *deca-ampere* (10 ampere).

[2]The description for the DaCapo benchmarks was taken directly from the DaCapo website: http://www.dacapobench.org/

Since the supply voltage is stable at 12V, energy consumption is computed as the sum of current samples multiplied by $12 \times 0.01 \times 10$. We measured the "base" power consumption of the OS when there is no JVM (or other applications) running. The reported results are the measured results *modulo* the "base" energy consumption.

5.4 Study Results

In this section, we report the results of our experiments. Results for **RQ1** and **RQ2** are presented in Section 5.4.1, which describes the impact of different thread management constructs in the presence of varying numbers of threads. In Section 5.4.2 we attempt to answer **RQ3** by investigating the impact of different task division strategies. Finally, in Section 5.4.3 we present answers to **RQ4** by exploring different data characteristics.

5.4.1 Energy Behaviors with Alternative Programming Abstractions and Varying Numbers of Threads

In this group of experiments, we fix the number of tasks and the size of the data, and study how variations on the number of threads and the choice of different thread management constructs impact energy consumption. The results of our experiments are presented in Figures 5.5 and 5.6. Here, the first set of figures are energy consumption results, whereas the other ones are the corresponding performance results.

The Λ Curve.

One interesting observation throughout our study is that energy consumption typically increases as the number of threads increases, and then gradually decreases as the number of threads approaches the number of CPU cores. In the energy consumption figures, the curves typically display a Λ shape, which we term the Λ *curve*. Nearly all benchmarks display the Λ curve.

We believe the Λ curve results from a combination of multicore processor characteristics and program performance traits. Under the default setting of the ondemand governor, power management modules of multicore CPUs work in an "adaptive" fashion: when a particular core stays idle, the operating frequency of the core will be dynamically adjusted to a lower level. When a 32-core CPU is only loaded with 4 threads for instance, a large number of cores will operate on the lowest frequency (the specific number of cores is likely to be slightly more than 4, because of the running of VM/OS threads). It is standard knowledge that power consumption is reduced when the operating frequency is lower. For that reason, a program running 4 threads is likely to consume less power than one running 8 threads. This helps us explain the $/$ part of the Λ curve.

To see why energy consumption often decreases after the initial increase, note that energy consumption, by definition, is the multiplication of power and time. As more threads are used,

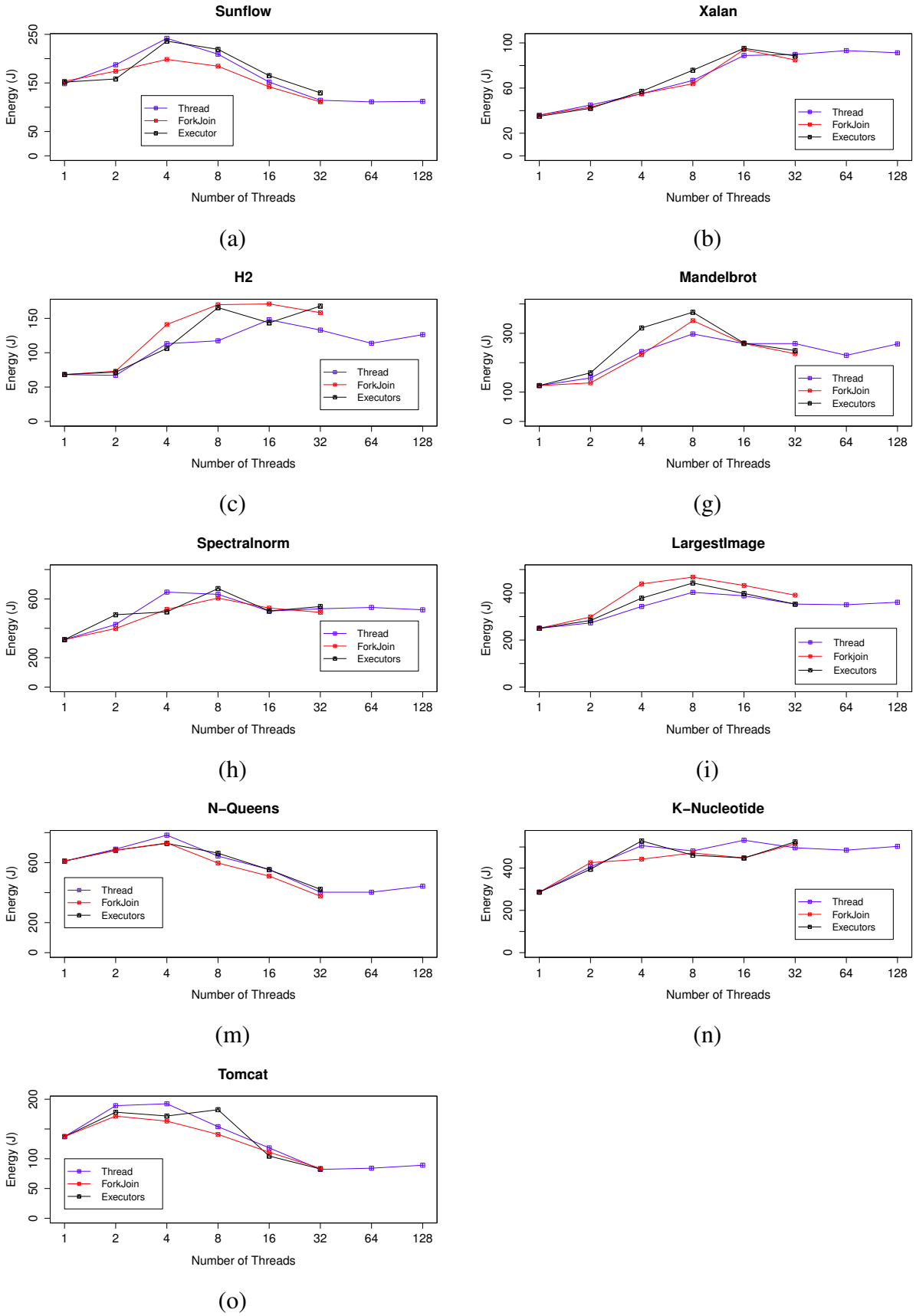
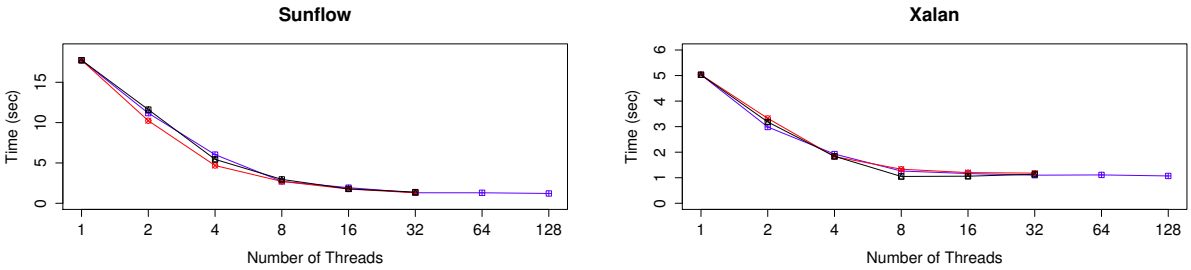
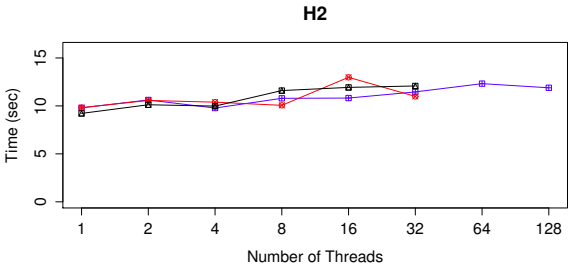


Figure 5.5: Energy Consumption with Alternative Programming Abstractions and Varying Numbers of Threads

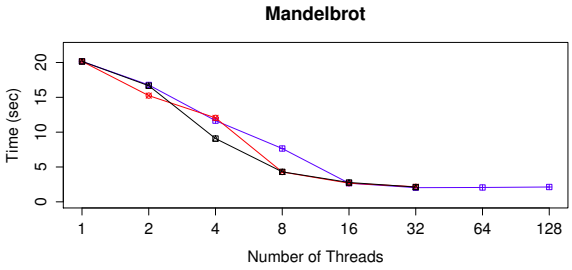


(d)

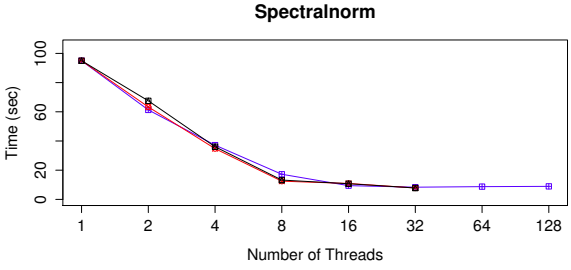
(e)



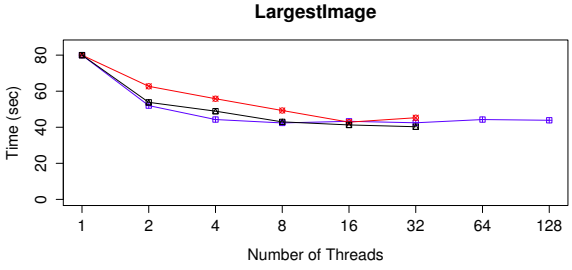
(f)



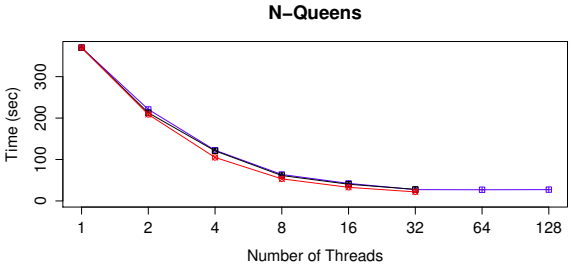
(j)



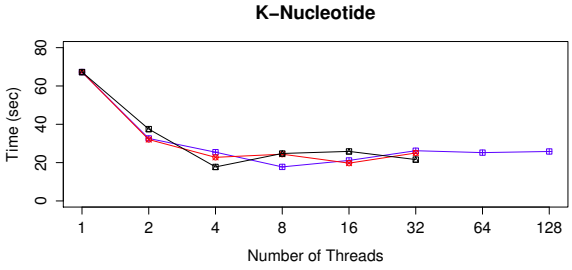
(k)



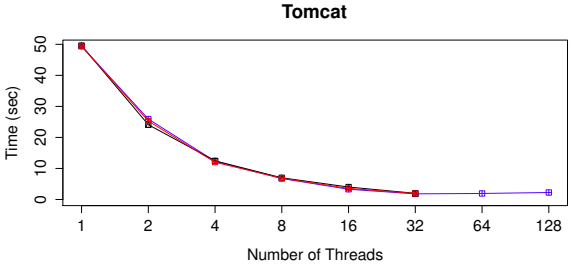
(l)



(p)



(q)



(r)

Figure 5.6: Performance with Alternative Programming Abstractions and Varying Numbers of Threads

program execution time tends to shorten. The extent of the drop — the \ part of the Λ curve — is determined by the increase in performance (and thus decrease in time) and the increase in power consumption. The greater the ratio between speedup and increase in power, the steeper the \ part of the curve will be.

The specific shape details of the Λ curve, including the "peaking" point and the slope of the increase/decrease, are application-specific. Take `sunflow` and `h2` (in the Thread style) for example. Power consumption for the two benchmarks is 8.54 *W* and 6.87 *W* on average, respectively, when using 1 thread, and 88.05 *W* and 14.27 *W* when using 32 threads. Execution time is 17.74 and 9.92 seconds, respectively, when using 1 thread, and 1.34 and 10.38 seconds when using 32 threads. Since the power consumption for `sunflow` increases about 10x and performance improves 13x, energy consumption in fact decreases. For `h2` however — a benchmark known to scale rather poorly as the number of cores increases — power consumption increases 2.07x but execution time also increases 1.04x, yielding 2.17x energy consumption. Thus, in this extreme case, the \ part of the curve does not exist.

Embarrassingly Parallel vs. Embarrassingly Serial. Our selection of benchmarks range from "embarrassingly parallel" ones (`sunflow`, `tomcat`, `spectralnorm`, and `n-queens`), to middle-of-the-road ones (`xalan`, `knucleotide`, and `mandelbrot`), to "embarrassingly serial" ones (`h2`, and to some extent `largestimage`)). The performance results of four Dacapo benchmarks — Figure 5.6(a)(b)(c)(i) — are consistent with recent studies (*e.g.*, KALIBERA et al. (2012)).

We find the (more) embarrassingly parallel benchmarks are likely to "peak" earlier on the Λ curve, *i.e.*, reaching the highest energy consumption with the smallest number of threads. For example, `sunflow`'s Λ curve peaks at 4 threads, whereas `xalan` peaks at 16. We think this is reasonable: the speed-up of `sunflow` is almost 8x when the number of threads increases from 1 to 8 (linear speedup), so the reduction in execution time can quickly offset the increase in power consumption early on. In comparison, `xalan` produces a 5x speedup with the same variation in threads and its performance does not improve with more threads. Hence, its Λ curve peaks later.

Faster is not Greener. In most of our benchmarks, additional threads would initially lead to improved performance; see Figure 5.6(a)(b) for example. Following the / part of the Λ curve however, the energy consumption increases as the number of threads increases initially. Furthermore, for 6 of our 9 benchmarks, the lowest energy consumption was achieved by the sequential (1 thread) version. For CPU computations only, being "faster" clearly has little correlation with being "greener". However, a precise and definitive answer to this question must consider not only CPU (or DRAM and interconnects, as the last chapter did), but also the entire system's energetic footprint. Unfortunately, the current state of the art on energy tooling does not offer support to such kind of analysis. We expect to revisit this question when better support is available.

Moreover, since benchmarks "peak" at different parts of the Λ curve, it is not possible to

generalize that an improvement in time could be seen as an improvement in energy, and *vice versa*.

Which Programming Style Should I Use? As Figures 5.5 and 5.6 show, it is possible to detect differences in the amount of energy used when different concurrent programming abstractions are employed. For some benchmarks, this difference is small, *e.g.*, `xalan` in Figure 5.5(b) and Figure 5.6(b). However, the difference is more noticeable in others. Every programming abstraction may have its "15 minutes of fame." In one configuration of `sunflow`, ForkJoin outperforms Thread and Executor by reducing energy consumption by 30%, as shown in Figure 5.5(a). In one configuration of `h2` however, ForkJoin underperforms Thread and Executor by increasing energy consumption by 50%, as shown in in Figure 5.5(c). Our experiments do show that there are scenarios where one style is more likely to outperform the others, which we summarize now.

First, the Thread style performs well in I/O-bound (such as `largestimage`) benchmarks. One possible explanation is that in I/O-bound benchmarks, the instruction pipeline has a higher likelihood to stall. In such a scenario, the Thread style defers context switching and/or load balancing to the OS, which appears to be efficient. The Executor style and the ForkJoin style build an additional layer of thread management on top. Unfortunately, this higher layer of decision making may disagree with the OS, missing some opportunities for context switch in the presence of long-latency I/O operations.

Second, the energy consumption of the ForkJoin-style programming is sensitive to the degree of parallelism latent in the benchmarks. It outperforms the other two strategies when the benchmarks are embarrassingly parallel (*e.g.*, Figures 5.5(a)(e)(g)(i)), but underperforms the other two strategies in the presence of more serial benchmarks, such as `h2`. We believe this can be explained through the nature of the work stealing algorithm: it excels through balancing the deques of individual threads. For benchmarks involving significant serial portions, synchronization (such as a barrier) is often used during the execution of a task. The work stealing algorithm is oblivious to such intra-task synchronizations, preventing tasks from being stolen and thus suppressing load balancing. In other words, the impacts of stealing a task with long synchronization delays and one without are clearly different, and the natural strength of work stealing in balancing tasks among threads is broken when long intra-task synchronization delays are present. Along this line, the Executor style performs slightly better, but still not as efficient as the Thread style. One possible reason is that an Executor needs to manage a queue of worker threads. Updates to the queue are protected from clients by a lock, thus increasing synchronization costs when a new task is submitted. Such overhead does not exist in the Thread style.

Energy-Performance Trade-offs. An energy-related question arises when we move from single-threaded programming to multi-threaded programming, or from 16 threads to 32 threads: what is the relationship between energy consumption and performance? One well-known metric to evaluate the energy/performance trade-off is the Energy-Delay Product (EDP): the product of

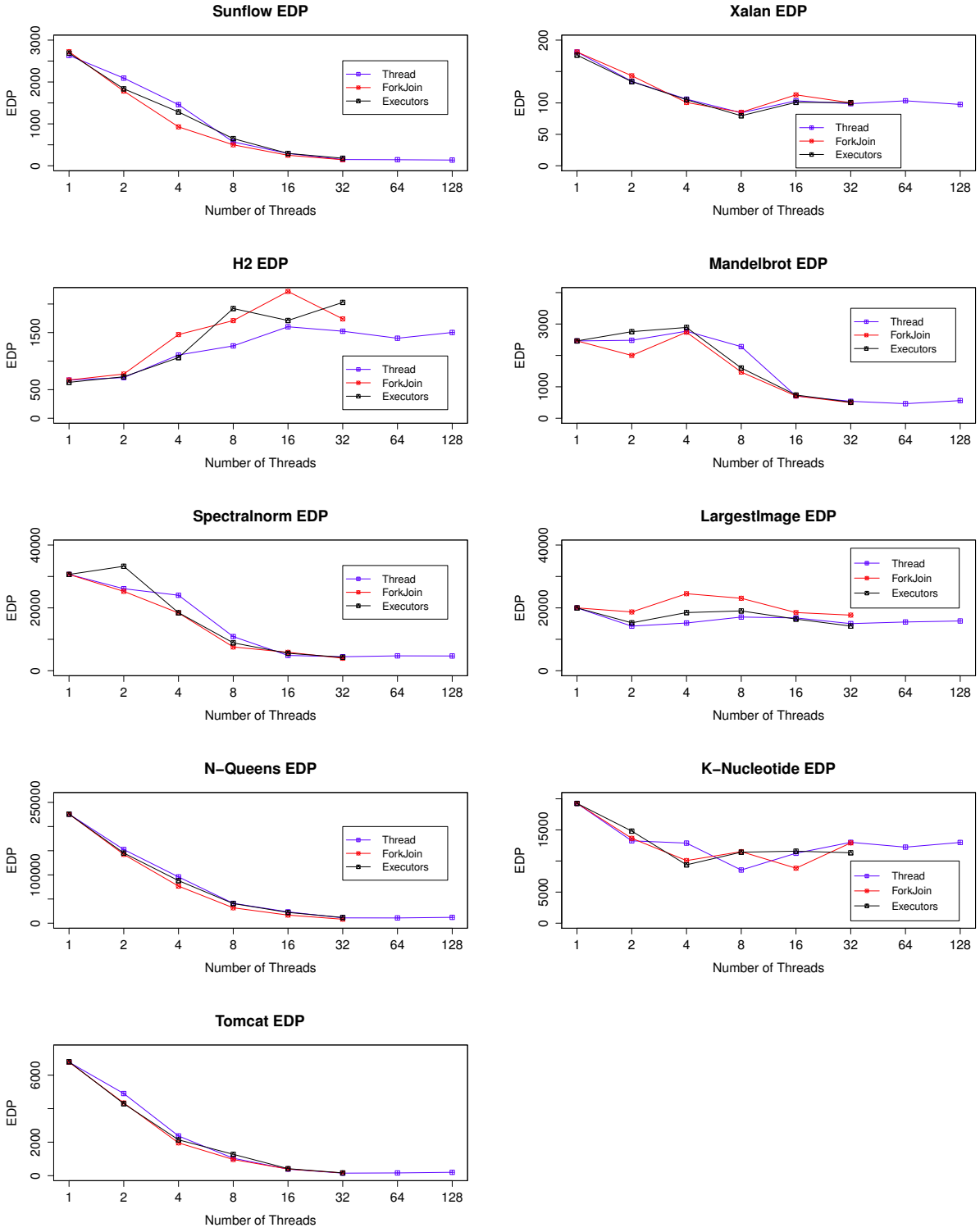


Figure 5.7: EDP (a smaller value is better)

energy consumption and execution time. We compute the EDP for the benchmarks, with results presented in Fig. 5.7, where a smaller EDP value indicates the more favorable trade-off.

We observed that a parallel execution is generally more favorable for energy-performance trade-offs than its single-threaded counterpart. This is particularly true for embarrassingly parallel programs: the EDP for `sunflow` with 32 threads is only 5.8% of its single-threaded execution. The degree of improvement on EDP appears to be in sync with the potential of parallelism in applications, and for specific benchmarks, increasing the number of threads is most likely *not* aligned with the improvement of EDP. For instance, when the number of threads increases for `xalan` from 8 to 16, EDP for all three programming constructs deteriorates significantly. The most unfortunate case among our benchmarks is perhaps `h2`. As the number of threads increases, the benchmark produces no gain in performance, but its energy consumption triples. As a result, EDP degrades as we move from sequential to parallel execution.

Overpopulating Cores with Threads. For the Thread style of thread management, we have also constructed experiments where the number of threads goes beyond the number of cores. In all experiments, we did not notice significant change in energy consumption. This suggests that the JVM and the OS are well-versed in handling cases where threads outnumber cores. Make no mistake: the number of context switches does increase as the cores become more overpopulated with threads. For instance, in the `sunflow` benchmark, the number of context switches increases 3.57x when the number of threads varies from 32 to 128 threads, as Figure 5.8 shows.

We choose not to perform experiments over the cases where there are more threads than CPU cores for Executor and ForkJoin styles. The "comparable" (Section 5.2) implementation would create a thread pool that outnumbers the number of cores. We do not believe that is the intended use for these thread management constructs.

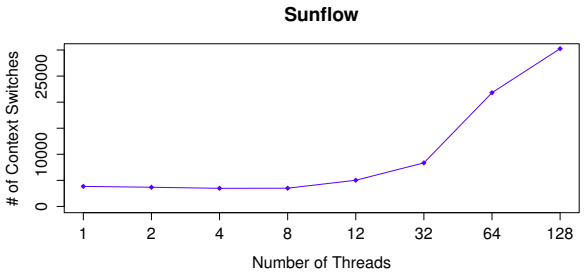


Figure 5.8: Context Switches and Thread Overpopulation

5.4.2 Energy Behaviors and Task Division Strategies

In this section, we fix the number of threads and the size of data, and study how the variations on the number of tasks have effects on energy consumption. To thoroughly explore the experimental space, we further refine our benchmarks into two versions: a *task-centric* division strategy and a *data-centric* division strategy as we first introduced in Section 5.2.

Task Granularity with Task-Centric Division. In this style, we divide the work based on `TASKN`. Figure 5.9 demonstrates the effect of task granularity on `xalan` benchmark. We observed a similar energy consumption behavior in the other benchmarks. Here the data exhibit remarkable uniformity: the number of tasks submitted/executed as logically independent units of work has little impact on energy consumption, independently of the thread management construct.

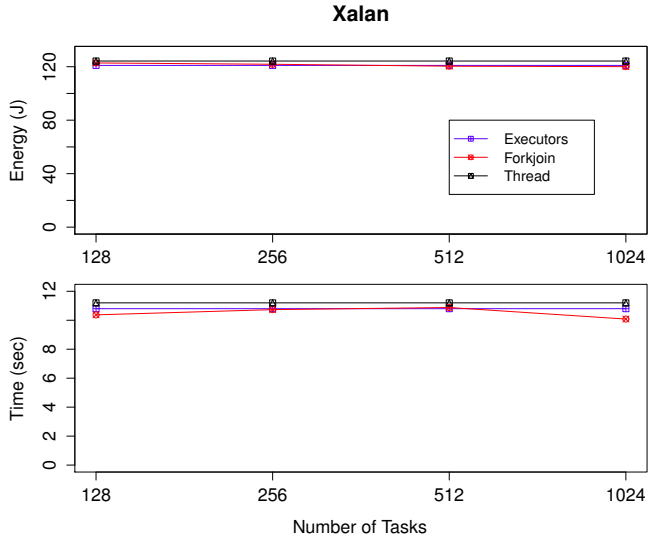


Figure 5.9: Energy/Performance and Task-Centric Division

At first glance, the results may be disappointingly "boring." We believe, however, that no news is news. The data reveal that task granularity matters little to energy consumption of concurrent programs. For instance, we have at a point increased the number of tasks to 1024, for a benchmark whose overall `DATAN` is 2048. In other words, every task only takes 2 pieces of data. In this case, no noticeable energy consumption increase was observed. Its version in the Executor style submits 1024 tasks to the `ExecutorService` and its version in the ForkJoin style recursively creates 1024 `RecursiveAction` objects. Though such programming patterns may appear to be "extreme," our experiments show they place little burden on energy consumption.

Task Granularity with Data-Centric Division. Under a data-centric approach, ForkJoin can also be seen as a divide-and-conquer algorithm, where in each recursive call new tasks are spawned until a certain threshold is reached. Using this approach, Figure 5.10 shows the energy/performance behavior of different sequential threshold configurations for the `sunflow` benchmark, where each recursive call sapwns two tasks to divide work into halves. We choose not to perform experiments of the case using Thread and Executor styles because their programming patterns do not naturally fall into the divide-and-conquer style as ForkJoin does.

There are three observations from this set of experiments. First, energy consumption and execution time both increase when the sequential threshold changes from 135 to 405, a 2.66x increase in energy consumption and a 2.64x increase in execution time. In this example, the benchmark operates over an array of 2048 positions. Thus, when we use 405 as the sequential

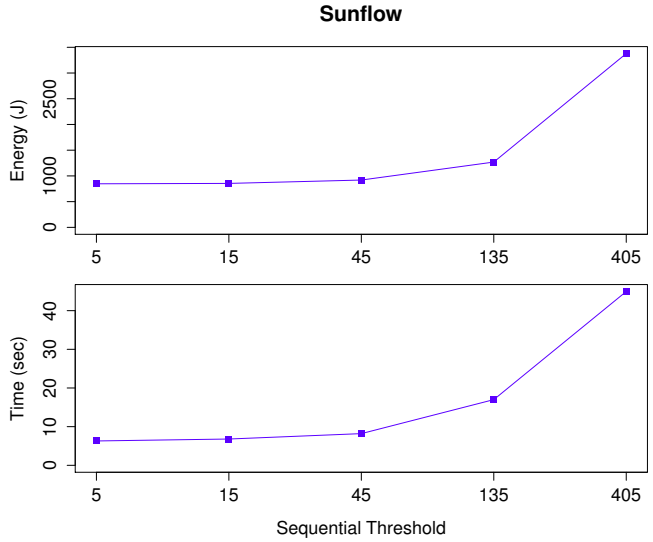


Figure 5.10: Energy/Performance and Data-Centric Division

cutoff threshold, the benchmark creates less than 10 tasks and operates on at most as many CPU cores. With the majority of the cores idle, the benchmark is not able to take advantage of the multiprocessors. As the sequential threshold reduces to 135, the program operates on more cores. As the Λ curve suggests, both the energy consumption and the execution time reduce for `sunflow`.

Second, the overhead of scheduling a high number of tasks does not seem to impact energy consumption. This phenomenon appears to recur in all benchmarks, and it is consistent with our findings for the task-centric experiments.

Third, energy consumption and execution time do not always increase in sync. For example, there is a small energy consumption variation (7.85%) when the sequential threshold changes from 45 to 135. Performance, on the other hand, degrades 23.8%. One possible reason is that, when tasks become more coarse-grained, it is less likely that a ForkJoin thief will steal a task, because the total number of available tasks decreases. Thus, after few unsuccessful attempts, the processor goes idle and the average power consumption decreases.

Asymmetric Workload. So far, we have created tasks where the data is divided uniformly. Another important characteristic to take into consideration is the use of asymmetric workloads. With different amounts of work, some ForkJoin workers will finish their work faster than others. Hence, the likelihood of steals may increase. Figure 5.11 shows the average number of steals per task granularity in the presence of symmetric load, a random asymmetric work division, and an 80-20 asymmetric work division.

The figure shows that the number of steals is strongly correlated to the symmetric vs. asymmetric nature of task workloads. Further, the number of steals is also correlated to task granularity: the smaller the tasks, the greater the number of steals. We observed an average energy savings of 3.26% using asymmetric workloads. We have experienced similar results in CPU-bound benchmarks, as Figure 5.12 shows.

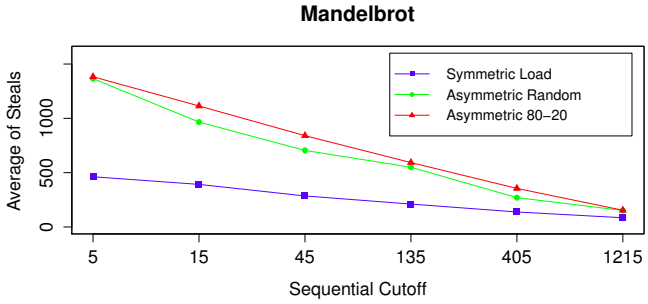


Figure 5.11: Number of Steals and Task Granularity

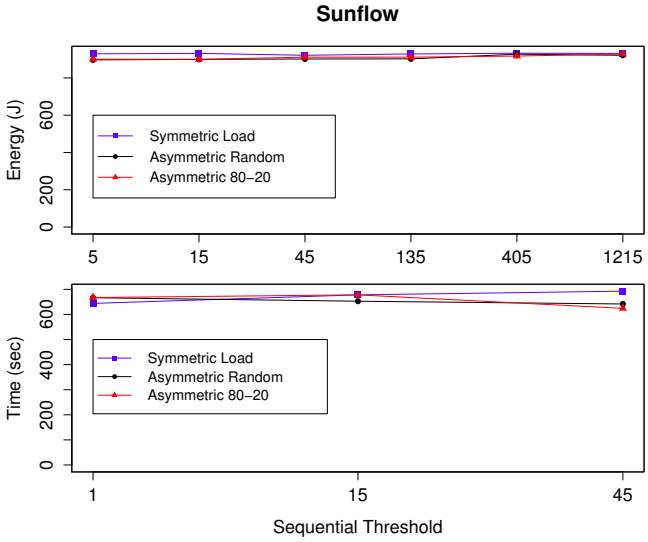


Figure 5.12: Energy/Performance with Asymmetric Workload

The Width of Forking. The ForkJoin-style can be configured to divide the work into *n* desirable tasks, instead of two per recursive call, which we term the *width* of forking. We have analyzed 4 different forking widths. For the `sunflow` benchmark in Figure 5.13, we observed a negligible difference of energy consumption from 2 to 4 forks, and from 4 to 8 forks per recursive call (about 0.96% and 1.21% respectively). From 8 forks to 16 forks, however, we observed an increase of 5.78% over the total energy consumption, and a similar increase in the execution time of 5.67%. This result is consistent with the other benchmarks. The experiment here suggests that excessive forking width can lead to increased energy consumption.

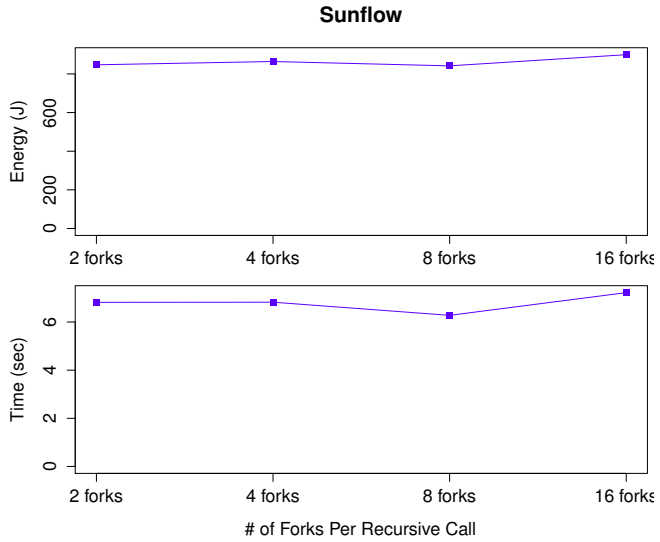


Figure 5.13: Energy/Performance and Forking Width

5.4.3 Energy Behaviors and Data

We now focus on **RQ4**, studying the impact of data — its size and access patterns — on program energy behaviors.

Data Size. Fixing the number of threads and the number of tasks, we now study how the variations on data size have effect on energy consumption. Figure 5.14 shows the energy behavior for the `xalan` benchmark, where the analogous `DATAN` value (as used in examples Fig. 5.1/5.2/5.3/5.4) represents the number of XML files to be converted. `THREADN` and `TASKN` were fixed at 32 and 256, respectively.

As predicted, energy consumption increases when a larger number of files are processed. Observe however, the increase in energy consumption is not necessarily linear to data size. Generally speaking, the precise relationship is application-specific: it depends on the algorithm complexity relative to the data size. In cases of data-parallel benchmarks, one phenomenon we observe is that the curve is often *convex*, especially for the part of the curve where the data sizes are relatively small. Take `xalan` for instance. When data size increases from 50 to 100, the energy consumption and performance remain almost unchanged. We think this has to do with the programming pattern itself. In data-parallel programs, there is usually a barrier at the end of

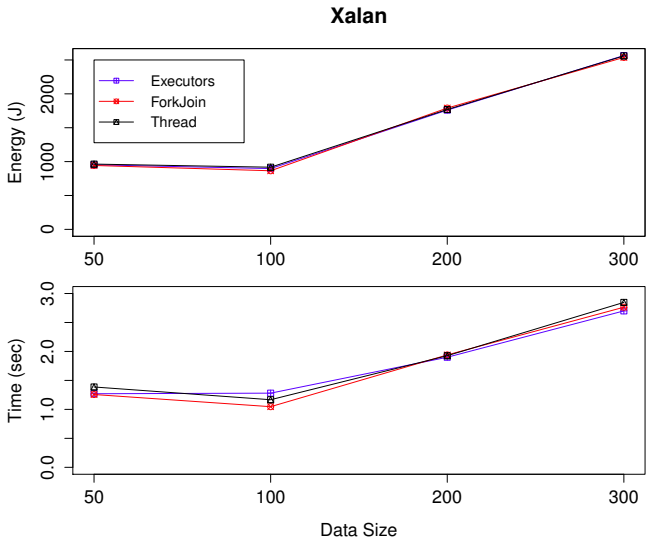


Figure 5.14: Energy/Performance and Data Size

data processing, and performance is determined by the slowest processing thread. When overall data size is small, the execution time of processing each data "slice" is also small. Variations on processors and scheduling may contribute to a larger proportion on the progress of individual threads, and differences in data size may be masked. When data size increases, the masking effect is reduced.

In `xalan`, the energy behaviors with the 3 thread management constructs are nearly identical, but there is a small but detectable difference in performance for the three constructs, with ForkJoin taking the least time and Executor taking the most. Since energy is the accumulated effect of power over time, this indicates ForkJoin is likely to have completed the task faster with a higher power consumption. Work stealing systems are most known for their ability for load balancing, where CPU core idling is reduced, improving performance while presenting fewer opportunities for cores to fall into lower power modes. This phenomenon is reduced when data size becomes larger, because data processing time would be proportionally larger, reducing the relative effect of frequent steals.

Data Sharing vs. Copying. We now study how memory-intensive tasks may impact energy consumption. Many ForkJoin benchmarks (in the style of data-centric division) operate on an indexable data structure, with subtasks operating on partitions of this data structure. During recursion, it is often necessary to split the data structure into smaller pieces on which the newly forked tasks can work on. One possible solution is to *copy* part of the data structure and use it for the newly forked tasks. Given an array-based data structure, each recursive call in this scenario will create n new arrays, where n is the width of forking. However, an alternative solution is to *share* this array, with newly forked tasks operating on contiguous partitions of this data structure. In all the experiments we have reported so far, sharing is the default strategy for data use. In the next set of experiments, we modify each benchmark to one that the forked tasks operate on *copies* of the data structure, instead of working on in-place data. In Figure 5.15 we compare the

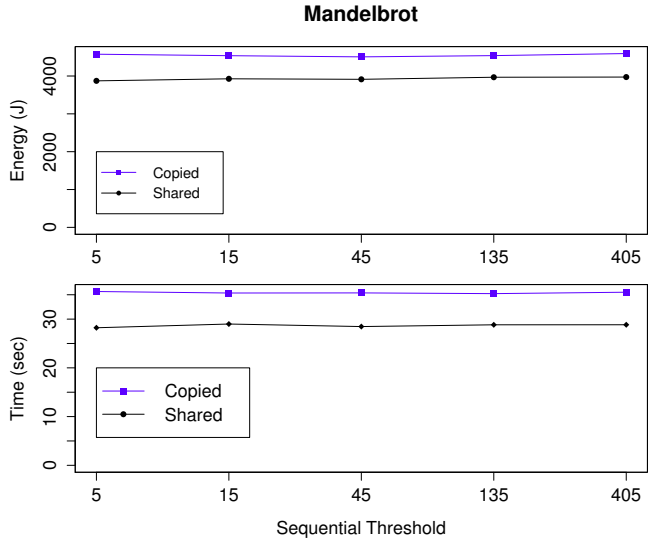


Figure 5.15: Energy/Performance and Data Sharing Strategies

two approaches using the `mandelbrot` benchmark.

As the figure shows, we experienced an energy consumption increase of 15.38% when copying is used. In the meantime, performance degrades by 20.85%. In other words, copying has severe impact on both energy and performance. In cases where the newly forked tasks are unlikely to lead to data races, this set of experiments demonstrate that a ForkJoin programmer should use shared data structures as much as possible. Furthermore, observe that copying has a more severe impact on performance than energy. This is indeed natural: when long-latency main memory request is issued, the issuing cores can often be reduced to a lower frequency, and a lower level of power consumption. (Recall again that energy is the multiplication of power and execution time.)

```
public void compute() {
  ...
  Solver[] tasks = new Solver[size];
  for (int i = 0; i < tasks.length; i++) {
    int[] newElements = new int[depth + 1];
    System.arraycopy(currentElements, 0,
                     newElements, 0, depth);
    tasks[i] = new Solver(newElements);
    tasks[i].fork();
  }
  ...
  for (int i = 0; i < tasks.length; i++) {
    if(tasks[i] != null) tasks[i].join();
  }
}
```

Figure 5.16: ForkJoin: Spreading Out Data Copying

Data Locality. Next, we investigate the impact of data locality on energy consumption. We modify the (data copying flavor of the) `n-queens` benchmark into two versions: Figure 5.16

```
public void compute() {
   ...
   List<Solver> tasks = new ArrayList<>(size);
   for (int i = 0; i < tasks.size(); i++) {
     int[] newElements = new int[depth + 1];
     System.arraycopy(currentElements, 0,
                      newElements, 0, depth);
     tasks.add(new Solver(newElements));
   }
   ...
   invokeAll(tasks);
}
```

Figure 5.17: ForkJoin: Aggregating Data Copying

and Figure 5.17. The two versions are functionally identical. In the first version, the execution of a task follows the sequence of `abababababc` where `a` is copying memory for a subtask, `b` is forking the subtask, and `c` is computing the current task. In the second version, the execution of task follows the sequence of `aaaacbbbb`[3].

Which version should fare better? On the surface, the second version indeed admits less parallelism on the execution of the current task: it forks the subtasks only after the current task has finished. Therefore, it cannot be executed in parallel with any of the `a` steps or the `c` step. Our benchmarking results on the other hand show the opposite: the second version yields energy savings of 10.11% and a performance improvement of 10.66%.

We hypothesize that data locality plays an important role. Note that in the first version, we interspersed data copying with thread forking (together with other operations in a loop iteration). Any of the latter operations may potentially pollute the cache, increasing the chance of memory round-trips. In the second version however, the same memory area is repeated requested, leading to significant data locality.

To further strengthen our belief that data locality is the main cause here, we also investigated the same two-version approach, but using a data sharing strategy. There is no noticeable difference in energy consumption and performance for the two versions.

5.5 Threats to Validity

In experimental systems research, a fundamental challenge is the vast number of factors across the compute stack. For instance, it is a valid question to ask whether different OS scheduling policies (YUAN; NAHRSTEDT, 2003; MERKEL; BELLOSA, 2006), different processor and interconnect layouts (KUMAR et al., 2003; SOLERNOU et al., 2013), and different VLSI circuit designs (A.CHANDRAKASAN; SHENG; BRODERSEN, 1992), have impact on results. They clearly all do. Our study takes a route common in experimental programming language

[3]The `invokeall` method in the second version is part of the Java ForkJoin API. It forks all tasks and then joins them all. Through inspecting its source code, we find no “magic” that would otherwise skew the results.

research, by constructing experiments over representative system software and hardware, and the results are empirical by nature.

To take a step further, we seek to gain a preliminary understanding of how variations in the underlying system impact our results. In particular, we focus on configuration variations of the language runtime. The primary goal is to understand the stability and portability of our results.

Heap Size. Heap size settings are known to impact JVM performance (GEORGES; BUYTAERT; EECKHOUT, 2007). Figure 5.18 shows the energy consumption and performance under different settings of maximum heap sizes (to trigger GC) for `sunflow`; the rest of the JVM settings are identical to those described in Section 5.3. When maximum heap size is restricted to a very low level – such as 20MB for `sunflow` – both energy consumption and performance go higher significantly. We speculate the additional overheads result from VM allocation and garbage collection. Variations in energy consumption that stem from heap size appear to be small if the maximum heap size is higher. While examining this benchmark without setting a fixed maximum heap size, we observed that its heap usage reaches a peak of more than 50MB before GC is triggered. Fixing the heap size at 20MB may have triggered significantly more GC.

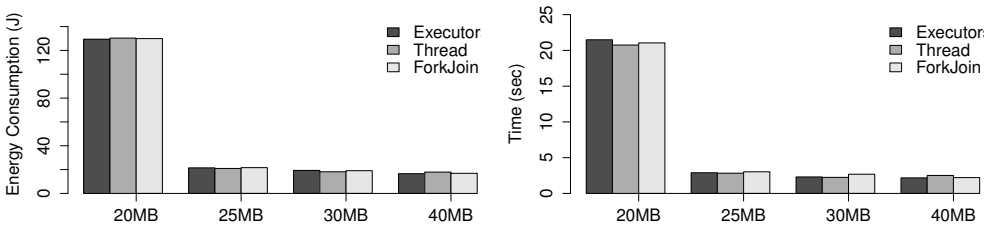


Figure 5.18: Heap Size Effect (`sunflow`, 32 threads, 256 tasks, 256 as image data size)

Garbage Collection Strategy. To gain a preliminary understanding of how GC strategies may pose a threat to the validity of our results, we constructed experiments over 5 GC options over Hotspot: (a) SerialGC: the stop-the-world serial collector, (b) ParallelGC: the parallel collector, (c) PrallelOldGC: the parallel collector with data compression, (d) ConcMarkSweepGC: concurrent mark sweep collector, and (e) G1GC: the garbage-first collector. All have been specified by ORACLE (2014). Figure 5.19 shows the results for `xalan`.

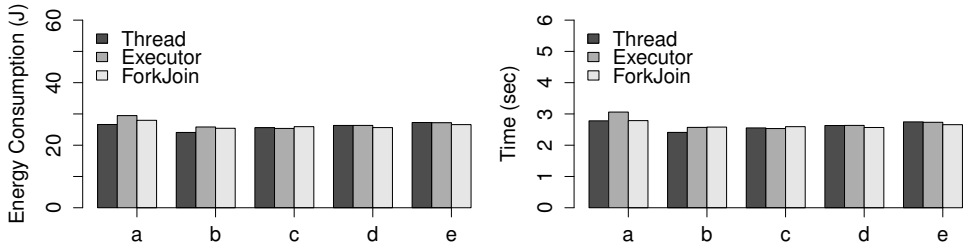
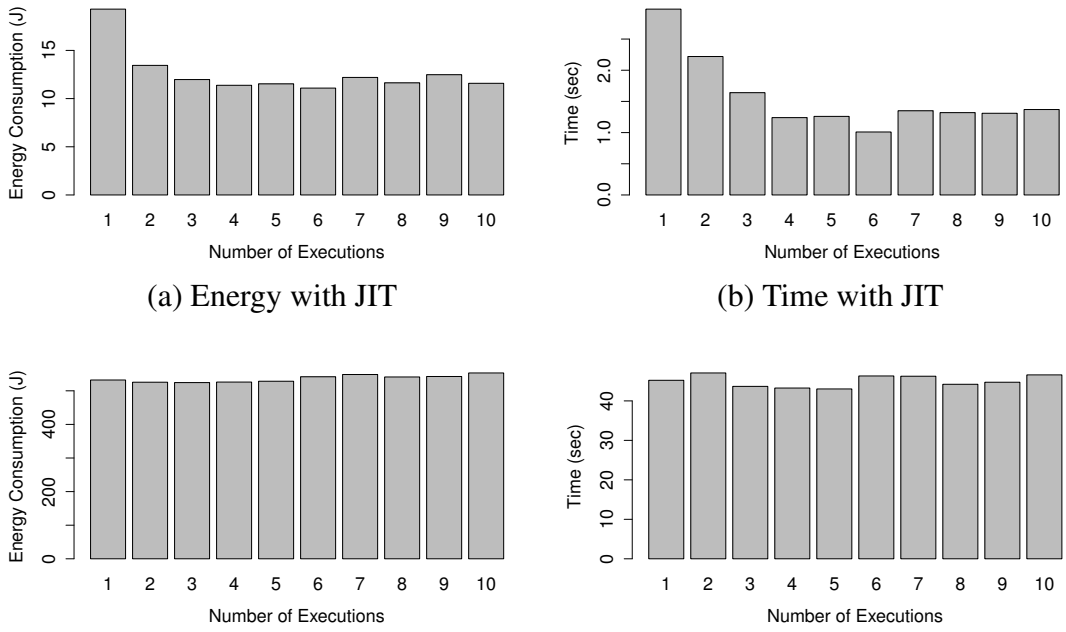


Figure 5.19: GC Effect (`xalan`, 32 threads, 64 tasks, 300 transformation files. GC strategies are: `a`: SerialGC, `b`: ParallelGC, `c`: `PrallelOldGC`, `d`: `ConcMarkSweepGC`, `e`: `G1GC`)

As shown, GC strategies do have observable impact on program energy consumption. In the context of this study, the effect is relatively mild, within $\pm 10\%$. A precise relationship between GC and energy consumption is a complex topic beyond the scope of this study.

Just-In-Time Compilation. Just-In-Time (JIT) compilation dynamically optimizes the program and is known to have significant impact on performance. Predictably, JIT also has direct impact on energy consumption. Figure 5.20 shows the effect of JIT on `sunflow`.



(a) Energy with JIT

(b) Time with JIT

(c) Energy without JIT

(d) Time without JIT

Figure 5.20: JIT Effect (`sunflow`, 32 threads, 256 tasks, 256 as image data size, 10 runs on X-axis)

Here, the X-axis represents 10 "hot" runs of `sunflow`, *i.e.*, a top-level loop that encompasses 10 executions within one JVM. With JIT, early runs incur higher energy/time overhead than later runs, as illustrated in Figure 5.20(a) and Figure 5.20(b). Also note that energy/performance behaviors do stabilize after a number of runs. With JIT disabled, both energy consumption and performance are uniform, as shown in Figure 5.20(c) and Figure 5.20(d). Both of them however are also significantly worse than their JIT counterparts.

Moreover, the growth of energy and time is not proportional. Execution time increases by 33x from using JIT to not using JIT, whereas energy consumption increases more than 45x. For instance, for the 10th `sunflow` execution, the average power consumption using JIT was $85.47W$, and when not using it was $118.35W$. After a more detailed inspection in the data, we observed that although the JIT executions recorded the highest power consumption ($175.3W$ using JIT and $166.3W$ not using JIT), non-JIT dominates the executions with higher power consumptions (3rd quartile: $163.2W$), that is, consuming more power, than the approach using JIT (3rd quartile: $154.6W$).

In Section 5.3, we explained our data collection strategy as averaging the last runs of JIT-enabled executions. This decision stems from our observations here: (1) JIT-disabled executions incur energy/performance overhead unrealistic to common use of Java applications, and (2) later runs of JIT-enabled executions do stablize in terms of energy consumption and performance.

Platform Variations. As a final experiment, we ran some of the benchmarks on a different machine: an 8-core AMD FX-8150 processor (Bulldozer architecture) with 16GB of DDR 1600 memory, running Debian 3.2.46-1 Linux (kernel 3.2.0-4-amd64) and Oracle HotSpot 64-Bit server VM, JDK version 1.7.0_45, build 18. Figure 5.21 shows the results for `n-queens` benchmark.

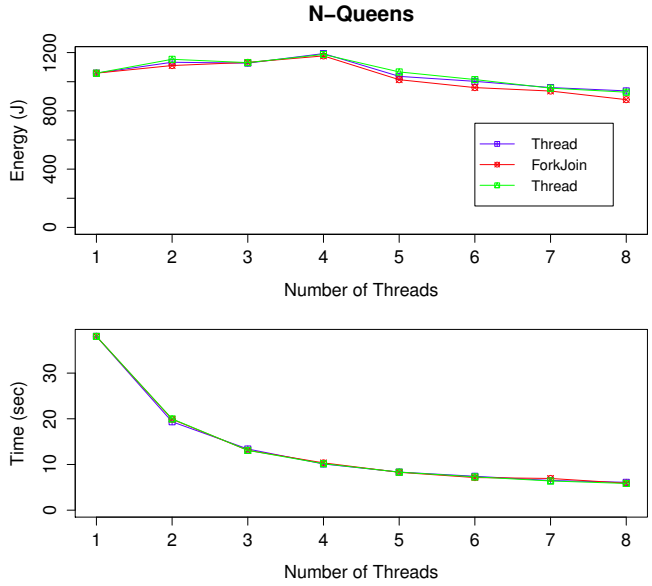


Figure 5.21: Energy/Performance on Alternative Platform

The benchmarking results show similar trends. For instance, the Λ curve recurs, peaking at 4 threads – the same behavior for this benchmark when using 32 processors. The thread management styles behave similarly when compared to the 32-core machine. Still, ForkJoin style outperforms Thread and Executor.

5.6 Summary

In this chapter, we present a study on how concurrent programming practices may have impact on energy consumption. Our results suggest that different constructs for managing concurrent execution can impact energy consumption in different ways, and energy consumption is determined by the choice of thread management constructs, the number of threads, the granularity of tasks, the size of the data, and the nature of data access. This study is a step toward a better understanding of the interplay between energy efficiency and performance.

6

Understanding and Overcoming Bottlenecks in Java ForkJoin Applications

The lack of refactoring incurs technical debt.

—MIRYUNG KIM

In last chapter, we present a comprehensive empirical study on real-world FORKJOIN applications, identifying potential bottlenecks against parallelism in these applications, illustrating their impacts on systems performance, and demonstrating how simple changes on source code can make a big difference (Section 6.3). We summarize our findings as 6 potential bottlenecks latent in FORKJOIN applications, particularly focusing on how data management and thread management interact with FORKJOIN's work-stealing scheduler(Section 6.4). We finish this chapter by presenting our refactoring tool (Section 6.5), with our evaluation (Section 6.5.2).

6.1 Overview

As parallel applications become pervasive, the significance of providing performant, easy-to-use parallelism support has never been so critical for the design of the Java language. Unlike parallel programming in C-like languages or scientific programming in MPI-like frameworks, the design of Java parallel programming models needs to place more emphasis on programmer productivity. In this backdrop, the FORKJOIN framework LEA (2000) rises as an important parallel framework for Java. FORKJOIN is intuitive for programming task-parallel and data-parallel jobs with a divide-and-conquer nature. Since its inception in 2010, it has become the bedrock of numerous applications — many of which we will study in this chapter — and more importantly, it has been the foundation of newer Java concurrent libraries, such as Java Streams TUTORIALS (????) and many important state-of-the-art parallel language frameworks, such as Scala (HALLER; ODERSKY, 2009), Groovy GROOVY (2015) and Clojure CLOJURE (2015).

Apart from ease of use, efficiency is another important goal of the FORKJOIN design, just

as any modern parallel programming framework. Unknown to many application programmers, FORKJOIN employs a state-of-the-art work-stealing runtime LEA (2000). While work stealing provides many benefits in resource utilization and scalability, efficient stealing dictates careful coordination across the layers of applications, runtime systems, and the OS. This requires knowledge about how these different layers interact, of which many Java programmers are unaware. It is not uncommon that the executions of seemingly legitimate programs give rise to unexpectedly poor performance. The goal of this work is to provide a better understanding—as well as raise the awareness—of the subtleties and common performance pitfalls in FORKJOIN programming through a comprehensive study of characteristics and behaviors of real-world FORKJOIN applications.

Since performance and energy efficiency of parallel applications is a topic that has been extensively studied, the first question one may ask is how different a FORKJOIN program is from other parallel programs and whether these differences are large enough to be worthy of a new study. We observe that FORKJOIN applications face a distinctive set of challenges that did not exist in other parallel frameworks:

- *Data Management with Work Stealing*: the Java runtime primarily allocates objects in the heap, and deallocation is managed by garbage collection. This is in sharp contrast with other established work-stealing frameworks such as Cilk, where data are routinely represented as arrays of primitive data types. As a result, how data are managed — allocated, distributed, localized, aggregated, deallocated among worker threads in a work-stealing scheduler — plays a much more pivotal role in the performance of FORKJOIN applications.

- *Thread Management with Work Stealing*: work stealing presents a unique flavor of management on synchronization and thread states. Unfortunately, legacy parallel Java programs heavily rely on locks (*e.g.*, synchronized methods) and explicit thread state managements (*e.g.*, sleep) PINTO et al. (2015). As an application-level framework, FORKJOIN is faced with diverse applications developed by programmers with uneven skills.

Do these challenges manifest themselves in modern parallel Java applications? If so, what kinds of bottlenecks can they create in parallelism? How severe are these bottlenecks in terms of hurting performance and energy? Is there generalizable wisdom that can be shared with FORKJOIN programmers to avoid the bottlenecks?

This Chapter This chapter is a quest for answers to these questions. We present a comprehensive empirical study on real-world FORKJOIN applications, identifying potential bottlenecks against parallelism in these applications, illustrating their impacts on systems performance, and demonstrating how simple changes on source code can make a big difference. We summarize our findings as 6 potential bottlenecks latent in FORKJOIN applications, partic-

ularly focusing on how data management and thread management interact with FORKJOIN's work-stealing scheduler.

Our study is the *first* empirical study to bridge the gap between modern software engineering and systems in the arena of work stealing. It is unique in its *application-driven*, *systems-oriented* perspective.

First, our study is *application-driven*. In systems research, developing a new runtime, OS, or architecture to optimize work stealing has a long history (*e.g.*, DING et al. (2012); WANG et al. (2010); AGRAWAL et al. (2008)), including some over managed runtimes like Java (GUO et al., 2010; MICHAEL; VECHEV; SARASWAT, 2009; KUMAR et al., 2012). Although there are numerous debates on the comparative strengths or weaknesses of FORKJOIN and other parallel frameworks, the focus of this chapter is neither on assessing FORKJOIN in the landscape of parallel programming, nor on making a judgment call on any mechanism in FORKJOIN. Instead, the goal of our study is to understand how real-world applications behave and how small changes can be made to unleash the power of FORKJOIN in these applications. The bottlenecks we have identified are not of the FORKJOIN design, but of *applications* that use FORKJOIN. In addition, FORKJOIN is the first implementation of work stealing that has attracted a large number of application developers, and hence, understanding FORKJOIN applications provides useful insights in coming up with a fair evaluation of the idea of work stealing from a practical viewpoint.

Second, our study is *systems-oriented*. Unlike traditional software engineering studies, we are aimed at finding the root causes of the bottlenecks on the systems stack, such as how each bottleneck may potentially hamper the behavior of work stealing, memory allocator, garbage collector, and cache. Our analysis considers a diverse set of performance metrics, including energy consumption, work stealing runtime status, memory consumption, as well as execution time. Beyond its implication to Java parallelism support, we believe our study may be also interesting towards the understanding of the design of runtime systems.

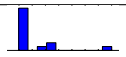
This chapter makes the following contributions:

- We present a comprehensive application-driven investigation into 32 real-world FORKJOIN projects from GitHub, with a total of 791K LOC.

- We summarize potential hurdles of parallelism in FORKJOIN applications into 6 bottlenecks, and present an in-depth systems-oriented analysis on the root causes of these bottlenecks in a work stealing runtime and JVM.

- We develop a bottleneck detection and refactoring tool FJDETECTOR that can perform interactive source-code-level optimizations of FORKJOIN applications. The optimized applications can produce an average of 26% of performance improvement and 23% of energy savings. Our optimization patches were confirmed by the majority of application developers we communicated with.

Table 6.1: Quantitative characteristics of the analyzed projects.

#	Mean	Median	Std.	Histogram
LOC	19,157.73	3,099.0	58,869.86	
Commits	103.6	23.0	188.59	

6.2 Methodology and Caveats

This section describes the programs that we analyze, as well as the infrastructure and methodology that we use to conduct the study.

Methodology To find real-world programs using ForkJoin, we searched in Github for the key word "ForkJoin" and selected a set of 32 open-source projects, covering a wide range of application domains from supervisor management to raytracing. Our selection criteria are: (1) they must be of reasonable length; we removed all programs that are tutorial in nature; (2) they should demonstrate continuous development, *i.e.*, multiple commits over a duration; (3) they should be recent, but not currently under rapid changes. For example, we did not select any projects whose first commit and last commit are both within 6 months; (4) they must be able to compile and run. Table 6.1 provides the overall quantitative characteristics of all projects included in our study. The detailed information of a sample of these programs is shown in Table 6.2.

We ran each selected application in a machine with a 2×8-core (32-core when hyper-threading is enabled) Intel(R) Xeon(R) E5-2670 CPU (2.60GHz) and 64GB of DDR3 1600 memory, running Debian 6 (kernel 3.0.0-1-amd64) and Oracle HotSpot 64-Bit Server VM (build 25.5-b02, mixed mode, JDK version 1.8.0_05-b13). The machine has three cache levels (L1, L2 and L3), whose sizes are 64KB per core (128KB total), 256KB per core (512KB total), and 3MB (smart cache), respectively. All experiments were performed in the OS-exclusive mode without any other loads running simultaneously.

The default settings of both the OS and the JVM were used. In particular, (1) the power management of Linux is the default *ondemand* governor, which dynamically adjusts CPU core frequencies based on system workloads. (2) For the JVM, the parallel garbage collector is used and just-in-time (JIT) compilation is enabled. The initial heap size and maximum heap size were set to be 1GB and 16GB respectively. We ran each benchmark 10 times; this is implemented by a top-evel 10-iteration loop over each benchmark. The reported data is the average of the last 3 runs to warm up the JIT optimizations (PINTO; CASTOR; LIU, 2014a). Hyper-threading is enabled and the Turbo Boost feature is disabled.

Energy consumption was measured using `jRAPL` (LIU; PINTO; LIU, 2015), a framework that contains a set of APIs for profiling Java programs running on CPUs with Running Average Power Limit (RAPL) (DAVID et al., 2010) support. Originally designed by Intel to enable chip-level power management, RAPL is widely supported in today's Intel architectures, including

Table 6.2: A sample of projects used in this study. LoC encompass only non-blank and non-commented lines of code computed using the `cloc` program.

Projects	*# LoC*	*# Commits*	*# Last Commit*	*# Bottlenecks*
DocumentIndexing	1,127	1	07-2013	1, 4
hyungjin	873	16	04-2015	1
mills	8,846	50	08-2015	1
itemupdown	4,925	2	08-2013	2
jAcer	4,476	35	12-2014	2
educational	1,323	7	05-2014	2
scalatuts	253	5	11-2013	2
knn	3,099	27	11-2014	2
doms-transformers	3,714	254	06-2014	2
ForkAndJoinUtility	127	12	03-2013	2
Solitaire	1,527	39	11-2011	2
mywiki	1,920	17	10-2012	2
MagicSquares	664	153	10-2013	2
ejisto	12,330	274	06-2014	2, 3
exhibitor	15,314	701	11-2014	2, 3, 4
cq4j	5,815	23	10-2013	2, 3
netflixoss	231,361	1	09-2013	2, 3
javaOneBR-2012	518	4	12-2012	2, 3
jadira	46,095	630	08-2015	3
ecco	5,849	119	02-2015	3
conflate	934	9	11-2013	3
bazzar-base	7,766	15	10-2013	3, 4
CSSTProto	10,721	17	01-2012	4
Fibonacci	79	2	08-2013	5
Mandelbrot	1,442	30	06-2015	5
Solitaire	1,527	39	11-2011	5
Matrices	2,356	15	04-2015	5
LockedBasedGrid	1,390	1	2013	5
Basic-Blocks	4,821	41	08-2015	5
warp	15,287	338	01-2015	6
j7cc	5,110	76	09-2013	6
lowlatency	3,018	18	02-2015	6

Xeon server-level CPUs and the popular i5 and i7 processors. RAPL-enabled architectures monitor energy consumption and store the information in Machine-Specific Registers (MSRs). The RAPL support can access both CPU core and uncore data (*i.e.*, caches and interconnects) as well as data on DRAM energy consumption. RAPL-based energy measurement has been used in recent work (*e.g.*, KAMBADUR; KIM (2014)), and its precision and reliability have been extensively studied (HäHNEL et al., 2012).

Caveats As with all empirical studies, there is an inherent risk that our findings may

	work stealing	fine-grained parallelism	dynamic allocation	GC	unstructured synchronization	programmable thread states
Fortran	no	no	no	no	yes	uncommon
Pthread	no	no	uncommon	no	prevalent	prevalent
OpenMP	no	no	uncommon	no	uncommon	uncommon
MPI	yes	yes	uncommon	no	uncommon	uncommon
Go	yes	yes	uncommon	yes	uncommon	uncommon
Cilk	yes	yes	uncommon	no	uncommon	uncommon
Java threads	no	no	prevalent	yes	prevalent	prevalent
X10	yes	yes	prevalent	yes	uncommon	uncommon
FORKJOIN	yes	yes	prevalent	yes	prevalent	prevalent

Table 6.3: Representative parallel programming frameworks.

not be representative. While the FORKJOIN applications we selected cover representative and important domains and workloads, there may still be missing categories. However, we took care to select diverse projects from GitHub. At the very least, these projects are actively maintained: developers from 9 out of the 15 projects replied to our recommended patches.

6.3 Case Study: An Overview

Our study is intended to inspire researchers and practitioners to develop techniques that can detect, fix, and avoid parallelism bottlenecks in real-world FORKJOIN applications.

Why* FORKJOIN*? We begin this section by placing the Java FORKJOIN framework in the context of existing parallel programming frameworks. As illustrated in Table 6.3, the FORKJOIN system is a unique combination of the following features :

- It employs a *work-stealing scheduler* that performs deque management and structured synchronization;
- It provides support for *fine-grained parallelism*. It is routine for a short 1-minute FORKJOIN application execution to produce tens of thousands of tasks, each of which may only execute in milliseconds;
- Due to its Java base, a FORKJOIN program makes heavy use of *dynamic allocation* and *unstructured synchronization* (*i.e.*, object locks), and employs *automatic memory management* (*i.e.*, garbage collection).
- FORKJOIN and the underlying Java has a rich set of APIs that expose numerous thread states for application programmers to manipulate.

The performance and energy consumption for systems with this unique set of features — and especially their combined effects — has rarely been empirically studied before.

A Bottleneck Example Next, we use a motivating example to answer the following three questions: (1) How are performance bottlenecks in FORKJOIN programs different from

those in other parallel applications? If they are very different, what are the unique features of the FORKJOIN bottlenecks? (2) Do these bottlenecks have significant impact on performance and energy consumption of the applications? (3) At what level can and should these bottlenecks be fixed?

In data-parallel programming, two strategies exist in achieving divide-and-conquer. In the first pattern, which we term *cascaded division*, the programmer recursively divides the data into slices until a *sequential slice*, *i.e.*, one meant to be handled by a task, is reached:

```
class Task extends RecursiveAction {
  public Task (User[] u, int lo, int hi) { ... }
  protected void compute() {
    if (hi - lo < 16) {
      for (int i = lo; i < hi; i++) f(u[i])
    } else {
      int mid = (lo + hi) / 2;
      invokeAll(new Task(u, lo, mid),
                new Task(u, mid, hi));
 } } }
```

In the second strategy, which we term *flat division*, the programmer may divide all data into sequential slices all at once:

```
class Task extends RecursiveAction {
  public Task(User[] users, int lo, int hi) { ... }
  protected void compute() {
    if (hi - lo == 16) {
      for (int i = lo; i < hi; i++) f(users[i]);
    } else {
      int tasks = 16;
      for (int i = 0; i < users.length; i = (i+tasks))
        new Task(users, i, i + tasks).fork();
 } } }
```

We observed that most FORKJOIN programmers adopted the cascaded division strategy. We speculate this may result from its prevalent use in online code samples and tutorials, including some from the JDK documentation itself. The less obvious question is *why* cascaded division is favored. As it turns out, this can be traced back to the history of work stealing design in Cilk, guided by a now well-known mechanism called the *work-first principle*. There, each forked task is executed in the *current* worker thread, and it is the *continuation* (whatever the next program counter points to) that is pushed onto the deque and may be later stolen by a thief. This principle, also dubbed as "thief steals continuation", suggests that flat division would inadvertently *linearize* stealing: at any given moment, there is only *one* work item among *all* deques ready for stealing. In other words, work-first work stealing crucially depends on cascaded division to be effective.

The lesser known fact is that FORKJOIN is *future*-based: the forked task is the one that

is pushed onto the deque and may be stolen by a thief. In this design, all tasks created in the flat division will immediately be pushed onto the deque and ready for stealing. In contrast, the cascaded division will "roll out" the tasks gradually, potentially reducing the opportunity for stealing.

We conducted experiments for the two division strategies, with the results shown in Figure 6.1. Figures at the top, middle and bottom show, respectively, stealing count, execution time and energy consumption. The X-axis represents data size, the length of the `users` array. To construct a fair comparison, we make sure the size of sequential slices in two strategies are the same (16 in our experiment). As it turns out, flat division enjoys a noticeable margin of improvement for both execution time and energy consumption, and there is a significant difference in the number of steals.

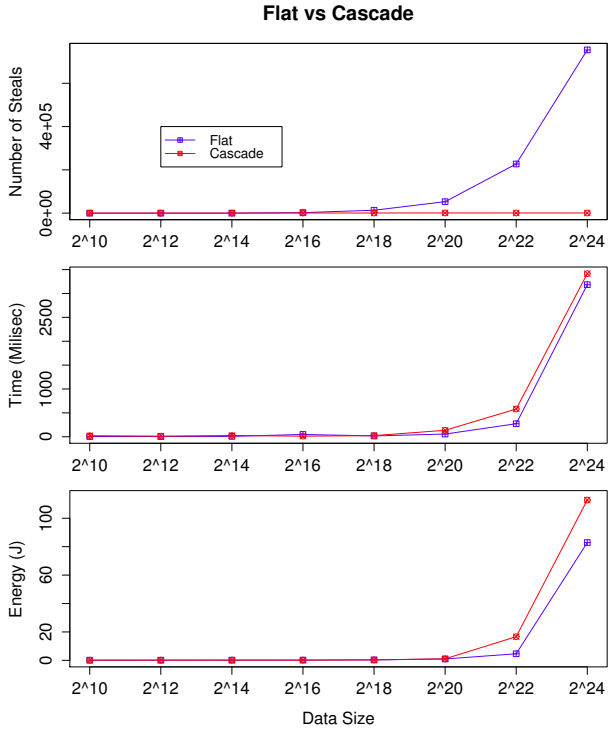


Figure 6.1: A comparison on execution time, energy consumption and steal counts between the flat and cascade style programming.

This example can help us answer the questions asked earlier: (1) parallel bottlenecks in FORKJOIN applications can be very different than those in other parallel applications due to the subtle interactions between the high-level application and the low-level work-stealing runtime; (2) from the performance measurements shown above, they have non-trivial impact on application performance and energy consumption; (3) while optimizations can be developed at different levels of the compute stack to improve performance, the bottleneck described here is highly related to program semantics — for example flat division outperforms only for data parallel programs — and simple changes in the source code may result in significant performance gains.

6.4 A Study of Parallelism Bottlenecks

Motivated by the observation that many parallelism bottlenecks can be easily removed by making minor changes to the source code, this section studies the sources of bottlenecks and groups them into six main patterns.

Bottleneck 1: Cascaded Division in Data Parallelism

The program discussed in §6.3 is an example of this bottleneck. We have found 3 instances of this usage among the 32 projects studied. One example is project `DocumentIndexing`, which searches for text files and indexes them. The benchmark begins with a given folder, and it creates a new task for each subfolder found. The number of tasks is directly related to the number of visited folders. During the execution of a moderate-sized workload, the program visited 1,000 folders and indexed 2,000 text files. Figure 6.2 shows the results.

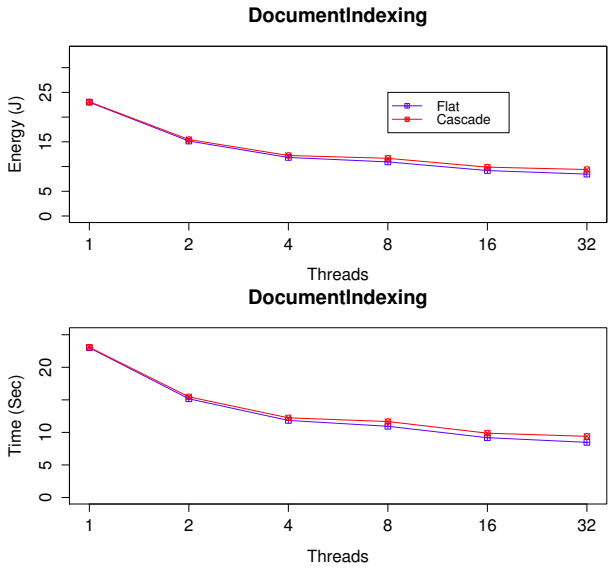


Figure 6.2: A comparison of energy and performance between the flat and cascade programming styles, for varying numbers of threads in `DocumentIndexing`.

Figure 6.2 presents a similar behavior of the one observed in Figure 6.1, which corroborate our initial observation. The flat programming style presents a slightly improvement over the cascade one. This is an interesting fact in particular because the benchmark present in Figure 6.1 is CPU-intensive, whereas the current one is IO-intensive.

Bottleneck 2: Copy on Fork For data-intensive applications, a performance-sensitive dimension of design is data distribution, *i.e.*, how data are spread through parallel execution units. In divide-and-conquer frameworks — including FORKJOIN — the general strategy is to represent the data as an indexible structure, *e.g.*, a (potentially multidimensional) array, which in turn can be partitioned into slices and fed to individual parallel execution units. Both CPU frameworks such as OpenMP, MPI, or Cilk and GPU frameworks such as CUDA follow the same general route.

This simple process may pose challenges to a FORKJOIN programmer. In particular, data in Java are often represented as objects. The combination effect of aliasing and shared-memory

programming implies that data distribution "by reference" at forking time may introduce race conditions.

As a conservative approach, many FORKJOIN programmers choose to copy data at the forking time. Observe the `copyOfRange` method below:

```
import static Arrays.*;
class Task extends RecursiveAction {
  public Task (User[] u) { ... }
  protected void compute() {
    if (u.length < N) { local(u); }
    else {
      int split = u.length / 2;
      User[] u1 = copyOfRange(u, 0, split);
      User[] u2 = copyOfRange(u, split, u.length);
      invokeAll(new Task(u1), new Task(u2));
} } }
```

Beyond the obvious consequences such as memory bloat (XU et al., 2009), excessive copying turns out to be uniquely unfriendly to FORKJOIN, for a number of reasons. (1) As a fine-grained parallelism framework, most tasks are completed within milliseconds. Copying upon fork implies the dominating growth of short-lived objects, creating a severe burden for garbage collection. (2) The cascaded division common in FORKJOIN applications means that data are copied at every level of recursion, potentially leading to an O(log n) growth in memory. In contrast, copying for flat data partitioning can only lead to a constant growth in memory. (3) Unlike copying with flat data partitioning where all allocations are done once and for all, a strategy somewhat friendly for the memory allocator due to batching, copying with cascaded data partitioning leads to frequent yet intermittent allocation requests, hampering performance.

Among the 32 programs we have studied, we found 18 occurrences of this bottleneck, in 15 FORKJOIN programs. Fixing the bottleneck requires simple modification of the source code that shares the input data structure and lets subtasks work on distinct regions of the data structure. Figure 6.3 shows the energy gains from fixing this bottleneck. Clearly, the energy consumption is reduced in all the refactored programs. The average reduction in energy consumption is 12.63%. The execution time decreases proportionally. Interestingly, 9 out of the 15 analyzed projects cross the 10% barrier of energy savings. However, 5 of the analyzed projects have energy savings of less than 5%. For the projects above 5%, the minimum energy saving was 8.23% (for `itemupdown`), and the maximum was 23.51% (for `MagicSquares`). After inspecting these projects, we have observed that the amount of energy savings is related to the width of forking. That is, the more the program creates redundant copies of the data structure, the more effective our refactoring is.

Next we provide an in-depth analysis for benchmark `MagiSquares`. Figure 6.4 shows the comparison results before and after eliminating copies for varying numbers of threads in `MagiSquares`.

`MagicSquare` is an implementation of the magic square puzzle. It can be described as "*a square array of numbers consisting of the distinct positive integers* $1,2,\ldots,n^2$ *arranged*

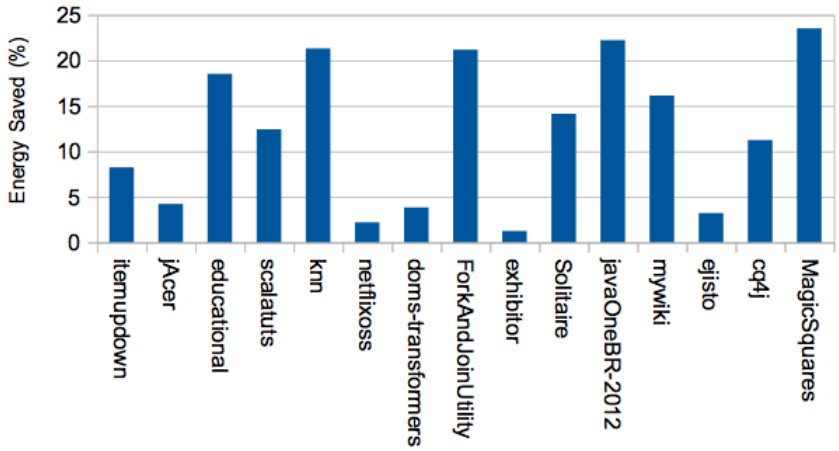


Figure 6.3: A summary of the energy savings when removing the *Copy on Fork* bottleneck.

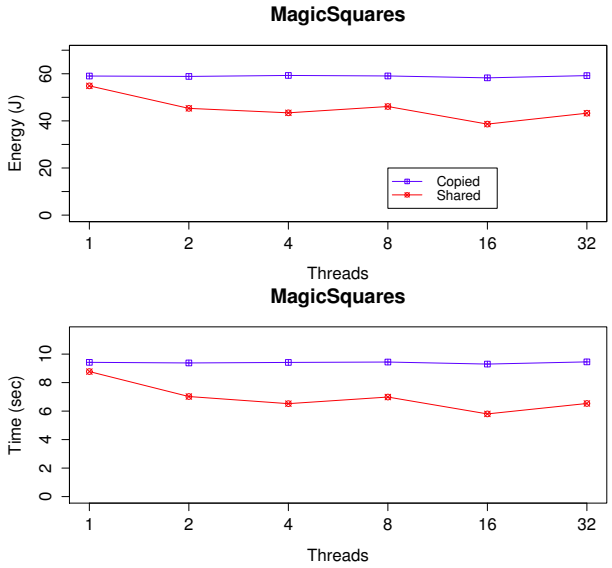


Figure 6.4: A comparison on energy and performance with varying numbers of threads before and after copies are removed in `MagicSquares`.

such that the sum of the n numbers in any horizontal, vertical, or main diagonal line is always the same number"[1]. It is a CPU intensive computation. This program has only one test case, which exercises the whole program. The data-parallel computation is based on the number of permutations available, which represents all possible rows, columns, and diagonals. Each parallel task attempts to construct a matrix whose first row is the permutation and whose first column is another permutation that begins with the same entry and contains no other duplicate entries. The algorithm attempts to find sum permutations to fill in the remaining rows and columns. When sharing the data structure, we saved the program from creating 128 additional data structures (with integer data type), leading to a 23.51% energy saving, when running with 32 threads.

Moreover, one of the unique advantages of `jRAPL` is that it provides ability to observe how different hardware components behave in terms of energy usage. Using this feature, we present inspect the energy consumptions of three different hardware components: DRAM, CPU,

[1] http://mathworld.wolfram.com/MagicSquare.html

and Uncore. Figure 6.5 shows the results.

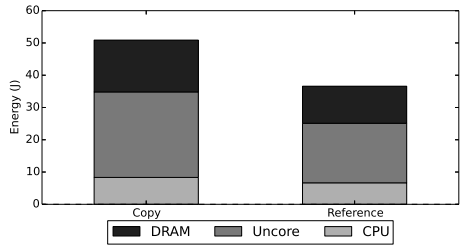


Figure 6.5: Energy consumptions for each hardware component before and after removing excessive copies (`MagicSquares`, 32 threads).

As we can see, roughly the same amount of CPU energy was consumed before and after removing the copies (*i.e.*, 8.32 Joules and 6.67 Joules, respectively). However, the difference is more obvious when the energy consumptions of DRAM and Uncore are compared. Due to the excessive object creation, DRAM and Uncore of the original version consume 1.39× and 1.43× more energy than the optimized version.

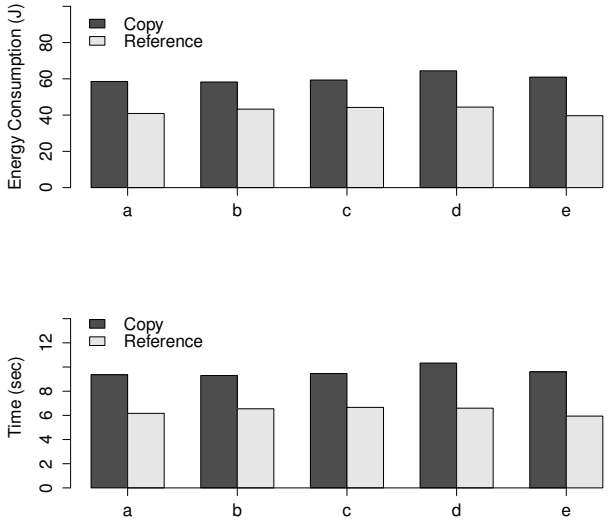


Figure 6.6: A Comparison of GC costs (`MagicSquares`, 32 threads; GC algorithms are: `a`: SerialGC, `b`: ParallelGC, `c`: ParallelOldGC, `d`: ParallelNewGC, `e`: G1GC).

Since *Copy on Fork* creates large volumes of small, shortly-lived data structure objects, it is interesting to understand how different GC algorithms may impact our results, we conducted experiments over 5 GC options in Hotspot: (a) SerialGC: the stop-the-world serial collector, (b) ParallelGC: the parallel collector, (c) ParallelOldGC: the parallel collector with data compression, (d) ConcMarkSweepGC: concurrent mark sweep collector, and (e) G1GC: the garbage-first collector. Figure 6.6 shows the results for `MagicSqure`. For almost all algorithms, the fix can speed up garbage collection by 20%–40%.

Bottleneck 3: Copy on Join The counterpart of *Copy on Fork* is *Copy on Join*: after having joined on its subtasks, a task must usually combine the results of the subtasks into a result for the larger problem. Consider the following program, extracted from the `cq4j` benchmark.

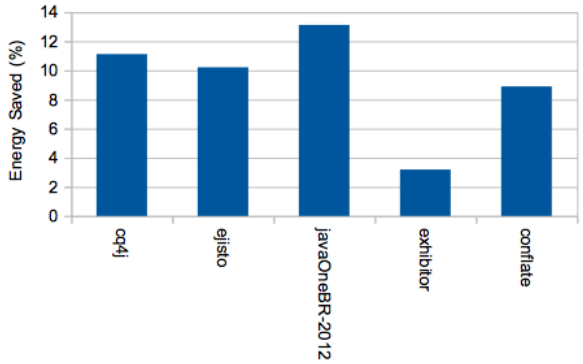


Figure 6.7: A summary of the energy savings after removing the Copy on Join bottleneck.

```
protected List<T> compute() {
  int size = dataSource.size();
  if (size < FORK_SIZE) {
    return computeDirectly();
  } else {
    List<T> result = new ArrayList<T>();
    int mid = size / 2;
    RecursiveFilteringTask<T> first = new RecursiveFilteringTask<T>(filter,
        dataSource.subList(0, mid));
    first.fork();
    RecursiveFilteringTask<T> second = new RecursiveFilteringTask<T>(filter,
         dataSource.subList(mid, size));
    second.fork();
    result.addAll(first.join());
    result.addAll(second.join());
    return result;
  }
}
```

As one reader might observe, this particular code snippet suffer from the same bottleneck previously explained (creating sublists of the current data structure). However, this benchmark also presents a different bottleneck. At the end of the execution, an expensive operation `addAll` is invoked to copy merge collections. *Copy on Join* has many negative consequences similarly to *Copy on Fork*, with one additional unique drawback: since joining in a work stealing system is implemented by barriers, *Copy on Join* increases the wait time at barriers, particularly unfriendly for *energy consumption*. Note that this is an established fact PARK et al. (2007a); LI; MARTINEZ; HUANG (2004); RIBIC; LIU (2014); PINTO; CASTOR; LIU (2014a): barrier wait at the low level is either implemented as spin locks or context switch, both of which can lead to energy waste without contributing to program progress.

We have found 5 occurrences of this bottleneck in the 32 programs studied. A fix of this bottleneck is similar to that of *Copy on Fork*: a shared data structure can be passed into subtasks

to carry results. After applying these changes in 5 programs, we have achieved overall 3% – 13% energy savings. The detailed results are shown in Figure 6.7.

Bottleneck 4: Scattered Data We investigate the impact of data locality on the performance and energy consumption. An important pattern we found is that the execution of a task follows the sequence of *abababc*, where *a* performs memory copies for a subtask, *b* forks the subtask, and *c* does the computation of the current task. The following figure shows a code snippet of this case, extracted from benchmark `CSSTProto`.

```
protected R compute() {
  if (len == 1) {
    RecursiveTask<R> task = createAtomicTask(from);
    return task.invoke();
  } else {
    ForkJoinTask<R>[] tasks = new ForkJoinTask[len];
    for (int i = 0; i < len; i++) {
       ForkJoinTask<R> task = createAtomicTask(from+i)
       task.fork();
       tasks[i] = task;
    }
    R result = tasks[0].join();
    tasks[0] = null;
    for (int i = 1; i < len; i++) {
       R next = tasks[i].join();
       tasks[i] = null;
       result = merge(result, next);
    }
    return result;
}
```

However, this implementation relies on the method named `createAtomicTask`, which creates a new copy of the current task, to avoid potential data races. Notwithstanding, it has an impact on energy consumption and performance. This is first because the copy operation has the potential of polluting caches, increasing the chance of memory round-trips. Second, the number of context switches might also increase, due to the sparse task creations. A possible solution to this problem is to create a list of tasks and, during the `for` loop, add each new task object to the list. After the execution of the `for` loop, one might call the `invokeAll` method, which is responsible for forking and joining all tasks in the list. With this fix, we have observed an energy saving of 9.82% for `CSSTProto`. Regarding cache behavior, we observed that the original implementation had a 34.24% cache misses, whereas the fix reduced it to 31.98%. We also observed a reduction on context switches, from 24,550 to 23,193. Yet, the number of branch misses also reduced: from 1.14% of all branches to 1.14% [2], which we believe is due to the boilerplate code used our initial example; `invokeAll` eliminates the first for loop, then reducing the overall number of branches and, as a consequence, the number of branch misses.

Bottleneck 5: Exacting Intra-Task Synchronization As locks play a central role in Java shared-memory programming and metadata representation, unstructured synchronization

[2] We used the `perf` linux tool to calculate cache misses, context switches, and branch misses.

is pervasive in Java applications. Synchronization occurs via invoking `synchronized` methods or code blocks, or using popular concurrent library classes such as `CountDownLatch`. Improving performance and energy efficiency for systems where unstructured synchronization is the only mechanism to achieve concurrency safety — such as Pthreads or the Java `Thread` library— is a well understood topic.

Mixing unstructured synchronization in a structured parallel system such as work stealing leads to additional subtle interactions between the application runtime and the OS. When unstructured synchronization happens *in the middle* of the task execution, it effectively stalls stealing from that worker. Unfortunately, the stalled worker cannot forgo the current task and select another task from its deque — even if there are many other task items in it — because tasks on the deque in a work stealing system carry inherent logical dependencies, analogous to stack frames. At best, the worker itself can be context-switched by the OS. Observe however, even though there may be thousands of *tasks* in the work-stealing runtime, the number of *workers* — the JVM representation of OS threads — is few, typically smaller than the number of CPU cores. In other words, OS-level context switch may at best help *other* applications in a time-sharing environment, but will not contribute to improving the performance or energy efficiency of the application *itself*.

The most principled solution to avoid the bottleneck is to eradicate unstructured synchronization from Java. There is encouraging progress in recent Java development to support asynchronous abstractions, such as futures). However, it may take time before Java practitioners fully embrace these features (PINTO et al., 2015). In this study, we investigate into legacy programs, attempting to understand how unstructured synchronization is used in the real world.

Overall, we found 7 occurrences of this bottleneck. Surprisingly, we found in a significant number of projects, an easier solution exists: many synchronizations are simply to implement *exact computations*, which can be safely *relaxed* (CARBIN et al., 2012) without creating any impact on correctness (MISAILOVIC; SIDIROGLOU; RINARD, 2012). We use a few concrete programs to illustrate this bottleneck.

Mandelbrot A mandelbrot is a mathematical set of points whose boundary is a distinctive and easily recognizable two-dimensional fractal shape. The `synchronized` block is then used when a task needs to render the fractal image. This is done by calling the `setRGB` method, available on the `BufferedImage` class, as showed in the following code snippet.

Figure 6.8 shows the results for this benchmark, varying the number of threads. Figure on top shows energy (in Joules), whereas figure on bottom shows execution time (in seconds). In this benchmark, a task has a range of values of which it should work on. For our input data (width:

```
if (!isBenchmarking && mandelbrot.isLiveRendering) {
  synchronized (mandelbrot.lock) {
    mandelbrot.renderImage.setRGB(j, i, color.getRGB());
  }
  mandelbrot.repaint();
}
```

1000, height: 10000), the benchmark creates a total of 2,048 tasks. As we can see, there is a great difference between the synchronized version and the unsynchronized one. This difference can be found in both energy consumption and execution time. On average, the unsynchronized version consumes 42% less energy then its counterpart (38% faster). Interestingly, when the number of cores increases, one might expect that the synchronization overhead would create a higher performance penality. However, we found that the proportional difference between the synchronized version and the unsynchronized one, in terms of energy consumption and execution time, remained the same.

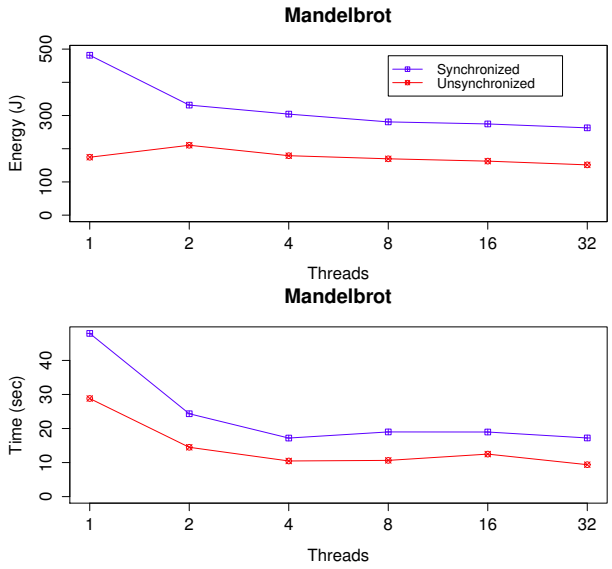


Figure 6.8: A comparison of energy and performance, with and without synchronization, for varying numbers of threads in `MandelBrot`.

After inspecting the implementation, we observed that the method `setRGB` is already `synchronized`, so there is no need to use another synchronization construct to wrap up this single method call. In fact, we could not find any visible difference between the images generated by executions with and without the synchronization. We sent the modified source code as a patch to its developer, who then acknowledged the over-synchronization and accepted our patch[3]. Due to the CPU-intensive nature of this program, we also analyze the energy behaviors of the program under different CPU frequencies using DVFS. Figure 6.9 shows the results.

As expected, the execution time increases as we decrease frequency. However, interestingly, the energy consumption increases in much slower speed. For instance, while the execution time increases 1.97× when the CPU frequency is bumped down from 2.6GHz to 1.2GHz, the unsynchronized version, the energy consumption increases only by 14.29% in the unsynchronized version.

The major reason why unstructured synchronization is much more harmful in FORKJOIN is that a FORKJOIN program can create significantly more threads than a regular parallel applications. For example, program `Fibonacci` creates a total of 331,160,282 tasks when

[3] `https://github.com/catree/SimpleMandelbrotDemo/pull/1`

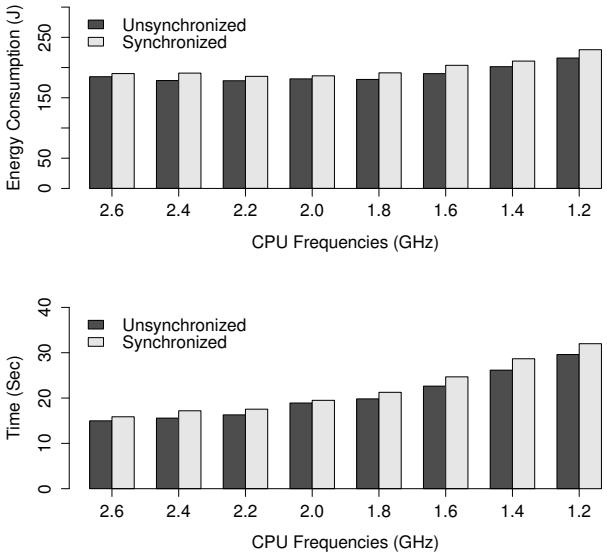


Figure 6.9: Energy consumptions under different CPU frequencies before and after the synchronization was removed (`Mandelbrot`, 32 threads).

performing on number 40. Removing an unnecessary synchronization in the program leads to a 20.52× reduction in energy consumption and a 117.02× reduction in running time. It can be extremely rare for a parallel program under another programming model to create 300 million threads even for processing large workloads.

Bottleneck 6: Sleepy Workers A more extreme case — but along the same line of *Intra-Task Synchronization* — is the use of `Thread.sleep` during task execution. Just as the previous bottleneck, the invocation of this thread management primitive stalls stealing, and explicitly requests OS context switches. From a logical perspective, the intention of the programmer may be to put the *task* to sleep, but unfortunately, the work stealing runtime will place the *worker* to sleep. As described earlier, the worker cannot forgo the sleep-inducing task and pick up other tasks from its deque; neither can the idle CPU core help other workers of the same application. What is worse is that unless the OS has other applications running, an idle core under the widely used on-demand governor would put the core in a low-power state, which later needs a long time to wake up. In a work stealing runtime where competitive performance is of its first priority, user-level sleeping is often more detrimental than beneficial. We found 3 occurrences of this bottleneck.

CTask The `CTask` benchmark presents the worst scenario of this bottleneck. During the sequential execution, this benchmark puts every current task to sleep for a second. Figure 6.10 shows this impact on both performance and energy consumption.

The sleep construct creates significant penalities in both performance and energy consumption. The execution without sleep can be 13,495.26× more energy efficient than the execution with (and 1,917× reduction in running time). After inspecting the source code, we

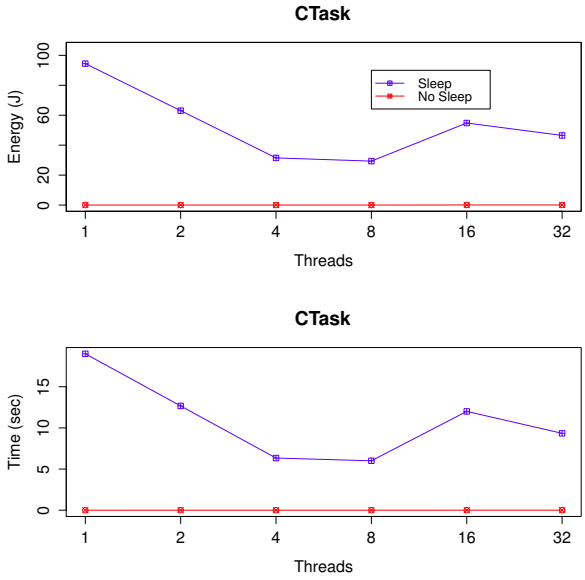


Figure 6.10: A comparison on energy and Performance, with and without thread sleeping, for varying numbers of threads in `ctask`.

observed that the developer used sleep to force the program to wait for a result from another computation. However, this sleep is completely unnecessary, since the computation on which the sleep is waiting is a *synchronous* operation.

6.5 Detecting Refactoring Opportunities

In §6.4, we observed that one of the ways to misuse FORKJOIN is to create copies of data structures within parallel computations when sharing these data structures would be more efficient. For the benchmarks we analyzed, improvements of up to 23.5% in energy consumption (26% in performance) could be obtained by applying this optimization. Based on these observations, we built a tool named FJDETECTORcapable of automatically detecting and refactoring copy-related bottlenecks. Human intervention is only required to judge whether the tool should actually perform the refactoring once a bottleneck is identified. For most of the benchmarks mentioned in Figure 6.7, the copy-related bottleneck was removed using FJDETECTOR.

6.5.1 FJDETECTOR

FJDETECTORstarts its analysis by building the ASTs for the files of the target application. For each Java source file, FJDETECTORscans the class hierarchy of the application, looking for indicators of relevant properties, for instance, whether the class inherits either `RecursiveAction` or `RecursiveTask`.

Detecting bottlenecks requires the understanding of whether (1) the program exhibits data parallelism and (2) it contains a divide-and-conquer-based implementation. For (1), we check if the computation is done on a *data structure*, such as `array` or `List`. Since most of

the `List` methods provide accesses over arrays, our approach handles them in a similar way. FORKJOIN computations are usually described in terms of inner-classes, where the data is passed through the inner-class constructor. Hence, for each parameter of the constructor, we verify (a) if it is a data structure, (b) if it is splitted and copied inside the `compute` method, and (c) if the variables containing the copy results are passed into new instances of the `Task` class.

To understand if a divide-and-conquer approach is used, we first verify if the control flow of method `compute` has an `if-then-else` structure, *e.g.*, if the size of the data structure `u` is greater than `x`, then invoke `compute` on `u`; otherwise, split `u` in two smaller parts. In the FORKJOIN framework, the `if` and `else` parts can be described as sequential and parallel operations, respectively. However, programmers have numerous ways to implement this idiom. To improve usefulness, FJDETECTORcovers three common scenarios: (1) sequential computation in the `if` block and parallel computation in the `else` block; (2) parallel computation in the `if` block and sequential computation in the `else` block; and (3) sequential computation in the `if` block plus a `return` at the end of the block, and the parallel computation in the remainder of the method. Finally, FJDETECTORdetects I/O operations inside the FORKJOIN code by looking for the signatures of I/O-related methods.

Once a bottleneck is confirmed by the developer, FJDETECTORperforms refactoring, *i.e.*, a set of transformations on the FORKJOIN code.Our transformations remove copies by computing indices for each subtask and letting them work on distinct regions of the same (shared) data structure.

6.5.2 FJDETECTOR Results

We have applied FJDETECTORto 15 of the benchmarks listed in §6.2, since they exhibit Bottleneck 2 (§6.4). For all these benchmarks the use of FJDETECTORboth saved energy and improved performance. In 9 of them, the reduction in energy consumption was greater than 10%. In this section, we assess FJDETECTORin terms of the following evaluation questions:

- **EQ1.** Is our approach *useful*?
- **EQ2.** How intrusive is FJDETECTOR?

To answer **EQ1**, we have sent modified versions of the benchmarks to their developers as patches. If these matches are useful, they will eventually be merged into the benchmarks. To assess the intrusiveness of FJDETECTOR, we measured the number of lines of code that FJDETECTORadds to and removes from the benchmarks in order to refactor them. A large number of modifications makes the code harder to understand and modify for its developers.

Subjects Our benchmark set spans a wide range of open-source projects, from 127 to 231K lines of code, implemented by between 1 and 14 different developers. The benchmarks were selected due to the presence of Bottleneck 2 (§6.4). They vary across multiple

Table 6.4: The benchmarks used in this study. Columns *Add* and *Del* indicate the number of additions and deletions applied by FJDETECTOR. The symbols ✓, × and — on the *Repplied?* and *Accepted?* columns mean, respectively, 'yes', 'no', 'no response yet'.

Projects	*Add*	*Del*	*Replied?*	*Accepted?*	*Savings*
`itemupdown`	13	7	—	—	8.23%
`jAcer`	14	8	✓	✓	4.21%
`educational`	13	17	—	—	18.51%
`scalatuts`	12	6	✓	✓	12.41%
`knn`	20	8	✓	✓	21.3%
`netflixoss`	17	13	—	×	2.18%
`doms-transformers`	20	9	✓	—	3.82%
`ForkAndJoinUtility`	13	6	✓	✓	21.17%
`exhibitor`	21	15	✓	—	1.23%
`Solitaire`	14	5	—	—	14.12%
`javaOneBR-2012`	13	4	✓	✓	22.21%
`mywiki`	17	18	—	—	16.12%
`ejisto`	18	9	✓	✓	3.2%
`cq4j`	14	7	—	—	11.23%
`MagicSquares`	12	11	✓	—	23.51%

dimensions, including lines of program code, number of developers, developer age, and project domain (NAGAPPAN; ZIMMERMANN; BIRD, 2013). Table 6.4 lists the selected benchmarks.

Results of EQ1 Using our corpus of 15 projects, we applied our approach in 18 places (we have applied the same refactoring in 4 different classes of project `knn`). We then sent these modified versions as patches to the owners of the corresponding repositories via the pull request feature of Github. On Table 6.4, columns "Replied?" and "Accepted?" flag the projects that have replied and accepted our patch with a ✓ symbol. At the time when this chapter was written, 9 projects had replied showing an intention to accept our patch. However, one of them (project `doms-transformers`) answered that the "*code is currently not actively maintained, but I'm leaving the pull request open in case we ever return to it*"[4]. This particular patch is not expected to be merged soon. For the remaining 8 projects that replied, 7 of them have already accepted and merged our patches.

In contrast, `netflixoss` was the only project that closed our patch with no response. This particular project seems to be a fork from another existing project (it has 231,361 lines of Java code performed by a single developer in a single commit), and does not seem to be maintained anymore. The owners of the remaining 7 projects did not provide any comments for our patches. At first, this might suggest that the patches were not useful. However, in the mining software repositories literature, it is well-known that an addressed issue may not be integrated into an official release for some time (COSTA et al., 2014). For instance, one contribution for the Linux Kernel usually takes 1-3 months to complete the code review, and between 1 and 3 months

[4]https://github.com/statsbiblioteket/doms-transformers/pull/1

for the change be integrated (JIANG; ADAMS; GERMAN, 2013). The intermittent nature of Github projects (KALLIAMVAKOU et al., 2014) only exacerbates this.

Results of EQ2 To answer EQ2, we measured the number of new statements that were added to and the number existing statements that were deleted from the benchmarks. A large number of modifications can produce code that is hard to understand and modify. So, a refactoring that results in a small number of modifications is desirable.

Overall, our approach has added 231 statements and removed 143 ones to the 15 benchmarks. Considering that one of them has 4 instances of Bottleneck 2, the mean number of modifications for each transformation was 12.8 additions and 7.9 deletions. Thus, our approach is not very intrusive. Most of the additions are due to the addition of a new constructor, which means that preexisting code, e.g., the `compute` method, is the target of only a few modifications. The refactoring of the parallel code added an average 5.3 new statements. Deletions have different explanations. For instance, most of the deletions on project `exhibitor` are due rewriting the parallel computation (10 out of the 15 deletions). Initially, this project used a more verbose approach, iterating through the data structure, creating and forking each new parallel task, and joining them at the end. We simplified this computation by just using the `invokeAll` method, as shown in the (simplified) code snippet below:

```
protected List<ServerStatus> compute() {
  for (List<ServerSpec> subList : Lists.partition(specs, size / 2)) {
    Task task = new Task(exhibitor, subList);
    task.fork();
    tasks.add(task);
  }
  for (Task task : tasks) {
    statuses.addAll(task.join());
  }
}
```

```
protected List<ServerStatus> compute() {
  // ...
  int split = (from + to)/2;

  invokeAll(
    new Task(exhibitor, specs, from, split),
    new Task(exhibitor, specs, from + split, to)
  );
  // ...
}
```

Project `mywiki`, on the other hand, presents an interesting approach. In this particular project, the copying process is done without using built-in library functions, for instance:

This approach used 9 out of the 18 deleted lines. This approach not only requires more lines of code to accomplish the same task, but it is also error-prone and omission-prone. Furthermore, the built-in library copy functions are written in low-level libraries in order to be optimized

```
protected Object[] splitArray(Object[] array, int start, int end) {
  int length = end - start;
  Object[] part = new Object[length];
  for ( int i = start; i < end; i++ ) {
    part[i-start] = array[i];
  }
  return part;
}
```

for performance. Another interesting observation from this code snippet is that the programmer used a similar solution when compared to our Bottleneck 2, *e.g.,* she creates a `length` variable whose value is obtained by subtracting the `end` and `start` variables. However, even though she employed a very similar solution, she missed an optimization opportunity by keeping the data copying over sub-tasks. This shows that, albeit straightforward to understand, it is not easy to identify this optimization opportunity. This finding is also supported by a comment from the owner of the `javaOneBR-2012` project: "*Thanks for the improvement, I didn't notice that*"[5].

6.6 Summary

This chapter describes a comprehensive study on parallelism bottlenecks in the Java FORKJOIN applications. Based on an in-depth analysis of more than 30 open-source FORKJOIN applications on GitHub, we classify the bottlenecks in 6 different kinds, and present comparisons on performance and energy before and after these bottlenecks are fixed. We have also developed a tool that can semiautomatically detect copy-related bottlenecks and refactor a program to remove bottlenecks.

[5]https://github.com/mariofts/javaOneBR-2012/pull/1

7

Related Work

If I have seen further it is by standing on the shoulders of giants.

—ISAAC NEWTON

In this chapter we present related work. Section 7.1 presents the works related to energy consumption, and Section 7.2 presents the works related to refactoring.

7.1 Software Energy Consumption

Studying energy efficiency at the application level is an emerging direction. In this section we describe the studies overlapping with the scope of our work.

Energy Management. The most established energy management approaches are focused on the hardware level (HOROWITZ; INDERMAUR; GONZALEZ, 1994; DAVID et al., 2010) and the OS level (GE et al., 2007; MERKEL; BELLOSA, 2006; YUAN; NAHRSTEDT, 2003). TIWARI; MALIK; WOLFE (1994) correlated energy consumption with CPU instructions. VIJAYKRISHNAN et al. (2001) and FARKAS et al. (2000) performed two early studies on the energy consumption of the JVM. In recent years, a number of studies have explored energy management strategies at the application level as an attempt to empower the application programmer to take energy-aware decisions, since some design choices might influence energy efficiency.

Within the programming language community, it is an active area of research to design energy-aware programming languages, with examples such as Eon (SORBER et al., 2007), Green (BAEK; CHILIMBI, 2010), EnerJ (SAMPSON et al., 2011), Energy Types (COHEN et al., 2012), and LAB (KANSAL et al., 2013). None of these software-centric energy management approaches focuses on multi-threaded programs. In these systems, recurring patterns of energy management tasks are incarnated as first-class citizens. Approximated programming (CARBIN et al., 2012; CARBIN; MISAILOVIC; RINARD, 2013) trades and reasons about occasional "soft errors", *i.e.*, errors that may reduce the accuracy of the results, for a reduction in energy consumption. The authors coined the *acceptability properties* term, which is based on relaxed programming constructs. In this language, variable assignments can be nondeterministic, that is,

an assignment `relax X = P` can assign the variable `X` to any set of values that satisfies the relaxation predicate `P`. The relationship between this line of work and our work is complementary: existing work provides language support to facilitate energy optimization, whereas our work experimentally and empirically evaluates some of those language constructs.

Energy Measurement. Energy measurement is a broad area of research. Prior work has attempted to model energy consumption at the individual instruction level (TIWARI et al., 1996), system call level (DONG; ZHONG, 2011), bytecode level (SEO; MALEK; MEDVIDOVIC, 2008c,a), and source code level (LIU; PINTO; LIU, 2015). Recent progress also includes fine-grained measurement for Android programs (COUTO et al., 2014; HAO et al., 2013; LI et al., 2013), with detailed energy measurement of different hardware components such as camera, Wi-Fi and GPS. They also observed that there is no strong correlation between performance and energy consumption. RAPL-based energy measurement has appeared in recent literature (SUBRAMANIAM; FENG, 2013; LIU; PINTO; LIU, 2015; KAMBADUR; KIM, 2014); its precision and reliability have been extensively studied (HäHNEL et al., 2012).

Empirical Studies. Existing research that dealt with the trade-off of comparing individual characteristics of an application and energy consumption has covered a wide spectrum of applications. These characteristics vary from data structures (DAYLIGHT et al., 2002; HUNT; SANDHU; CEZE, 2011; MANOTAS; POLLOCK; CLAUSE, 2014), VM services (CAO et al., 2012), cloud offloading (KWON; TILEVICH, 2013), code obfuscation (SAHIN et al., 2014), and design patterns (SAHIN et al., 2012; LI; HALFOND, 2014). To the best of our knowledge, our study is the first in specifying and implement refactorings for energy efficiency focusing on a multi-core environment. In a recent study, SAHIN; POLLOCK; CLAUSE (2014) analyzed how different refactorings impact in energy consumption. In this work, the authors analyzed the implications of applying 6 of the most commonly used refactorings (Convert Local Variable to Field, Extract Local Variable, Extract Method, Introduce Indirection, Inline Method, Introduce Parameter Object) under a subject of 9 applications, totalizing 197 refactored applications. Even though their work considers the energy efficiency of refactorings, the authors are not specifying and implementing refactorings to improve software energy consumption.

The mobile arena is also an important topic of research. HINDLE (2012) investigated the relationship between software changes (several versions of the Mozila Firefox app) and power consumption. The author observed that intentional performance optimization introduced a steady reduction in power consumption. More recently, HINDLE et al. (2014) proposed an energy consumption framework to be used in mobile devices. The authors suggest that this framework is more accurate than real meters in measuring energy consumption of smartphones because it does not take the battery usage in consideration. PATHAK; HU; ZHANG (2011) categorized energy bugs through analyzing the posts from 4 online forums. They produced a comprehensive taxonomy ranging from battery problems, SIM card problems, OS configuration problems, to no-sleep bugs. In comparison, our study of StackOverflow — a programmer-centric

website — allows us to zero in on energy-aware *software development*, instead of taking the more system-oriented view for mobile devices. The Q&A form of StackOverflow further offer rich contextual information of the energy-related keywords, providing insights such as what *solutions* software practitioners know about energy consumption problems. In a subsequent study, PATHAK; HU; ZHANG (2012) presented an in-depth investigation in order to understand which is the root cause for energy consumption problems in mobile applications. Like our study, they also observed that advertisement plays an important role, consuming up to 75% of energy consumption in free apps.

There is also some research that focus on handset *users*, not *software developers*. ZHANG; HINDLE; GERMÁN (2014) analyze the impact of user applications, such as browser and text editors, on energy efficiency. Likewise, WILKE et al. (2013) compared how the choice of mobile user applications can impact on energy consumption. HEIKKINEN et al. (2012) studied energy problems of mobile handsets through a questionnaire-based study. Recently, WILKE et al. (2013) analyzed the Apps from Google Play market place. Their study focuses on a domain complementary to ours: the impact of energy consumption problems on the post-programming stages of software lifecycle. Their work is particularly interesting in correlating energy consumption problems with the user ratings to the Apps, the pricing of Apps, and different natures of Apps.

Most of the existing work which focus on energy efficiency of concurrent programs concentrates on energy behaviors in the presence of synchronization. PARK et al. (2007b) developed several synchronization-aware runtime techniques to balance the trade-off between energy and performance. GAUTHAM et al. (2012) studied the relative energy efficiency of synchronization implementation techniques (such as spin locks and transactions). A recent short paper, LIU (2012a) called for energy management based on different synchronization patterns, a concrete instance of which based on futures has been formally defined (LIU, 2012b). TREFETHEN; THIYAGALINGAM (2013) surveyed energy-aware software, including multi-threaded programs with different workload settings. BARTENSTEIN; LIU (2013) designed a data-centric approach to improve energy efficiency for multi-threaded stream programs. RIBIC; LIU (2014) designed an algorithm to improve the energy efficiency of the work-stealing runtime of Intel Cilk Plus by managing the relative speed of threads. However, our work is unique in its focus on the impact of programming models for managing thread execution and program design choices on energy consumption.

There are many approaches for energy management of multi-threaded programs at the architecture- and OS-levels. Examples in the former category include investigating the impact of Dynamic Voltage and Frequency Scaling on multi-core architectures (IYER; MARCULESCU, 2002), meeting power budget based on hardware performance counters (ISCI et al., 2006), and leveraging hardware heterogeneity (KUMAR et al., 2003) and processor topology (SOLERNOU et al., 2013). Examples in the latter include studying the impact of energy consumption based on workloads (GE et al., 2007), thread schedules (YUAN; NAHRSTEDT, 2003; MERKEL;

BELLOSA, 2006), and thread migration (RANGAN; WEI; BROOKS, 2009). Our work and related work cited here are complementary. Together, they attempt to understand energy behaviors of multi-threaded programs through the perspectives of different levels of the compute stack.

7.2 Refactoring

In object-oriented systems, refactoring is the process of improving the structure of existing code through behaviour-preserving transformations, themselves called refactorings. OPDYKE (1992) coined the term refactoring in his thesis. He informally proposes refactorings as behavior-preserving program transformations that improve quality factors. After this initial study, refactoring has been greatly studied in several contexts (ALVES et al., 2006; MENS; TOURWE, 2004; BROWN; LOIDL; HAMMOND, 2012; FOWLER et al., 1999; PINTO; KAMEI, 2013a; NEGARA et al., 2013; COLE; BORBA, 2005), but only a few of them are related to the concurrency field. This is unfortunate for at least two reasons. First, refactoring can also be applied to concurrent programs in order to improve a number of quality attributes, such as thread-safety and performance (DIG; MARRERO; ERNST, 2009). Second, when applied to concurrent programs, some refactorings that work reliably for sequential code may introduce concurrency bugs (SCHäFER et al., 2010).

7.2.1 Refactoring for Concurrency

Most of the refactoring for concurrency studies rely on the use of high-level concurrent libraries (DIG; MARRERO; ERNST, 2009; ISHIZAKI; DAIJAVAD; NAKATANI, 2011; DIG et al., 2009; SCHäFER et al., 2010, 2011; OKUR et al., 2014; OKUR; ERDOGAN; DIG, 2014; LIN; RADOI; DIG, 2014), mainly because they are easier to use. One example of such library is the `java.util.concurrent`, briefly described in Chapter 2 of this document. In such studies, the authors focus on specifying and implementing refactorings from low level concurrent code to the high-level counterparts.

DIG; MARRERO; ERNST (2009) introduced CONCURRENCER, a tool that automates three refactorings: from `int` to `AtomicInteger`; from `HashMap` to `ConcurrentHashMap`, and from a recursive algorithm to a parallel version using `ForkJoin`. The authors show that the automated approach is effective in identifying and applying such transformations. In a similar work, DIG et al. (2009) proposed a new tool called RELOOPER, which refactors sequential loops to execute in parallel using the `ParallelArray` library. In recent work, OKUR et al. (2014) proposed a set of refactoring tools used to safely refactor from low level threading constructs to high-level counterparts in the C# platform. Similarly, LIN; RADOI; DIG (2014) proposed a refactoring tool to use `AsyncTask` in the Android platform.

A similar study was conducted by SCHäFER et al. (2010). The authors stressed that the correctness of traditional refactorings on concurrent code is not well-studied. The authors

showed examples of how basic refactorings can break concurrent programs, even if they work properly in sequential code. The authors then propose a systematic approach to ensuring the correctness of commonly used refactorings on concurrent code. SCHäFER et al. (2011) also proposed REENTRANCER, a tool that transforms programs to be reentrant, enabling safe parallel execution.

One way to tackle concurrency problems is by reducing mutability. Following this direction, KJOLSTAD et al. (2011) introduced IMMUTATOR, a tool that automates the analysis and transformations required to make a class immutable. Another way of improving the behavior of existing concurrent software is by fixing concurrent code that could lead to bugs. LIN; DIG (2013) presented an empirical study of CHECK-THEN-ACT idioms used in `java.util.concurrent` collections. Even though the individual operations of these collections are thread-safe, when operations are combined (*e.g.* first checks if the queue is empty and, if not, removes elements from it), it could lead to bugs when executed under multiple threads. The authors then implemented a tool to detect and correct such CHECK-THEN-ACT misused idioms. More broadly, RADOI et al. (2014) introduced MOLD, a tool that transforms sequential Java programs into Scala programs that can be executed either on a single computing node via parallel Scala collections, or in a distributed manner, using a MapReduce framework.

7.2.2 Refactoring for Energy Efficiency

Finally, even though refactoring towards improve energy efficiency in concurrent software is a promising research topic (FRASER et al., 2011a), to the best of our knowledge, no study was found in the literature. However, through an investigation on premiere software engineering venues, we identify and discuss 14 contributions that can be further instantiated in refactoring tools used to improve software energy efficiency — and the challenges behind this process (PINTO; SOARES-NETO; CASTOR, 2015a). Here we describe three of them:

1. **Opportunity: Cloud offloading.** KWON; TILEVICH (2013) have described a technique to offload CPU intensive computations from a mobile device to the cloud, then reducing battery usage. However, not all possible CPU intensive computations can be offloaded, since offloading is not free. It does pay a toll on energy consumption, mainly due GSM and Wi-Fi power consumption for transmitting data over the network. The trade-off of when using this technique relies on the execution time of the computation; if it is small, it does not pay the network costs. However, when careful applied, this technique can reduce the overall energy consumption of a mobile application in up to 50%. However, to take advantage of the benefits of the cloud, developers face a high entry barrier. They need expertise on many topics: communication protocols, data storage, databases, and cloud infrastructure. Moreover, the manual set up of the cloud environment is tedious, error-prone and omission-prone. A refactoring engine can greatly lower the entry barrier to allow beginner developers

to partition their mobile applications, so that the energy intensive functionality can be executed in the cloud.

Challenges. The refactoring engine has two main challenges. The first one is to determine whether the computation is worth refactoring, that is, if offloading will not turn out to be more energy inefficient than performing the computation locally. Refactoring engines can take advantage of runtime analysis and energy consumption estimation tools to help programmers to decide when to refactor. Second, if a programmer agrees with the refactoring, the refactoring engine needs to set up all the environment to receive the computation in the cloud. While starting a virtual machine with default settings can be seem as trivial, set up a particular configuration to work with a particular piece of refactored code would require more sophisticated analysis. However, recent efforts have showed that such challenges can be overcome (HILTON et al., 2014).

2. **Opportunity: Software testing.** As a means of ensuring the reliability of a software, software testing have become one fundamental activity during the software development process. Software engineers are often motivated to write several test cases to their programs. In nontrivial applications, however, executing test cases can be an extremely time-consuming due to the great number of them. Selecting the most important test cases to be executed, without losing testing coverage, is a topic of great interest that researchers have extensively worked on. This is a particular concern for embedded software, which has to routinely perform test cases on a live deployed system, which in turn has its life-time limited to battery power. LI et al. (2014) proposed a technique used to minimize the energy consumption of test suites. This technique selects test cases based not only on their coverage but also on their energy usage. Results revealed that the technique is effective at generating test suites that consume up to 95% less energy, while maintaining testing coverage requirements. Prior to deploy, a refactoring engine can be used to select only useful high-quality energy efficient test cases.

   Challenges. First, the refactoring engine should be seemly integrate with the proposed approach. Since the proposed approach needs to modify the existing test suite, this should be done automatically by the refactoring tool. Second, the energy savings of the proposed approach is based on the quality of the test cases. If the test suite doe not have enough quality, the refactoring can impact negatively on the testing coverage. Such analysis should be performed by the refactoring engine. Finally, since test suites written in an ad hoc manner are not supported by the proposed approach, the refactoring engine should also be capable to refactor such test cases to use a high-level framework, such as JUnit or TestNG.

3. **Opportunity: HTTP Requests.** LI et al. (2014) presented the first large scale

study on the energy efficiency of mobile applications. Among the findings, they describe two remarkable ones: (1) a few set of APIs used in applications dominate non-idle energy consumption and (2) an HTTP request is the most energy consuming operation of the network. Likewise, NOUREDDINE et al. (2012) also observed that the highest power consumption methods on the Jetty Web Server came from classes that manage HTTP requests. Still, in COHEN et al. (2012), the authors examined the top 5 energy hotspots of mobile applications and, for most of the target systems, HTTP usage consumed the most energy. We believe that refactoring engines can play a role here. A refactoring engine should be able to identify such energy hungry APIs, and replace them for an energy-friendly ones.

Challenges. Even though not every energy-intensive API has an energy-friendly counterpart, some of them do have. An example of such component is the power efficient work queue[1]. Refactoring tools should keep track of the cutting edge energy efficient implementations. For those of which do not have such energy-efficient implementation, there are probably other implementations available. Webservices, for example, can be implemented using at least two commonplace protocols: SOAP and REST, which greatly differ in their internal characteristics. While REST is more flexible and light-weight, SOAP is more detailed and heavy-weight. In the absence of an energy-efficient implementation, refactoring tools should support the transition to more light-weight components.

[1] `http://lwn.net/Articles/548281/`

8

Concluding Remarks

The future is not laid out on a track. It is something that we can decide.

—ALAN KAY

This Chapter reviews the problem stated, the solution proposed and the main contributions of this thesis to the state-of-art. This Chapter also presents other results that were obtained along the Ph.D., although not included in this document. We finish by present further extensions of the main contributions of this work.

8.1 The Problem

This thesis tackles two timely but overlooked energy related problems:

1. The *lack of knowledge* for writing energy efficient parallel applications;
2. The *lack of tools* used to detect, refactor and prevent energy bloats from occurring in an early stage of software development.

8.2 The main contributions

This thesis makes the following contributions:

1. A developer-oriented view of energy-aware software development;
2. An understanding of energy-behaviors of different thread-safe collections;
3. An understanding of energy-behaviors of different thread management approaches;
4. A Catalog of bottlenecks found in ForkJoin Java applications;
5. A refactoring approach;
6. A refactoring engine;

8.3 Other contributions

In addition to the results described in greater details in the other chapters of this thesis, many others were obtained during the journey of my Ph.D. Most of them were done in a collaboration with colleagues from the CIn-UFPE, and from the State University of New York. These results are briefly described next.

- In TORRES et al. (2011) we provided a small-scale analysis of the adoption of the `java.util.concurrent` library in a handset of well-known Java programs. We further improved this paper in large-scale study covering more than 2,000 mature and non-trivial Java projects, spaning a timeframe of more than 8 years. Results suggest that more than 75% of the latest versions of the projects either explicitly create threads or employ some concurrency control mechanism (PINTO et al., 2015).

- In SARAIVA et al. (2012) we performed a systematic mapping study on Aspect-Oriented Software Maintainabiliy (AOSM). Using the guidelines of Kitchenham and Charters', we searched in well-known digital libraries engines. A total of 138 primary studies were selected, which describe 67 aspect-oriented (AO) maintainability metrics. This catalogue provides an objective guide to researchers looking for maintainability metrics to be used as indicators in their quantitative and qualitative assessments.

- In ABREU et al. (2012) we investigated which methods, techniques and tools have been used to assist the development of educational software. Using the guidelines of Kitchenham and Charters', we searched in well-known digital libraries engines. A total of 65 primary studies were selected, which describe 11 mechanisms the assist the development of educational software and 11 teaching resources. The result of this work generates the underlying technical and pedagogical foundations for, in further research, the development of a methodology or the improvement of existing techniques, considering the intrinsic characteristics of educational software.

- In PINTO; KAMEI (2013b), adopting a mixed-method approach, we analyzed the contributions of more than 12,400 Brazilian OSS programmers in more than 15,000 projects during the period of a year. Our results show that exists an OSS trend in Brazil: most part of the contributors are active, performing around 30 contributions per year, and they contribute to OSS mostly due altruism PINTO; KAMEI (2014).

- In PINTO; KAMEI (2013a) we qualitatively and quantitatively analyzed investigated StackOverflow in order to understand what are the barriers to adoption and the desirable features in refactoring tools. We manually analyzed more than 1,400 messages — 324 questions and 1,115 answers to those questions — from more than 1,200 refactoring users. Results showed that the most desirable features are

refactoring recommendations and refactoring for dynamic languages. On the other hand, usability problems, such as unknown error messages, are the most common ones.

- In LIU; PINTO; LIU (2015) we studied the energy impact of alternative data management choices by programmers, such as data access patterns, data precision choices, and data organization. Second, we attempt to build a bridge between application-level energy management and hardware-level energy management, by elucidating how various application-level data management features respond to Dynamic Voltage and Frequency Scaling (DVFS). Finally, we apply our findings to real-world applications, demonstrating their potential for guiding application-level energy optimization.

8.4 Future Work

We divide future work in two parts: (1) future work directly related to this PhD thesis, and (2) future work that we envision for other energy consumption research.

8.4.1 A Look Ahead for this Thesis

In particular, we intend to complement this work with the following future work:

Chapter 3. As regarding the StackOverflow study, we plan to extend the findings of this study with a survey, in order to observe if our findings could be generalizable to other data sources. We also plan to analyze commit messages and bug reports as ways to investigate how frequently energy bugs are being introduced during the software evolution process.

Chapter 4. When considering the thred-safe collections study, we first plan on enlarging the scope of our study. Although we considered a significant number of subjects, adding additional collection, and their methods, would potentially allow us to refute or confirm some of our observations in addition to perform the removal experiments for all collections available. With insights of this study, we plan to introduce the concept of relaxed collection. One step towards this goal is to reduce their accuracy (CARBIN et al., 2012). Since Java8 introduced the concept of `Streams`, which use implicitly parallelism and is well-suitable for data-parallel programs, an approximate solution for a given function, for instance sum the values of all elements, over a huge collection can take a fraction of memory, time and, last but not least, energy consumption

Chapter 5. For the thread management constructs, we intent to replicate this study using `jRAPL` (LIU; PINTO; LIU, 2015). Since such framework provides a power-based perspective on different hardware levels, we can provide additional discussions of our results. Second, we plan to investigate the energy consumption of different concurrent programming models, such as the actor programming model (AGHA, 1986). In particular, this investigation would introduce an

energy-perspective brick on the wall of the mutable × immutable programming models. Finally, we also plan to derive a cookbook of energy efficient design choices for concurrent software.

Chapter 6. Finally, on the refactoring contribution, we plan to extend our tool to cover additional `ForkJoin` bloats, such as the "Data Locallity" one, discussed in Chapter 5. Second, we plan to investigate how our refactoring can be further extended to cover different instantiations of this same runtime bloat on other object-oriented programming languages, so that non-Java communities would also benefit from them.

8.4.2 A Look Ahead for Energy Consumption Research

Although far from complete, this section provides a discussion on how software energy consumption research will mitigate the lack of knowledge and the lack of tools problems in a near future.

8.4.2.1 Lack of Knowledge

In order to mitigate the lack of knowledge, we believe that energy-aware researchers will act in two main fronts: conducting empirical studies and synthesizing the main findings of these studies into cookbooks.

Empirical studies. Even though there are several efforts on this direction (KWON; TILEVICH, 2013; LI et al., 2014; LINARES-VáSQUEZ et al., 2014; LIU; PINTO; LIU, 2015; PINTO; CASTOR; LIU, 2014a; PINTO et al., 2015; TREFETHEN; THIYAGALINGAM, 2013), these studies do not cover the whole spectrum of programming languages design and implementations. If we start from the beginning, the question of when performance and energy consumption are not correlated still need better attention. Even though recent effort was invested in this direction (CORRAL et al., 2014), there is still no generalization or clear answer for that. If we move one step further, we can reach design patterns and refactoring techniques. We believe that a new set of design patterns will emerge, using energy consumption as a standpoint — what we call thin-patterns. Such patterns will be redesigned to use less resource as possible. Similarly, refactoring research will derive new refactorings focusing on energy efficiency. These refactorings will play an important role in the transitioning to the sustainable software engineering. Moreover, we believe that threading techniques will be focus of several studies. Some well-known Java threading constructs have been studied before (PINTO; CASTOR; LIU, 2014a), but other ones such as Software Transaction Memory and the Actor Programing Model still need more attention.

Cookbooks. Most of the software development methodologies used in practice pay little attention to the design of an energy efficient system. Also, most good programming and design cookbooks and catalogues, like the refactoring and design pattern catalogues, do not even mention energy consumption. Since programers often rely on these catalogues of good practices, we envision

that in the future, researchers will summarize results of empirical studies in these textbooks. These textbooks will then be useful for better educating application software developers about software energy consumption and also empowering the to create the next generation of energy consumption tools.

8.4.2.2 Lack of Tools

Before we start discussing specific tools in details, it is worth to mention that although there are existing software tools for energy measurement, such tools have well-known drawbacks. First, energy measurement tools may pay an additional overhead on energy consumption, mostly due to the sampling mechanism. Even though recent efforts have mitigate this problem (LIU; PINTO; LIU, 2015), software-based approaches are often regarded as less rigorous than hardware-based ones. Second, energy measurement tools do not provide the granularity level that developers are more interested in (PINTO; CASTOR; LIU, 2014b). For instance, there is no tool support to measure energy consumption per thread or per process. In the following years, we expect much more contributions on this important direction. The presence of solid, platform-independent, and easy to use software measurement tools is the key to success of this emerging research field.

Refactoring tools. It is well-known that refactoring is a common practice among software developers. However, recent research suggests that existing refactoring tools are underused – about 90% of refactorings that developers could do with modern refactoring tools are done manually instead). When taking energy consumption into account, developers currently do not have enough knowledge to refactor their source code in order to consume less energy, so they cannot perform "green refactorings" without the assistance of specialized tools. Refactoring tools can take advantage of cutting-edge research, and incorporate such knowledge in refactoring engines. However, even though researchers have been speculating on this subject during the last years (FRASER et al., 2011b), little effort has been placed on introducing novel refactoring tools for improving the energy efficiency of a software system. This lack of contributions is not related to a lack of opportunities. As aforementioned, a recent study has pointed out more than 14 opportunities for green refactoring tools (PINTO; SOARES-NETO; CASTOR, 2015b). In the following decade, we expect that refactoring for energy efficiency will become a solid research field and will drive the development of energy-efficient pathways. In order to support this belief, mainstream development environments should provide support for green refactoring. Refactoring engine engineers should then learn how language constructs and design implementations impact on energy consumption. To help on this direction, researchers should focus on energy consumption studies with empirical evidences. Several studies are currently being conducted on this subject, and we expect that even more will be available in the years that come. With empirical support and solid evidences, refactoring engine engineers can apply this knowledge on novel refactoring tools. In the following decade, we expect that refactoring for

energy efficiency will become a solid research field and will drive the development of energy-efficient pathways. In order to support this belief, mainstream development environments should provide support for green refactoring. Refactoring engine engineers should then learn how language constructs and design implementations impact on energy consumption. To help on this direction, researchers should focus on energy consumption studies with empirical evidences. Several studies are currently being conducted on this subject, and we expect that even more will be available in the years that come. With empirical support and solid evidences, refactoring engine engineers can apply this knowledge on novel refactoring tools.

Reengineering tools. Differently than Refactoring tools, which are more static-based, reengineering tools can also be dynamic-based. As a promising example, method reallocation and method offloading are two strategies that can become a standard practice to implement energy-aware methods. Based on measurement tools that are able to estimate the energy consumption per method, or per line of source code, we envision that in the future, such estimation tools could be integrated to IDEs in a way that the estimation can be done at coding time, and that eventually, at building time, such results could be passed to a module that can recommend to implement method reallocation or method offloading. For instance, to implement reallocation, the module may recommend what methods to keep as they are, and what methods to re-code in a different programming language to be called as an hybrid application or shared library. In terms of method offloading, we envision that the recommender module can determine whether to place a method in the application, or to offload it to a third party.

Debugging tools. Debugging tools are commonly used by software developers in order to catch errors in program formulation. If we think that errors in green software development are energy inefficient pieces of code, we can think about debuggers as tools smart enough to help programmers to identify such source code inefficiencies. We believe that in a near future, debugging tools will have the capability of inspecting the energy consumption used in a given line of code, as well as their common ability to identify which value was attributed to a given variable. Debugging tools can go further and highlight the energy intensive lines of code, which in turn will ease the perception of these inefficiencies by the programmers.

Testing tools. We believe that testing tools for improving energy efficiency will receive much more attention in the years that come. At least, new testing techniques will be evaluated using energy consumption as another important metric. At best, energy testing will become a research area. Several possible areas of research can be further visualized. One of them is what we call "green assertions", that is, the possibility to one define an energy budget, in Joules, for which the test case should assert whether the computation fits in the budget. The test should fail if the energy consumed is greater than the suggested budget. For instance, the following example defines that the `render()` method should consume, at most, 200 Joules. Otherwise, the test should fail.

```
double maxEnergy = 200;
```

```
assertTrue(render(), expected, maxEnergy);
```

More sophisticated approaches can be derived from this initial example. For instance, the programmer can define a energy budget delta. Any energy consumption value between the delta range should be accepted. Also, the test can be fast-fail, that is, the test can fail as soon as the testing tool figure out that the test case under execution exceeds the budget. Continuous integration (CI) tools should also take advantage of this knowledge, and grade test cases according to their "greeness". A straightforward way to grade test cases is to judge them in a binary fashion, that is, based on their success/fail status. However, a fine-grainded granularity can also be visualized. For instance, if the test case consumes only, say, 30% of the suggested budget, it can be seen as "A". On the other hand, if it consumes around 80% of the budget, it can be seen as "B". All test cases over the budget are clearly "C" test cases. With this family of labels, developers can easily visualize how energy-efficiency is the software system just by refactoring existing test cases.

Visualization tools. In the context of software evolution research, visualization has proven to be a key technique, due to the large amounts of information that need to be processed and understood. We strongly believe in the value of such research, and we also believe that visualization tools will play an important role in the software energy consumption research agenda. Visualization techniques are useful to support the understanding of software systems in order to discover and analyze their anomalies. In our context, energy consumption hotspots can be seen as our anomalies, which should be properly identified and fixed/removed. However, how to identify that a given method became more energy intensive from a given version to another? This question still lacks an answer that can hardly be given by the usually provided line/bar charts that depict the evolution of one particular aspect of a system over time. In order to mitigate this problem, ideas can be borrowed from stablished software engineering visualization techniques, such as the software as a city project (WETTEL; LANZA; ROBBES, 2011), in which software system as are visualized as interactive, navigable 3D cities. In a nutshell, the classes are represented as buildings in the city, while the packages are depicted as the districts in which the buildings reside.

Using this idea, software energy consumption researchers can provide an energy efficient view of a whole software system. With this paradox in mind, we can catch a glimpse of a visualization tool that can correlate energy consumption with his closest relative, performance. Since it is well-known the energy and performance do not always follow the same path (PINTO; CASTOR; LIU, 2014a), we can think about a class, or in this case a building, as the relation between energy consumption and performance. Energy consumption can represent the width, whereas performance can play as height. Thus a tall, though thin, building is a one where the execution time of the methods that pertain to that classes are high, but the energy consumption is not. Such information is valuable because, until now, the trade-off between energy consumption and execution is still an open topic of research.

However, since the original idea is based on source code analysis, and energy consumption should be measured during runtime only, this visualization tool should be based on an existing energy report generated by another tool. As a practical example, the testing tool aforementioned can provide such information. Since maximize code coverage is regarded as an important software development practice, there will be little effort placed on the software engineer side.

Static analysis tools. Finally, one of the main challenges of software energy consumption research is to bring the analysis to the static level. Currently, software energy consumption instrumentation can only be conducted at the runtime level, and often has several limitations, such as sophisticated (and expensive) hardware equipments (*e.g.*, PINTO; CASTOR; LIU (2014a)) or specific hardware configurations (*e.g.*, LIU; PINTO; LIU (2015)). This fact has the potential of limiting the usability of software energy consumption tools. In the following decade, we believe that new static-based solutions will emerge in this scenario. There are at least two important motivating factors that support this decision:

- First, it does not require one to run the program. Currently, it is extremely labor intensive to conduct a large-scale energy consumption study mainly because the software under analysis needs to run correctly, which means that the internal source code should be compiled and external dependencies should be fixed. A static analysis tool will greatly reduce this workload, since it is not required to run the system.
- Second, developers under time pressure have little chance to use another runtime tool to analyze energy consumption. With static analysis ones, tool vendors can integrate them into well-known IDEs, which programmers often use and rely on, thus decreasing the barrier of adopting such tools.

The main challenge for deriving static analysis tools for energy consumption, is the need of a body of knowledge of how language constructs and design decisions impact on energy consumption. Although several empirical studies have been conducted on this subject (*e.g.*, HAO et al. (2013); LIU; PINTO; LIU (2015); PINTO; CASTOR; LIU (2014a)), they are far from covering the whole spectrum of programming language designs and implementations. However, due to the emerging character of the field, we believe that new studies will be conducted in the following years, which in turn will help researchers to create static analysis tools.

References

ABREU, F. et al. Métodos, Técnicas e Ferramentas para o Desenvolvimento de Software Educacional: um mapeamento sistemático. In: BRAZILIAN SYMPOSIUM OF INFORMATICS IN THE EDUCATION. **Proceedings...** [S.l.: s.n.], 2012. (SBIE 2012).

A.CHANDRAKASAN; SHENG, S.; BRODERSEN, R. Low Power CMOS Digital Design. **IEEE Journal of Solid State Circuits**, [S.l.], v.27, p.473–484, 1992.

AGHA, G. **Actors**: a model of concurrent computation in distributed systems. Cambridge, MA, USA: MIT Press, 1986.

AGRAWAL, K. et al. Adaptive Work-stealing with Parallelism Feedback. **ACM Trans. Comput. Syst.**, [S.l.], v.26, p.7:1–7:32, 2008.

ALVES, V. et al. Refactoring product lines. In: GENERATIVE PROGRAMMING AND COMPONENT ENGINEERING, 5. **Proceedings...** [S.l.: s.n.], 2006. p.201–210.

ASAFU-ADJAYE, J. The relationship between energy consumption, energy prices and economic growth: time series evidence from asian developing countries. **Energy Economics**, [S.l.], v.22, n.6, p.615 – 625, 2000.

BAEK, W.; CHILIMBI, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 2010. **Proceedings...** [S.l.: s.n.], 2010. p.198–209. (PLDI '10).

BANERJEE, U. et al. Automatic program parallelization. **Proceedings of the IEEE**, [S.l.], v.81, n.2, p.211–243, Feb 1993.

BARTENSTEIN, T. W.; LIU, Y. D. Green Streams for Data-intensive Software. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.532–541. (ICSE '13).

BECK, K.; ANDRES, C. **Extreme Programming Explained**: embrace change (2nd edition). [S.l.]: Addison-Wesley Professional, 2004.

BELLOSA, F. The benefits of event: driven energy accounting in power-sensitive systems. In: ACM SIGOPS EUROPEAN WORKSHOP: BEYOND THE PC: NEW CHALLENGES FOR THE OPERATING SYSTEM, 9. **Proceedings...** [S.l.: s.n.], 2000. p.37–42. (EW 9).

BENINI, L.; BOGLIOLO, A.; DE MICHELI, G. A Survey of Design Techniques for System-level Dynamic Power Management. **IEEE Trans. Very Large Scale Integr. Syst.**, [S.l.], v.8, n.3, p.299–316, June 2000.

BIRCHER, W. L.; JOHN, L. K. Analysis of Dynamic Power Management on Multi-core Processors. In: ND ANNUAL INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 22. **Proceedings...** [S.l.: s.n.], 2008. p.327–338. (ICS '08).

BLACKBURN, S. M. et al. The DaCapo Benchmarks: java benchmarking development and analysis. In: ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, 21. **Proceedings...** [S.l.: s.n.], 2006. p.169–190. (OOPSLA '06).

BRANDOLESE, C. et al. The Impact of Source Code Transformations on Software Power and Energy Consumption. **Journal of Circuits, Systems and Computers**, [S.l.], v.11, n.05, 2002.

BROWN, C.; LOIDL, H.-W.; HAMMOND, K. ParaForming: forming parallel haskell programs using novel refactoring techniques. In: TRENDS IN FUNCTIONAL PROGRAMMING, 12. **Proceedings...** [S.l.: s.n.], 2012. p.82–97. (TFP'11).

BU, Y. et al. A Bloat-aware Design for Big Data Applications. In: INTERNATIONAL SYMPOSIUM ON MEMORY MANAGEMENT, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.119–130. (ISMM '13).

BUTENHOF, D. R. **Programming with POSIX Threads**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

CAO, T. et al. The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 39. **Proceedings...** [S.l.: s.n.], 2012. p.225–236. (ISCA '12).

CARBIN, M. et al. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 33. **Proceedings...** [S.l.: s.n.], 2012. p.169–180. (PLDI '12).

CARBIN, M.; MISAILOVIC, S.; RINARD, M. C. Verifying quantitative reliability for programs that execute on unreliable hardware. In: ACM SIGPLAN INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, OOPSLA 2013, PART OF SPLASH 2013, INDIANAPOLIS, IN, USA, OCTOBER 26-31, 2013, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.33–52.

CLOJURE. **A dynamic programming language that targets the Java Virtual Machine**. [Online; accessed 12-Aug-2015], `http://www.clojure.org/`.

COHEN, M. et al. Energy types. In: ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA 2012, PART OF SPLASH 2012, TUCSON, AZ, USA, OCTOBER 21-25, 2012, 27. **Proceedings...** [S.l.: s.n.], 2012. p.831–850.

COLE, L.; BORBA, P. Deriving refactorings for AspectJ. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, AOSD 2005, CHICAGO, ILLINOIS, USA, MARCH 14-18, 2005, 4. **Proceedings...** [S.l.: s.n.], 2005. p.123–134.

CORRAL, L. et al. Can Execution Time Describe Accurately the Energy Consumption of Mobile Apps? An Experiment in Android. In: INTERNATIONAL WORKSHOP ON GREEN AND SUSTAINABLE SOFTWARE, 3. **Proceedings...** [S.l.: s.n.], 2014. p.31–37. (GREENS 2014).

COSTA, D. A. da et al. An Empirical Study of Delays in the Integration of Addressed Issues. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, VICTORIA, BC, CANADA, SEPTEMBER 29 - OCTOBER 3, 2014, 30. **Anais...** [S.l.: s.n.], 2014. p.281–290.

COUTO, M. et al. Detecting Anomalous Energy Consumption in Android Applications. In: PROGRAMMING LANGUAGES - 18TH BRAZILIAN SYMPOSIUM, SBLP 2014, MACEIO, BRAZIL, OCTOBER 2-3, 2014. PROCEEDINGS. **Anais...** [S.l.: s.n.], 2014. p.77–91.

DAVID, H. et al. RAPL: memory power estimation and capping. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 16. **Proceedings...** [S.l.: s.n.], 2010. p.189–194. (ISLPED '10).

DAYLIGHT, E. G. et al. Incorporating Energy Efficient Data Structures into Modular Software Implementations for Internet-based Embedded Systems. In: INTERNATIONAL WORKSHOP ON SOFTWARE AND PERFORMANCE, 3. **Proceedings...** [S.l.: s.n.], 2002. p.134–141. (WOSP '02).

DE WAEL, M.; MARR, S.; VAN CUTSEM, T. Fork/Join Parallelism in the Wild: documenting patterns and anti-patterns in java programs using the fork/join framework. In: INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICES OF PROGRAMMING ON THE JAVA PLATFORM: VIRTUAL MACHINES, LANGUAGES, AND TOOLS, 2014. **Proceedings...** [S.l.: s.n.], 2014. p.39–50. (PPPJ '14).

DIG, D. et al. Relooper: refactoring for loop parallelism in java. In: ACM SIGPLAN CONFERENCE COMPANION ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, 24. **Proceedings...** [S.l.: s.n.], 2009. p.793–794. (OOPSLA '09).

DIG, D.; MARRERO, J.; ERNST, M. D. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 31. **Proceedings...** [S.l.: s.n.], 2009. p.397–407. (ICSE '09).

DING, X. et al. BWS: balanced work stealing for time-sharing multicores. In: EUROSYS '12. **Anais...** [S.l.: s.n.], 2012. p.365–378.

DONG, M.; ZHONG, L. Self-constructive High-rate System Energy Modeling for Battery-powered Mobile Systems. In: INTERNATIONAL CONFERENCE ON MOBILE SYSTEMS, APPLICATIONS, AND SERVICES, 9. **Proceedings...** [S.l.: s.n.], 2011. p.335–348. (MobiSys '11).

ESMAEILZADEH, H. et al. Dark Silicon and the End of Multicore Scaling. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 38. **Proceedings...** [S.l.: s.n.], 2011. p.365–376. (ISCA '11).

ESMAEILZADEH, H. et al. What is Happening to Power, Performance, and Software? **IEEE Micro**, [S.l.], v.32, n.3, 2012.

ESMAEILZADEH, H. et al. Power Challenges May End the Multicore Era. **Commun. ACM**, [S.l.], v.56, n.2, p.93–102, Feb. 2013.

FARKAS, K. I. et al. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. In: ACM SIGMETRICS INTERNATIONAL CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS, 2000. **Proceedings...** [S.l.: s.n.], 2000. p.252–263. (SIGMETRICS '00).

FEREDAY, J. Demonstrating rigor using thematic analysis: a hybrid approach of inductive and deductive coding and theme development. **International Journal of Qualitative**, [S.l.], v.5, 2006.

FOWLER, M. et al. **Refactoring**: improving the design of existing code. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

FRASER, S. et al. Going Green with Refactoring: sustaining the "worldwide virtual machine". In: ACM INTERNATIONAL CONFERENCE COMPANION ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS COMPANION. **Proceedings...** [S.l.: s.n.], 2011. p.171–174. (SPLASH '11).

FRASER, S. et al. Going Green with Refactoring: sustaining the "worldwide virtual machine". In: ACM INTERNATIONAL CONFERENCE COMPANION ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS COMPANION. **Proceedings...** [S.l.: s.n.], 2011. p.171–174. (OOPSLA '11).

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The Implementation of the Cilk-5 Multithreaded Language. In: ACM SIGPLAN 1998 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION. **Proceedings...** [S.l.: s.n.], 1998. p.212–223. (PLDI '98).

GAUTHAM, A. et al. The implications of shared data synchronization techniques on multi-core energy efficiency. In: OF THE. **Proceedings...** [S.l.: s.n.], 2012. (HotPower'12).

GE, R. et al. CPU MISER: a performance-directed, run-time system for power-aware clusters. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (ICPP 2007), SEPTEMBER 10-14, 2007, XI-AN, CHINA, 2007. **Anais...** [S.l.: s.n.], 2007. p.18.

GEORGES, A.; BUYTAERT, D.; EECKHOUT, L. Statistically Rigorous Java Performance Evaluation. In: ND ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS AND APPLICATIONS, 22. **Proceedings...** [S.l.: s.n.], 2007. p.57–76. (OOPSLA '07).

GOETZ, B. **Java theory and practice**: concurrent collections classes. Accessed: 2014-09-29, `http://www.ibm.com/developerworks/java/library/j-jtp07233/index.html`.

GROOVY. **The Groovy programming language**. [Online; accessed 12-Aug-2015], `http://www.groovy-lang.org/`.

GU, Y.; LEE, B. S.; CAI, W. Evaluation of Java Thread Performance on Two Different Multithreaded Kernels. **SIGOPS Oper. Syst. Rev.**, [S.l.], v.33, n.1, p.34–46, Jan. 1999.

GUO, Y. et al. SLAW: a scalable locality-aware adaptive work-stealing scheduler. In: IPDPS. **Anais...** [S.l.: s.n.], 2010. p.1–12.

HäHNEL, M. et al. Measuring Energy Consumption for Short Code Paths Using RAPL. **SIGMETRICS Perform. Eval. Rev.**, [S.l.], v.40, n.3, p.13–17, Jan. 2012.

HALLER, P.; ODERSKY, M. Scala Actors: unifying thread-based and event-based programming. **Theor. Comput. Sci.**, [S.l.], v.410, n.2-3, p.202–220, Feb. 2009.

HAO, S. et al. Estimating mobile application energy consumption using program analysis. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '13, SAN FRANCISCO, CA, USA, MAY 18-26, 2013, 35. **Anais...** [S.l.: s.n.], 2013. p.92–101.

HEIKKINEN, M. V. et al. Energy efficiency of mobile handsets: measuring user attitudes and behavior. **Telematics and Informatics**, [S.l.], v.29, n.4, 2012.

HILTON, M. et al. Refactoring Local to Cloud Data Types for Mobile Apps. In: INTERNATIONAL CONFERENCE ON MOBILE SOFTWARE ENGINEERING AND SYSTEMS, 1. **Proceedings...** [S.l.: s.n.], 2014. p.83–92. (MOBILESoft 2014).

HINDLE, A. Green mining: a methodology of relating software change to power consumption. In: MINING SOFTWARE REPOSITORIES (MSR), 2012 9TH IEEE WORKING CONFERENCE ON. **Anais...** [S.l.: s.n.], 2012. p.78–87.

HINDLE, A. et al. GreenMiner: a hardware based mining software repositories software energy consumption framework. In: WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, 11. **Proceedings...** [S.l.: s.n.], 2014. p.12–21. (MSR 2014).

HOROWITZ, M.; INDERMAUR, T.; GONZALEZ, R. Low-power digital design. In: LOW POWER ELECTRONICS, 1994. IEEE SYMPOSIUM. **Anais...** [S.l.: s.n.], 1994.

HUNT, N.; SANDHU, P. S.; CEZE, L. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures. In: WORKSHOP ON INTERACTION BETWEEN COMPILERS AND COMPUTER ARCHITECTURES, 2011. **Proceedings...** [S.l.: s.n.], 2011. p.63–70. (INTERACT '11).

ISCI, C. et al. An Analysis of Efficient Multi-Core Global Power Management Policies: maximizing performance for a given power budget. In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 39. **Proceedings...** [S.l.: s.n.], 2006. p.347–358. (MICRO 39).

ISHIZAKI, K.; DAIJAVAD, S.; NAKATANI, T. Refactoring Java programs using concurrent libraries. In: WORKSHOP ON PARALLEL AND DISTRIBUTED SYSTEMS: TESTING, ANALYSIS, AND DEBUGGING. **Proceedings...** [S.l.: s.n.], 2011. p.35–44. (PADTAD '11).

IYER, A.; MARCULESCU, D. Power Efficiency of Voltage Scaling in Multiple Clock, Multiple Voltage Cores. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 2002. **Proceedings...** [S.l.: s.n.], 2002. p.379–386. (ICCAD '02).

JIANG, Y.; ADAMS, B.; GERMAN, D. M. Will My Patch Make It? And How Fast?: case study on the linux kernel. In: WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, 10. **Proceedings...** [S.l.: s.n.], 2013. p.101–110. (MSR '13).

KALIBERA, T. et al. A black-box approach to understanding concurrency in DaCapo. In: ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, OOPSLA 2012, PART OF SPLASH 2012, TUCSON, AZ, USA, OCTOBER 21-25, 2012, 27. **Proceedings...** [S.l.: s.n.], 2012. p.335–354.

KALLIAMVAKOU, E. et al. The Promises and Perils of Mining GitHub. In: WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, 11. **Proceedings...** [S.l.: s.n.], 2014. p.92–101. (MSR 2014).

KAMBADUR, M.; KIM, M. A. An experimental survey of energy management across the stack. In: ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, OOPSLA 2014, PART OF SPLASH 2014, PORTLAND, OR, USA, OCTOBER 20-24, 2014, 2014. **Proceedings...** [S.l.: s.n.], 2014. p.329–344.

KANSAL, A. et al. The latency, accuracy, and battery (LAB) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. In: ACM SIGPLAN INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, OOPSLA 2013, PART OF SPLASH 2013, INDIANAPOLIS, IN, USA, OCTOBER 26-31, 2013, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.661–676.

KIRBAS, S. et al. The effect of evolutionary coupling on software defects: an industrial case study on a legacy system. In: ACM-IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, ESEM '14, TORINO, ITALY, SEPTEMBER 18-19, 2014, 2014. **Anais...** [S.l.: s.n.], 2014. p.6.

KJOLSTAD, F. et al. Transformation for class immutability. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE 2011, WAIKIKI, HONOLULU , HI, USA, MAY 21-28, 2011, 33. **Proceedings...** [S.l.: s.n.], 2011. p.61–70.

KUMAR, R. et al. Single-ISA Heterogeneous Multi-Core Architectures: the potential for processor power reduction. In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 36. **Proceedings...** [S.l.: s.n.], 2003. p.81–. (MICRO 36).

KUMAR, V. et al. Work-stealing Without the Baggage. In: ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS. **Proceedings...** [S.l.: s.n.], 2012. p.297–314. (OOPSLA '12).

KWON, Y.; TILEVICH, E. Reducing the Energy Consumption of Mobile Applications Behind the Scenes. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, EINDHOVEN, THE NETHERLANDS, SEPTEMBER 22-28, 2013, 2013. **Anais...** [S.l.: s.n.], 2013. p.170–179.

LEA, D. **Concurrent Programming in Java. Second Edition**: design principles and patterns. 2nd.ed. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1999.

LEA, D. A Java fork/join framework. In: JAVA GRANDE. **Anais...** [S.l.: s.n.], 2000. p.36–43.

LEBRETON, H.; VIVET, P. Power Modeling in SystemC at Transaction Level, Application to a DVFS Architecture. In: IEEE COMPUTER SOCIETY ANNUAL SYMPOSIUM ON VLSI, 2008. **Proceedings...** [S.l.: s.n.], 2008. p.463–466. (ISVLSI '08).

LEE, E. A. The Problem with Threads. **Computer**, [S.l.], v.39, n.5, p.33–42, May 2006.

LI, D. et al. Calculating Source Line Level Energy Information for Android Applications. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.78–89. (ISSTA 2013).

LI, D. et al. Integrated energy-directed test suite optimization. In: ISSTA. **Anais...** [S.l.: s.n.], 2014. p.339–350.

LI, D. et al. An Empirical Study of the Energy Consumption of Android Applications. In: ICSME. **Anais...** [S.l.: s.n.], 2014. p.121–130.

LI, D.; HALFOND, W. G. J. An Investigation into Energy-saving Programming Practices for Android Smartphone App Development. In: INTERNATIONAL WORKSHOP ON GREEN AND SUSTAINABLE SOFTWARE, 3. **Proceedings...** [S.l.: s.n.], 2014. p.46–53. (GREENS 2014).

LI, D.; TRAN, A. H.; HALFOND, W. G. J. Making Web Applications More Energy Efficient for OLED Smartphones. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 36. **Proceedings...** [S.l.: s.n.], 2014. p.527–538. (ICSE 2014).

LI, J.; MARTÍNEZ, J. F. Power-performance Considerations of Parallel Computing on Chip Multiprocessors. **ACM Trans. Archit. Code Optim.**, [S.l.], v.2, n.4, p.397–422, Dec. 2005.

LI, J.; MARTINEZ, J. F.; HUANG, M. C. The Thrifty Barrier: energy-aware synchronization in shared-memory multiprocessors. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE, 10. **Proceedings...** [S.l.: s.n.], 2004. p.14–. (HPCA '04).

LIN, Y.; DIG, D. Check-then-Act Misuse of Java Concurrent Collections. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION, 6. **Proceedings...** [S.l.: s.n.], 2013. (ICST).

LIN, Y.; RADOI, C.; DIG, D. Retrofitting Concurrency for Android Applications Through Refactoring. In: ND ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 22. **Proceedings...** [S.l.: s.n.], 2014. p.341–352. (FSE 2014).

LINARES-VáSQUEZ, M. et al. Mining Energy-greedy API Usage Patterns in Android Apps: an empirical study. In: WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, 11. **Proceedings...** [S.l.: s.n.], 2014. p.2–11. (MSR 2014).

LIU, K.; PINTO, G.; LIU, D. Data-Oriented Characterization of Application-Level Energy Optimization. In: INTERNATIONAL CONFERENCE ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING, 18. **Proceedings...** [S.l.: s.n.], 2015. (FASE'15).

LIU, Y. D. Energy-efficient synchronization through program patterns. In: FIRST INTERNATIONAL WORKSHOP ON GREEN AND SUSTAINABLE SOFTWARE, GREENS 2012, ZURICH, SWITZERLAND, JUNE 3, 2012. **Anais...** [S.l.: s.n.], 2012. p.35–40.

LIU, Y. D. Variant-Frequency Semantics for Green Futures. In: FIFTH WORKSHOP ON PROGRAMMING LANGUAGE APPROACHES TO CONCURRENCY- AND COMMUNICATION-CENTRIC SOFTWARE, PLACES 2012, TALLINN, ESTONIA, 31 MARCH 2012. **Proceedings...** [S.l.: s.n.], 2012. p.1–6.

MANOTAS, I.; POLLOCK, L.; CLAUSE, J. SEEDS: a software engineer's energy-optimization decision support framework. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 36. **Proceedings...** [S.l.: s.n.], 2014. p.503–514. (ICSE 2014).

MENARINI, M. et al. Green web services: improving energy efficiency in data centers via workload predictions. In: INTERNATIONAL WORKSHOP ON GREEN AND SUSTAINABLE SOFTWARE, GREENS 2013, SAN FRANCISCO, CA, USA, MAY 20, 2013, 2. **Anais...** [S.l.: s.n.], 2013. p.8–15.

MENS, T.; TOURWÉ, T. A Survey of Software Refactoring. **IEEE Trans. Softw. Eng.**, [S.l.], v.30, n.2, p.126–139, Feb. 2004.

MENS, T.; TOURWE, T. A survey of software refactoring. **Software Engineering, IEEE Transactions on**, [S.l.], v.30, n.2, p.126–139, 2004.

MERKEL, A.; BELLOSA, F. Balancing Power Consumption in Multiprocessor Systems. In: ACM SIGOPS/EUROSYS EUROPEAN CONFERENCE ON COMPUTER SYSTEMS 2006, 1. **Proceedings...** [S.l.: s.n.], 2006. p.403–414. (EuroSys '06).

MICHAEL, M. M.; VECHEV, M. T.; SARASWAT, V. A. Idempotent work stealing. In: PPOPP '09. **Anais...** [S.l.: s.n.], 2009. p.45–54.

MISAILOVIC, S.; SIDIROGLOU, S.; RINARD, M. C. Dancing with Uncertainty. In: RACES. **Anais...** [S.l.: s.n.], 2012. p.51–60.

MORRISON, P.; MURPHY-HILL, E. Is Programming Knowledge Related to Age? An Exploration of Stack Overflow. In: WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, 10. **Proceedings...** [S.l.: s.n.], 2013. p.69–72. (MSR '13).

MOURA, I. et al. **What Solutions Do Developers Use in Practice to Save Energy?** Unpublished manuscript.

MURPHY-HILL, E.; BLACK, A. Refactoring Tools: fitness for purpose. **Software, IEEE**, [S.l.], v.25, n.5, p.38–44, Sept 2008.

MURPHY-HILL, E.; JIRESAL, R.; MURPHY, G. C. Improving Software Developers' Fluency by Recommending Development Environment Commands. In: ACM SIGSOFT 20TH INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings...** [S.l.: s.n.], 2012. p.42:1–42:11. (FSE '12).

NAGAPPAN, M.; ZIMMERMANN, T.; BIRD, C. Diversity in Software Engineering Research. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND THE ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING (ESEC/FSE). **Proceedings...** [S.l.: s.n.], 2013.

NANZ, S.; WEST, S.; SILVEIRA, K. S. da. Examining the Expert Gap in Parallel Programming. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 19. **Proceedings...** [S.l.: s.n.], 2013. p.434–445. (Euro-Par'13).

NEGARA, S. et al. A Comparative Study of Manual and Automated Refactorings. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 27. **Proceedings...** [S.l.: s.n.], 2013. p.552–576. (ECOOP'13).

NOUREDDINE, A. et al. Runtime Monitoring of Software Energy Hotspots. In: ASE. **Anais...** [S.l.: s.n.], 2012. p.160–169.

OKUR, S.; DIG, D. How Do Developers Use Parallel Libraries? In: ACM SIGSOFT 20TH INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings...** [S.l.: s.n.], 2012. p.54:1–54:11. (FSE '12).

OKUR, S.; ERDOGAN, C.; DIG, D. Converting Parallel Code from Low-Level Abstractions to Higher-Level Abstractions. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 28. **Proceedings...** [S.l.: s.n.], 2014. (ECOOP'14).

OKUR, S. et al. A Study and Toolkit for Asynchronous Programming in C#. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 36. **Proceedings...** [S.l.: s.n.], 2014. p.1117–1127. (ICSE 2014).

OPDYKE, W. F. **Refactoring Object-oriented Frameworks**. 1992. Tese (Doutorado em Ciência da Computação) — , Champaign, IL, USA. UMI Order No. GAX93-05645.

ORACLE. **Java HotSpot Garbage Collection**. [Online; accessed 21-Mar-2014], `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-140228.html`.

OVERBEY, J. et al. Refactorings for Fortran and High-performance Computing. In: SECOND INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR HIGH PERFORMANCE COMPUTING SYSTEM APPLICATIONS. **Proceedings...** [S.l.: s.n.], 2005. p.37–39. (SE-HPCS '05).

PANKRATIUS, V.; SCHMIDT, F.; GARRETON, G. Combining functional and imperative programming for multicore software: an empirical study evaluating scala and java. In: SOFTWARE ENGINEERING (ICSE), 2012 34TH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2012. p.123–133.

PAPAMARCOS, M. S.; PATEL, J. H. A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 11. **Proceedings...** [S.l.: s.n.], 1984. p.348–354. (ISCA '84).

PARK, S. et al. Managing Energy-Performance Tradeoffs for Multithreaded Applications on Multiprocessor Architectures. In: INTERNATIONAL CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS. **Proceedings...** [S.l.: s.n.], 2007.

PARK, S. et al. Managing energy-performance tradeoffs for multithreaded applications on multiprocessor architectures. In: ACM SIGMETRICS INTERNATIONAL CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS, SIGMETRICS 2007, SAN DIEGO, CALIFORNIA, USA, JUNE 12-16, 2007, 2007. **Proceedings...** [S.l.: s.n.], 2007. p.169–180.

PATHAK, A.; HU, Y. C.; ZHANG, M. Bootstrapping Energy Debugging on Smartphones: a first look at energy bugs in mobile devices. In: ACM WORKSHOP ON HOT TOPICS IN NETWORKS, 10. **Proceedings...** [S.l.: s.n.], 2011. p.5:1–5:6. (HotNets-X).

PATHAK, A.; HU, Y. C.; ZHANG, M. Where is the Energy Spent Inside My App?: fine grained energy accounting on smartphones with eprof. In: ACM EUROPEAN CONFERENCE ON COMPUTER SYSTEMS, 7. **Proceedings...** [S.l.: s.n.], 2012. p.29–42. (EuroSys '12).

PEIERLS, T. et al. **Java Concurrency in Practice**. [S.l.]: Addison-Wesley Professional, 2005.

PENTIKOUSIS, K. In Search of Energy-efficient Mobile Networking. **Comm. Mag.**, [S.l.], v.48, n.1, 2010.

PERING, T.; BURD, T.; BRODERSEN, R. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, 1998. **Proceedings...** [S.l.: s.n.], 1998. p.76–81. (ISLPED '98).

PINTO, G.; CASTOR, F. On the Implications of Language Constructs for Concurrent Execution in the Energy Efficiency of Multicore Applications. In: COMPANION PUBLICATION FOR CONFERENCE ON SYSTEMS, PROGRAMMING, AND APPLICATIONS: SOFTWARE FOR HUMANITY, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.95–96. (SPLASH '13).

PINTO, G.; CASTOR, F. Characterizing the Energy Efficiency of Java's Thread-Safe Collections in a Multicore Environment. In: SPLASH'2014 WORKSHOP ON SOFTWARE ENGINEERING FOR PARALLEL SYSTEMS (SEPS). **Proceedings...** [S.l.: s.n.], 2014. (SEPS '14).

PINTO, G.; CASTOR, F.; LIU, Y. D. Understanding Energy Behaviors of Thread Management Constructs. In: ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, 2014. **Proceedings...** [S.l.: s.n.], 2014. p.345–360. (OOPSLA '14).

PINTO, G.; CASTOR, F.; LIU, Y. D. Mining Questions About Software Energy Consumption. In: WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, 11. **Proceedings...** [S.l.: s.n.], 2014. p.22–31. (MSR 2014).

PINTO, G. et al. A Comprehensive Study on the Energy Efficiency of Java Thread-Safe Collections. **The Journal of Systems and Software**, [S.l.], 2015.

PINTO, G. et al. **Detecting and Refactoring Energy Bloat on Data-Parallel Computations**. Unpublished manuscript.

PINTO, G. et al. A large-scale study on the usage of Java's concurrent programming constructs. **Journal of Systems and Software**, [S.l.], v.106, n.0, p.59 – 81, 2015.

PINTO, G.; KAMEI, F. What Programmers Say About Refactoring Tools?: an empirical investigation of stack overflow. In: ACM WORKSHOP ON WORKSHOP ON REFACTORING TOOLS, 2013. **Proceedings...** [S.l.: s.n.], 2013. p.33–36. (WRT '13).

PINTO, G.; KAMEI, F. Analisando as Contribuições de Desenvolvedores Brasileiros em Projetos Distribuídos de Software Open-Source: um estudo inicial. In: WORKSHOP ON DISTRIBUTED SOFTWARE DEVELOPMENT, 7. **Proceedings...** [S.l.: s.n.], 2013. (WDDS 2013).

PINTO, G.; KAMEI, F. The Census of the Brazilian Open-Source Community. In: INTERNATIONAL CONFERENCE ON OPEN SOURCE SYSTEMS, 10. **Proceedings...** [S.l.: s.n.], 2014. (OSS 2014).

PINTO, G.; SOARES-NETO, F.; CASTOR, F. **Refactoring for Energy Efficiency**: a reflection on the state of the art. 2015. (GREENS 2015).

PINTO, G.; SOARES-NETO, F.; CASTOR, F. Refactoring for Energy Efficiency: a reflection on the state of the art. In: INTERNATIONAL WORKSHOP ON GREEN AND SUSTAINABLE SOFTWARE, 4. **Proceedings...** [S.l.: s.n.], 2015. (GREENS 2015).

RADOI, C. et al. Translating Imperative Code to MapReduce. In: ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES & APPLICATIONS, 2014. **Proceedings...** [S.l.: s.n.], 2014. p.909–927. (OOPSLA '14).

RANGAN, K. K.; WEI, G.-Y.; BROOKS, D. Thread Motion: fine-grained power management for multi-core systems. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 36. **Proceedings...** [S.l.: s.n.], 2009. p.302–313. (ISCA '09).

RIBIC, H.; LIU, Y. D. Energy-efficient work-stealing language runtimes. In: ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, ASPLOS '14, SALT LAKE CITY, UT, USA, MARCH 1-5, 2014. **Anais...** [S.l.: s.n.], 2014. p.513–528.

SAHIN, C. et al. Initial explorations on design pattern energy usage. In: FIRST INTERNATIONAL WORKSHOP ON GREEN AND SUSTAINABLE SOFTWARE, GREENS 2012, ZURICH, SWITZERLAND, JUNE 3, 2012. **Anais...** [S.l.: s.n.], 2012. p.55–61.

SAHIN, C. et al. How Does Code Obfuscation Impact Energy Usage? In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, VICTORIA, BC, CANADA, SEPTEMBER 29 - OCTOBER 3, 2014, 30. **Anais...** [S.l.: s.n.], 2014. p.131–140.

SAHIN, C.; POLLOCK, L. L.; CLAUSE, J. How do code refactorings affect energy usage? In: ACM-IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, ESEM '14, TORINO, ITALY, SEPTEMBER 18-19, 2014, 2014. **Anais...** [S.l.: s.n.], 2014. p.36.

SAMPSON, A. et al. EnerJ: approximate data types for safe and general low-power computation. In: ND ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 32. **Proceedings...** [S.l.: s.n.], 2011. p.164–174. (PLDI '11).

SARAIVA, J. et al. Aspect-Oriented Software Maintenance Metrics: a systematic mapping study. In: INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING, 16. **Proceedings...** [S.l.: s.n.], 2012. (EASE 2012).

SCHAEFER, M.; MOOR, O. de. Specifying and Implementing Refactorings. In: ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS. **Proceedings...** [S.l.: s.n.], 2010. p.286–301. (OOPSLA '10).

SCHäFER, M. et al. Correct refactoring of concurrent java code. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 24. **Proceedings...** [S.l.: s.n.], 2010. p.225–249. (ECOOP'10).

SCHäFER, M. et al. Refactoring Java programs for flexible locking. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33. **Proceedings...** [S.l.: s.n.], 2011. p.71–80. (ICSE '11).

SCHALLER, R. R. Moore's law: past, present, and future. **IEEE Spectr.**, [S.l.], v.34, n.6, p.52–59, June 1997.

SEO, C.; MALEK, S.; MEDVIDOVIC, N. Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems. In: INTERNATIONAL SYMPOSIUM ON COMPONENT-BASED SOFTWARE ENGINEERING, 11. **Proceedings...** [S.l.: s.n.], 2008. p.97–113. (CBSE '08).

SEO, C.; MALEK, S.; MEDVIDOVIC, N. Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems. In: CHAUDRON, M.; SZYPERSKI, C.; REUSSNER, R. (Ed.). **Component-Based Software Engineering**. [S.l.: s.n.], 2008. p.97–113. (Lecture Notes in Computer Science, v.5282).

SEO, C.; MALEK, S.; MEDVIDOVIC, N. Estimating the Energy Consumption in Pervasive Java-Based Systems. In: SIXTH ANNUAL IEEE INTERNATIONAL CONFERENCE ON PERVASIVE COMPUTING AND COMMUNICATIONS (PERCOM 2008), 17-21 MARCH 2008, HONG KONG. **Anais...** [S.l.: s.n.], 2008. p.243–247.

SINGER, L.; FILHO, F. M. F.; STOREY, M. D. Software engineering at the speed of light: how developers stay current using twitter. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE '14, HYDERABAD, INDIA - MAY 31 - JUNE 07, 2014, 36. **Anais...** [S.l.: s.n.], 2014. p.211–221.

SOLERNOU, A. et al. The Effect of Topology-Aware Process and Thread Placement on Performance and Energy. In: **Supercomputing**. [S.l.: s.n.], 2013. p.357–371. (Lecture Notes in Computer Science, v.7905).

SORBER, J. et al. Eon: a language and runtime system for perpetual systems. In: INTERNATIONAL CONFERENCE ON EMBEDDED NETWORKED SENSOR SYSTEMS, 5. **Proceedings...** [S.l.: s.n.], 2007. p.161–174. (SenSys '07).

SUBRAMANIAM, B.; FENG, W.-c. Towards Energy-proportional Computing for Enterprise-class Server Workloads. In: ACM/SPEC INTERNATIONAL CONFERENCE ON PERFORMANCE ENGINEERING, 4. **Proceedings...** [S.l.: s.n.], 2013. p.15–26. (ICPE '13).

SUTTER, H. The free lunch is over: a fundamental turn toward concurrency in software. **Dr. Dobb's Journal**, [S.l.], v.30, n.3, August 2005.

TANENBAUM, A. S. **Modern Operating Systems**. 3rd.ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

TIWARI, V. et al. Instruction Level Power Analysis and Optimization of Software. **Journal of VLSI Signal Processing**, [S.l.], v.13, p.1–18, 1996.

TIWARI, V.; MALIK, S.; WOLFE, A. Power Analysis of Embedded Software: a first step towards software power minimization. **IEEE Transactions on VLSI Systems**, [S.l.], v.2, p.437–445, 1994.

TORRES, W. et al. Are Java programmers transitioning to multicore?: a large scale study of java floss. In: WORKSHOP ON TRANSITIONING TO MULTICORE. **Proceedings...** [S.l.: s.n.], 2011. p.123–128. (SPLASH '11 Workshops).

TREFETHEN, A.; THIYAGALINGAM, J. Energy-aware software: challenges, opportunities and strategies. **Journal of Computational Science**, [S.l.], v.1, n.0, p.–, 2013.

TSANTALIS, N.; CHATZIGEORGIOU, A. Identification of Extract Method Refactoring Opportunities. In: SOFTWARE MAINTENANCE AND REENGINEERING, 2009. CSMR '09. 13TH EUROPEAN CONFERENCE ON. **Anais...** [S.l.: s.n.], 2009. p.119–128.

TSENG, S.-Y.; CHANG, M.-W. DVFS Aware Techniques on Parallel Architecture Core (PAC) Platform. In: INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE AND SYSTEMS SYMPOSIA, 2008. **Proceedings...** [S.l.: s.n.], 2008. p.79–84. (ICESSSYMPOSIA '08).

TUTORIALS, T. J. **Parallelism (The Java™ Tutorials > Collections > Aggregate Operations)**. Accessed: 2015-07-02, `https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html`.

VIJAYKRISHNAN, N. et al. Energy behavior of Java applications from the memory perspective. In: USENIX JAVA VIRTUAL MACHINE RESEARCH AND TECHNOLOGY SYMPOSIUM (JVM'01. **Anais...** [S.l.: s.n.], 2001. p.207–220.

WANG, L. et al. An adaptive task creation strategy for work-stealing scheduling. In: CGO '10. **Anais...** [S.l.: s.n.], 2010. p.266–277.

WANG, S.; LO, D.; JIANG, L. An Empirical Study on Developer Interactions in StackOverflow. In: ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, 28. **Proceedings...** [S.l.: s.n.], 2013. p.1019–1024. (SAC '13).

WETTEL, R.; LANZA, M.; ROBBES, R. Software Systems As Cities: a controlled experiment. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33., New York, NY, USA. **Proceedings...** ACM, 2011. p.551–560. (ICSE '11).

WILKE, C. et al. Comparing Mobile Applications' Energy Consumption. In: ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, 28. **Proceedings...** [S.l.: s.n.], 2013. p.1177–1179. (SAC '13).

WILKE, C. et al. Energy Consumption and Efficiency in Mobile Applications: a user feedback study. In: GREEN COMPUTING AND COMMUNICATIONS (GREENCOM), 2013 IEEE AND INTERNET OF THINGS (ITHINGS/CPSCOM), IEEE INTERNATIONAL CONFERENCE ON AND IEEE CYBER, PHYSICAL AND SOCIAL COMPUTING. **Anais...** [S.l.: s.n.], 2013. p.134–141.

WLOKA, J.; SRIDHARAN, M.; TIP, F. Refactoring for reentrancy. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND THE ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, 7. **Proceedings...** [S.l.: s.n.], 2009. p.173–182. (ESEC/FSE '09).

XU, G. CoCo: sound and adaptive replacement of java collections. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 27. **Proceedings...** [S.l.: s.n.], 2013. p.1–26. (ECOOP'13).

XU, G. et al. Go with the Flow: profiling copies to find runtime bloat. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 2009. **Proceedings...** [S.l.: s.n.], 2009. p.419–430. (PLDI '09).

YUAN, W.; NAHRSTEDT, K. Energy-efficient Soft Real-time CPU Scheduling for Mobile Multimedia Systems. In: NINETEENTH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES. **Proceedings...** [S.l.: s.n.], 2003. p.149–163. (SOSP '03).

ZHANG, C.; HINDLE, A.; GERMÁN, D. M. The Impact of User Choice on Energy Consumption. **IEEE Software**, [S.l.], v.31, n.3, p.69–75, 2014.

ZHANG, Y. et al. Refactoring Android Java Code for On-demand Computation Offloading. In: ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS. **Proceedings...** [S.l.: s.n.], 2012. p.233–248. (OOPSLA '12).

ZHUANG, Z.; KIM, K.-H.; SINGH, J. P. Improving Energy Efficiency of Location Sensing on Smartphones. In: INTERNATIONAL CONFERENCE ON MOBILE SYSTEMS, APPLICATIONS, AND SERVICES, 8. **Proceedings...** [S.l.: s.n.], 2010. p.315–330. (MobiSys '10).