January 2019

Improving Energy Consumption Of Java Programs

Mohit Kumar
Wayne State University

IMPROVING ENERGY CONSUMPTION OF JAVA PROGRAMS

by

MOHIT KUMAR

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2019

MAJOR: COMPUTER SCIENCE

Approved by:

_____

Advisor Date

_____

_____

_____

_____

DEDICATION

This dissertation wouldn't have been possible without the love, support, and care from my family. My mother (Usha Rani), who always nurture me with her positive thoughts. My father (Jagbir Singh), who always make sure that I had anything I needed. My sister (Sakshi), who always makes me laugh by reminiscing about happy memories. My wife (Namita), without whom I couldn't have thought of starting it.

ACKNOWLEDGMENTS

My Ph.D. journey started in 2014 at Wayne State University. However, this journey became worthwhile when I joined MIST Lab in 2015. I would like to express my heartfelt thanks to Dr. Weisong Shi for accepting me as his Ph.D. student. His support and guidance helped me to work on various research projects. I also want to thanks my MIST lab mates who never hesitated to help me. In their presence, I never felt alone in the lab and able to connect with Chinese culture and food.

I also want to thanks my committee members - Dr. Daniel Grosu, Dr. Nathan Fisher, and Dr. Devesh Tiwari. They provided various valuable suggestions that helped to enhance this dissertation.

Moreover, I want to thanks the National Science Foundation (NSF) for supporting this work partly by grant CNS-1561216.

Last but not least, I want to thanks my wife for always supporting, believing, and encouraging me. I am deeply grateful to her as she was always there when I need her.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODES

CHAPTER 1 INTRODUCTION

Energy efficiency of Information and Communications Technology (ICT) systems has doubled roughly every nineteen months since the invention of the first computer in 1946 [2]. However, the growth in ICT systems over the years has outpaced this improvement in energy efficiency due to the affordable price and availability. According to the International Telecommunication Union (ITU), worlds two-third population lives in an area covered by the mobile broadband network and 95% of the population has access to a mobile network which has resulted in more number of mobile-cellular subscriptions than the human population. Around 47% of the world household has a computer today and 52% of the human population is using the internet [3], which requires either a PC or hand-held device access. All of these ICT systems are driven by energy resulting in a severe load on the electric grid and contributing to greenhouse gas emissions.

Most of the energy is generated using fossil fuels which release various greenhouse gases in the atmosphere. As fossil fuels will vanish one day, it is very important to integrate sustainability into daily life. Sustainability is defined as the use of products in such a way that they can meet the needs of the present without impacting the ability of future generation to satisfy their own needs. Today, ICT amounts for 10% of the world energy which will keep on growing in future [4] and 3% of the overall carbon footprint which is now more than the level of CO_2 emission as that of aviation industry [5]. Each personal computer generates a ton of carbon dioxide every year [6]. Watching an hour of video weekly on a single mobile or tablet consumes annually more electricity than two new refrigerators consume in a year [4]. Increase in carbon footprint is causing several environmental changes over the world like unusual droughts, flood, storms, and higher average temperature.

ICT systems are driven by hardware and software. Most of the green IT initiatives concentrate on the hardware side. Earlier works on energy efficiency proposed different hardware technologies like low-power circuits, chip multiprocessors, fine grain clock gating, power gating, and dynamic voltage/frequency scaling [7, 8, 9, 10, 11]. The amount of electricity consumed by different hardware components has been significantly reduced over the years. However, when we look at the

Table 1.1: RAPL domains

Domain	Component
Package	CPU package
PP0	All cores and caches
PP1	GPU
DRAM	DRAM

software side, a lot has to be done to improve its energy efficiency. There has been a lot of research in estimating software energy consumption, however software developers still lack the knowledge necessary to improve the energy efficiency of software. Software energy savings are considered to be greater than the energy savings in hardware, but they are harder to achieve [12] as there is no technology that can ascertain the energy consumption of all the components in any of the ICT systems. Without accurate energy consumption measurements, it is not possible to optimize the energy efficiency at the software level.

1.1 Problem Statements & Plans

In this section, we discuss different problems that we have worked on to make software energy-efficient.

1.1.1 Calculating idle energy

Idle energy is defined as the amount of energy consumed by a system when it is not performing any task. As defined in [13], it is the sum of static and dynamic energy, as systems have a different number of background processes running all the time. In this work, we try to reduce the dynamic energy to stabilize and calculate the idle energy. The *active* energy is defined as the amount of energy consumed by a system while performing a specific task such as web browsing, printing, emailing, listening to music or playing a game.

Intel introduced the Running Average Power Limit (RAPL) feature starting with their Sandy Bridge processors, for measuring the energy consumption of onboard hardware components. It provides energy consumption information of different hardware components as listed in Table 1.1. It uses a software power model which estimates the energy consumption by leveraging hardware

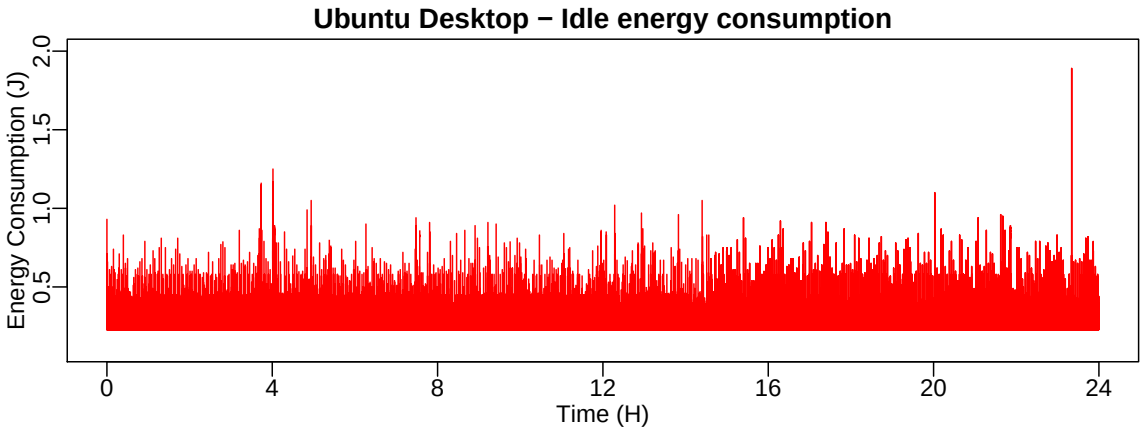


Figure 1.1: Idle energy consumption of Ubuntu Desktop.

performance counters. A user can configure and read RAPL information through Mode Specific Registers in privileged kernel mode. We use the Linux `perf` tool, which leverages RAPL technology, to measure energy consumption. However, it doesn't stabilize the idle energy consumption before performing energy measurements. Not stabilizing the idle energy consumption results in outliers as shown in Fig. 1.1, which causes inaccurate measurements. `systemd` is a critical suite of software for the Linux operating system that manages and operates various units like service, target, path, mount, etc [14]. Some of these units can trigger other units and work together to add functionality. In this work, we utilize the service unit to stabilize and calculate the idle energy consumption of the whole system as it is the most commonly utilized unit by system administrators. We use `systemctl` command to stop a service [15] and to stabilize the idle energy. We then calculate the active energy by subtracting the idle energy form the total energy consumed by the system.

1.1.2 Analyzing Java programs energy

Software developers are not aware of how to improve energy consumption. Therefore, we conduct a comprehensive study of the Java programs to provide software developers with energy saving suggestions. Java is one of the most commonly-used languages in ICT systems. We evaluate the energy consumption of data types, operators, control statements, String, exceptions, objects, and Arrays in Java using Intel RAPL technology. For data types, we evaluate primitive data types,

access modifiers, local and static variables, and scientific notations. For operators, we evaluate arithmetic, assignment, compound, pre- and post-increment, pre- and post-decrement, and short-circuit operators. For control statements, we evaluate if-then-else, conditional operator and loops. For exceptions, we evaluate try-catch block calls with and without exception. For String, we evaluate various ways of concatenating, converting and comparing String. For objects, we compare wrapper classes. For threads, we investigate different types of thread implementations. For Arrays, we compare different ways to copy and traverse Arrays. The key findings of this work are:

- For smaller iteration and variable sizes, different Java code statements consume the same energy as compare to their counterparts. We analyze different iteration and variable sizes and find out when the energy consumption differs.

- `int` is the most energy-efficient primitive data type. Static variables consume significantly high energy than local variables. Representing `float` and `double` number in scientific notation result in lesser energy consumption.

- Modulus operation is the most costly arithmetic operator. Conditional operator results in lower energy consumption than if-then-else statement. Loops initialization and termination expression can significantly impact a loop energy consumption.

- Try-catch block scope can impact its energy consumption. `StringBuilder append` method is the most energy-efficient way to concatenate `String` in Java. `String equals` method consumes lesser energy than `String compareTo` method. `System.arraycopy()` results in the least energy consumption for copying an array. Array column traversal results in higher energy consumption.

- Different JDKs results in almost the same energy consumption of different Java code statements. Execution time shows a high correlation to energy consumption.

1.1.3 Analyzing Java command-line options energy

Java has different command-line options that can be used to tune the JVM. These options can significantly affect the energy behavior of Java applications. However, there is no study characterizing the energy behavior of these command-line options. Therefore, we conduct a comprehensive study to evaluate the energy efficiency of Java command-line options. We use Intel RAPL technology to log the energy consumption values. We evaluate the active energy consumption of SPECjvm2008 benchmarks using different JDKs (Open and Oracle) and Java command-line options. The Java command-line options include `client`, `server`, `Xbatch`, `Xcomp`, `Xfuture`, `Xint`, `Xmixed`, `Xrs`, `AggressiveOpts`, `AggressiveHeap`, `Inline`, `AlwaysPreTouch`, `Xnoclassgc`, `UseSerialGC`, `UseParallelGC`, `UseConcMarkSweepGC`, and `UseG1GC`. The following are the key findings of our work:

- For most of the command-line options, Oracle JDK is more energy-efficient than Open JDK. Open JDK consumes up to 9% more energy than Oracle JDK.
- `Xint` command-line option results in the lowest energy efficiency of most benchmarks with up to 125% increase in energy consumption as compared to the default `server` command-line option.
- `UseG1GC` command-line option results in the highest energy efficiency of most benchmarks with up to 14% decrease in energy consumption as compared to the default `server` command-line option.
- Energy and time show a high correlation with a maximum value of 0.98 and a minimum value of 0.94.

1.1.4 Eclipse plugin for Java energy-saving suggestions

The only tool available for Java energy-saving suggestions [16] is limited to Java collections. Therefore, we present an Eclipse IDE plugin named Java Energy Profiler and Optimizer (JEPO)

which can provide suggestions for various Java code statements. JEPO can not only help in giving suggestions to software developers to change the source code for better energy efficiency but also can automatically measure the energy of Java source code at method granularity. JEPO leverages the Linux *rdmsr* tool to measure energy consumption. JEPO supports the energy consumption measurements at method granularity only while running Eclipse on a Linux operating system. For other operating systems, JEPO support only the energy-saving suggestions for source code.

JEPO has one toolbar button and two pop-up menu buttons. The software developer can see the real-time energy-saving suggestions in JEPO view by clicking on the toolbar button. JEPO pop-up menu buttons provide functionality for measuring energy consumption at method granularity and to provide suggestions for the whole project. The energy measurement at method granularity is achieved by injecting code at the start and end of the method using the Javassist library [17]. We make changes to various machine learning classifiers in Waikato Environment for Knowledge Analysis (WEKA) machine learning software and run the classifiers on Massive Online Data to find out how effective is our changes. We are able to achieve up to 14.46% improvement in package energy consumption, up to 14.19% improvement in core energy consumption, and up to 12.93% improvement in execution time. The changes result in only 0.48% drop in accuracy of the classifiers.

1.1.5 Summary of contributions

The contributions of our work are:

1. We stabilize the idle energy to measure the active energy accurately. We are able to achieve a standard variation of 0.0006 in the idle energy consumption of ICT systems.

2. We analyze Java programming language for data types, operators, control statements, exceptions, objects, collection classes, and map classes. We compare different Java JDKs for Java code statements.

3. We evaluate Java command-line options `client`, `server`, `Xbatch`, `Xcomp`, `Xfuture`, `Xint`, `Xmixed`, `Xrs`, `AggressiveOpts`, `AggressiveHeap`,

`Inline`, `AlwaysPreTouch`, `Xnoclassgc`, `UseSerialGC`, `UseParallelGC`, `UseConcMarkSweepGC`, and `UseG1GC`. We compare these Java command-line options for both Open and Oracle JDKs. We also calculate the correlation between active energy consumption and execution time.

4. We present an Eclipse plugin that a software developer can use for source code energy-efficiency optimization suggestions and energy measurements.

1.2 Outline

The rest of the document is organized as follows:

Chapter 2 discusses various terms, units to understand energy consumption in ICT systems. It also describes different type of ICT systems, different ways to estimate energy demand in ICT systems, hardware performance counters, energy efficiency benchmarks, software role in energy consumption and different ways to measure power in ICT systems.

Chapter 3 stabilizes the idle energy of two ICT systems. We first describe the tool used to measure the energy consumption and then stabilize the ICT systems by masking all the enabled and disabled services.

Chapter 4 investigates the energy efficiency of the Java programming language by evaluating data types, operators, control statements, String, exceptions, objects, and Arrays. We compare different Java components for JDK 7,8,9,10,11, and 12 in terms of energy consumption and execution time.

Chapter 5 analyzes various Java command-line options using Open and Oracle JDK on two different ICT systems. We compare all Java command-line options based on energy consumption and execution time.

Chapter 6 presents JEPO, an Eclipse IDE plugin to provide suggestions and energy measurements at method granularity to software developers. We evaluate JEPO using the WEKA library and MOA dataset.

Chapter 7 provides the conclusion of this dissertation. Chapter 8 ends it with a discussion on the future work in software energy efficiency.

CHAPTER 2 BACKGROUND

As energy efficiency can play an important role in reducing the carbon footprint of ICT systems, its better to understand how these systems consume energy. In this section, we will first go over the different terms that will help us to measure the energy consumption of a system. Then, we will explain different type of ICT systems, their energy demand and their role in energy consumption. After that, we will describe hardware performance counters that are used to measure energy consumption. Next, we will explain the benchmarks that are generally used to evaluate the energy efficiency of a system. Finally, we will illustrate different ways to measure power.

2.1 Energy & Power

Before defining energy, we need to understand four key terms *force*, *newton*, *joule*, and *watt*.

A *force* is a push or pull upon an object resulting from the objects interaction with another object. A *force* is measured in the SI unit of *newton*. A *newton* is defined as the amount of *force* required to accelerate a mass of one kilogram at a rate of one meter per second squared.

$$F = 1N = 1\frac{kg * m}{s^2}$$

A *joule* is a unit of energy defined as the amount of work done to move an object through a distance of meter using a *force* of one *newton*. It is also defined as the energy dissipated as heat when an electric current of 1 meter passes through a resistance of one ohm for one second.

$$1J = 1N * m = 1\frac{kg * m^2}{s^2}$$

A *watt* is a unit of *power* and defined as the amount of work done to hold the velocity of an object constant at one meter per second against a constant opposing *force* of one *newton*.

$$1W = 1\frac{J}{s} = 1\frac{N * m}{s} = 1\frac{kg * m^2}{s^3}$$

Power is the amount of energy consumed per unit time and energy is the energy to do work. A

Table 2.1: Energy Consumption Units

Joules	Unit
1 joule	1 watt second
3.6 joules	1 watt milliHour or mWHr
60 joules	1 watt minute
3600 joules	1 watt hour
3,600,000 joules	1 watt kilohour or 1 kWHr

more energy-efficient device will consume less energy per unit of time. For example, a laptop that consumes energy at a rate of 15 W is more efficient than one that consumes energy at a rate of 20 W. The amount of energy consumed by a system is calculated in watt. Table 2.1 shows different energy consumption units that are used to measure the energy consumption of ICT systems.

Power can be categorized as Idle or Active power. Idle Power is defined as the amount of power consumed by a system when its not performing any task or in other words, it is the power consumed when the system sits idle in wait of next task. As defined in [13], it is a sum of static and dynamic power as systems are not found to be in the fully static state while measured. Idle power plays a significant role in the total power consumption of a system. For future energy-efficient ICT systems, it is very crucial to minimize the idle power of a system. Active Power is defined as the amount of power consumed by a system while performing a specific task. A task can be web browsing, printing, emailing, web browsing, listening to music or playing a game. The energy consumed by a system should be proportional to the system computation. Voltage scaling and improved circuit designs have been used to minimize the active power of components in ICT systems.

The total energy consumption of ICT systems annually is expressed in kWHr whereas the energy consumed by systems on a daily basis is measured in wattours (WHrs) or milliwattHours (mWHrs)[1]. Notebooks, laptops, tablets, and mobiles use lithium-ion batteries and each strives for the best battery life. The capacity of these batteries is expressed in WHrs or mWHrs. Many factors influence the battery life of these systems like CPU, GPU, display, and memory. Heat dissipation also plays a major role in total energy consumption of these systems. Heat is a natural

byproduct of the electricity running through metal pathways in these systems. High computation needs cause more energy use, resulting in more heat production. Half of the energy at data centers is attributed to heat dissipation [18], raising a serious question on the energy efficiency of the cooling infrastructures in data centers.

2.2 ICT systems classification

ICT systems configuration varies significantly from low computation power embedded devices to high computation power Supercomputer. In this section, we will explain the different categories of ICT systems.

2.2.1 Embedded Systems

Embedded system is a microcomputer designed to perform a specific function or a set of functions with real-time performance constraints. It can be a part of bigger hardware machinery or a standalone device. We interact with a lot of embedded systems in our daily life from a small device like microwave, mp3 players to a large appliance like a refrigerator. A user cant change the functions of an embedded system but can select pre-defined functions [19]. Around 98% of the processors manufactured every year belong to the embedded systems [20]. Embedded systems are very energy-efficient as their CPU can keep working in an idle state at one-tenth or less power than the usual power drawn at peak computation [21]. An increase in temperature of embedded devices can double the chances of device failure [22]. Embedded systems have a very restricted power budget from a few milliwatts to two watts [5], however, a very high-performance requirement.

2.2.2 Handheld Devices

Handheld devices have shown an enormous growth in the past few years. One of the important reason for this growth is the decline in cost. There are two major sections of the handheld devices in the market: cell phones and tablets. The first commercial cell phone was launched in 1983 [23], at a price of $3995. Over the years, technological innovations have helped to reduce the price of multimedia cell phones to $30, making them affordable to a large part of the community. This has resulted in more than four billion mobile phone users worldwide. However, this comes with an increase in carbon footprint. Manufacturing and transporting a cell phone results in around

eighteen KG of CO2. Afterward, the average annual use can lead to an electricity consumption of up to two kWh [24]. Global tablet sales have been on a decline in 2015 [25], which can be bad news as a tablet consumes far less energy than a personal computer and can be as productive as a personal computer.

2.2.3 Personal Computer

Desktops, nettops, laptops, and netbooks fall under personal computer category and the numbers of personal computer sold in 2019 are reaching two billion [26] and expected to hit four billion counts in 2020 [27]. Operating one computer for thirty-two hours at a consumption rate of sixty Watt adds one Kg of carbon to the environment. PC has the same trend in cost and performance as handheld devices where the cost keeps on declining and the performance keeps on getting better. Apple has done significant improvement in power consumption of their products, as MacBook only consumes 0.5W in sleep mode, but still reported a carbon footprint of 25,200,000 metric tons in 2018 [28].

2.2.4 Data Center

Data centers can be found as server rooms in small-to-medium sized organizations, to the enterprise data centers in big corporations, to the server farms that run cloud computing hosted by Amazon, Facebook, Google, and others. Data centers are dedicated to the centralized accommodation, interconnection, and operation of IT and network telecommunication equipment providing data storage, processing, and transport services. The utilization of a server in a datacenter is quite a low ranging between 10% to 50% of their maximum utilization level [21]. The energy cost per square meter of the data center is up to a hundred times higher than for office accommodations [18]. Techniques like virtualization and work consolidation have been proposed to achieve energy-efficient data centers [29]. Aggregating the workload on a single machine increases the utilization and the efficiency rate of a machine ensuring the mitigation of greenhouse gasses. Kwasinski et al.[30] proposes Distribution Green Data Centers (DGDC) which are small distributed centers spread over a geographical region for better energy efficiency. DGDC uses local power resources and the emphasis is given to renewable energy. Even less amount of energy waste in transmitting

Table 2.2: ICT Systems TDP rating [1]

ICT System	TDP
Embedded Systems	<5 W
Smart Phone and Tablets	4-12 W
Laptop	45-60 W
Desktop	90-130 W
Small Server	80-165 W
Supercomputer and Data Center servers	300W to thousands

energy from the power station to local data centers results in more energy saving. DGDC have simplified cooling system which consumes less energy than one in big data centers.

2.2.5 Supercomputer

Supercomputers have the highest computation power which helps the community in several ways weather forecasting, financial analysis, disaster prediction, scientific applications, and national defense. However, this scale of computation requires enormous power consumption, not only to run the supercomputer but also to cool it [31]. Since 1992, there has been a 10,000-fold increase in the performance of the supercomputers whereas performance per watt has seen an only 300-fold improvement [32]. The fastest supercomputer Summit in TOP500 at a speed of 148 petaFLOPs requires 10 megawatt of peak power whereas the most energy-efficient system DGX SaturnV Volta in GREEN500 requires 97 kilowatt but has a speed of 1 petaFlops. Annual power cost is on the verge of exceeding the acquisition cost of supercomputers, which will change the performance above all mindset in the supercomputing community. Energy efficiency is also one of the bottlenecks in developing next-generation exascale systems [33]. The one way to achieve this is by improving the energy efficiency of the hardware, but its not sufficient to design exascale systems [34]. Energy-efficient Software will be a major breakthrough in designing next-generation exascale systems.

2.3 ICT systems energy demand

ICT systems have a variable energy demand. There are two ways to estimate it.

Table 2.3: Performance Counter in Processor Family

Processor	Number of PMC
AMD Athlon 64	4
Intel Xeon 7500	7
Power 8	6
R10000	2

2.3.1 Thermal Design Power (TDP)

TDP is the maximum amount of heat generated in a system that the system can dissipate. Lower TDP systems like embedded systems and mobiles use passive cooling like radiation heat transfer, conduction, and convection. Higher TDP systems like PC, HPC or data centers use active cooling like fans, water cooling or thermoelectric coolers. Table 2.2 shows the TDP rating for different ICT systems.

2.3.2 Performance per Watt

Performance per watt is the maximum amount of computation achieved for every watt of power consumed by the system. It does not only represents the power consumed for computation but also for the heat dissipation. The Green 500 uses performance per watt to evaluate the energy efficiency of Supercomputers. There has been a little growth in performance per watt in Supercomputers as compared to performance increase over the years [32].

2.4 Hardware Performance Counters

Hardware Performance Counters are special registers that are located on microprocessors to monitor hardware-related activities [35]. A lot of work is done in estimating power consumption using Hardware Performance Counters [36, 37, 38, 39]. Each microprocessor has a different set of Hardware Performance Counters. Hardware performance counters can be used in two ways: for aggregate measurement and for statistical sampling [40]. Aggregate measurement is used to get information while running a specific program or part of a program. At the start of the program or in between of the program, counters start collecting different events data and show the gathered results when the program finishes. Statistical sampling is used to gather information from specific

performance counters, at regular intervals using timer or overflow interrupts. The advantage of using statistical sampling is that the program need not to be changed while taking a measurement, however, the disadvantage is that it is hard to match the recorded values to any particular application or program. Some performance counters are programmable which can be used to collect user-specific events. A user can select at most the same number of events concurrently as the number of hardware counters available. Table 2.3 shows the number of performance counters in different types of processor family.

2.5 Benchmarks

Benchmarks are the software that has been developed to measure the energy consumption of ICT systems. The most important associations for energy efficiency measurement are the Transaction Processing Performance Council (TPC), the Standard Performance Evaluation Corporation (SPEC) and Storage Performance Council (SPC) [41]. SPEC announced the first industry-standard benchmark to measure the energy efficiency of server-class computers - SPECpower_ssj2008, it measures the power consumption in relation to performance. SPC published SPC-1C/E as its first energy consumption benchmark in 2009 and provides the linkage between power consumption and performance for their existing benchmark SPC-1C. TPC-Energy benchmark was announced in 2010 and measures the energy consumption in relation to the amount of work completed.

2.6 Software Role

Software plays a major role in the total power consumption of an ICT system[13, 34, 37]. In one of the experiments conducted at Intel, the idle power of system changed from 8.6 W to 13.1 W due to the addition of software [1]. For this study, the idle power of the system was measured with a change in software in three ways.

In the first scenario, the idle power was measured on a brand new 100% charged OEM notebook of 56 WHr battery that has an operating system and minimal drivers. The LCD display was set to the lowest brightness with sleep mode disabled. The Ethernet cable was plugged in, which was connected to corporate LAN. The system was then left idle until the battery drains out. It took a total of 6 hours and 32 minutes to get the notebook turned off. As it was a 56 WHr battery, the

average power was 8.6W.

In the second scenario, IT build was added on the notebook and the same process is repeated. IT build is the software that the corporate IT department put on a new system before issuing it to the new employee. This software includes office applications, system state monitoring applications, virus protection and utilities which help IT department to manage the system. In this case, the battery ran out in 4 hours and 43 minutes giving an idle power of 11.9 W.

In the third scenario, more software was added to the system that is specific to an employee work such as text editors, IDEs, weather gadget or news feed. These applications were started and kept idle with no user interaction until the whole battery drained out. In this case, the battery ran out in 4 hours and 30 minutes giving an idle power of 13.1 W.

The addition of software on the above system causes the reduction of battery life of 33%. This shows that the software can severely impact the power consumption of a system.

Pinto et al. [42] conducted a study to understand what application programmers think about software energy consumption problems. They collected data from StackOverflow of 300 questions, 550 answers from 800 users to know:

1. Whether the application programmers are aware of the energy consumption problem in software?

2. What are the software energy questions they are asking?

3. What do developers think are the main reasons of the software energy consumption?

4. What are the steps taken by application developers to save energy?

The study shows that the application programmers' interest in software energy consumption is increasing over the years. It has increased by 100% in 2012 as compared to the number of questions in 2011 and 183% in 2013 as compared to 2012. This is interesting because more the programmers asked the question, more they are serious about this issue. Application developers were asking questions regarding measurement, knowledge, code design, and context-specific questions about

software energy consumption. Programmers think that unnecessary resource usage, faulty GPS usage, background activities, excessive synchronization, high GPU usage, background wallpapers and advertisement in mobile applications are the reasons for software high energy consumption. The solutions implied by programmers are keeping I/O to the minimum, perform bulk operations, avoid polling, use concurrent programming, race to idle, efficient data structures, lazy initialization, and hardware coordination.

2.7 Power Measurement

Power measurement approaches are very crucial for the optimization of both hardware and software energy-efficiency. Power measurement can be categorized into three parts: hardware-based method, software-based method, and hybrid method.

2.7.1 Hardware-Based methods

Hardware-based methods can be divided into two groups direct power measurement and integrated power sensors. Direct power measurement method uses meters to directly measure the power of any specific hardware. Joseph et al. use this method to measure the power consumption of individual components of a microprocessor and compare it to the estimated power consumption calculated using hardware performance counters [43]. Kamil et al. use inline and clamp meters to measure power consumption in single nodes to a full-scale supercomputer [44]. They were able to model the power consumption of the whole system from a subset of the system. This method is good to evaluate software-based method of power estimation but can not be used to estimate the power of software applications. Integrated power sensors are used to estimate the power of high-performance servers. Its hard to estimate the power of low-level hardware using this technique due to its complexity. One of the drawbacks of using an integrated circuit is that it dissipates a lot of energy.

2.7.2 Software-Based Methods

Software-based methods are less accurate but less complex in estimating power consumption. Software methods use power models that are developed at various levels circuit, instruction, component, etc. Software methods can be grouped as architectural-level power models and system-

level power models. Architectural level models are used to estimate the power dissipation during the architecture design stage. Several models have been proposed to estimate the power dissipation of chips, microprocessor, registers, memory, on-chip buses, disks, routers, etc. Brooks et al. come up with Wattch to estimate the CPU power consumption [45]. Liu et al. developed a tool for power estimation in CMOS VLSI chips by leveraging gate count, memory size, logic, and layout style [46]. The power estimation at register-transfer level is proposed in [47] using measures based on entropy and informational energy. Ye et al. developed Simplepower, a framework to estimate power consumption of register-transfer level, memory system, and on-chip buses [48]. Softwatt was developed as a complete system power simulator to capture the interaction between different components in [49]. It was used to estimate the power consumption in CPU, memory, disk, application and operating system. The advantage of the architectural level models was that they were more efficient in power estimation, however, less accurate.

System level power models are used to supply live power information to OS and applications. These models are very helpful in virtual environments where the guest OS dont have direct control over the hardware resources. A lot of models in the system level group use PMC to estimate the power dissipation [37, 39, 50, 51]. In [52], Joulemeter was created to provide same power metering functionality for VM that is available on physical server hardware. Hypervisor-observable states were used to track the hardware usage of a single VM. Joulemeter power capping functionality was used to manage the power consumption of individual VM. Dhiman et al. developed an online power prediction model for virtualized environments using Gaussian mixture models [53]. It outperformed the linear and multivariate regression model with an average prediction error of less than 10%. JouleUnit, an energy profiling framework was proposed in [54] for estimating power consumption of applications using workload and sensors values for different ICT devices. Wang et. al. proposes two-level power model in [55], where they used the fewest PMC to estimate the power dissipation of application source code. Using the power model, they implement SPAN to find out the power dissipation of the source code in an application. SPAN can be used to find the more power-hungry part of a source code. The only problem in the SPAN approach is that the

developers need to add SPAN source code to the application source code for power estimation. Mair et al. present the misconception in estimating power consumption using CPU performance monitoring counters [56]. They discuss five myths in PMC power estimation research. First, the previous work in power estimation doesnt provide with PMC sampling rate and time for which the samples were taken. Second, there was no mention of the temperature at which sampling was done. However, temperature plays a significant in affecting the power consumption of a system. Third, memory events like cache misses are a good indicator of power consumption of ICT systems, which is not the case in a multicore architecture. Fourth, compiler optimization doesnt affect power consumption. Fifth, there is no way to compare the previous research studies as there is no common metrics to compare it.

2.7.3 Hybrid-Based Methods

Hybrid methods take data at both hardware level and software level to estimate the power dissipation. Flinn et al. designed PowerScope which used hardware instrumentation and kernel software support to estimate power consumption of applications [57]. It has been shown to reduce 46% of the power consumption of a video playing application. Isci and Martonosi proposed a technique for measuring power consumption by combining real total power measurement calculated using a multimeter with per unit power consumption estimation using performance counters [58]. Ge et al. designed a framework PowerPack that can measure the power consumption of different components of a system like disks, memory, NICs, and processors and map these measurements to application functions [59]. It also supported multicore and multiprocessor based node and parallel applications in HPC systems.

2.8 Summary

This chapter introduces various terms to define energy and power. It discusses different type of ICT systems like embedded systems, handheld devices, personal computer, data center, and supercomputer. It explains TDP and performance per watt which are used to estimate energy demand in ICT systems. It talks about hardware performance counters and energy efficiency benchmarks. It describes the role of software in ICT systems energy consumption. Finally, it presents various

approaches to measure power in ICT systems.

CHAPTER 3 CALCULATING IDLE ENERGY

Same as power, energy can be classified as idle and active energy. Unstabilized idle energy results in outliers which can cause inaccurate measurements. In this chapter, we stabilize the idle energy on two ICT systems to calculate their idle energy consumption.

3.1 Set Up

We leverage two different ICT systems to conduct our experiments: Intel Fog Node and Laptop. The configuration of these two systems is presented in Table 3.1. For the Laptop, the charger is plugged in a wall outlet all the time. Both systems are disconnected from the internet all the time.

We use the Linux `perf` tool to gather energy consumption values of package and core domains. The sampling rate is set to 10Hz. The following command is executed for each run:

```
$ sudo perf stat -a -r 1 -I 100\
      -e 'power/energy-pkg/'\
      -o pack.txt\
      java programFile
```

where `-a` specifies collection from all CPUs, `-r` indicates how many times the command will be repeated, `-I` specifies the time interval (msec), `-e` specifies the event selector, and `-o` specifies the name of the output file.

3.2 Idle Energy

The total energy consumption is the sum of active and idle energy. As `perf` reports the total energy, one can subtract the idle energy out of the total energy to find the active energy of an application. However, the idle energy of a system can vary a lot due to the background services running on an operating system. These variations make it hard to measure an accurate idle energy of a system. To measure the idle energy of both systems consider here, we conduct an experiment in which we first stabilize the idle energy and then calculate the idle energy of both systems by removing outliers and computing the mean of values. We first measure the idle energy of both systems without any stabilization for 24 hours with a sampling rate of 10Hz to determine how the

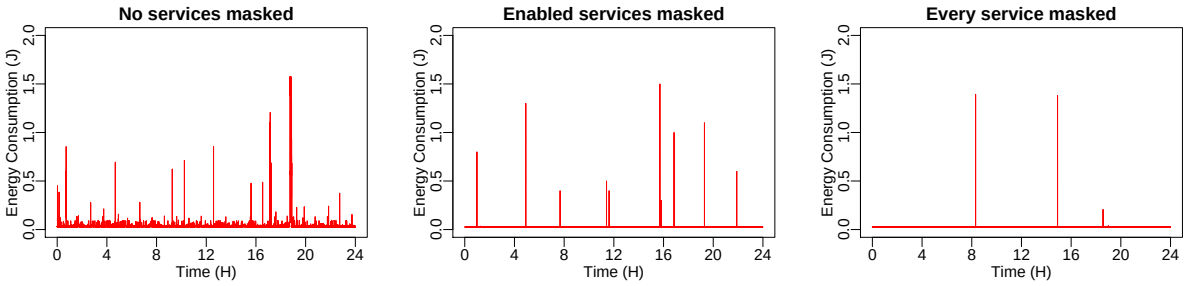
Table 3.1: System specification

System Component	Intel Fog Node Configuration	Laptop Configuration
CPU	Intel(R) Xeon(R) E3-1275 v5	Intel(R) Core(TM) i5-3317U v5
Socket	1	1
Number of cores	4	2
Number of threads	8	4
Architecture	x86_64	x86_64
Kernel	4.13.0-37-generic	4.4.0-116-generic
OS	Ubuntu Server 16.04.4 LTS	Ubuntu Server 16.04.3 LTS
CPU governor	powersave	powersave
Memory	32GB SODIMM 2133 MHz	4GB DDR3 1600 MHz
L1d cache	32KB	32KB
L1i cache	32KB	32KB
L2 cache	256KB	256KB
L3 cache	8MB	3MB
JDK	64-Bit Server VM	64-Bit Server VM
JDK build	25.151-b12	25.151-b12
JDK version	1.8.0_151	1.8.0_151
Initial Heap Size	526MB	63MB
Maximum Heap Size	8.4GB	1GB

idle energy varies. In Fig. 3.1a and 3.2a, we show the idle energy consumption of the two systems. We can see that the idle energy can change abruptly at any time for both systems.

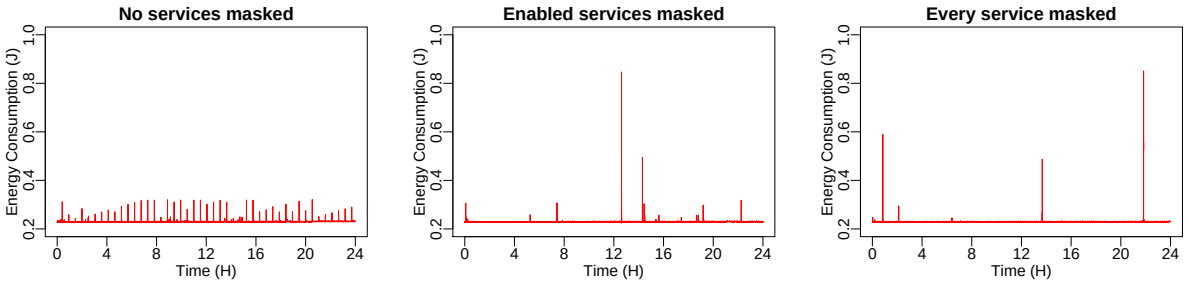
3.3 Stopping System Services

Next, we stop the background services using `systemctl` command to stabilize the idle energy of the systems. For both systems, we disable all the enabled services. We again measure the idle energy and we observe that there is a lot of variation. The reason is that the disabled services can still be enabled because if a service is disabled, then it is not loaded during boot time but it can be loaded if a service is started and it depends on the disabled service. Next, we mask all the enabled services using `systemctl` command to stabilize the idle energy. If a service is masked, then it cannot be loaded even if it is required by some other service. This time we are able to stabilize the idle energy with a very few outliers as shown in Fig. 3.1b and 3.2b. We go one step further and mask all the disabled services and then obtain fewer outliers, as shown in Fig. 3.1c and 3.2c. However, outliers were still there as we can't mask some of the services like log in, user



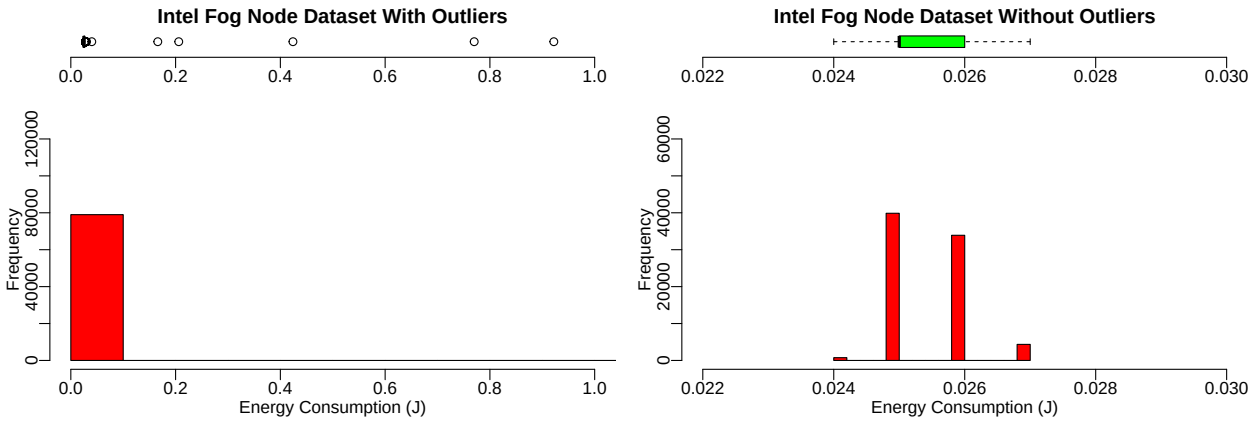
(a) (b) (c)

Figure 3.1: Intel Fog Node package idle energy consumption: (a) with all services unmasked; (b) with enabled services masked, and (c) with all services masked.



(a) (b) (c)

Figure 3.2: Laptop package idle energy consumption: (a) with all services unmasked; (b) with enabled services masked, and (c) with all services masked.

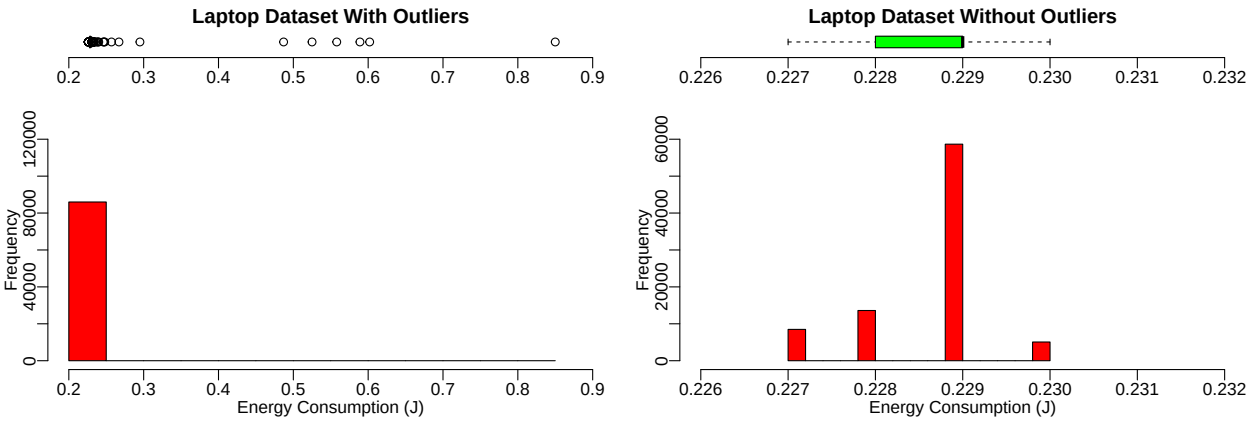


(a) (b)

Figure 3.3: Intel Fog Node dataset: (a) with outliers, and (b) without outliers.

manager and dbus.

The next step is to remove the outliers from the 24-hour dataset shown in Fig. 3.1c and 3.2c



(a) (b)

Figure 3.4: Laptop dataset: (a) with outliers, and (b) without outliers.

and calculate the mean of the values. We use Tukey's method to remove the outliers [60]. For Intel Fog Node and Laptop, the outliers represent 0.001% and 0.002% of the total data set, respectively. We remove the outliers from both datasets. The histogram and boxplot before and after removing outliers for the Intel Fog Node are shown in Fig. 3.3a and 3.3b, and for the Laptop in Fig. 3.4a and 3.4b. The standard deviation for Intel Fog Node is 0.0006, and for Laptop is 0.0007. The standard deviation indicates that both datasets have low variation. The mean idle energy consumption per one-tenth of a second for Intel Fog Node and Laptop is found to be 0.025 J and 0.229 J, respectively. We can now calculate the active energy by subtracting the idle energy from the total energy.

One of the drawbacks of masking all the services is that it can make the system unstable as it stops some services that are essential for the systems. Even when we unmask all the masked services, we were not able to restart the system. In such cases, we have to reinstall the operating system. Due to this problem, we limit the stabilization of idle energy to measure the energy consumption of Java command-line options. For Java programming language components and JEPO, we rely on taking multiple measurements, removing outliers and calculating mean of the values.

3.4 Summary

In this chapter, we are able to stabilize the idle energy of two ICT systems by masking all the enabled and disabled services of Ubuntu Server OS. The stabilization helps us to achieve a standard

deviation of 0.0006 in idle energy. However, stabilization also makes the system unstable which limits the application of this stabilization technique.

CHAPTER 4 ENERGY EFFICIENCY OF JAVA PROGRAMS

In this chapter, we conduct a comprehensive study to evaluate the energy efficiency from the perspective of the Java programs. Java is one of the most commonly-used languages in ICT systems. We leverage two ICT systems to evaluate the energy consumption of data types, operators, control statements, String, exceptions, objects, and Arrays in Java using Intel RAPL technology for different iteration size of micro-benchmarks. For data types, we evaluate primitive data types, access modifiers, local and static variables, and scientific notations. For operators, we evaluate arithmetic, assignment, compound, pre- and post-increment, pre- and post-decrement, and short-circuit operators. For control statements, we evaluate if-then-else, conditional operator and loops. For exceptions, we evaluate try-catch block calls with and without exception. For String, we evaluate various ways of concatenating, converting and comparing String. For objects, we compare wrapper classes. For threads, we investigate different types of thread implementations. For Arrays, we compare different ways to copy and traverse Arrays.

We use the jRAPL [61] framework to gather energy consumption values of package and core domains. For better accuracy, we measure the total energy consumption of each code hundred times. We then check for outliers in those hundred measurements using Tukey's method. We remove the outliers and calculate the mean of remaining values to come up with the final value of energy consumption. In cases where the means are close, we use the independent sample t-test (two means) or one-way ANOVA test (more than two means) to determine whether the means are the same. For both tests, we consider alpha value as 0.05 which means that if the p-value is smaller than or equal to 0.05, we reject the null hypothesis that the mean values are the same. If the p-value is greater than 0.05, it means that the mean values are the same. If the means are the same, we conclude that there is no difference in energy consumption. Next, we present a detailed analysis of all these Java components. Part of this work is published in [62, 63].

4.1 Energy Consumption Traits

In this section, we evaluate the energy consumption of different Java code snippets. Each plot in the next subsections of the chapter represents the total energy consumption of a single iteration

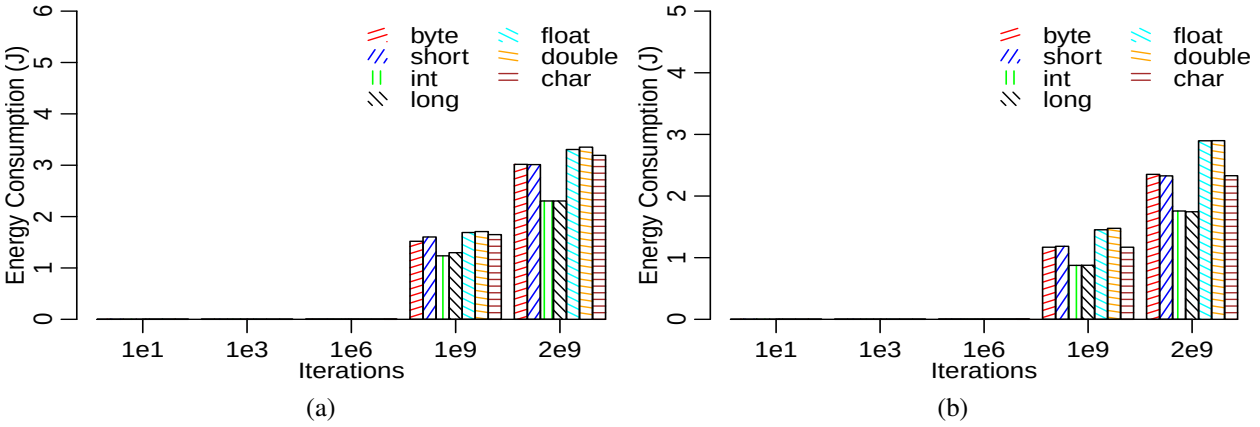


Figure 4.1: Primitive data types: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
1  double[] before = EnergyCheckUtils.getEnergyStats();
2  start = System.nanoTime();
3  byte var = 0;
4  for (int j = 0; j < iter; j++) {
5   var *= j % 2 == 0 ? var++ : var--;
6
7  }
8  finish = System.nanoTime();
9  double[] after = EnergyCheckUtils.getEnergyStats();
```

Code 4.1: byteVariable.java

of a code snippet. The number of iterations in each code is adjusted to support the default heap size. We logged the package, core, and dram energy but only present the package energy consumption values as the core and dram energy measurements values are negligible compared to those of the package. For some of the results, we do not present the code used to obtain them since the code can be found in our previous work [62].

4.1.1 Variables

In Fig. 4.1a and 4.1b, we compare all of the primitive data types except `boolean` in terms of energy consumption. For each data type, we declare a variable and execute a conditional operator statement for various iteration. The code for `byte` is shown in Code 4.1. Line 1,2,8, and 9 relates to the code to measure energy consumption and execution time. For the rest of the codes in this chapter, we don't show these lines. Other primitive data types code are shown in Code 4.2, 4.3, 4.4, 4.5, 4.6, and 4.7. For iteration size 1e1, 1e3 and 1e6, all primitive data types consume the same

```
short var = 0;
for (int j = 0; j < iter; j++) {
 var *= j % 2 == 0 ? var++ : var--;
}
```

Code 4.2: shortVariable.java

```
int var = 0;
for (int j = 0; j < iter; j++) {
 var *= j % 2 == 0 ? var++ : var--;
}
```

Code 4.3: intVariable.java

```
long var = 0;
for (int j = 0; j < iter; j++) {
 var *= j % 2 == 0 ? var++ : var--;
}
```

Code 4.4: longVariable.java

```
float var = 0;
for (int j = 0; j < iter; j++) {
 var *= j % 2 == 0 ? var++ : var--;
}
```

Code 4.5: floatVariable.java

amount of energy for both systems. For other iterations, `int` consumes the least amount of energy whereas `double` consumes the most amount of energy for both systems same as in [64] and [65]. However, for smaller iteration sizes of up to 1e6, all primitive data types consume the same energy. `double` consumes up to 67% more energy than `int`. The higher energy for `long` is expected because `long` has a size of 64 bits, whereas `int` has a size of 32 bits. As `long` has more storage overhead, we conclude that the code using `long` consumes more energy. The reason for the higher energy consumption of `byte`, `short`, and `char` is the implicit conversion of `byte`, `short`, and `char` variables to `int` in bytecode. The `float` and `double` consume significantly more energy than any other data types due to the extra overhead to store decimal numbers. According to these results, `int` is the most energy-efficient where as `double` is the least energy-efficient primitive data type.

Variables in Java can be declared as local and static variables. Local variables of a method are stored in the stack and lead to lower execution times. They can be changed in their specific

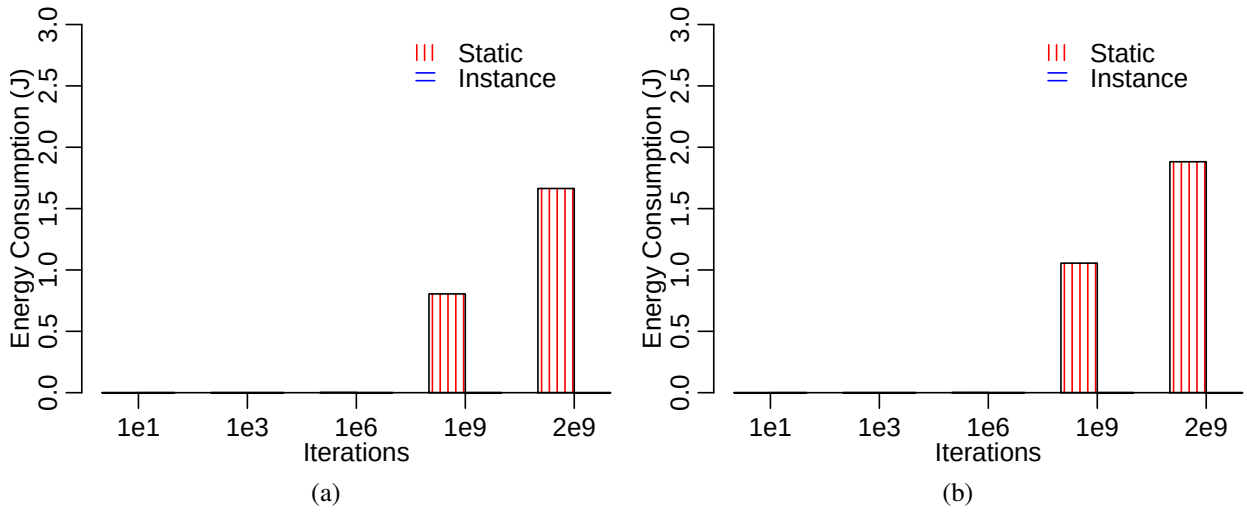


Figure 4.2: Instance and static variables: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
double var = 0;
for (int j = 0; j < iter; j++) {
 var *= j % 2 == 0 ? var++ : var--;
}
```

Code 4.6: doubleVariable.java

```
char var = 0;
for (int j = 0; j < iter; j++) {
 var *= j % 2 == 0 ? var++ : var--;
}
```

Code 4.7: charVariable.java

```
int var = 0;

for (int j = 0; j < iter; j++) {
 var = j % 2 == 0 ? var++ : var--;
}
```

Code 4.8: localVariable.java

scope during the program execution. Static variables are stored in the heap and have a global scope. For the local and static variable experiments, we create a local and static variable and assign conditional operator result to it as shown in Code 4.8 and 4.9. The results of the energy consumption measurements for both cases are shown in Fig. 4.2a and 4.2b. For iteration size 1e1, 1e3 and 1e6, local and static variable consume the same amount of energy for both systems. For

```
static int var = 0;

public static void main(String[] args) throws IOException {

 for (int i = 0; i < repeat; i++) {

  for (int j = 0; j < iter; j++) {
   var = j % 2 == 0 ? var++ : var--;
  }

 }
```

Code 4.9: staticVariable.java

```
public class publicVariable {

 public long var = 0;

 public static void main(String args[]) {

  int iter = Integer.parseInt(args[0]);
  publicVariable obj = new publicVariable();

  for (int i = 0; i < iter; i++) {

   obj.var *= obj.var;
  }
 }
}
```

Code 4.10: publicVariable.java

one billion iteration size, the code using static variable consumes 8,300% and 7,100% more energy than the one using local variable for the Intel Fog Node and Laptop, respectively. For two billion iteration size, the code using static variables consumes 17,700% and 14,300% more energy than the one using local variable for the Intel Fog Node and Laptop, respectively. The slower heap memory and more overhead due to extra instructions in bytecode is the reason for the behavior of the code with the static variables.

Encapsulation in Java allows the control of access for members of a class. A member can have four types of modifiers - public, private, protected, and default. For all types, we execute the compound multiplication assignment statement on each member type two billion times. The public variable code is shown in Code 4.10. The other access modifiers are implemented in the same way. However, when we check the bytecode for each access modifier, we find them to be

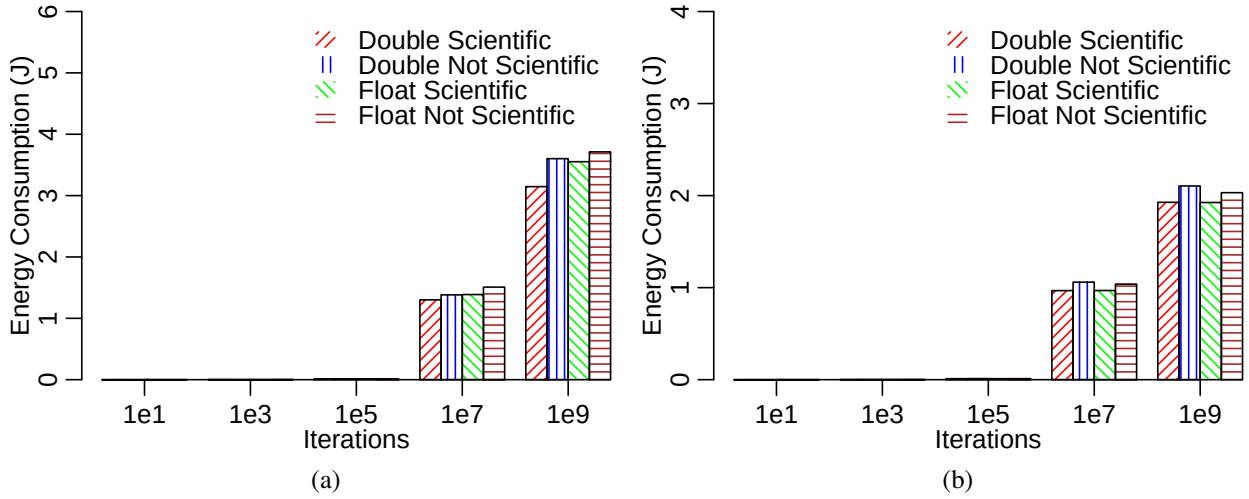


Figure 4.3: Float and Double scientific notation: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
1  double var = 1234567.00;
2
3  for (int j = 0; j < iter; j++) {
4   var += j;
5  }
```

Code 4.11: doubleNormal.java

```
1  double var = 1.234567e6;
2
3  for (int j = 0; j < iter; j++) {
4   var += j;
5  }
```

Code 4.12: doubleScientific.java

same. Therefore, access modifiers do not have any effect on energy consumption of a variable.

Scientific notation is another way to write a decimal number. In scientific notation, e or E is used to express the decimal number. Both `float` and `double` numbers can be written as scientific notation. We compare the decimal and float number to their counterparts scientific notation. The code for `double` number and its scientific counterpart are shown in Code 4.11 and 4.12. For `float`, the same is shown in Code 4.13 and 4.14 . For iteration size 1e1, 1e3, and 1e5, `float` and `double` numbers consume the same amount of energy as scientific notation for both systems as shown in Fig. 4.3a and 4.3b. For iteration size 1e7 and 1e9, `double` and `float` numbers con-

```
float var = 1234567.0f;

for (int j = 0; j < iter; j++) {
 var += j;
}
```

Code 4.13: floatNormal.java

```
float var = 1.234567e6f;

for (int j = 0; j < iter; j++) {
 var += j;
}
```

Code 4.14: floatScientific.java

```
long a = 0;

for (long j = 1; j < iter; j++) {
 for (long k = 1; k < iter; k++) {
  a = j + k;
 }
}
```

Code 4.15: add.java

```
long a = 0;

for (long j = 1; j < iter; j++) {
 for (long k = 1; k < iter; k++) {
  a = j - k;
 }
}
```

Code 4.16: subtract.java

sume up to 14% more energy than the scientific notation. Simplifying the arithmetic operations helps scientific notation to consume less energy.

4.1.2 Operators

Java programming language provides arithmetic operators for addition, subtraction, multiplication, division, and modulus. These operators are also called binary operators as they take two operands. The operands of these operators can be integers or real numbers. For each arithmetic operator, we perform arithmetic operations using different `long` and `double` operands and store the value in the same `long` and `double` variable. The code for different operators are shown in Code 4.15, 4.16, 4.17, 4.18, and 4.19. For iteration size 1e1, 1e3 and 1e4, all operators consume the

```
long a = 0;

for (long j = 1; j < iter; j++) {
 for (long k = 1; k < iter; k++) {
  a = j * k;
 }
}
```

Code 4.17: multiply.java

```
long a = 0;

for (long j = 1; j < iter; j++) {
 for (long k = 1; k < iter; k++) {
  a = j / k;
 }
}
```

Code 4.18: divide.java

```
long a = 0;

for (long j = 1; j < iter; j++) {
 for (long k = 1; k < iter; k++) {
  a = j % k;
 }
}
```

Code 4.19: modulus.java

same amount of energy for both systems using `long` operands as shown in Fig. 4.4a and Fig. 4.4b. For iteration size 1e5, the modulus and division operator consume a significantly higher amount of energy. The addition operator consumes the least whereas the modulus operator consumes the highest and up to 1,620% more energy. For `double` operands, iteration size 1e1, 1e3, and 1e4 shows the same results. For iteration size 1e5, the modulus operator shows the same behavior for `double` operands, but division operator consumes lesser amount of energy as the other operators. The reason for faster divison of `double` operands is the faster execution of exponent part. The modulus operator consumes up to 277% more energy than the addition operator. When we check the bytecode for differences between mod and add, the only difference was the change from `drem` to `dadd` instruction. Higher number of CPU cycles consume by these instructions result in higher energy consumption of division and modulus operator.

|| and `&&` operators are called short-circuit operators in Java. When the first operand of the `&&`

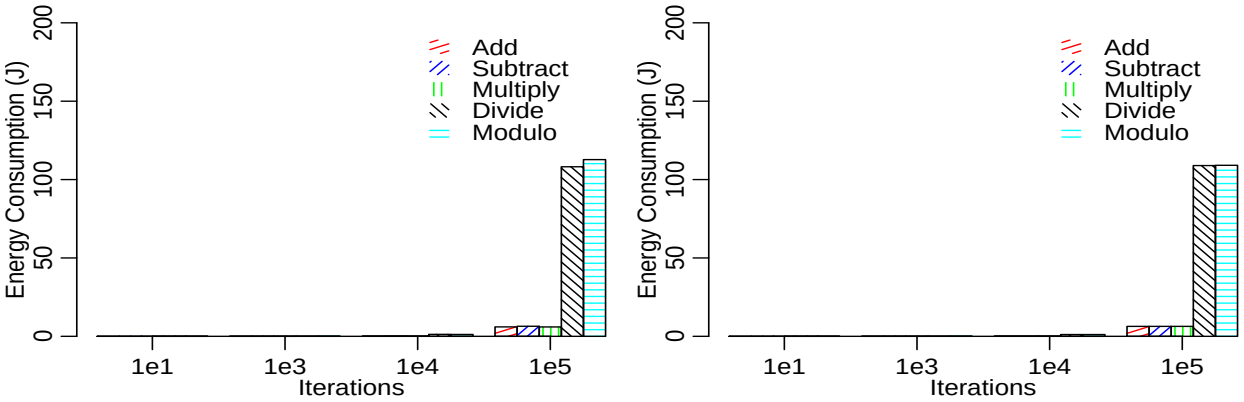


Figure 4.4: Arithmetic operators using `long` variables: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
int var = 0;

for (int j = 0; j < iter; j++) {
 if (var > -1 || var > 1 || var > 1)
  var = j;
}
```

Code 4.20: circuitLeftOr.java

```
int var = 0;

for (int j = 0; j < iter; j++) {
 if (var < -1 || var < -1 || var > -1)
  var = j;
}
```

Code 4.21: circuitRightOr.java

operator evaluates to false, the result of the whole condition is false, and when the first operand of the || operator evaluates to true, the result of the whole condition is true. We compare the two cases where the true operand of the || operator is in the first and last position. We create a for loop with various iteration sizes, with the if condition having two || operators and three operands. For the first case (First), the result is returned after the evaluation of the first operand as shown in Code 4.20. For the second case (Last), the result is returned after evaluation of the last operand as shown in Code 4.21. The first and second cases are shown in Fig. 4.5a and 4.5b. For iteration size 1e1, 1e3 and 1e6, both cases consume the same amount of energy for both systems. For other iteration

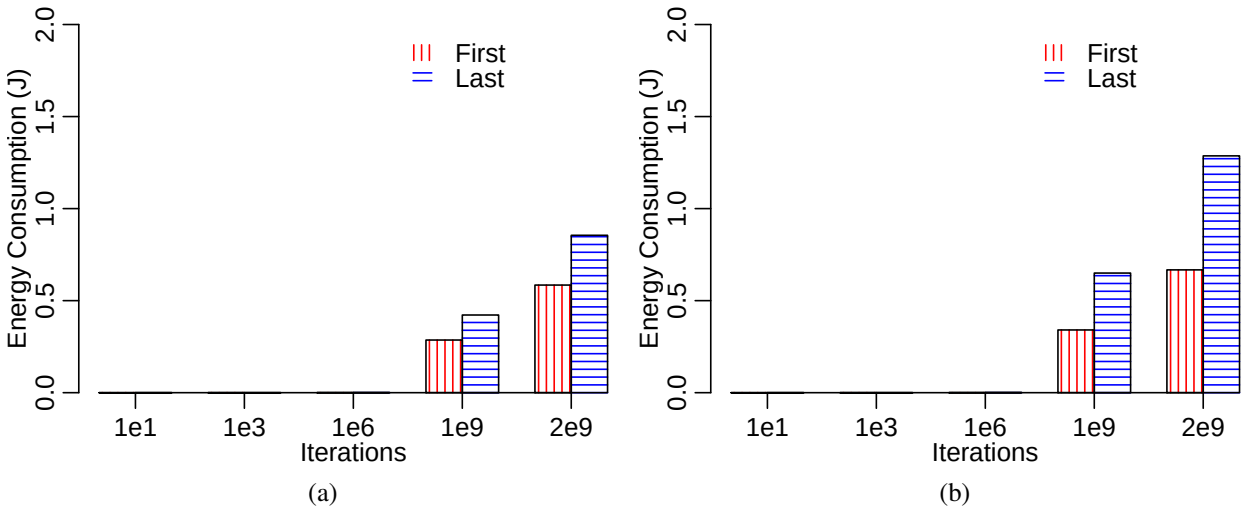


Figure 4.5: First operand as true and last operand as true in || operator expression: (a) Intel Fog Node, and (b) Laptop package energy consumption.

sizes, the Or Last case consumes up to 100% more energy than the Or First case for both systems. Therefore, it is better to always put those cases first in short circuit operators that are the most common. The higher energy consumption is due to extra operands execution in Or Last case. The `&&` operator shows the same behavior.

The assignment operator and arithmetic operators can be combined in two ways - normal and compound. In the normal assignment, the operands are first evaluated on the right and then the result is assigned to a variable on the left. In the compound assignment, an arithmetic operator is combined with an assignment operator. We compare these two cases. However, when we check the bytecode, we find that both cases have the same bytecode. Therefore, it does not matter whether the normal or compound assignment is used in Java. Next, we compare the compound addition and subtraction with post-increment, pre-increment, post-decrement, and pre-decrement operators for various iterations. As shown in Fig. 4.6a and 4.6b, all operators consume the same energy for different iteration sizes. The code for compound addition, post-increment and pre-increment is shown in Code 4.22, 4.23, and 4.24.

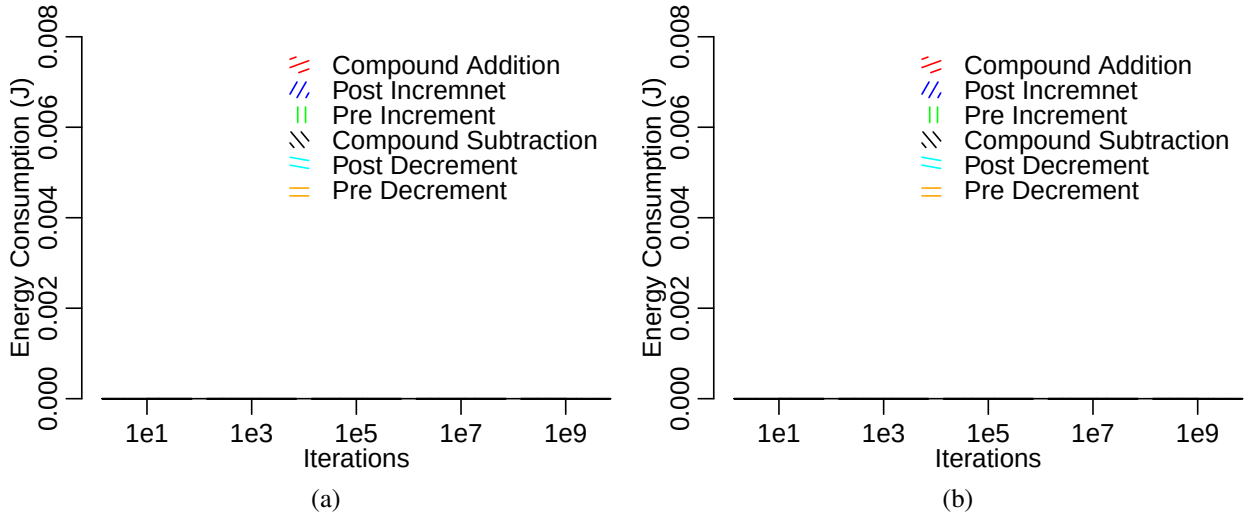


Figure 4.6: Increment and decrement operators: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```java
int var = 0;

for (int j = 0; j < iter; j++) {
 var += 1;
}
```

Code 4.22: compoundAddition.java

```java
int var = 0;

for (int j = 0; j < iter; j++) {
 var++;
}
```

Code 4.23: postIncrement.java

```java
int var = 0;

for (int j = 0; j < iter; j++) {
 ++var;
}
```

Code 4.24: preIncrement.java

```java
String var = "";

for (int j = 0; j < iter; j++) {
 var = j % 2 == 0 ? "Even" : "odd";
}
```

Code 4.25: conditional.java

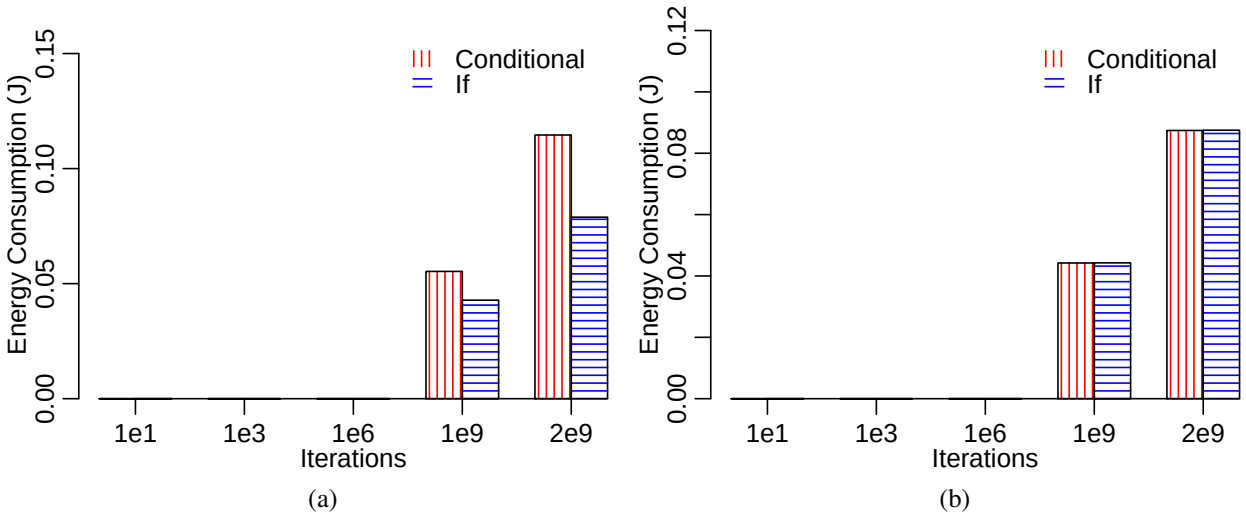


Figure 4.7: Conditional operator and if-then-else: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
1   String var = "";
2
3   for (int j = 0; j < iter; j++) {
4
5    if (j % 2 == 0)
6     var = "Even";
7    else
8     var = "Odd";
9
10  }
```

Code 4.26: ifThenElse.java

4.1.3 Control Statements

Conditional or ternary operators in Java allow assigning a value to a variable based on the boolean expression value. It allows writing an if-then-else statement in one line. It is called ternary operator because it takes three operands. In Code 4.25, we execute various iteration sizes in a for loop and test whether the iteration number is even or odd using the conditional operator. We do the same for the if-then-else case in Code 4.26. For the if-then-else case, the bytecode has an extra instruction `astore` to store a reference into a local variable. Fig. 4.7a and 4.7b, show the energy consumption of the two cases for the Intel Fog Node and the Laptop. For iteration size 1e1, 1e3 and 1e6, both the cases consume the same amount of energy for both systems. For the

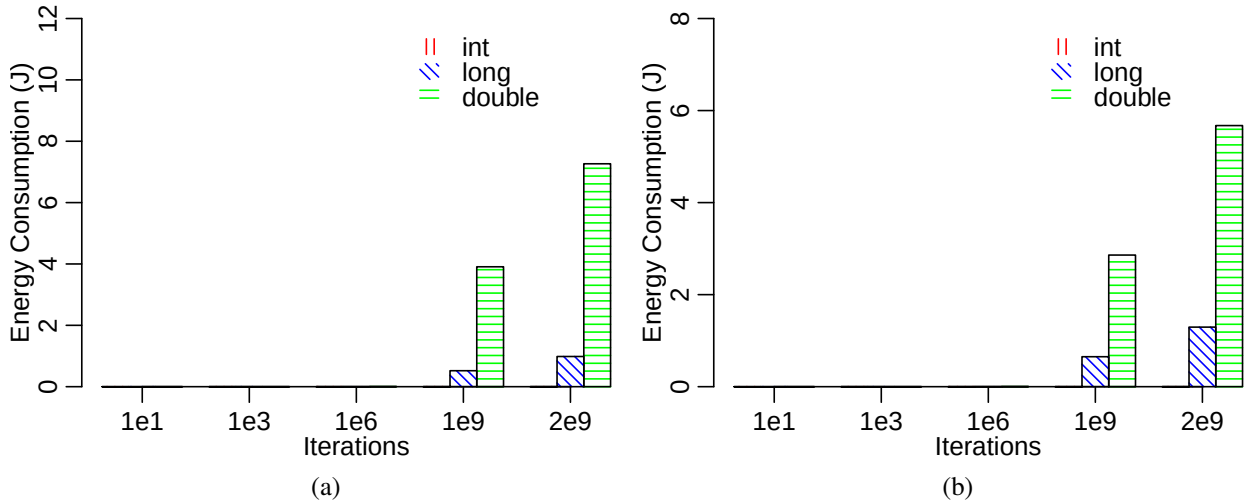


Figure 4.8: Iteration variable (different data types): (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
1  int var = 0;
2
3  for (int j = 0; j < iter; j++) {
4   var++;
5  }
```

Code 4.27: loopInt.java

```
1  int var = 0;
2
3  for (long j = 0; j < iter; j++) {
4   var++;
5  }
```

Code 4.28: loopLong.java

Intel Fog Node, conditional operator consumes 25% and 37% more energy for one billion and two billion iteration size, respectively. For the Laptop, conditional operator consumes less than 1% higher energy for the two iteration sizes.

Loops are frequently used to execute a block of statements or a single statement several times. For our analysis, we first change the iteration variable of a for loop to a different data type - `int`, `long`, and `double`, and then execute post-increment operator for various iteration sizes for each. The code for `int`, `long`, and `double` iteration variable is shown in Code 4.27, 4.28, and 4.29. The results for these cases are shown in Fig. 4.8a and 4.8b. For iteration size 1e1, 1e3 and 1e6,

```
int var = 0;

for (double j = 0; j < iter; j++) {
 var++;
}
```

Code 4.29: loopDouble.java

```
ArrayList<Integer> loop = new ArrayList<Integer>();

for (int j = 0; j < iter; j++) {
 loop.add(j);
}

for (int j = 0; j < loop.size(); j++) {
 loop.get(j);
}
```

Code 4.30: methodTerminate.java

```
ArrayList<Integer> loop = new ArrayList<Integer>();

for (int j = 0; j < iter; j++) {
 loop.add(j);
}

int size=loop.size();

for (int j = 0; j < size; j++) {
 loop.get(j);
}
```

Code 4.31: varTerminate.java

both the cases consume the same amount of energy for both systems. The `int` consumes the least amount of energy whereas the `double` consumes the most and up to 573% more energy.

We also examine two different ways of initializing the termination expression in a for loop: using a variable and using a method call. In both cases, we first initialize an `ArrayList` and add different number of integers to it. Then, in the method case, we use the `size` method to specify the termination expression of the for loop, as shown in Code 4.30, whereas in the variable case, we first store the size of the list in an integer variable and then use that variable in the for loop termination expression to avoid multiple calls to the `size` method as shown in Code 4.31. For iteration size 1e1, 1e3 and 1e5, both the cases consume the same amount of energy for both systems. For iteration size 1e7, the energy consumption in the method termination case is up to 3%

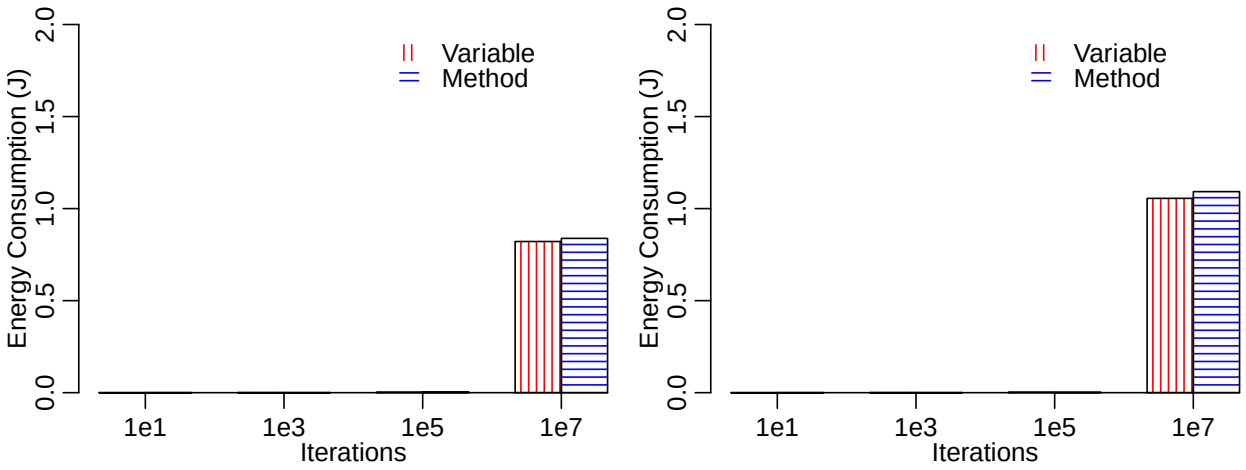


Figure 4.9: Termination expression variable and method: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
1  int var = 1, j = 0;
2
3  do {
4
5   var *= j;
6   j++;
7
8  } while (j < iter);
```

Code 4.32: doWhile.java

more than the variable case for both systems, as shown in Fig. 4.9a and 4.9b. When we check the bytecode for both cases, the variable case result in additional instructions, however, it consumes lesser energy due to the removal of method calls overhead. A different method with significant overhead can result in higher energy consumption in the method case than in the variable case.

`for`, `while` and `do-while` are the loop statements in Java. In our experiments, each type of loop executes a compound multiplication assignment statement for different iteration sizes. Code 4.32, 4.33, and 4.34 show the implementation for the `do-while`, `while`, and `for` loop. All loop statements consume the same amount of energy as shown in Fig. 4.10a and 4.10b. However, when we check the bytecode for each loop, the one corresponding to the `for` loop is found to have the most number of instructions, while the one for the `do-while` loop is found to have the least number of instructions. Next, we compare the `for` statement to the enhanced `for` statement

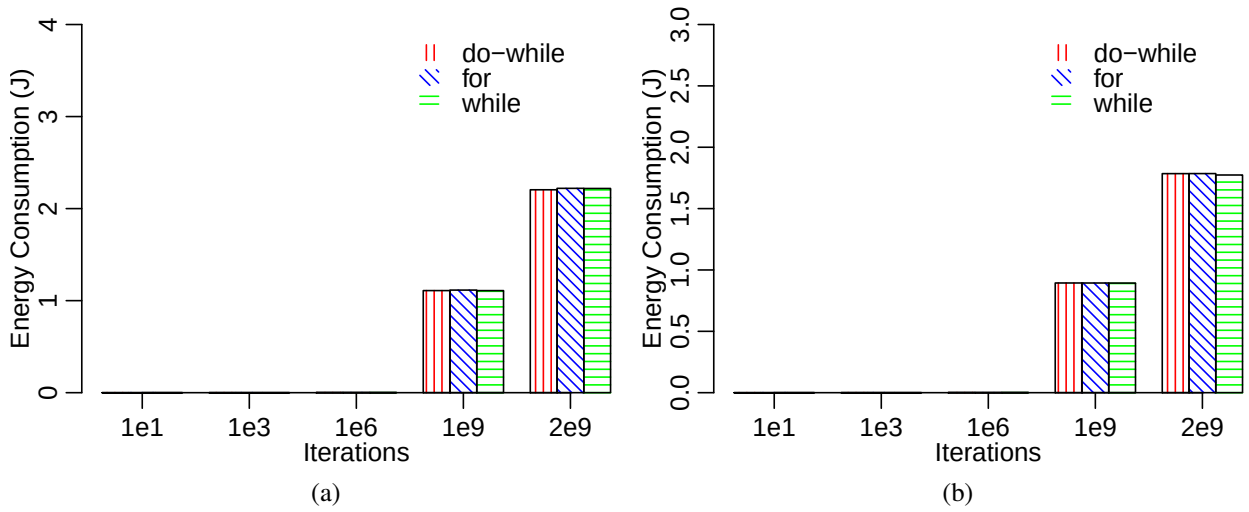


Figure 4.10: `For`, `while` and `do-while`: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
int var = 1, j = 0;

while (j < iter) {

 var *= j;
 j++;

}
```

Code 4.33: while.java

```
int var = 1;

for (int j=0; j < iter; j++) {

 var *= j;

}
```

Code 4.34: for.java

which is mainly used to iterate over a collection. It does not contain the termination and increment expression and allows traversing through the collection without explicitly knowing the index. It can only be used when one needs to iterate from the first to the last value in a collection. We access various sizes array using the `for` loop and the enhanced `for` loop. The enhanced `for` loop implementation is shown in Code 4.35. We find that both cases have the same energy consumption.

```
int a = 0;

int[] arr = new int[iter];

for (int i = 0; i < arr.length; i++) {

 arr[i] = i;

}

for (int i : arr) {

 a *= i;

}
```

Code 4.35: enhancedVsFor.java

```
int var = 1;

for (int j = 1; j < iter; j++) {
 try {
  var = var / j;
 } catch (Exception E) {
  System.out.println("Denominator can't be zero!!!");
 }
}
```

Code 4.36: exceptionInLoop.java

```
int var=1;

try {
 for (int j = 1; j < iter; j++) {
  var = var / j;
 }
} catch (Exception E) {
 System.out.println("Denominator can't be zero!!!");
}
```

Code 4.37: exceptionOutLoop.java

4.1.4 Exceptions

Java uses try-catch blocks to handle exceptions. We examine whether a try-catch block has any associated energy cost when it is used in a program with no exception thrown. We compare the energy consumption of multiple calls of the try-catch block inside a loop with a single call (Code 4.36) to a try-catch block outside the loop (Code 4.37). The first case results in various size executions of a try-catch block, while the second case results in the single execution of a try-catch

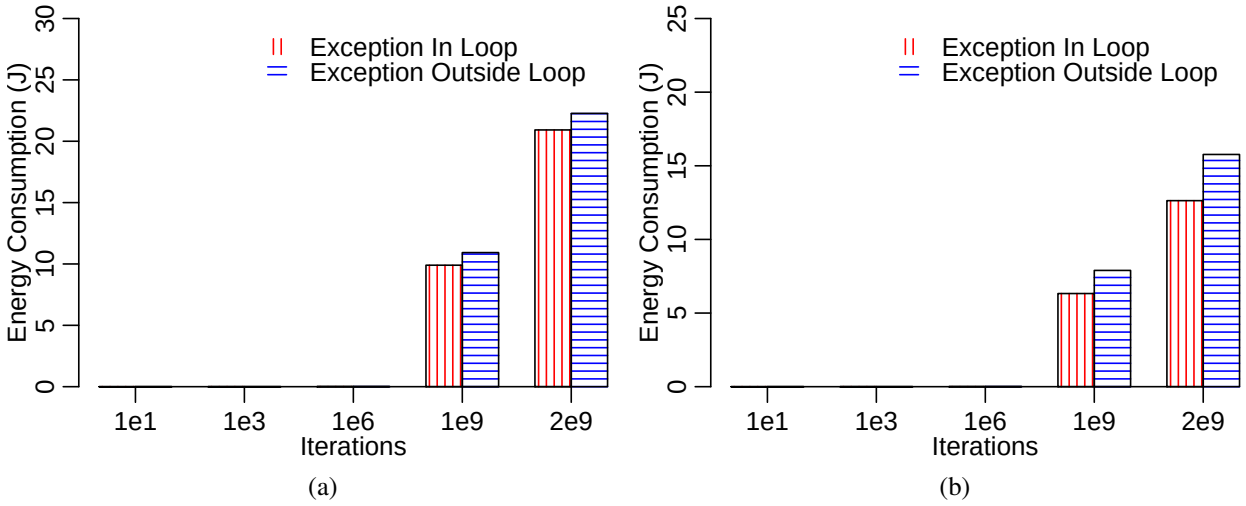


Figure 4.11: Try-catch in loop and not in loop: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
1  int a=1,b=1;
2
3  for(int i=0;i<2000000000;i++){
4   try{
5   a=a/0;
6   }catch(Exception E){
7    a=a/b;
8   }
9  }
```

Code 4.38: exceptionThrownLoop.java

block. For iteration size 1e1, 1e3 and 1e6, both the cases consume the same amount of energy for both systems. However, for iteration size 1e9 and 2e9, we find that the single execution of try-catch block result in up to 24% more energy consumption as shown in Fig. 4.11a and 4.11b. When we check the bytecode, both cases have almost the same bytecode with a small difference in the order of the instructions. The higher energy consumption can be a result of scope difference.

Executing a catch block has a major overhead as it involves reading the stack for the thrown exception. We measure the impact of the catch block by running two cases - exception thrown and no exception thrown - using the `Exception` class. All exceptions in Java are subtypes of the `Exception` class. In the exception thrown case, we put a try-catch block inside a for loop which iterate two billion times as shown in Code 4.38. For each iteration, an exception is thrown due to

```
int a=1,b=1;

for(int i=0;i<2000000000;i++){
 try{
 a=a/b;
 }catch(Exception E){
  a=a/b;
 }
}
```

Code 4.39: exceptionNotThrown.java

```
  String str = "";

  for (int j = 0; j < iter; j++) {
   str += j + "";
  }
```

Code 4.40: concatenationOperator.java

division by zero. Then, we write the same statements with no exception thrown as shown in Code 4.39. The exception case consumes 2.46% and 3.29% more energy for the Intel Fog Node and the Laptop, respectively. The results can differ for a different exception type as different exceptions have different overheads.

4.1.5 String

`String` is immutable and cannot be altered once created in Java. Applying different methods on a `String` does not change the original `String`. For example, `String.substring (int beginIndex)` returns a new `String` that is a substring of the original `String`. `String` supports the concatenation operator (+) which combines two or more strings. The concatenation operator is resolved at compile time if all the strings that need to be combined can be resolved at compile time. However, if an expression cannot be resolved at compile time, the concatenation operator will execute at run time and causes extra overhead [66]. `String` has function named `concat`, `StringBuilder` and `StringBuffer` has function named `append` which we compare with the `String` concatenation operator in Fig. 4.12a and 4.12b. For each case, we concatenate an integer variable and string for various iteration sizes. The code using the concatenation operator, `concat` method, `StringBuilder append` method and `StringBuffer append` method is shown in Code 4.40, 4.41, 4.42, and 4.43. For iteration size 1e1, 1e3 and 1e4,

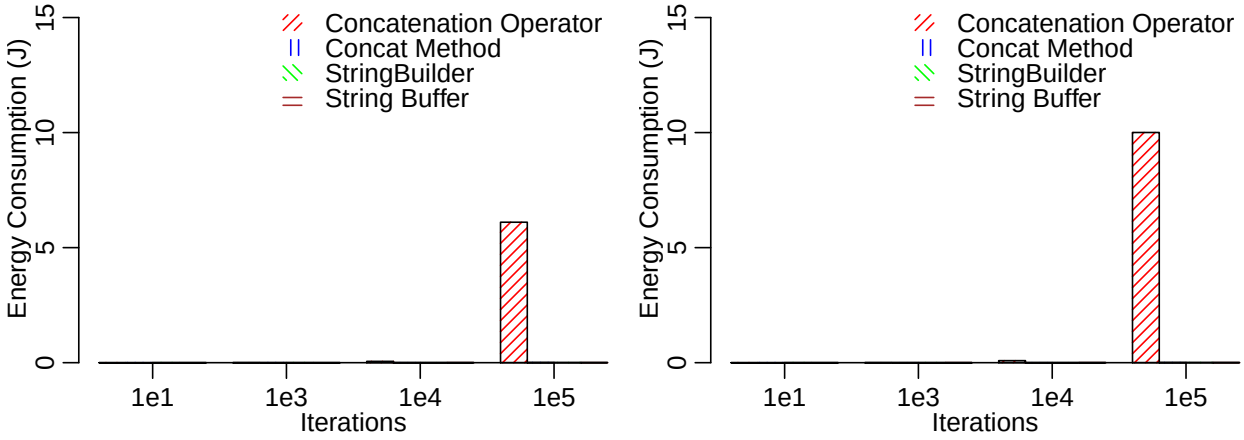


Figure 4.12: `String` concatenation: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
1   String str = "";
2
3   for (int j = 0; j < iter; j++) {
4    str.concat(j + "");
5   }
```

Code 4.41: concatMethod.java

both the cases consume the same amount of energy for both systems. The concatenation operator consumes up to 1,48,069% more energy than other methods after iteration size 1e3 for both systems. `StringBuilder append` method consumes the least amount of energy same as in [64] and [65], however, concatenation of up to 1e4 `String` values result in same energy consumption of all methods. The reason for the higher energy consumption of concatenation operator is multiple calls to `StringBuilder append` method, creating a new `String` object on each iteration, whereas the `StringBuilder append` method use resizable array to store the strings and change it to `String` only when required. The `String concat` method result in higher energy consumption than `StringBuilder append` method due to copying the `String` values to `char` array and creating a new `String` object on each iteration. `StringBuilder` performs better than `StringBuffer` as `StringBuffer` is synchronized.

Java provides various methods to covert a `String` to a primitive data type. For example, to

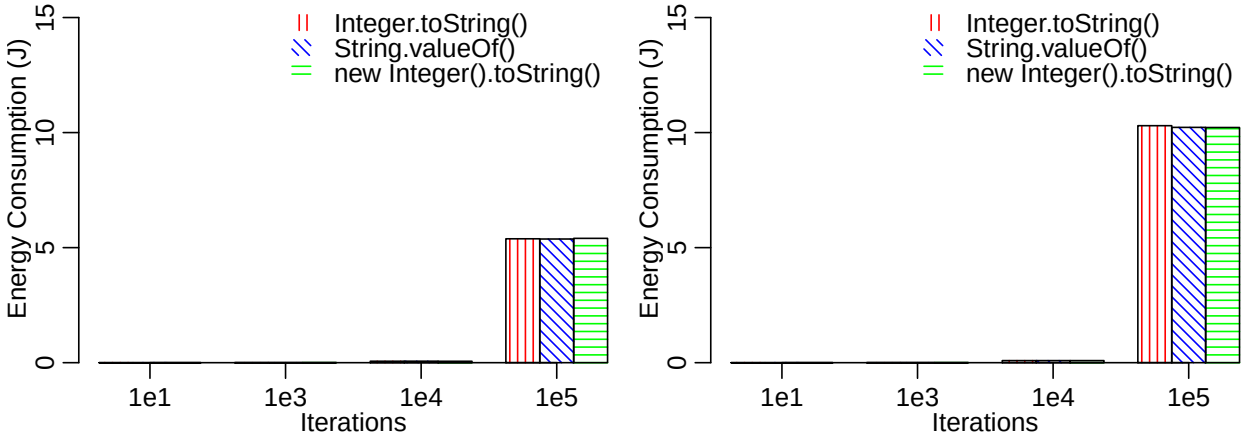


Figure 4.13: `String` conversion: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
1   StringBuilder str = new StringBuilder();
2
3   for (int j = 0; j < iter; j++) {
4    str.append(j + "");
5   }
```

Code 4.42: StringBuilder.java

```
1   StringBuffer str = new StringBuffer();
2
3   for (int j = 0; j < iter; j++) {
4    str.append(j + "");
5   }
```

Code 4.43: StringBuffer.java

convert a `String` variable var to integer one can use following methods - Integer.toString(var), String.valueOf(var) and new Integer().toString(var). For each method, we convert various number of integers as shown in Code 4.44, 4.45, and 4.46. For iteration size 1e1, 1e3 and 1e4, both the cases consume the same amount of energy for both systems. For both systems, all methods consume same energy for all iterations as shown in Fig. 4.13a and 4.13b. The reason for the same energy consumption is using the same functionality to convert the `int` variable to `String`. First the `int` variable is converted to `char` buffer using the `char` array and then this `char` array is converted to `String` using `String` class constructor.

```
String str = "";

for (int j = 0; j < iter; j++) {
 str += Integer.toString(j);
}
```

Code 4.44: integerToString.java

```
String str = "";

for (int j = 0; j < iter; j++) {
 str += String.valueOf(j);
}
```

Code 4.45: stringValueOf.java

```
String str = "";

for (int j = 0; j < iter; j++) {
 str += new Integer(j).toString();
}
```

Code 4.46: newIntegerToString.java

`String` has various methods to compare whether two strings are equal. A developer can use `String equals` or `compareTo` method, or `Objects equals` method for `String` comparison. `String equals` first check whether the two `String` objects are same or not. If they are same, then the `equals` method return true. If they are not equal, both `String` object are compared for each character. `String compareTo` compares two `String` objects lexicographically and return 0 if the two `String` objects are equal. `Objects equals` method check whether the two `String` objects refer to same object or not. It doesn't check for `String` objects contents. For `String equals` and `compareTo` method, we compare various `String` as shown in Code 4.47 and 4.48. We don't compare `Objects equals` method to `String equals` or `compareTo` method as we have to compare `String` objects content. For iteration size up to 1e3, all methods consume the same energy for both systems as shown in Fig. 4.14a and 4.14b. For other iteration sizes, `String compareTo` method consumes up to 33% more energy than `String equals` method. The higher energy consumption for `String compareTo` method is due to extra overhead to compare `String` objects lexicographically.

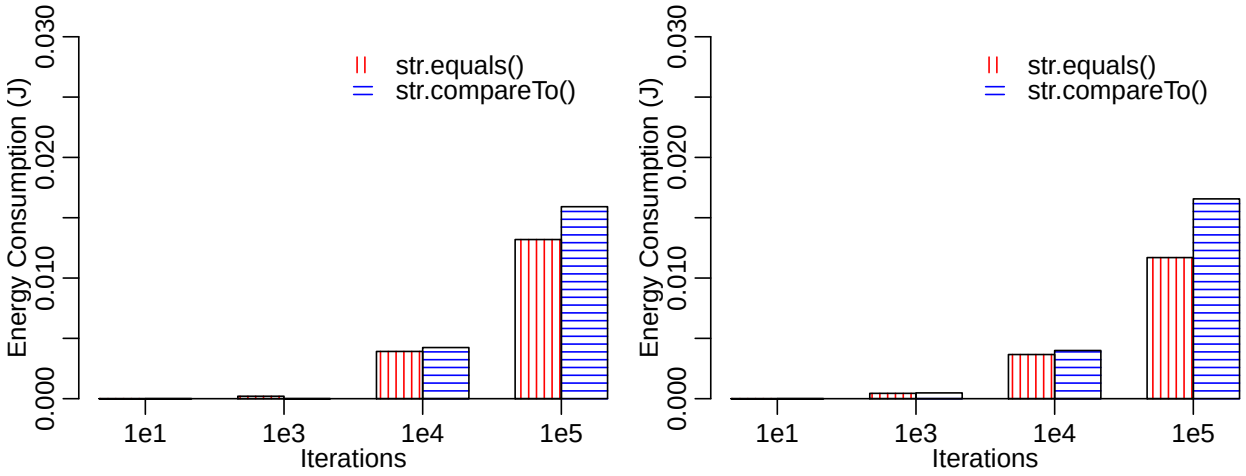


Figure 4.14: `String` comparison: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
String[] str1 = new String[iter];
String[] str2 = new String[iter];

for (int j = 0; j < iter; j++) {
 str1[j] = j + "";
 str2[j] = j + "";
}

for (int j = 0; j < iter; j++) {
 str1[j].equals(str2[j]);
}
```

Code 4.47: stringEqual.java

```
String[] str1 = new String[iter];
String[] str2 = new String[iter];

for (int j = 0; j < iter; j++) {
 str1[j] = j + "";
 str2[j] = j + "";
}

for (int j = 0; j < iter; j++) {
 str1[j].compareTo(str2[j]);
}
```

Code 4.48: stringCompareTo.java

4.1.6 Objects

Wrapper classes in Java are used to convert a primitive data type into an object and vice-versa. For example, `Integer` wrapper class is used to convert primitive data type `int` into an object.

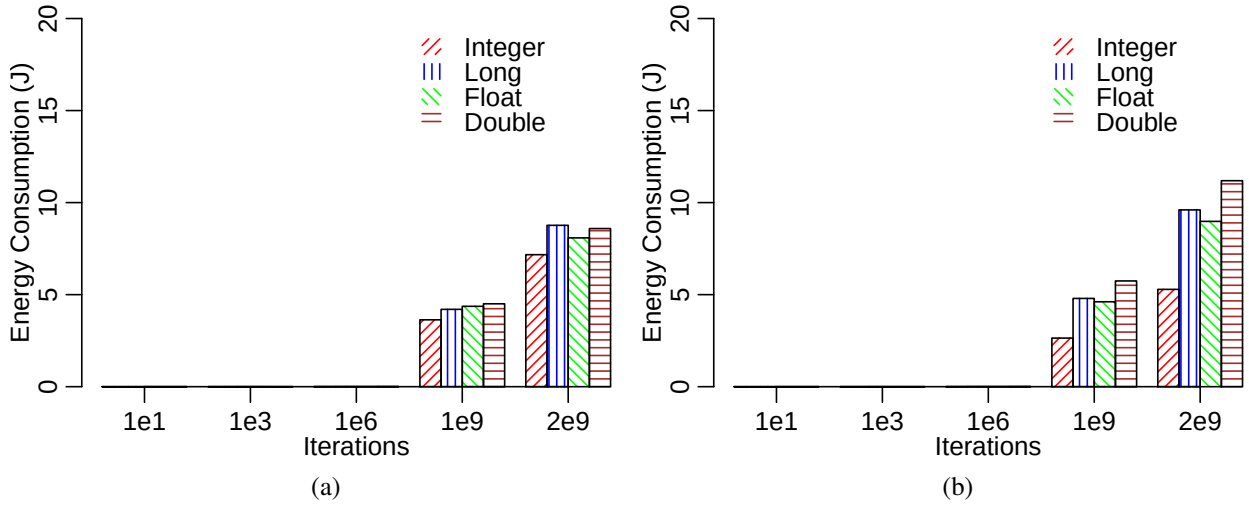


Figure 4.15: Wrapper variables: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
1   Integer var = 0;
2
3   for (int j = 0; j < iter; j++) {
4    var *= j % 2 == 0 ? var++ : var--;
5   }
```

Code 4.49: Integer.java

It is used in cases such as storing `int` values in a collection as they only accept objects as values. We compare `Integer`, `Long`, `Float` and `Double` wrapper classes to analyze their energy consumption values. We initialize a variable of each type to zero and execute a conditional operator statement for various iteration sizes. The code for `Integer` `Long`, `Float` and `Double` wrapper classes are shown in Code 4.49, 4.50, 4.51, and 4.52. For iteration size up to 1e6, all wrapper classes consumes same energy for both systems as shown in Fig. 4.15a and 4.15b. `Integer` class consumes the least amount of energy where as `Double` class consumes the most amount of energy for higher iteration sizes for both systems. Therefore, developers can prefer to use `Integer` wrapper class when dealing with collections. `Double` consumes up to 115% more energy than `Integer`.

4.1.7 Threads

Threads allow dividing a task into smaller parts and then each part is executed concurrently. This way of executing multiple threads at the same time is called multithreaded programming.

```
Long var = 0L;

for (int j = 0; j < iter; j++) {
 var *= j % 2 == 0 ? var++ : var--;
}
```

Code 4.50: Long.java

```
Float var = 0F;

for (int j = 0; j < iter; j++) {
 var *= j % 2 == 0 ? var++ : var--;
}
```

Code 4.51: Float.java

```
Double var = 0D;

for (int j = 0; j < iter; j++) {
 var *= j % 2 == 0 ? var++ : var--;
}
```

Code 4.52: Double.java

Threads can be used in Java either by implementing a `Runnable` interface or extending `Thread` class. The `Runnable` interface allows sharing of the same object instance but the `Thread` class results in unique objects for each thread. As a Java class can only extend a single class, it is better to use the `Runnable` interface to allow the class to extend any other class. The `Runnable` interface and the `Thread` class consume the same energy when we create three different threads and execute the compound multiplication assignment statement various times for each thread.

4.1.8 Arrays

Arrays are very commonly used in a programming language to store or operate on data. Many a time, an array is copied to other to avoid making changes to original array. An array can be either copy manually or using the methods in Java - `clone()`, `System.arraycopy()` and `Arrays.copyOf()`. We compare all these ways of copying array for different array sizes. The code for manually copying, `clone()`, `System.arraycopy()` and `Arrays.copyOf()` are shown in Code 4.53, 4.54, 4.55, and 4.56. For both systems, array size of up to 1e5 result in the same energy consumption as shown in Fig. 4.16a and 4.16b. For array size 1e7, `System.arraycopy()` results in the least amount of energy consumption where as

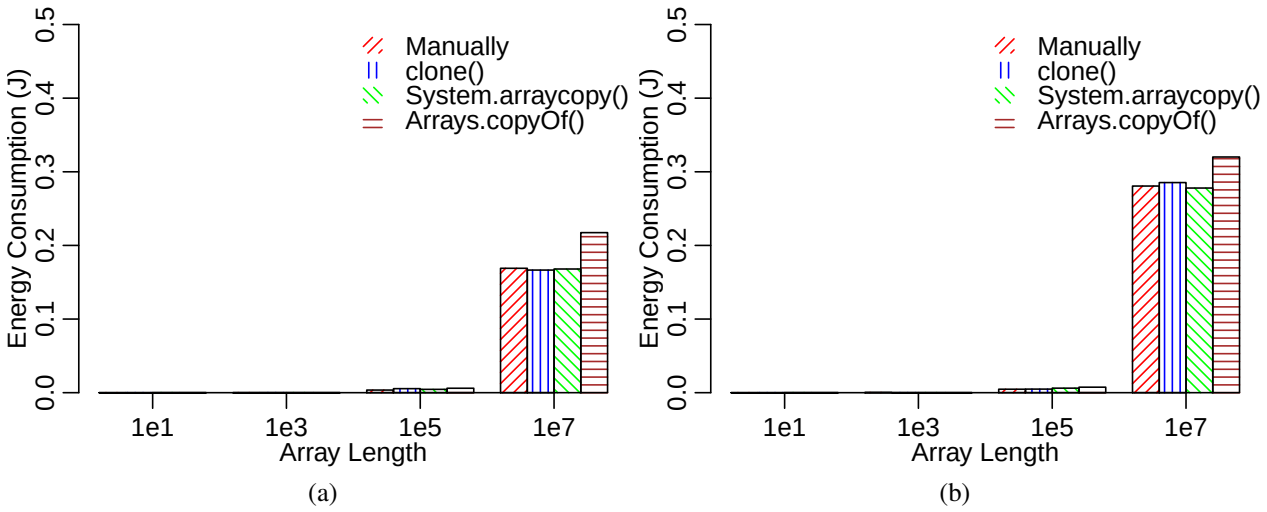


Figure 4.16: Array copy: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
1   int[] var = new int[iter];
2   int[] varC = new int[iter];
3
4   for (int j = 0; j < iter; j++) {
5    var[j] = j;
6   }
7
8   for (int j = 0; j < iter; j++) {
9    varC[j] = var[j];
10  }
```

Code 4.53: manualArrayCopy.java

`Arrays.copyOf()` result in the most amount of energy consumption. `Arrays.copyOf()` consumes 14% more energy than `System.arraycopy()`. `Arrays.copyOf()` utilizes `System.arraycopy()` to copy an array, however, it creates a new array and uses extra instructions to calculate arguments for `System.arraycopy()` method. `clone()` method results in higher energy consumption as it creates a new array object and copies the values of an existing array to it. `System.arraycopy()` was found to consume lesser energy when compared to manual copy in [64]. From this study, we get to know that Array size of smaller than 1e5 consume same energy and `System.arraycopy()` performs better than `clone()` and `Arrays.copyOf()`.

Accessing an array of more than one dimension is possible either by row or column. We compare the energy consumption of various array sizes while traversing them by row and column.

```
int[] var = new int[iter];
int[] varC = new int[iter];

for (int j = 0; j < iter; j++) {
 var[j] = j;
}

varC = var.clone();
```

Code 4.54: clone.java

```
int[] var = new int[iter];
int[] varC = new int[iter];

for (int j = 0; j < iter; j++) {
 var[j] = j;
}

System.arraycopy(var, 0, varC, 0, iter);
```

Code 4.55: systemArrayCopy.java

```
int[] var = new int[iter];
int[] varC = new int[iter];

for (int j = 0; j < iter; j++) {
 var[j] = j;
}

varC = Arrays.copyOf(var, iter);
```

Code 4.56: arraysCopyOf.java

We want to know at what array size it matters whether we are traversing array by row or column. The code for row and column traversal is shown in Code 4.57 and 4.58. For array size up to 1e3 in a two dimension matrix, both row and column traversing consumes same energy as shown in Fig. 4.17a and 4.17b. However, an array size of 1e4 result in up to 793% more energy consumption for column traversing. The reason for higher energy consumption for column traversal is the out of order access of memory locations. The same result were shown by a matrix of 1024x1024 in [64], however, the smaller size matrix shows same energy consumption for row and column traversal.

4.2 Energy And Execution Time

In this section, we analyze the relation between the energy and execution time for JDK 7, 8, 9, 10, 11 and 12. Each JDK release a number of new features. We calculate the correlation

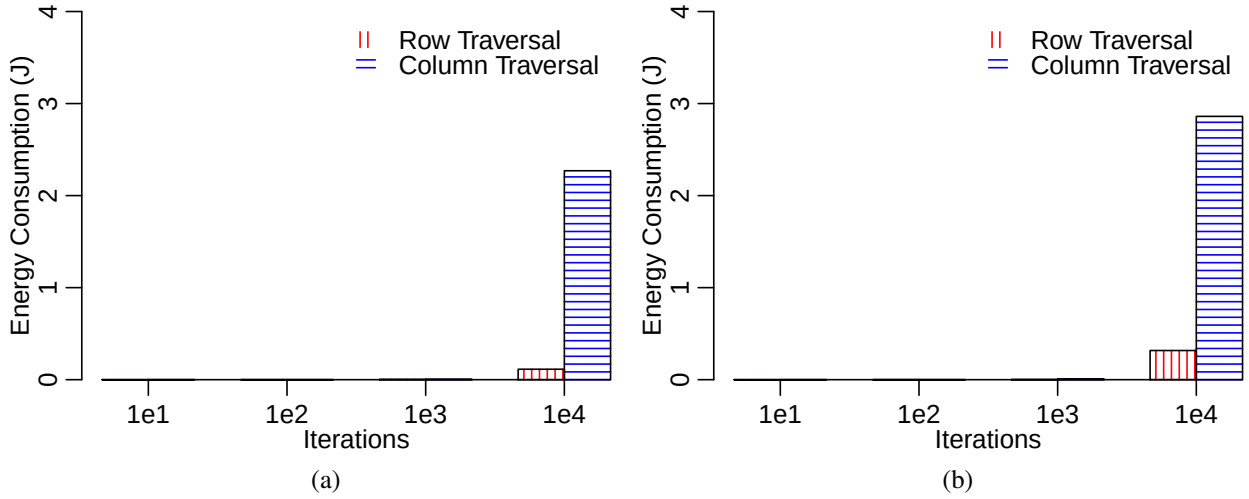


Figure 4.17: Matrix traversal row and column: (a) Intel Fog Node, and (b) Laptop package energy consumption.

```
int[][] var = new int[iter][iter];

for (int j = 0; j < iter; j++) {
 for (int k = 0; k < iter; k++) {
  var[j][k] = j + k;
 }
}
```

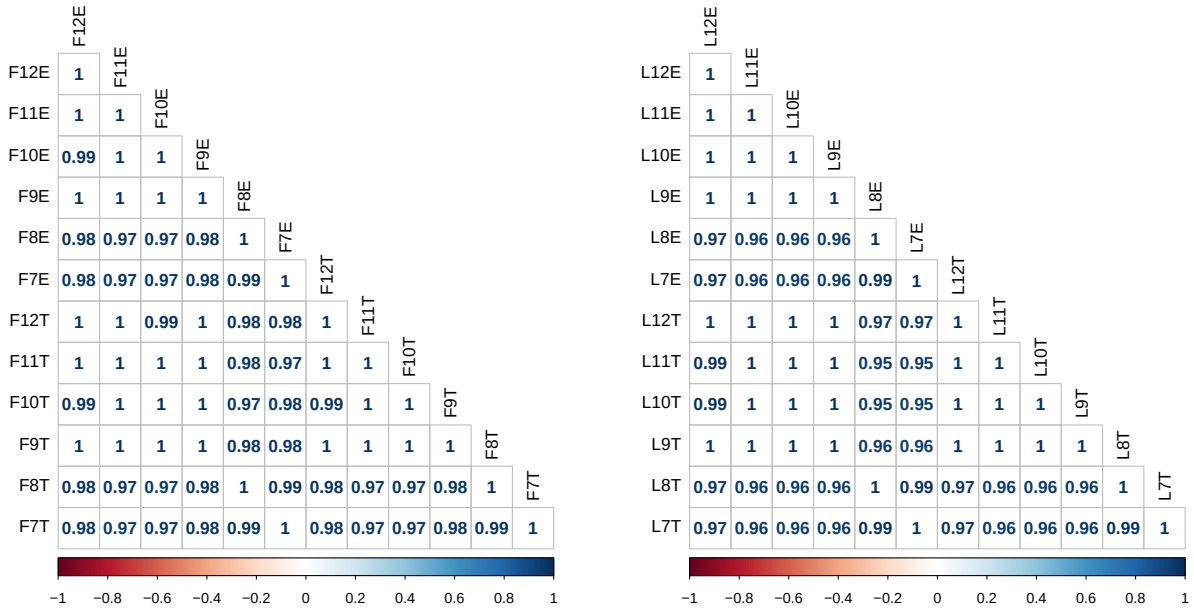
Code 4.57: rowTraversal.java

```
int[][] var = new int[iter][iter];

for (int j = 0; j < iter; j++) {
 for (int k = 0; k < iter; k++) {
  var[k][j] = j + k;
 }
}
```

Code 4.58: columnTraversal.java

between the energy and execution time for both systems using the values of each code snippet from Section 4 on different JDKs. The results for the Intel Fog Node and the Laptop are shown in correlation matrix in Fig. 4.18a and 4.18b. E denotes the energy, T denotes the execution time, F denotes the Intel Fog Node and L denotes the Laptop. For the Intel Fog Node, the correlation for energy consumption is at least 0.97, whereas, for the Laptop, the correlation is 0.96. Execution time shows the same behavior which means that there is a high linear relation between the energy



(a) (b)

Figure 4.18: Energy and time correlation for different JDKs: (a) Intel Fog Node, and (b) Laptop.

and execution time. The high correlation between different JDKs energy consumption and time execution values shows that our results are stable for the different JDKs.

4.3 Threats to Validity

The results of our earlier work [62] can vary from this work due to the measurement set up difference and various Java command-line options. In this section, we go over various threats to the validity of this work. These threats can have a big impact on the results.

JVM type: JVM is offered in two flavors - server and client. These two types of JVM have different compilers that are appropriate to the performance needs of the client and server applications. JDK contains both types which can be enabled by specifying `-client` or `-server`. The server JVM is suited for running server applications which require the fastest possible operating speed instead of a fast start-up. Client JVM is specifically adjusted to execute less of the complex optimizations resulting in reduced application start-up time and smaller memory footprint. For our experiments, we use the server JVM. For most of the codes, using the client JVM will result in the same energy consumption as most of our codes are lightweight. However, for codes such as those

for `LinkedList` deletion, the server JVM is more energy-efficient.

Garbage Collector type: The type of garbage collector used to remove unused objects in heap memory can impact both processing time and energy consumption. Java offers four types of garbage collectors: serial (`-XX:+UseSerialGC`), parallel (`-XX:ParallelGCThreads`), concurrent mark sweep (`-XX:+UseConcMarkSweepGC`) and garbage first (`-XX:+UseG1GC`). The serial garbage collector uses a single thread and freezes all the application threads during garbage collection. The parallel garbage collector is the default garbage collector and works the same as the serial garbage collector but uses multiple threads for garbage collection. The concurrent mark sweep garbage collector minimizes the pauses during the garbage collection by performing the garbage collection concurrently with the application threads. The garbage first garbage collector is parallel, concurrent and compacts the free heap space as soon as it reclaims the memory. For our experiments, we use the default parallel garbage collector. Again, as most of our codes are lightweight, using any of the garbage collectors will result in almost the same energy consumption values for most of the codes. However, for codes like that for `LinkedList` deletion, the selection of best garbage collector is not only code dependent but also hardware dependent. For the Intel Fog Node, the concurrent mark sweep garbage collector and for the Laptop, the parallel garbage collector result in the lowest energy consumption values for the `LinkedList` deletion code.

JIT compilation type: JIT compilation helps improve the performance of Java applications at run time. Depending on the application performance need, JIT compilation can be changed to always used (`-Xcomp`), disabled (`-Xint`), or just on hot methods (`-Xmixed`). The last one is the default JIT compilation. An interpreter is used to execute the bytecode whenever JIT is disabled. As JIT produces faster execution of Java code, turning it off and on can impact the total energy consumption of the code. For our experiments, we use the default just on hot methods JIT compilation. Changing the compilation to `-Xint` results in higher energy consumption values for almost all codes in our experiments.

Table 4.1: A summary of the observations

Component	Observation
Variables	`int` is the most efficient primitive data type. Static variables consume up to 17,700% more energy compared to local variables. Access modifiers have no impact on a variable energy consumption. Scientific notation results in lower energy consumption of decimal numbers.
Operators	Modulus arithmetic operator consumes up to 1,620% more energy than other arithmetic operators. Putting the most common cases first in short circuit operators can save up to 87% energy. Normal and compound assignments have the same amount of energy consumption. Post-increment, pre-increment , post- decrement and pre-decrement operators consume same energy.
Control Statements	Ternary operator consumes up to 37% more energy than if-then-else statement. `int` is the most energy-efficient iteration variable in a `for` loop. `for`, `while` and `do-while` loop statements consume same energy. Enhanced `for` statement consumes the same energy as the `for` statement. Method termination expression can consume higher energy than the variable termination expression.
Exception	Try-catch block scope can change how it consumes energy. Executing a catch block results in higher energy consumption.
String	`StringBuilder append` method consumes up to 1,48,069% lower energy than `String` concatenation operator. `Integer toString`, and `String valueOf` consume same energy. `String compareTo` method consumes up to 33% more energy than `String equals` method.
Object	Wrapper classes object are more energy-expensive as compared to primitive data types. `Double` wrapper class object consume up to 115% more energy than `Integer` wrapper class object.
Thread	Creating `Threads` by implementing a `Runnable` interface or extending `Thread` class consume same energy.
Arrays	`System.arraycopy()` is the most energy-efficient way to copy Arrays. Two-dimensional Array column traversal result in up to 793% more energy.

4.4 Related Work

Application programmers were shown to be aware of software energy consumption problems in [42]. In this study, the researchers analyzed 300 questions and 550 answers from 800 users on StackOverflow. The questions asked regarding the energy consumption problems were found to be diverse, interesting and challenging. However, the researchers found the answers to these ques-

tions to be often flawed or vogue. In another study, thousands of energy related questions, posted on StackOverflow, are analyzed to understand the issues faced by software developers while developing energy-efficient code [67]. In this study, the software developers are found to be concerned about energy consumption issues when there were issues related to improper implementations, sensor and radio utilization. A survey of more than 100 programmers has shown that software developers have limited knowledge about how software consumes energy and how the energy consumption among software can be reduced [68]. In their study, more than 80% of the programmers didn't consider the energy consumption of the software while developing it. Even interesting is that only 3% of the programmers received a complaint about the software energy consumption of the software they developed. It tells that even the users of the software are not aware of the energy consumption of a software. The reason for such results can be that the demand for energy-efficient computing is not a part of the education, knowledge, and training of programmers. Software energy efficiency research challenges are discussed in [69]. Some of these challenges include characterizing software behavior and energy consumption relationship, measurement of energy consumption of highly heterogeneous systems, insufficient generalized principles and lack of a unified approach.

The energy consumption of sorting algorithms in embedded and mobile environments was examined in [70]. No correlation was found between the time complexity of the sorting algorithms and their energy consumption. However, the sorting algorithms for memory requirements were found related to energy consumption. Quality contracts that express dependencies between software and hardware components for energy efficiency of software systems were used in [71, 72]. The impact of languages (C/C++/Java/Python), compiler optimization (GNU C/C++ compiler with O1, O2, and O3 flags), and implementation choices (e.g. using malloc instead of new to create dynamic arrays and using vector vs. array for Quicksort) on Fast Fourier Transform, Linked List Insertion/Deletion, and Quicksort was examined in [73]. The analysis found that improvement in serial code performance resulted in better energy efficiency, choice of programming languages was found to impact the energy use of program, and compiler options and data structure found to significantly impact the energy consumption of program. Java thread management constructs

- explicit thread creation, fixed-size thread pooling, and work stealing - relation to energy consumption was explored in [74]. In this study, energy consumption is found related to 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. Energy-efficient multithreaded program runtimes are shown to save 11-12% of energy in [75]. 6 commonly-used refactoring techniques have shown to impact the energy consumption of 197 applications in [76]. Refactoring not only just impact the energy consumption but also result in increase and decrease in the amount of energy consumption of the applications. The change in the energy efficiency of software by using different classes that implement the same interface was investigated in [77]. The study showed that the use of dynamic data structures in software applications leads to energy savings between 16.95% and 97.50%. The authors used machine learning tools to select the right data structure for the right workload and modify the classes for adaptive green data structures which resulted in better energy efficiency. Java collections were studied in terms of energy efficiency in [78], [79] and [80]. In those studies, the selection of a wrong collection type led to 300% more energy consumption than the most efficient type. Intelligent light-weight scheduler, Neurosurgeon, shown to reduce mobile energy consumption by 59.5% in [81]. Haskell, a pure functional language, is shown to save 60% energy while changing one data sharing primitive to another [82]. In this study, the relationship between energy efficiency and performance was not clear. Advanced software engineering methods like Dynamic Software Product Lines (DSPLs) [83] and Aspect-Oriented Software Development [84] are utilized to develop self-adaptive applications for energy efficiency in [85]. Performance-based guidelines have shown a saving of battery life of mobile applications for up to an hour in [86]. Object Oriented Programming (OOP) features and design patterns impact on software is investigated in [87]. In this study, inheritance was found to have no impact on energy efficiency and virtual functions, dynamic binning, overloading resulted in lower energy efficiency. For design patterns, impact of façade, prototype, and template method design patterns had no effect on energy efficiency, flyweight design pattern resulted in better energy efficiency and decorator design pattern resulted in lower energy efficiency. In one of our earlier work [62], we investigated Java's

energy consumption on data types, access modifiers, arithmetic operators, compound assignment, conditional operators, loop statements, exception, threads and all the classes in Java Collections Framework in terms of energy efficiency. In this chapter, we perform a rigorous analysis of data types, operators, control statements, String, exceptions, objects, and Arrays in terms of energy efficiency. None of the works described above has performed an exhaustive analysis of the Java components like us for energy consumption analysis.

4.5 Summary

Software energy efficiency research still has a long way to go as most of the energy efficiency research concentrates on hardware. Software developers have been oblivious to application energy efficiency for years. They do not pay much attention to the energy consumption of the software they develop. There are no well-established guidelines that they can follow to write energy-efficient code. We hope that this work will stimulate the development of such guidelines for developing energy-efficient software. We measure and compare the energy efficiency of data types, operators, control statements, String, exceptions, objects, and Arrays in Java as summarized in Table 4.1. We find a strong linear relationship between energy consumption and execution time for different JDKs. We hope that these results will help software developers to build more energy-efficient Java applications in the future.

CHAPTER 5 ENERGY CONSUMPTION ANALYSIS OF JAVA COMMAND-LINE OPTIONS

Java has different command-line options that can be used to tune the JVM. These options can significantly affect the energy behavior of Java applications. However, there is no study characterizing the energy behavior of these command-line options. We evaluate the active energy consumption of SPECjvm2008 benchmarks using different JDKs (Open and Oracle) and Java command-line options. The Java command-line options include `client`, `server`, `Xbatch`, `Xcomp`, `Xfuture`, `Xint`, `Xmixed`, `Xrs`, `AggressiveOpts`, `AggressiveHeap`, `Inline`, `AlwaysPreTouch`, `Xnoclassgc`, `UseSerialGC`, `UseParallelGC`, `UseConcMarkSweepGC`, and `UseG1GC`. This work is published in [88].

SPECjvm2008 consists of 11 benchmarks which are split into sub-benchmarks as shown in Table 5.1. Compiler benchmark has two sub-benchmarks - `compiler.compiler` and `compiler.sunflow`. `compiler.compiler` compiles javac itself. `compiler.sunflow` compiles the `sunflow` sub-benchmark from SPECjvm2008. This benchmark has its own File-Manger to manage memory. `compress` benchmark uses a modified Lempel-Ziv method to compress data. It is deterministic as it first finds common substrings and then replaces them with a variable size code. This benchmark is ported from 129.compress benchmark from CPU95, however, it is modified to compress real data from files instead of compressing synthetically generated data. Crypto benchmark consists of three sub-benchmarks - `crypto.aes`, `crypto.rsa` and `crypto.signverify` - which focuses on different areas of crypto. `crypto.aes` performs encryption and decryption using the AES and DES protocols with an input data of size 100 bytes and 713 KB. `crypto.rsa` performs encryption and decryption using the RSA protocol with an input data of size 100 bytes and 16 KB. `crypto.signverify` sign and verify using MD5withRSA, SHA1withRSA, SHA1withDSA and SHA256withRSA protocols with an input data size of 1 KB, 65 KB, and 1 MB. `derby` benchmark focuses on BigDecimal computations and database logic using an open-source database written in pure Java. `MPEGaudio` benchmark

utilizes JLayer, an LGPL mp3 library, for mp3 decoding and is floating-point heavy. Scimark benchmark is a floating point benchmark which is consist of five sub-benchmarks - `fft`, `lu`, `sor`, `sparse`, and `monte_carlo`. Each sub-benchmark has two versions with different dataset size, except `monte_carlo` (as it uses only scalars). The large dataset has a size of 32MB for stressing the memory whereas the small dataset has a size of 512 KB to stress the JVM. `serial` benchmark utilizes data from the JBoss benchmark to serialize and deserialize primitives and objects. `sunflow` benchmark utilizes half the number of hardware threads to test graphics visualization. Each of the hardware thread results in four internal threads inside the benchmark. XML benchmark has two sub benchmarks - `xml.transform` and `xml.validation`. `xml.transform` stresses the JRE's implementation of javax.xml.transform by applying style sheets to XML documents. `xml.validation` stresses the JRE's implementation of javax.xml.validation by validating XML instance documents against XML schemata. `startup` benchmark starts each of the above-discussed benchmarks for one operation. For every benchmark run, a new JVM is launched and time is measured from starting the JVM to finishing off the benchmark iteration. SPECjvm2008 has two run categories - Base and Peak. Base category run doesn't allow the tuning of the JVM. Therefore, in this work, we utilize the Peak category as we evaluate various command-line options to tune the JVM. Except for `startup`, each benchmark goes through one iteration in which several operations (each invocation of a benchmark is one operation) are executed for certain duration, by defaults 240 seconds. Each iteration finishes at least 5 operations. The duration of an iteration is never less than the specified time, however, it increases if at least five operations are not executed within the specified duration of time. For this work, we utilize the default duration of the iteration. The warmup is skipped as it is not possible to remove the warmup energy from the total energy of a benchmark run.

5.1 Energy Consumption Analysis

In this section, we evaluate the energy consumption of different Java command-line options. For better accuracy, we measure the total energy consumption of each command-line option ten times. We then check for outliers in those ten measurements using Tukey's method. We replace

Table 5.1: SPECjvm2008 benchmarks

Benchmarks	Sub-Benchmarks
Compiler	compiler.compiler, compiler.sunflow
Compress	compress
Crypto	crypto.aes, crypto.rsa, crypto.signverify
Derby	derby
MPEGaudio	mpegaudio
Scimark.X.large Scimark.X.small	scimark.fft.large, scimark.lu.large, scimark.sor.large,
	scimark.sparse.large, scimark.fft.small, scimark.lu.small,
	scimark.sor.small, scimark.sparse.small, scimark.monte_carlo
Serial	serial
Sunflow	sunflow
XML	xml.transform, xml.validation
Startup	startup.helloworld, startup.compiler.compiler, startup.compiler.sunflow,
	startup.compress, startup.crypto.aes, startup.crypto.rsa,
	startup.crypto.signverify, startup.mpegaudio, startup.scimark.fft,
	startup.scimark.lu, startup.scimark.monte_carlo, startup.scimark.sor,
	startup.scimark.sparse, startup.serial, startup.sunflow,
	startup.xml.transform, startup.xml.validation

the outliers measurements with new measurements and again check for outliers. We repeat this process until no outlier is left. The reason for outlier removal is two-fold: (i) the active `systemd` units or the garbage collector does not interfere with the active energy measurements, and (ii) the variation in measurements remain low. Next, we subtract the idle energy from the total energy consumption to determine the active energy of each benchmark. We then calculate the mean of all the ten observations to determine the total energy consumption and the execution time of each benchmark. In cases where the means are close, we use the independent sample t-test (two means) or one-way ANOVA test (more than two means) to determine whether the means are the same. For both tests, we consider alpha value as 0.05 which means that if the p-value is smaller than or equal to 0.05, we reject the null hypothesis that the mean values are not the same. If the p-value is greater than 0.05, it means that the mean values are the same. If the means are the same, we conclude that there is no difference in energy consumption. We log the package, and the core energy but only present the package energy consumption values as the core energy measurements values are negligible compare to those of the package. `compiler` benchmark is not shown in

Table 5.2: Energy consumption for client option

Benchmark	client							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10409.24	243.06	2535.53	244.24	10396.71	242.90	2537.34	244.81
crypto								
crypto.aes	10314.78	246.68	2596.89	250.46	10377.92	246.22	2599.21	248.59
crypto.rsa	10074.29	242.02	2439.42	243.55	9623.52	241.80	2346.07	242.31
crypto.signverify	9715.88	241.96	2469.04	243.58	9700.36	241.76	2446.63	243.17
derby	10916.68	259.56	2600.44	422.25	10953.10	259.51	2606.55	421.16
mpegaudio	10464.72	243.77	2517.06	245.30	10450.29	243.43	2511.49	246.42
scimark								
fft.small	11456.44	241.94	2689.68	242.97	11447.94	241.81	2684	243.46
lu.small	14771.78	241.61	2854.29	243.27	14645.49	241.54	2857.34	242.95
monte_carlo	9982.14	242.52	2367.98	244.66	9947.21	242.21	2348.31	244.54
sor.large	7639.63	247.87	2491.90	257.20	7580.28	252.00	2459.64	254.95
sor.small	8166.53	242.57	2066.18	244.06	8123.92	243.06	2063.16	244.18
sparse.large	6130.27	264.51	2695.02	253.13	6148.83	251.79	2630.65	255.68
sparse.small	9754.74	243.24	2845.30	245.05	11143.90	242.62	2793.65	243.7
serial	11149.01	243.58	2573.32	247.05	11198.22	243.47	2568.53	246.29
sunflow	10111.17	242.71	2587.03	245.25	10115.83	243.43	2581.64	243.38
xml								
xml.transform	11362.00	254.01	2748.23	267.81	11356.21	253.84	2753.63	267.16
xml.validation	11742.37	241.67	2535.31	243.22	11741.60	241.65	2536.2	242.89
startup								
compress	32.81	1.66	23.41	3.80	32.22	1.70	22.78	3.04
crypto.aes	48.27	2.67	38.13	5.21	50.41	2.80	40.87	5.57
crypto.rsa	28.68	1.26	21.63	2.50	25.90	1.05	20.19	2.33
crypto.signverify	28.62	1.30	21.7	2.61	25.92	1.27	20.17	2.47
mpegaudio	51.49	2.04	38.82	4.78	53.26	2.12	39.52	4.62
scimark.fft	26.43	1.26	19.92	2.43	24.78	1.29	19.37	2.48
scimark.lu	22.64	0.7	18.33	2.27	21.80	1.00	17.77	2.30
scimark.monte_carlo	32.91	1.89	24.96	3.42	32.24	1.90	24.28	3.38
scimark.sor	31.05	1.87	23.11	3.29	30.18	1.93	22.51	3.21
scimark.sparse	30.32	1.54	23.50	3.07	29.67	1.53	23.06	2.97
serial	56.27	2.20	43.66	5.20	56.34	2.23	44.91	5.20
sunflow	54.51	1.80	42.13	4.46	54.92	1.81	42.46	4.40
xml.transform	264.16	13.33	207.03	27.06	269.16	13.52	211.76	27.51
xml.validation	48.94	1.62	41.22	4.43	48.88	1.61	42.28	4.53

any of the result tables as it is not supported by Java SE 8. Also, `fft.large` and `lu.large benchmark` results are not shown as they abort when run on both systems. Each table in the next subsections of this chapter represents the total active energy consumption and the execution time of SPECjvm2008 benchmarks. For each table, Open and Oracle represent the different JDKs. Under each JDK we have the two ICT systems - IFN and Laptop. Under each ICT system is the measurement of energy consumption (E) and execution time (T). `server` command-line option refers to default mode as both systems use server JVM by default.

Table 5.3: Energy consumption for server option

Benchmark	server							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10446.44	242.68	2533.75	244.61	10409.98	242.7	2532.61	244.36
crypto								
crypto.aes	10368.44	246.26	2599.44	250.21	10366.61	246.16	2597.92	247.87
crypto.rsa	10125.59	242.22	2440.72	243.51	9636.60	241.82	2345.5	242.41
crypto.signverify	9740.42	242.04	2458.03	243.66	9725.98	241.76	2452.33	243.24
derby	10948.96	258.71	2603.97	430.39	10949.42	259.49	2603.04	422.69
mpegaudio	10474.17	243.76	2519.12	245.18	10443.40	243.68	2516.17	245.40
scimark								
fft.small	11474.26	241.95	2684.87	243.51	11430.56	241.91	2683.26	243.50
lu.small	14698.87	241.61	2852.07	242.88	14697.20	241.61	2856.52	242.68
monte_carlo	9974.18	242.57	2365.7	244.86	9949.61	242.27	2349.11	244.32
sor.large	7591.76	249.50	2469.21	250.93	7567.83	248.40	2462.37	251.50
sor.small	8158.51	243.03	2063.81	244.01	8123.63	243.15	2060.24	243.63
sparse.large	6073.82	262.46	2658.77	250.84	5965.71	255.23	2635.06	261.88
sparse.small	9739.27	242.70	2821.39	244.70	11109.87	242.70	2786.7	244.95
serial	11157.09	243.07	2572.46	246.17	11168.14	243.15	2570.49	245.76
sunflow	10134.22	243.56	2581.23	244.03	10113.49	242.89	2579.16	243.59
xml								
xml.transform	11359.57	253.98	2740.60	266.96	11355.62	253.96	2744.93	267.07
xml.validation	11738.44	241.72	2532.73	242.67	11735.22	241.66	2529.64	242.94
startup								
compress	32.56	1.68	23.51	3.01	32.00	1.63	22.84	3.01
crypto.aes	48.46	2.62	38.44	5.09	50.32	2.74	40.92	5.36
crypto.rsa	29.38	1.27	21.78	2.64	25.72	1.06	20.47	2.42
crypto.signverify	28.24	1.28	22.05	2.68	25.77	1.24	20.23	2.52
mpegaudio	52.42	2.10	37.7	4.20	52.09	2.17	39.61	4.63
scimark.fft	26.17	1.25	19.73	2.56	24.73	1.26	19.43	2.49
scimark.lu	22.69	0.7	18.47	2.25	21.90	1.06	17.87	2.20
scimark.monte_carlo	33.21	1.92	24.70	3.38	32.11	1.90	24.22	3.37
scimark.sor	31.36	1.94	23.07	3.16	29.98	1.94	22.5	3.24
scimark.sparse	31.05	1.50	23.58	3.14	29.71	1.48	22.97	2.97
serial	56.24	2.13	44.22	4.98	56.70	2.28	44.67	5.07
sunflow	54.29	1.83	42.11	4.17	53.77	1.83	42.10	4.50
xml.transform	265.36	13.31	209.15	26.45	268.01	13.34	211.96	26.81
xml.validation	49.18	1.58	41.63	4.40	48.23	1.55	41.82	4.68

5.1.1 `-client` and `-server`

The JDK supports two type of JVM - `client` and `server`. These two JVMs have the same runtime environment code base, however, they use a different type of compilers. `client` JVM compiler offers lesser optimization, which results in faster compiling for short-running applications. `server` JVM offers an advance adaptive compiler, which supports complex optimization for maximizing peak operating speed of long-running applications. We compare the energy consumption of these two options in Table 5.2 and 5.3. For the `client`, we can see that 20 benchmarks on the IFN and 19 benchmarks on the Laptop have lower energy consumption when executed on Oracle JDK instead of Open JDK. For the `server`, these number jump to 25 and

Table 5.4: Energy consumption for Xbatch option

Benchmark	Xbatch							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10446.58	244.73	2539.76	248.89	10434.53	244.26	2531.36	247.81
crypto								
crypto.aes	10357.00	247.71	2564.25	252.54	10334.18	247.00	2604.10	256.38
crypto.rsa	9963.95	244.33	2441.61	248.16	9611.29	243.56	2335.73	247.88
crypto.signverify	9678.07	243.68	2460.32	247.92	9709.19	243.37	2452.40	248.01
derby	10900.72	264.74	2590.83	449.60	10909.60	265.00	2603.77	443.09
mpegaudio	10474.96	244.82	2516.81	250.02	10462.94	244.51	2507.13	248.26
scimark								
fft.small	11481.31	243.83	2687.80	247.76	11397.42	243.72	2688.03	247.19
lu.small	14771.46	243.39	2861.04	247.37	14698.83	243.27	2855.71	247.43
monte_carlo	9965.10	244.19	2356.34	249.05	9955.24	243.79	2376.20	247.84
sor.large	7581.93	248.75	2519.88	259.17	7601.54	251.06	2488.45	258.56
sor.small	8140.68	244.62	2064.47	248.15	8128.72	243.94	2061.62	247.44
sparse.large	6184.37	263.10	2672.73	270.11	6121.03	260.16	2658.01	264.90
sparse.small	9855.46	243.94	2849.72	249.40	11056.37	244.24	2814.7	247.37
serial	11142.34	244.74	2578.36	250.48	11163.01	244.68	2568.80	250.58
sunflow	10126.31	244.62	2576.64	248.32	10115.35	243.67	2572.80	248.19
xml								
xml.transform	11379.61	263.44	2776.10	296.92	11370.77	264.01	2783.25	298.86
xml.validation	11715.73	243.92	2550.41	248.66	11702.98	243.34	2543.29	248.30
startup								
compress	36.51	2.54	26.25	6.47	35.83	2.48	24.84	5.87
crypto.aes	50.91	3.65	41.12	8.79	53.47	3.72	42.60	8.77
crypto.rsa	32.90	2.27	24.76	6.48	29.58	2.24	22.53	5.30
crypto.signverify	32.89	2.30	24.41	6.28	29.33	2.23	22.20	5.52
mpegaudio	54.97	3.18	41.40	8.22	55.72	2.98	41.44	7.32
scimark.fft	29.91	2.22	22.36	5.90	27.83	2.15	21.34	5.35
scimark.lu	26.53	1.98	20.58	5.97	25.38	1.87	19.76	5.35
scimark.monte_carlo	35.59	2.93	27.27	6.98	34.80	2.91	26.30	6.54
scimark.sor	33.77	2.88	25.30	6.82	32.90	2.79	24.43	6.09
scimark.sparse	34.01	2.68	25.93	6.81	32.50	2.38	24.77	6.13
serial	59.89	3.21	46.05	8.51	59.29	3.06	48.90	8.70
sunflow	57.07	2.68	44.22	7.77	57.42	2.71	44.54	7.34
xml.transform	265.59	14.34	212.21	32.17	270.47	14.34	214.65	30.17
xml.validation	51.67	2.57	44.68	8.16	52.48	2.62	44.51	7.51

24 benchmarks. For the IFN, 18 benchmarks consume more energy for `server` option while using Open JDK. Using Oracle JDK instead causes `client` option to consume more energy for 23 benchmarks. For the Laptop, 18 and 17 benchmarks consume lesser energy for `server` option while using Open and Oracle JDK, respectively.

Two benchmarks - `crypto.rsa` and `sparse.small` - stands out with large variation in energy consumption for different JDK types on the IFN. `sparse.small` not only shows the highest variation on the IFN but also shows higher energy efficiency using Open JDK. For the Laptop, `crypto.rsa` shows the highest variation for different JDK versions, however, `sparse.small` doesn't show the same behavior as on the IFN. For both systems, sparse results in higher energy

Table 5.5: Energy consumption for Xcomp option

Benchmark	Xcomp							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10611.87	262.40	2606.01	292.46	10483.43	259.89	2614.30	293.03
crypto								
crypto.aes	10525.60	265.32	2735.97	298.14	10583.84	263.58	2748.83	304.45
crypto.rsa	10393.64	261.92	2563.34	289.22	9874.67	261.46	2359.65	284.86
crypto.signverify	9932.05	256.80	2587.29	290.16	9864.95	258.57	2585.08	294.11
derby	11302.56	295.97	2677.06	556.63	11322.94	294.7	2706.95	553.45
mpegaudio	10378.80	260.51	2556.23	285.86	10493.28	261.15	2553.31	287.57
scimark								
fft.small	12301.04	260.72	2786.22	287.36	12034.09	260.44	2804.43	293.74
lu.small	14689.45	261.11	2965.53	289.95	14720.60	258.77	2973.35	292.54
monte_carlo	10122.55	259.9	2564.05	291.25	10109.62	262.09	2554.14	295.78
sor.large	7811.18	265.49	2580.44	301.82	7699.60	272.76	2598.38	305.03
sor.small	8302.24	259.26	2183.70	291.01	8302.74	256.19	2179.22	294.08
sparse.large	6031.19	269.54	2795.32	313.87	6039.10	276.13	2781.01	314.29
sparse.small	9810.80	263.58	2940.89	292.31	11103.12	259.24	2969.39	294.35
serial	11170.7	258.73	2624.39	288.52	11202.40	263.36	2636.49	292.11
sunflow	10470.78	262.77	2710.41	291.93	10467.30	263.32	2704.47	295.31
xml								
xml.transform	11698.98	292.24	2898.32	356.25	11724.08	288.04	2877.7	364.46
xml.validation	11720.61	263.75	2527.70	303.53	11726.44	263.51	2550.24	305.47
startup								
compress	173.49	16.23	132.53	48.40	189.49	18.80	141.66	51.57
crypto.aes	181.54	21.06	146.91	50.23	199.52	20.13	158.88	54.68
crypto.rsa	161.61	18.33	130.52	48.21	177.62	23.06	139.68	52.15
crypto.signverify	162.42	15.96	129.90	47.12	178.91	17.45	139.40	51.49
mpegaudio	189.38	17.48	147.01	50.06	204.50	16.61	157.37	54.91
scimark.fft	162.67	16.10	128.35	48.38	166.74	22.05	137.50	52.20
scimark.lu	143.20	21.20	126.76	47.66	171.23	18.35	135.88	51.55
scimark.monte_carlo	166.19	16.51	133.11	49.18	179.74	18.18	142.60	51.80
scimark.sor	169.31	16.44	131.15	48.34	177.95	20.85	140.72	52.27
scimark.sparse	161.31	14.24	131.80	50.05	177.07	18.25	141.14	52.59
serial	190.74	20.27	152.52	51.73	202.96	18.60	162.44	54.47
sunflow	192.46	18.42	149.84	50.89	199.74	18.18	160.82	54.13
xml.transform	392.74	27.69	314.66	72.65	406.46	29.74	330.58	76.82
xml.validation	178.5	22.72	150.06	51.53	198.12	20.34	161.27	53.48

consumption for the smaller dataset instead of the larger dataset. Most of the command-line options that we are going to discuss next show the same behavior for energy consumption variation of different benchmarks.

5.1.2 `-Xbatch`, `-Xcomp`, `-Xint`, `-Xfuture`, and `-Xmixed`

JVM runs a method in interpreted mode until the background compilation is finished. `Xbatch` option disables this background compilation and runs the compilation in the foreground. As shown in Table 5.4, for the `Xbatch` option, Oracle JDK results in better energy efficiency for 21 benchmarks, for both systems. `Xbatch` also results in the lower energy efficiency on both systems for at least 21 benchmarks as compared to the default mode for both JDKs.

Table 5.6: Energy consumption for Xint option

Benchmark	Xint							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10262.54	305.11	3582.90	344.55	10448.47	300.76	3594.33	331.08
crypto								
crypto.aes	14236.14	371.68	5906.80	585.48	13826.80	353.63	5856.40	611.91
crypto.rsa	11185.43	280.84	3321.55	326.19	10255.85	268.66	3215.92	310.36
crypto.signverify	10191.72	299.46	3774.37	376.81	10865.56	313.89	3695.84	360.63
derby	13474.05	517.50	5815.05	946.75	13401.37	514.11	5932.39	953.83
mpegaudio	11605.41	283.70	3288.36	326.60	11439.94	282.00	3247.07	319.73
scimark								
fft.small	11358.65	255.44	2825.84	272.91	11261.74	252.42	2780.51	278.73
lu.small	10467.14	268.97	2925.48	284.71	10462.62	266.74	2869.83	277.54
monte_carlo	12170.18	291.06	4291.29	417.89	12357.51	315.08	4067.55	393.57
sor.large	13097.32	334.18	3851.68	364.24	13065.95	333.69	3837.35	363.64
sor.small	10237.10	265.26	3019.37	284.22	10199.15	263.83	2806.69	268.06
sparse.large	13592.60	347.62	4016.89	401.54	13457.79	344.92	4004.76	389.23
sparse.small	10268.71	261.65	3172.02	300.26	10228.77	261.84	3150.17	294.87
serial	12284.70	297.22	4011.33	376.96	11750.90	298.43	4084.54	384.48
sunflow	12701.02	290.03	3103.53	286.03	12706.04	290.76	3154.03	291.20
xml								
xml.transform	12082.03	317.74	3823.75	391.98	11668.35	328.32	3890.64	397.48
xml.validation	10913.75	268.58	2942.64	276.54	10362.28	270.72	2959.56	280.39
startup								
compress	39.25	2.52	30.51	4.74	37.79	2.37	29.63	4.73
crypto.aes	54.69	3.50	45.58	6.72	56.09	3.58	47.37	6.97
crypto.rsa	35.48	2.09	29.52	4.47	31.95	1.81	27.00	3.86
crypto.signverify	34.74	2.21	29.57	4.59	31.77	2.00	27.01	4.12
mpegaudio	57.88	2.97	46.05	6.43	58.12	2.86	46.30	6.34
scimark.fft	32.77	2.10	27.06	4.36	30.61	1.93	26.03	4.13
scimark.lu	28.98	1.83	25.66	3.90	27.86	1.74	24.50	3.82
scimark.monte_carlo	39.39	2.76	32.05	5.26	37.35	2.62	31.00	5.01
scimark.sor	37.54	2.74	29.60	4.92	35.80	2.62	29.17	4.75
scimark.sparse	36.76	2.41	30.52	4.71	35.52	2.25	29.50	4.61
serial	61.13	3.01	50.56	6.79	62.09	2.91	51.76	6.84
sunflow	59.39	2.61	49.14	6.12	59.49	2.56	48.41	6.04
xml.transform	268.57	14.30	214.08	28.14	271.84	14.30	219.26	28.58
xml.validation	54.83	2.42	48.09	5.91	54.32	2.38	48.65	6.19

`Xcomp` forces the compilation of a method on the first invocation instead of doing that after a set threshold of interpreted method invocations. As shown in Table 5.5, for the `Xcomp` option, Open JDK results in better energy efficiency for 24 benchmarks on the IFN and 23 benchmarks on the Laptop. For both systems, `Xcomp` results in higher energy consumption of at least 27 benchmarks than the default mode for both JDKs.

`Xint` causes the JVM to run in interpreted-only mode. This option disables the just-in-time compilation, resulting in a considerable slow down in execution. As shown in Table 5.6, for 22 benchmarks on the IFN and 20 benchmarks on the Laptop, Oracle JDK results in higher energy efficiency than Open JDK. Open JDK results in up to 9% increase in energy consumption. For

Table 5.7: Energy consumption for Xfuture option

Benchmark	Xfuture							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10394.09	243.05	2536.62	245.07	10369.21	243.15	2533.96	245.00
crypto								
crypto.aes	10346.48	245.14	2625.90	249.74	10299.94	247.55	2601.46	252.78
crypto.rsa	10056.32	242.21	2441.36	243.33	9639.41	241.92	2347.92	242.78
crypto.signverify	9705.03	242.17	2464.92	244.15	9727.14	241.95	2449.93	243.47
derby	10959.62	258.34	2608.21	435.27	10946.01	258.81	2612.46	424.62
mpegaudio	10470.29	243.85	2522.02	245.86	10457.07	243.74	2519.61	246.65
scimark								
fft.small	11424.82	241.96	2685.79	243.83	11419.80	241.94	2685.09	243.56
lu.small	14679.49	241.77	2858.47	243.37	14681.57	241.72	2855.75	242.78
monte_carlo	9970.07	242.46	2359.31	245.06	9960.79	242.47	2352.46	244.16
sor.large	7617.49	251.00	2485.74	251.57	7563.82	247.45	2473.90	253.00
sor.small	8155.82	243.37	2064.69	244.29	8128.50	243.19	2063.80	244.14
sparse.large	6132.48	250.70	2674.36	255.54	6043.13	256.17	2649.74	259.37
sparse.small	9775.73	242.90	2807.79	245.59	11050.43	242.80	2794.86	245.47
serial	11139.7	243.57	2569.92	245.65	11173.81	243.74	2565.62	246.15
sunflow	10106.49	242.64	2575.90	244.70	10112.79	243.27	2583.61	243.16
xml								
xml.transform	11356.15	254.01	2748.04	267.33	11365.67	254.20	2751.01	267.05
xml.validation	11714.47	241.97	2527.02	243.28	11733.48	241.88	2531.30	243.10
startup								
compress	34.33	1.79	24.60	3.33	33.70	1.84	24.19	3.23
crypto.aes	49.90	2.81	39.11	5.41	52.69	2.95	42.37	5.85
crypto.rsa	30.12	1.36	23.15	2.80	27.17	1.21	21.51	2.61
crypto.signverify	29.58	1.45	22.98	2.89	27.36	1.39	21.70	2.83
mpegaudio	52.76	2.20	39.92	4.89	54.12	2.28	40.41	4.46
scimark.fft	27.70	1.38	21.18	2.81	26.50	1.35	20.57	2.76
scimark.lu	24.59	1.09	19.49	2.47	23.41	1.10	19.10	2.46
scimark.monte_carlo	34.81	2.06	26.11	3.64	33.38	2.06	25.68	3.64
scimark.sor	32.53	2.00	24.07	3.40	31.59	2.04	23.80	3.44
scimark.sparse	31.86	1.68	24.53	3.26	31.27	1.70	24.18	3.25
serial	56.82	2.26	45.82	5.31	56.17	2.39	46.38	5.32
sunflow	54.95	1.83	43.29	4.52	55.89	1.93	43.68	4.83
xml.transform	265.90	13.44	210.44	27.57	269.91	13.55	213.29	27.82
xml.validation	50.60	1.74	42.17	4.52	50.44	1.84	42.87	4.55

both systems, `Xint` results in higher energy consumption of at least 27 benchmarks than the default mode for both JDKs. For `crypto.aes` and `derby`, `Xint` results in significant increase in energy consumption. `Xint` also causes different variation in energy consumption than the default mode for most benchmarks. `Xint` causes the highest energy consumption for most of the benchmarks with up to 125% increase in the energy than the default mode. Interestingly, `Xint` consumes up to 28% lesser energy than the default mode for `lu.small` benchmark.

`Xfuture` results in stricter class-file format checks. As shown in Table 5.7, for the `Xfuture` option, Oracle JDK results in higher energy efficiency as compared to Open JDK for 20 benchmarks on the IFN and 21 benchmarks on the Laptop. For the IFN, `Xfuture` results in the higher

Table 5.8: Energy consumption for Xmixed option

Benchmark	Xmixed							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10412.87	243.00	2535.40	244.94	10410.38	243.17	2528.90	244.03
crypto								
crypto.aes	10349.72	245.37	2604.51	249.28	10358.89	247.81	2618.19	249.28
crypto.rsa	10144.64	242.00	2445.81	243.75	9636.84	241.68	2344.38	242.40
crypto.signverify	9734.62	242.06	2463.22	243.28	9711.01	241.86	2452.12	242.87
derby	10911.19	258.93	2603.35	430.80	10940.34	259.07	2606.34	422.52
mpegaudio	10455.64	243.78	2516.12	246.16	10447.77	243.92	2513.7	245.13
scimark								
fft.small	11422.21	241.91	2690.15	243.73	11425.55	241.86	2684.52	242.86
lu.small	14705.27	241.57	2856.53	243.14	14695.62	241.60	2856.24	243.29
monte_carlo	9958.54	242.23	2363.33	244.79	9980.80	242.35	2349.21	245.10
sor.large	7596.7	246.39	2472.88	255.33	7603.19	248.96	2464.89	255.7
sor.small	8154.84	242.65	2063.56	243.90	8131.51	243.07	2061.87	243.88
sparse.large	6099.36	252.24	2650.87	267.47	6016.54	249.78	2662.98	256.73
sparse.small	9776.10	242.83	2814.26	245.49	11016.73	242.85	2792.74	244.48
serial	11133.65	243.38	2572.63	247.06	11191.42	243.30	2564.04	244.58
sunflow	10123.55	242.98	2583.79	243.63	10118.13	242.62	2582.20	244.44
xml								
xml.transform	11365.7	253.69	2745.34	267.87	11363.58	254.01	2746.62	267.79
xml.validation	11751.53	241.81	2533.82	242.75	11735.05	241.84	2531.37	243.23
startup								
compress	32.70	1.67	23.60	3.07	31.7	1.66	22.83	2.7
crypto.aes	48.15	2.68	38.04	5.14	51.00	2.81	41.05	5.49
crypto.rsa	28.60	1.21	21.74	2.57	25.81	1.05	20.63	2.42
crypto.signverify	28.24	1.25	21.68	2.61	26.07	1.28	20.38	2.48
mpegaudio	51.11	2.08	38.58	4.27	52.81	2.18	38.47	4.49
scimark.fft	25.88	1.23	19.7	2.59	24.82	1.26	19.33	2.44
scimark.lu	22.58	1.00	18.36	2.19	22.03	1.00	17.87	2.32
scimark.monte_carlo	33.47	1.94	24.80	3.40	31.97	1.91	24.33	3.40
scimark.sor	31.23	1.88	23.31	3.24	29.95	1.91	22.46	3.24
scimark.sparse	30.70	1.55	23.54	3.05	29.7	1.57	22.75	2.95
serial	56.37	2.19	44.24	4.91	56.33	2.16	45.04	4.85
sunflow	54.75	1.80	42.24	4.42	54.01	1.79	42.14	4.39
xml.transform	263.83	13.54	207.74	27.72	270.76	13.55	213.17	27.62
xml.validation	48.31	1.59	41.36	4.44	48.52	1.64	41.13	4.41

energy efficiency of most benchmarks for both JDKs, except `startup` benchmark where the default mode is more energy-efficient. For the Laptop, the default mode results in the higher energy efficiency of most benchmarks for both JDKs, except `startup` benchmark where the default mode is more energy-efficient for each sub-benchmarks.

`Xmixed` option executes all bytecode except hot-methods in interpreter mode. Hot methods are those methods which are invoked very often. As shown in Table 5.8, for the `Xmixed` option, Oracle JDK results in the higher energy efficiency than Open JDK for 20 and 24 benchmarks on the IFN and the Laptop, respectively. For the IFN, `Xmixed` results in the better energy efficiency of 20 benchmarks than the default mode for Open JDK. Using Oracle JDK instead results in the

Table 5.9: Energy consumption for Xrs option

Benchmark	Xrs							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10442.01	243.00	2528.27	244.79	10429.60	242.80	2525.96	244.12
crypto								
crypto.aes	10405.48	246.46	2619.23	249.22	10346.92	246.43	2591.94	252.55
crypto.rsa	10035.57	242.07	2435.52	243.40	9641.12	241.86	2345.29	242.35
crypto.signverify	9694.70	242.12	2460.80	243.16	9728.37	241.57	2446.26	243.11
derby	10905.30	258.40	2601.79	435.10	10921.05	258.83	2598.22	425.60
mpegaudio	10460.76	243.83	2516.83	244.71	10465.81	243.67	2516.50	245.76
scimark								
fft.small	11439.55	242.01	2688.53	243.71	11444.42	241.86	2684.7	243.56
lu.small	14720.97	241.61	2856.01	242.66	14717.31	241.59	2855.5	242.7
monte_carlo	9992.65	242.31	2360.74	244.47	9957.83	242.25	2360.58	244.21
sor.large	7607.56	247.17	2481.50	250.70	7572.94	246.40	2460.91	257.59
sor.small	8150.23	242.76	2064.86	243.94	8124.78	243.18	2056.23	243.78
sparse.large	6037.07	256.66	2662.32	260.41	6132.02	263.25	2638.53	262.74
sparse.small	9838.93	242.72	2792.12	245.33	11053.89	242.84	2835.85	244.56
serial	11149.11	243.51	2574.58	246.27	11180.89	243.54	2564.95	245.16
sunflow	10099.54	242.45	2583.31	244.28	10119.73	242.88	2578.49	243.86
xml								
xml.transform	11357.36	253.87	2745.24	267.72	11376.51	254.09	2741.90	267.19
xml.validation	11742.74	241.74	2539.13	242.86	11755.26	241.63	2527.3	243.14
startup								
compress	32.61	1.62	23.28	3.05	32.10	1.68	22.88	30
crypto.aes	48.17	2.62	38.13	5.22	49.96	2.78	40.86	5.59
crypto.rsa	28.58	1.22	22.03	2.70	26.03	1.06	20.48	2.47
crypto.signverify	28.21	1.30	21.63	2.58	25.70	1.23	20.25	2.57
mpegaudio	51.77	2.11	38.34	4.40	52.71	2.12	38.98	4.68
scimark.fft	25.98	1.21	19.96	2.58	24.79	1.26	19.39	2.46
scimark.lu	22.30	0.96	18.50	2.29	21.98	1.02	17.77	2.34
scimark.monte_carlo	33.39	1.95	24.82	3.43	31.95	1.94	24.14	3.41
scimark.sor	31.01	1.90	23.05	3.26	30.17	1.95	22.49	3.21
scimark.sparse	31.04	1.55	23.76	3.11	29.77	1.58	22.82	2.7
serial	57.32	2.22	44.29	5.37	56.02	2.20	45.35	5.54
sunflow	54.91	1.73	41.60	4.37	54.45	1.88	43.25	4.59
xml.transform	264.13	13.33	208.24	26.85	270.02	13.48	212.29	27.02
xml.validation	48.44	1.54	41.75	4.50	48.75	1.65	41.80	4.62

lower energy efficiency of `Xmixed` option for 20 benchmarks. For the Laptop, `Xmixed` results in higher energy consumption than the default mode for 18 benchmarks for both JDKs.

5.1.3 `-Xrs`

`Xrs` option prevents JVM from using some of the operating system signals. In this option, an operating system handles any raised signal. Enabling this option can reduce JVM performance. Table 5.9 shows the energy consumption and execution time results for `-Xrs` option. Oracle JDK consumes lesser energy than Open JDK for 17 and 24 benchmarks, on the IFN and the Laptop, respectively. For the IFN, `Xrs` causes lower energy consumption for most of the benchmarks on Open JDK than the default mode but higher on Oracle JDK. For the Laptop, `Xrs` causes higher

Table 5.10: Energy consumption for AggressiveOpts option

Benchmark	AggressiveOpts							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10430.77	242.60	2540.69	244.55	10446.31	242.94	2525.14	245.29
crypto								
crypto.aes	10333.82	245.09	2615.09	251.18	10391.75	246.89	2596.48	252.30
crypto.rsa	10093.55	241.96	2439.61	243.30	9641.81	241.77	2342.88	242.55
crypto.signverify	9721.19	241.90	2474.61	243.24	9702.15	241.79	2445.39	243.41
derby	10942.01	258.77	2603.10	433.45	10936.14	258.83	2600.19	425.79
mpegaudio	10460.21	243.75	2520.20	245.51	10440.90	243.85	2516.63	244.85
scimark								
fft.small	11424.15	241.93	2693.13	243.67	11444.15	241.93	2688.13	243.45
lu.small	14758.23	241.68	2860.70	242.71	14728.42	241.59	2854.33	242.89
monte_carlo	9992.69	242.49	2364.50	245.65	9944.9	242.17	2357.29	244.30
sor.large	7669.47	251.96	2481.66	251.74	7595.13	250.37	2462.48	252.39
sor.small	8157.59	243.08	2061.68	244.17	8116.87	242.41	2060.24	243.88
sparse.large	6189.28	249.26	2653.89	262.56	6078.00	260.25	2663.21	255.50
sparse.small	9712.78	242.89	2814.82	244.20	10988.66	242.63	2784.59	245.83
serial	11130.45	243.82	2575.49	246.80	11187.19	243.50	2569.68	246.46
sunflow	10112.11	243.05	2577.39	244.31	10107.84	242.95	2573.23	244.67
xml								
xml.transform	11350.89	253.70	2744.11	266.43	11358.90	253.87	2743.37	267.37
xml.validation	11714.22	241.79	2557.16	243.02	11756.98	241.79	2551.59	242.81
startup								
compress	33.25	1.61	24.07	3.05	32.29	1.72	23.15	3.04
crypto.aes	48.42	2.61	38.52	5.21	50.86	2.80	40.93	5.58
crypto.rsa	28.81	1.25	22.16	2.65	26.19	1.11	20.70	2.43
crypto.signverify	28.94	1.35	22.26	2.74	26.28	1.26	20.63	2.59
mpegaudio	51.62	2.09	38.96	4.69	53.09	2.14	39.50	4.33
scimark.fft	26.29	1.25	20.15	2.54	25.21	1.26	19.65	2.59
scimark.lu	22.96	0.98	18.66	2.27	22.42	1.00	18.09	2.26
scimark.monte_carlo	33.56	1.95	25.34	3.48	32.18	1.93	24.44	3.46
scimark.sor	31.35	1.89	23.28	3.23	30.60	1.96	22.89	3.24
scimark.sparse	30.93	1.53	23.97	3.09	30.15	1.53	23.39	3.11
serial	56.84	2.18	43.86	5.06	56.77	2.29	46.49	5.26
sunflow	54.8	1.81	42.23	4.38	54.66	1.81	43.01	4.47
xml.transform	263.97	13.43	206.16	26.44	270.16	13.40	211.65	26.85
xml.validation	49.07	1.61	40.77	4.55	49.50	1.63	42.15	4.65

energy consumption for most of the benchmarks on Open JDK than the default mode but lower on Oracle JDK. This shows that the same JDK shows different behavior on different ICT systems.

5.1.4 `-XX:+AggressiveOpts` and `-XX:+AggressiveHeap`

`AggressiveOpts` option enables the use of aggressive performance optimization features. As shown in Table 5.10, for the `AggressiveOpts` option, Oracle JDK results in lower energy consumption of 20 benchmarks on the IFN and 24 benchmarks on the Laptop as compared to Open JDK. For the IFN, `AggressiveOpts` is more energy-efficient for 20 benchmarks than the default mode for Open JDK, however, lesser energy-efficient for 24 benchmarks for Oracle JDK. For the Laptop, `AggressiveOpts` results in higher energy consumption of at least 18

Table 5.11: Energy consumption for AggressiveHeap option

Benchmark	AggressiveHeap							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10407.54	242.97	2537.29	244.98	10416.00	242.89	2525.05	244.23
crypto								
crypto.aes	10249.63	245.43	2573.7	249.30	10235.92	246.02	2605.50	247.62
crypto.rsa	10073.65	242.20	2442.98	243.15	9688.7	241.94	2345.88	242.27
crypto.signverify	9753.18	242.17	2465.89	243.54	9750.24	241.98	2445.65	243.27
derby	10949.53	259.68	2613.11	433.98	10987.98	260.52	2609.35	418.84
mpegaudio	10440.66	244.21	2521.82	246.49	10463.16	242.98	2517.00	247.28
scimark								
fft.small	11571.20	242.02	2695.13	243.66	11564.80	242.03	2687.07	243.86
lu.small	14870.19	241.74	2861.60	242.92	14843.28	241.69	2860.90	243.09
monte_carlo	9978.09	242.58	2349.23	244.41	9944.95	242.30	2351.55	244.44
sor.large	7655.9	251.27	2497.60	256.06	7625.09	251.11	2473.34	252.77
sor.small	8149.71	242.72	2070.35	243.54	8136.85	242.75	2066.24	243.26
sparse.large	5976.62	252.19	2655.32	251.43	5887.32	253.05	2658.93	263.67
sparse.small	10062.75	242.45	2834.97	244.35	11301.00	242.78	2825.09	244.39
serial	11185.55	243.54	2617.61	246.13	11223.04	243.09	2612.01	247.04
sunflow	10206.38	242.65	2574.84	244.46	10196.53	242.75	2575.83	243.72
xml								
xml.transform	11402.89	254.13	2755.03	267.87	11410.07	254.23	2755.15	267.27
xml.validation	11672.28	241.76	2583.13	242.55	11695.09	241.66	2585.27	243.13
startup								
compress	32.81	1.67	23.47	3.09	32.12	1.65	22.68	2.92
crypto.aes	48.27	2.64	37.96	5.15	50.63	2.82	40.37	5.51
crypto.rsa	28.61	1.25	21.69	2.44	26.03	1.07	20.13	2.33
crypto.signverify	28.08	1.31	21.58	2.83	25.89	1.28	20.03	2.49
mpegaudio	51.74	2.07	37.80	4.52	52.89	2.11	38.96	4.67
scimark.fft	26.16	1.21	19.83	2.55	25.17	1.27	19.27	2.48
scimark.lu	22.91	1.01	18.12	2.25	22.04	1.01	17.72	2.17
scimark.monte_carlo	33.07	1.94	24.71	3.42	32.46	1.95	24.00	3.35
scimark.sor	31.20	1.90	22.89	3.16	30.29	1.92	22.44	3.18
scimark.sparse	30.74	1.50	23.40	3.03	30.10	1.58	22.79	3.05
serial	56.49	2.12	43.73	5.12	56.75	2.30	43.83	4.84
sunflow	54.88	1.72	41.66	4.29	54.49	1.84	41.91	4.32
xml.transform	263.7	13.33	206.61	26.53	270.64	13.52	212.68	26.86
xml.validation	48.45	1.60	40.65	4.56	48.56	1.59	41.20	4.20

benchmarks than the default server option for both JDKs.

`AggressiveHeap` option enables Java heap optimization which is optimal for long-running computation-intensive jobs. As shown in Table 5.11, for the `AggressiveHeap` option, Oracle JDK results in the higher energy efficiency of 19 benchmarks than Open JDK for both systems. For both systems, `AggressiveHeap` results in higher energy consumption of most of the benchmarks than the default mode for both JDKs, except `startup` benchmark where `AggressiveHeap` is more energy-efficient for both systems.

Table 5.12: Energy consumption for Inline Disable option

Benchmark	Inline Disable							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10434.79	243.65	3030.05	248.06	10414.35	243.21	3024.35	245.80
crypto								
crypto.aes	10378.38	246.17	3110.30	252.31	10379.53	246.73	3108.63	250.46
crypto.rsa	10072.85	242.20	2936.91	244.05	9655.23	241.90	2834.60	242.34
crypto.signverify	9726.55	242.06	2953.80	243.62	9698.98	241.88	2946.22	243.61
derby	10903.47	259.28	3445.68	429.97	10931.76	259.10	3450.74	431.14
mpegaudio	10456.94	243.84	3016.61	245.58	10453.94	243.85	3011.79	245.35
scimark								
fft.small	11431.71	242.01	3178.46	243.57	11446.43	242.03	3175.47	243.04
lu.small	14754.71	241.66	3344.78	242.82	14697.10	241.58	3346.30	243.01
monte_carlo	9970.15	242.39	2849.19	244.60	9967.81	242.28	2839.42	244.42
sor.large	7623.28	250.64	3001.62	252.86	7585.27	248.10	2976.80	251.93
sor.small	8164.23	243.13	2558.16	244.91	8136.93	242.93	2556.43	244.07
sparse.large	6131.77	264.39	3158.25	271.35	6111.49	259.71	3205.46	251.79
sparse.small	9727.94	242.76	3282.65	244.32	11026.88	242.80	3296.45	245.84
serial	11135.55	243.38	3070.38	245.93	11196.62	243.56	3064.13	246.26
sunflow	10113.05	242.35	3067.92	244.85	10131.76	243.26	3077.20	245.21
xml								
xml.transform	11368.50	254.15	3289.82	267.80	11381.00	254.00	3288.54	267.86
xml.validation	11718.17	241.78	3029.50	243.27	11753.22	241.74	3027.53	242.95
startup								
compress	32.63	1.65	30.15	3.10	32.22	1.65	29.17	3.02
crypto.aes	48.57	2.66	48.85	5.08	50.89	2.81	52.02	5.54
crypto.rsa	29.11	1.28	27.36	2.61	26.18	1.08	25.33	2.44
crypto.signverify	28.47	1.34	27.69	2.67	25.80	1.24	25.77	2.53
mpegaudio	52.50	2.14	48.33	4.59	52.56	2.08	48.25	4.74
scimark.fft	26.49	1.27	25.43	2.44	24.7	1.28	24.56	2.45
scimark.lu	22.93	1.03	23.14	2.29	22.07	1.03	22.44	2.31
scimark.monte_carlo	33.97	1.91	32.29	3.51	32.08	1.95	31.31	3.40
scimark.sor	31.69	1.90	29.92	3.26	30.26	1.92	29.18	3.24
scimark.sparse	30.83	1.53	30.25	3.13	30.03	1.58	28.96	3.03
serial	55.67	2.13	54.7	5.1	56.63	2.16	55.00	5.30
sunflow	54.24	1.76	50.91	4.37	54.84	1.82	51.81	4.65
xml.transform	262.27	13.27	262.49	27.26	269.94	13.55	268.42	27.45
xml.validation	49.04	1.58	50.49	4.48	48.96	1.65	50.68	4.70

5.1.5 `-XX:-Inline`

Inline option enables replacing of a function call with function body. It is by default enabled in JVM and can be disabled by `-XX:-Inline` option. Disabling inline results in higher energy consumption than the default mode for at least 19 benchmarks on both systems for Oracle JDK as shown in Table 5.12. Open JDK shows the opposite behavior for both systems. For JDKs, Oracle JDK is more energy-efficient for 21 benchmarks on the Laptop but for only 14 benchmarks on the IFN.

Table 5.13: Energy consumption for AlwaysPreTouch option

Benchmark	AlwaysPreTouch							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10379.10	243.22	2534.91	244.20	10423.53	242.91	2528.15	243.82
crypto								
crypto.aes	10367.45	246.44	2577.97	254.28	10346.94	246.94	2619.45	250.73
crypto.rsa	10042.52	241.94	2445.96	243.49	9650.14	241.89	2343.86	242.35
crypto.signverify	9678.86	242.15	2453.09	243.75	9695.43	241.82	2454.00	243.14
derby	10901.13	259.24	2595.76	426.66	10931.72	259.14	2606.50	421.26
mpegaudio	10440.50	243.69	2508.82	244.55	10469.7	243.91	2519.61	245.94
scimark								
fft.small	11390.29	242.07	2688.41	243.31	11415.37	242.07	2682.22	243.39
lu.small	14658.59	241.70	2852.86	243.30	14622.22	241.70	2853.19	243.01
monte_carlo	9966.89	242.38	2371.91	244.69	9965.64	242.52	2367.01	244.33
sor.large	7603.25	247.61	2476.25	255.64	7571.01	251.92	2468.06	253.98
sor.small	8118.3	243.08	2063.21	244.14	8111.08	243.14	2061.25	244.04
sparse.large	6189.71	257.42	2685.37	257.34	6032.74	253.41	2664.37	257.84
sparse.small	9756.30	242.97	2814.36	244.49	11154.84	242.89	2815.48	244.47
serial	11111.93	243.23	2574.10	245.96	11157.64	243.47	2567.58	245.94
sunflow	10079.48	243.45	2577.17	243.94	10079.77	243.43	2582.21	244.55
xml								
xml.transform	11298.66	253.89	2742.39	266.67	11345.92	254.02	2749.49	267.27
xml.validation	11670.95	241.84	2523.88	242.82	11739.68	241.72	2530.84	242.84
startup								
compress	34.11	1.79	23.34	2.7	33.53	1.72	22.75	2.96
crypto.aes	49.32	2.76	38.01	5.10	52.42	2.95	40.72	5.47
crypto.rsa	29.7	1.36	21.95	2.55	27.62	1.20	20.49	2.58
crypto.signverify	29.39	1.39	21.65	2.75	27.33	1.40	20.26	2.56
mpegaudio	53.26	2.14	38.54	4.46	54.17	2.20	39.67	4.63
scimark.fft	27.90	1.38	20.05	2.57	26.63	1.40	19.33	2.46
scimark.lu	24.20	1.11	18.39	2.29	23.49	1.14	17.71	2.23
scimark.monte_carlo	34.21	2.00	24.97	3.42	33.33	2.01	24.28	3.40
scimark.sor	32.08	2.01	23.03	3.14	31.47	2.04	22.43	3.16
scimark.sparse	31.81	1.62	23.78	3.14	31.28	1.67	23.10	2.97
serial	56.65	2.23	44.59	5.26	57.44	2.33	44.64	5.48
sunflow	55.50	1.88	42.10	4.35	55.58	1.95	42.24	4.35
xml.transform	264.62	13.41	210.59	27.74	272.31	13.54	214.44	27.44
xml.validation	49.83	1.72	42.29	4.70	50.42	1.63	40.97	4.51

5.1.6 `-XX:+AlwaysPreTouch`

`AlwaysPreTouch` is disabled by default as it results in a delay in JVM start up. It enables the touching of every page on the Java heap during JVM initialization which causes memory allocation in heap memory. For `AlwaysPreTouch`, Open JDK is more energy-efficient for the 16 benchmarks on the IFN and Oracle JDK is more energy-efficient for the 17 benchmarks on the Laptop as shown in Table 5.13. For the IFN, `AlwaysPreTouch` results in the higher energy efficiency of most benchmarks for both JDKs, except `startup` benchmark where all sub-benchmarks have lower energy efficiency than the default mode. For the Laptop, `AlwaysPreTouch` consumes higher energy for at least 17 benchmarks than the default mode for both JDKs.

Table 5.14: Energy consumption for Xnoclassgc option

Benchmark	Xnoclassgc							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	TT
compress	10397.13	242.71	2530.63	244.90	10430.03	242.76	2531.67	243.79
crypto								
crypto.aes	10356.70	246.17	2581.25	248.68	10344.17	246.96	2598.35	248.25
crypto.rsa	10067.18	242.11	2441.48	243.56	9647.78	241.77	2347.02	242.26
crypto.signverify	9718.36	242.06	2456.49	243.03	9713.18	241.77	2453.8	243.18
derby	10922.57	259.43	2607.89	432.50	10952.82	259.19	2611.02	422.42
mpegaudio	10444.41	243.52	2517.83	246.71	10453.01	243.87	2517.06	245.04
scimark								
fft.small	11432.51	241.88	2683.80	243.34	11423.12	241.88	2684.92	243.59
lu.small	14731.55	241.65	2854.90	242.88	14674.87	241.50	2853.61	242.86
monte_carlo	9964.75	242.68	2356.48	244.24	9968.50	242.33	2364.60	244.35
sor.large	7582.58	250.42	2476.26	251.85	7556.39	247.56	2490.24	253.65
sor.small	8156.63	243.03	2064.72	243.90	8132.93	242.57	2061.13	243.54
sparse.large	6151.81	249.79	2663.50	254.04	6116.82	256.45	2645.58	258.08
sparse.small	9678.32	243.33	2800.84	245.30	11118.20	242.95	2793.22	244.56
serial	11121.66	243.46	2571.04	245.01	11189.05	243.36	2565.15	246.44
sunflow	10103.37	243.26	2581.16	244.26	10097.69	243.21	2581.75	245.08
xml								
xml.transform	11348.86	254.12	2741.08	266.76	11356.83	253.7	2752.30	266.75
xml.validation	11734.66	241.79	2528.40	243.01	11730.62	241.67	2537.71	242.89
startup								
compress	32.78	1.62	23.40	3.02	32.14	1.67	22.90	2.95
crypto.aes	48.56	2.71	37.97	5.15	51.00	2.78	41.01	5.52
crypto.rsa	28.87	1.21	21.93	2.69	26.02	1.07	20.29	2.44
crypto.signverify	28.60	1.32	21.64	2.68	25.83	1.24	20.18	2.52
mpegaudio	52.20	2.08	38.04	4.45	52.37	2.15	38.73	4.82
scimark.fft	26.31	1.25	20.11	2.54	24.64	1.23	19.28	2.48
scimark.lu	22.85	1.00	18.38	2.23	21.90	0.7	17.72	2.30
scimark.monte_carlo	32.96	1.88	24.98	3.40	32.23	1.90	24.16	3.43
scimark.sor	31.16	1.89	23.04	3.27	30.34	1.91	22.42	3.20
scimark.sparse	30.64	1.51	23.65	3.04	29.91	1.58	23.01	3.04
serial	55.96	2.14	43.38	5.00	56.89	2.24	46.65	5.57
sunflow	53.81	1.76	42.14	4.43	54.93	1.82	42.49	4.32
xml.transform	264.63	13.49	205.88	27.39	269.17	13.36	210.41	26.39
xml.validation	49.08	1.57	39.82	4.37	49.04	1.65	42.32	4.55

5.1.7 `-Xnoclassgc`, `-XX:+UseSerialGC`, `-XX:+UseParallelGC`, `-XX:+UseConcMarkSweepGC`, and `-XX:+UseG1GC`

`Xnoclassgc` option disables garbage collection of classes. Using `Xnoclassgc`, Oracle JDK results in the higher energy efficiency of 19 benchmarks on the IFN and 15 benchmarks on the Laptop as shown in Table 5.14. However, Open JDK consumes up to 12% less energy as compared to Oracle JDK. For the IFN, `Xnoclassgc` results in the higher energy efficiency of 24 benchmarks for Open JDK but lower energy efficiency of 22 benchmarks for Oracle JDK than the default mode. The same behavior is shown by the Laptop.

`UseSerialGC` option uses a single thread and freezes all the application threads during

Table 5.15: Energy consumption for UseSerialGC option

Benchmark	UseSerialGC							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10434.73	242.77	2533.53	244.50	10396.33	243.00	2531.67	244.33
crypto								
crypto.aes	10383.46	246.83	2592.25	254.10	10359.77	246.56	2604.08	248.48
crypto.rsa	10132.89	242.07	2440.25	243.71	9649.29	241.83	2340.35	242.45
crypto.signverify	9723.60	242.06	2458.82	243.64	9712.7	241.79	2446.01	243.10
derby	10610.54	259.63	2584.04	425.52	10632.77	259.73	2591.75	424.28
mpegaudio	10450.01	243.81	2514.19	245.03	10440.95	243.04	2514.28	245.44
scimark								
fft.small	11437.74	241.90	2690.38	243.57	11409.61	241.88	2685.78	243.04
lu.small	14616.01	241.66	2817.72	243.30	14544.15	241.63	2812.33	242.83
monte_carlo	9971.67	242.36	2353.34	244.63	9960.36	242.32	2357.95	245.40
sor.large	7751.69	249.95	2489.94	252.70	7692.43	249.77	2483.87	255.12
sor.small	8138.79	242.92	2072.7	244.83	8098.71	242.79	2071.87	244.94
sparse.large	5761.24	255.90	2651.00	245.90	5696.68	248.81	2639.79	256.19
sparse.small	9945.33	242.86	2787.57	244.62	11407.97	242.61	2765.67	244.79
serial	11045.67	243.13	2559.38	245.65	11081.10	243.12	2564.25	246.19
sunflow	9774.58	243.62	2491.17	244.21	9759.61	242.54	2500.23	245.94
xml								
xml.transform	10734.84	253.83	2635.95	266.7	10728.55	253.7	2640.31	267.42
xml.validation	8918.29	241.62	2336.65	242.88	8834.73	241.66	2336.21	243.41
startup								
compress	31.91	1.63	23.52	2.98	31.15	1.69	22.55	3.02
crypto.aes	47.40	2.61	37.76	5.15	49.66	2.78	40.84	5.48
crypto.rsa	28.18	1.19	21.69	2.64	25.48	1.03	20.36	2.39
crypto.signverify	27.52	1.33	21.83	2.66	25.44	1.27	20.19	2.52
mpegaudio	50.59	2.04	38.03	4.70	51.84	2.09	39.06	4.72
scimark.fft	25.60	1.22	19.85	2.51	24.30	1.22	19.21	2.53
scimark.lu	22.34	0.97	18.24	2.23	21.59	1.01	17.66	2.19
scimark.monte_carlo	32.44	1.89	24.7	3.49	31.28	1.89	24.08	3.43
scimark.sor	30.28	1.89	22.81	3.23	29.42	1.89	22.32	3.23
scimark.sparse	29.83	1.53	23.57	3.02	29.17	1.55	22.86	3.08
serial	55.10	2.13	44.03	5.18	55.52	2.24	45.36	4.85
sunflow	53.20	1.80	41.80	4.43	53.79	1.79	42.25	4.50
xml.transform	260.23	13.52	205.33	26.51	267.14	13.50	212.71	27.01
xml.validation	47.36	1.60	40.35	4.34	47.93	1.68	41.08	4.72

garbage collection. As shown in Table 5.15, for the `UseSerialGC` option, Oracle JDK results in the higher energy efficiency than Open JDK for 22 benchmarks on the IFN and 18 benchmarks on the Laptop. For both systems, `UseSerialGC` results in the higher energy efficiency of at least 22 benchmarks than the default mode for both JDKs.

`UseParallelGC` option uses multiple threads for garbage collection and has the same energy consumption as the `server` command-line option because parallel garbage collector is the default garbage collector of JVM. `UseConcMarkSweepGC` option minimizes the pauses during the garbage collection by performing the garbage collection concurrently with the application threads. For `UseConcMarkSweepGC`, Oracle JDK results in the higher energy efficiency than

Table 5.16: Energy consumption for UseConcMarkSweepGC option

Benchmark	UseConcMarkSweepGC							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	10433.09	243.32	2529.27	244.08	10401.93	243.31	2525.31	243.91
crypto								
crypto.aes	10321.06	247.11	2598.34	252.60	10363.69	246.52	2591.85	249.45
crypto.rsa	10123.12	242.11	2439.91	242.96	9660.88	241.84	2347.97	242.33
crypto.signverify	9739.67	242.06	2452.01	243.92	9743.21	241.87	2445.81	243.04
derby	10500.20	258.57	2530.31	418.63	10510.22	258.57	2529.02	425.87
mpegaudio	10453.13	243.90	2514.94	245.01	10430.84	243.10	2515.18	245.70
scimark								
fft.small	11487.74	241.87	2685.34	243.54	11484.40	241.84	2684.68	243.57
lu.small	14679.32	241.72	2830.50	243.03	14592.90	241.58	2823.12	242.67
monte_carlo	9977.62	242.69	2351.60	245.48	9984.29	242.34	2361.1	244.83
sor.large	7593.93	249.11	2502.64	258.87	7598.94	247.84	2461.65	252.11
sor.small	8176.51	242.73	2083.28	243.80	8130.51	242.40	2068.68	244.69
sparse.large	7103.00	257.31	2704.88	264.26	6896.17	259.73	2714.60	261.12
sparse.small	10088.55	242.67	2769.37	244.70	11293.53	242.54	2792.31	244.58
serial	11100.34	243.72	2564.73	246.82	11149.12	243.52	2562.00	246.21
sunflow	10074.91	243.05	2496.52	244.60	10057.33	242.76	2484.38	244.95
xml								
xml.transform	11228.96	254.12	2682.14	268.13	11259.29	253.94	2691.19	267.97
xml.validation	11337.46	241.86	2491.38	243.34	11389.85	241.62	2487.62	243.00
startup								
compress	32.98	1.72	23.68	3.03	32.20	1.66	22.79	3.00
crypto.aes	48.30	2.64	37.96	5.22	51.39	2.83	40.64	5.43
crypto.rsa	28.78	1.24	21.88	2.60	26.34	1.06	20.28	2.30
crypto.signverify	28.47	1.36	21.65	2.72	26.01	1.27	20.20	2.52
mpegaudio	51.67	2.08	38.71	4.42	52.85	2.21	39.39	4.67
scimark.fft	26.58	1.28	19.96	2.52	25.51	1.33	19.29	2.48
scimark.lu	22.91	0.7	18.40	2.24	22.31	1.05	17.74	2.27
scimark.monte_carlo	32.83	1.89	24.83	3.44	32.25	1.95	24.29	3.42
scimark.sor	31.52	1.90	23.00	3.21	30.37	1.93	22.42	3.20
scimark.sparse	30.55	1.46	23.60	3.08	30.10	1.59	22.93	3.05
serial	55.19	2.13	45.13	5.39	57.52	2.26	44.11	5.01
sunflow	53.93	1.83	42.42	4.40	54.81	1.83	43.05	4.52
xml.transform	262.96	13.41	207.96	26.85	271.08	13.44	209.95	27.59
xml.validation	49.92	1.62	41.41	4.26	50.32	1.67	41.61	4.58

Open JDK for 16 benchmarks on the IFN and 21 benchmarks on the Laptop as shown in Table 5.16. For the IFN, `UseConcMarkSweepGC` results in the higher energy efficiency of 19 benchmarks for Open JDK but only for 9 benchmarks for Oracle JDK as compared to the default mode. The Laptop shows higher energy efficiency for `UseConcMarkSweepGC` for both JDKs for at least 19 benchmarks.

`UseG1GC` is parallel, concurrent and compacts the free heap space as soon as it reclaims the memory. For the IFN, Open JDK is more energy-efficient for 22 benchmarks whereas, for the Laptop, Oracle JDK is more energy-efficient for 16 benchmarks as shown in Table 5.17. For both systems, `UseG1GC` consumes up to 14% lesser energy than the default mode for 17

Table 5.17: Energy consumption for UseG1GC option

Benchmark	UseG1GC							
	Open				Oracle			
	IFN		Laptop		IFN		Laptop	
	E	T	E	T	E	T	E	T
compress	9880.35	243.56	2463.06	244.67	10020.79	243.57	2461.73	244.60
crypto								
crypto.aes	10303.34	246.20	2587.50	251.22	10329.26	246.66	2608.38	249.10
crypto.rsa	10055.39	242.13	2425.68	243.70	9656.16	241.83	2303.61	242.44
crypto.signverify	9751.27	242.04	2464.59	243.02	9771.50	241.96	2453.31	243.29
derby	10913.00	259.61	2638.94	422.53	10932.28	260.29	2682.52	438.71
mpegaudio	10466.92	243.94	2456.13	245.69	10456.34	243.00	2470.6	246.70
scimark								
fft.small	11320.12	241.90	2693.62	243.61	11304.78	241.93	2692.59	243.40
lu.small	14559.84	241.66	2854.7	243.02	14632.02	241.66	2846.57	243.13
monte_carlo	9913.28	242.66	2356.13	244.14	9913.89	242.84	2353.92	244.25
sor.large	7583.39	247.91	2485.21	255.76	7613.76	250.91	2464.48	253.23
sor.small	8292.91	243.26	2089.08	244.76	8292.73	243.29	2097.27	245.12
sparse.large	6166.40	251.40	2660.11	258.88	6057.50	254.89	2651.66	251.48
sparse.small	9459.37	242.56	2644.83	245.97	9503.92	243.32	2630.37	245.27
serial	10955.61	243.49	2537.92	246.64	10870.44	243.20	2550.76	246.78
sunflow	11058.25	242.81	2560.67	243.88	11153.35	242.67	2561.94	243.54
xml								
xml.transform	11344.32	254.29	2735.98	268.48	11382.41	254.31	2748.81	269.31
xml.validation	11619.79	241.77	2493.87	243.45	11627.28	241.78	2497.92	243.20
startup								
compress	34.05	1.72	24.20	3.11	34.41	1.72	24.17	3.06
crypto.aes	48.91	2.74	38.79	5.30	52.94	2.83	42.10	5.58
crypto.rsa	29.76	1.32	22.83	2.65	28.17	1.11	21.81	2.64
crypto.signverify	29.10	1.38	22.52	2.76	28.12	1.34	21.67	2.61
mpegaudio	52.43	2.19	39.75	4.73	54.39	2.17	40.48	4.75
scimark.fft	26.88	1.31	20.62	2.56	27.12	1.30	20.65	2.64
scimark.lu	24.04	1.09	19.07	2.31	24.28	1.08	18.7	2.34
scimark.monte_carlo	33.41	1.96	25.78	3.50	34.02	1.98	25.51	3.55
scimark.sor	32.08	2.02	23.88	3.34	32.17	1.98	23.83	3.32
scimark.sparse	31.86	1.63	24.30	3.08	31.80	1.58	24.20	3.15
serial	56.65	2.21	44.81	5.11	59.50	2.34	46.44	5.61
sunflow	55.84	1.79	43.28	4.42	56.59	2.02	43.74	4.51
xml.transform	263.56	13.37	207.10	26.76	271.50	13.47	213.63	27.87
xml.validation	50.33	1.63	42.38	4.64	51.13	1.71	43.25	4.67

benchmarks. We select it as the most energy-efficient command-line option because for benchmarks except `startup` (lightweight version of other benchmarks), it consumes lesser energy than `UseSerialGC`. `crypto.rsa` results in the highest energy consumption variation of different JDKs on the IFN.

5.2 Energy & Time

In this section, we analyze the correlation between energy and time for each command-line option for each JDK and system. The values for the correlation are shown in Fig. 5.1. The first thing we notice is that Energy and Time have a high correlation (strong linear relationship) which varies with a maximum value of 0.98 for Oracle JDK on the IFN and a minimum value of 0.94 for

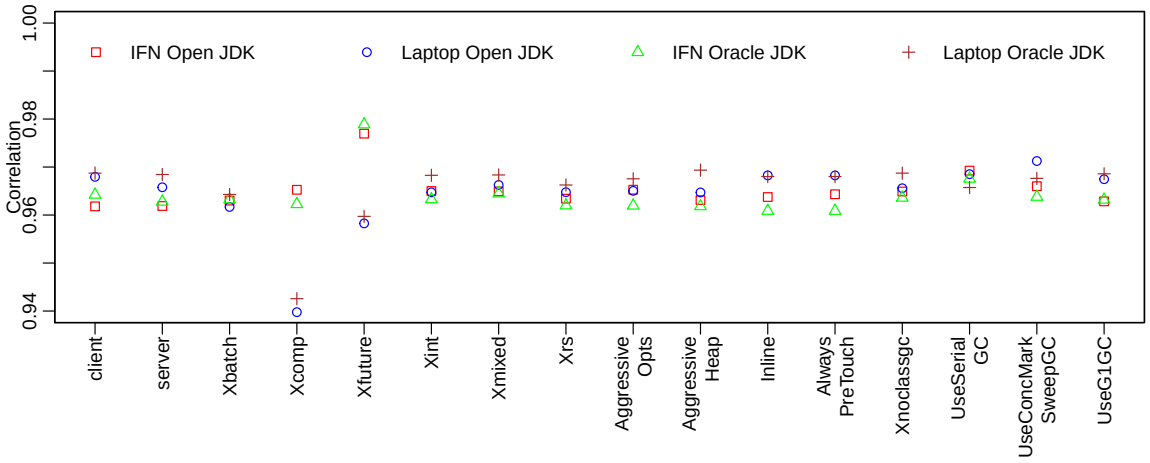


Figure 5.1: Energy & Time correlation.

Open JDK on the Laptop. The high correlation is expected because we stabilize the idle energy. Second, Open and Oracle JDK for the two systems results in almost same correlation. Third, for the IFN, Open JDK shows a higher correlation, whereas, for the Laptop, Oracle JDK shows a higher correlation. Last, `XComp` and `Xfuture` show a big difference between the correlation values of the two ICT systems.

5.3 Related Work

Most of the Java performance improvement work mention tuning JVM command line options [66, 89, 90]. JVM command line options show up to 5% improvement in Hadoop performance in [91]. Other works focus on refactoring of software to improve Java performance [73, 74, 77, 78, 80, 65, 62, 63].

OpenJDK and IBM I9 performance-power analysis is presented in [92] using SPECjvm2008 Base run category. It is the closest work we can find, however, it neither stabilize the idle energy and nor analyze Java command-line options in terms of energy efficiency.

5.4 Summary

In this chapter, we show how various command-line options cause Java applications to consume different energy. We evaluate these command-line options for active energy efficiency on two different ICT systems using the SPECjvm2008 benchmarks for Open and Oracle JDK. We stabilize the idle energy to get an accurate measurement of the active energy. For each command-line

options, we check which JDK performs better. Oracle JDK results in better energy efficiency for most of the command-line options. Next, we compare each command-line option to default JVM settings or server command-line option. We show that `Xint` causes the lowest energy efficiency and `UseG1GC` causes the highest energy efficiency. We find a strong linear relationship between active energy and execution time. We hope these results will help software users to choose between command-line options for a better energy efficiency of Java applications.

CHAPTER 6 JEPO: JAVA ENERGY PROFILER AND OPTIMIZER

Software is one of the most critical bottlenecks while trying to optimize the energy consumption of any ICT system. In this work, we present our JEPO* tool to help software developers to write energy-efficient code. This tool is an Eclipse IDE plugin and provides energy-efficient suggestions for Java programming language. It can provide suggestions dynamically while writing code or statically to refactor already written code. For providing suggestions, it analyzes each line of Java file and matches it to the pool of suggestions that we gather from the findings in our earlier work [62, 63]. JEPO can also help the software developers to measure energy consumption automatically at method granularity to determine the energy-hungry Java methods in a software. For measuring the energy, it injects energy and time measurement code at the start and end of each method in the project. This injected code leverage Intel RAPL technology to measure energy consumption.

6.1 JEPO

JEPO is an Eclipse plugin developed to provide suggestions for software developers to write energy-efficient code in real-time or to use suggestions to refactor already written code. It is a mix of a static and dynamic tool as it can not only give real-time suggestions while writing the code but also can be used to get suggestions for already written code. The suggestions are a result of our earlier work in which we analyze various components of Java programming language [62, 63]. These suggestions are hardcoded in the tool and displayed whenever the tool detect specific Java components like data types, operators, control statements, String, exceptions, objects, and Arrays.

JEPO analyzes each line of the code and checks for specific patterns to generate suggestions. These patterns relate to various components of Java programming language and are shown with suggestions in Table 6.1. For primitive data types - `byte`, `short`, `int`, `long`, `float`, `double` and `char` - `int` primitive data type is recommended for the best energy efficiency. Decimal numbers when typed as scientific notation consumes lesser energy. Wrapper classes are object representation of primitive data types. Integer wrapper class object consumes lesser energy than

*https://github.com/mohitkumar14/JEPO

Table 6.1: Java Components & Suggestions

Java Components	Suggestions
Primitive data types	int is the most energy-efficient primitive data type. Replace if possible.
Scientific notation	Scientific notation results in lower energy consumption of decimal numbers.
Wrapper classes	Integer Wrapper class object is the most energy-efficient. Replace if possible.
Static keyword	static keyword consumes up to 17,700% more energy. Avoid if possible.
Arithmetic operators	Modulus arithmetic operator consumes up to 1,620% more energy than other arithmetic operators.
Ternary operator	Ternary operator consumes up to 37% more energy than if-then-else statement.
Short circuit operator	Put most common case first for lower energy consumption.
String concatenation operator	StringBuilder append method consumes much lower energy than String concatenation operator.
String comparison	String compareTo method consumes up to 33% more energy than String equals method.
Arrays copy	System.arraycopy() is the most energy-efficient way to copy Arrays.
Array traversal	Two-dimensional Array column traversal results in up to 793% more energy.

any other wrapper class. Static keyword result in up to 17,700% increase in energy consumption of variables. Modulus is the most energy expensive arithmetic operator. Ternary operator consumes higher energy than if-then-else option. Putting most common cases in a short-circuit operator helps in saving energy. For string concatenation, `StringBuilder append` is the best way to concatenate string. String comparison method `compareTo` result in higher energy consumption than `equals` method. `System.arraycopy()` is the best way to copy array. Array column traversal is energy expensive than row traversal. All these suggestions work better when the above java components are used repeatedly in a program. More general suggestions can be found in our earlier work [62, 63].

JEPO can also help software developers to determine the energy-hungry methods in a Java project. This is achieved by injecting code in bytecode to read the machine specific registers (MSR) at the start and end of each method in the Java project using Javassist library [17]. We first

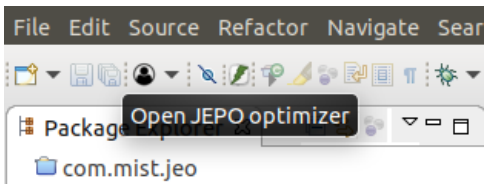


Figure 6.1: JEPO toolbar button

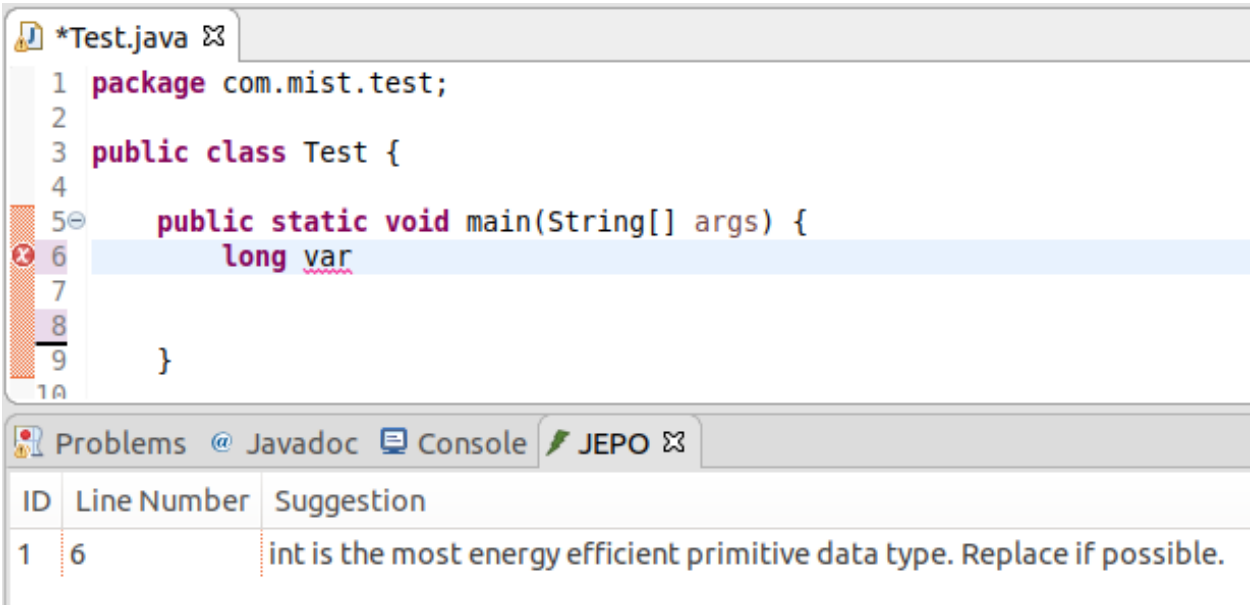


Figure 6.2: JEPO dynamic suggestion

search for all classes that has a main method in the project. If there is only one main class, then we choose it as our main class. If there are more than one, then we take user input to determine the correct main class for the project. After we finalize the main class, we create a new Java file named JEPOInsert in com.mist.jepo package. The purpose of this class is to inject the energy measurement code for each method in the project and then run the earlier selected main class. The injected code measure and store the MSR and the execution start or stop time whenever a method is executed. When the execution end, the energy consumption and execution time for all the executed methods are stored in a result.txt file in Java project directory and shown in JEPO view. If one method is executed more than once, then the measurements are stored for each execution.

JEPO include one toolbar button and two pop-up menu buttons. The toolbar button (shown in Fig.6.1) open JEPO view if it's not already open and then show the suggestions for the already open Java file. If the Java file is not already open then JEPO will show an empty view. The toolbar

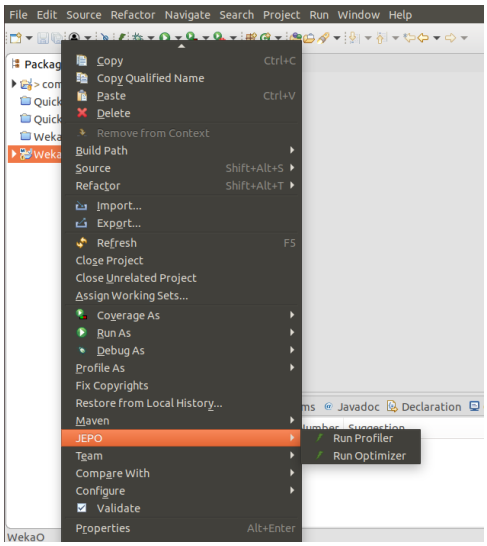


Figure 6.3: JEPO pop-up menu buttons

Problems @ Javadoc Declaration Console JEPO

Method Name	Execution Time (s)	Execution Energy (J)
com.mist.test.Test.main()	0.241	2.8242645263671875
com.mist.test.Test.count()	0.035	0.3778228759765625
com.mist.test.Test.count()	0.03	0.3336944580078125
com.mist.test.Test.count()	0.03	0.335174560546875
com.mist.test.Test.count()	0.035	0.4183349609375

Figure 6.4: JEPO profiler view

button view provides dynamic suggestions as shown in Fig. 6.2. The pop-up menu button can be accessed by doing right-click on any Java project. The pop-up menu will then show a button named JEPO with two sub-menu button - JEPO profiler and JEPO optimizer (Fig. 6.3). The JEPO profiler creates the JEPOInsert.java file to measure the energy consumption at method granularity. It shows the energy consumption for each method executed while running a Java project in JEPO view Fig. 6.4. The JEPO profiler functionality is limited to Intel processors that support RAPL and Ubuntu OS. The JEPO optimizer provides suggestions for all the classes in a Java project.

Table 6.2: WEKA classifiers metrics calculated using Eclipse Metrics Plug-in [93] and Class Dependency Analyzer (CDA) [94]

Classifiers	Dependencies	Attributes	Methods	Packages	LOC
J48	684	3263	7746	41	101172
Random Tree	668	3235	7611	41	99938
Random Forest	673	3270	7736	42	101812
REP Tree	668	3235	7619	41	100074
Naive Bayes	668	3229	7582	40	99221
Logistic	666	3216	7553	40	98812
SMO	677	3305	7796	43	102250
SGD	669	3222	7585	40	99304
KStar	671	3282	7576	41	99421
IBk	671	3268	7703	41	100339

6.2 Validation

For evaluating JEPO, we leverage WEKA an open-source machine learning software. We first make changes to WEKA as per JEPO suggestions and then evaluated the different classifiers on the Laptop described in Section 3.1.

WEKA software has 3373 classes in total. Its different classifiers specifications are shown in Table 6.2. Dependencies, attributes, methods, packages, and line of code (LOC) has almost the same properties for all classifiers. J48 implements a modified version of C4.5 which uses decision tree for classification. For building trees, RandomTree takes in account a given number of random features at each node without performing any pruning. RandomForest uses bagging on ensemble of random trees. REPTree uses information gain and variance reduction for constructing decision or regression tree. For pruning, reduced-error pruning method is used. Naive Bayes is a probabilistic classifier which is based on Bayes theorem. Logistic builds a multinomial logistic regression that uses a ridge estimator to guard against overfitting by penalizing large coefficients based on [95]. SMO uses polynomial or Gaussian kernels to implement the sequential minimal optimization algorithm for training a support vector qualifier [96, 97]. SGD is a stochastic gradient descent learning model with various loss functions. KStar and IBK are lazy classifiers which work only during the classification tome. KStar implements a nearest-neighbor classifier with general-

Table 6.3: MOA airlines data

Attributes	Type
Airline	Nominal
Flight	Numeric
Airport From	Nominal
Airport To	Nominal
Day Of Week	Nominal
Time	Numeric
Length	Numeric
Delay	Binary

ized distance function based on transformations whereas IBk implements a k-nearest-neighbour classifier.

The data used for classification is Massive Online Analysis (MOA) data [98], which is used to predict whether a flight will be delayed or not. The data has 8 attributes and 539,383 instances. We reduce the number of instances to 10,000 due to limited heap memory. The attributes are shown in Table 6.3. The attributes refer to the airline name, flight name, airport from where the flight departs, airport to which flight arrives, day of the week, time of the flight, distance of the flight, and whether the flight get delayed or not. There are 4 nominal, 3 numeric and one binary attribute. For airline and airports nominal values, the distinct values are 18 and 293, respectively.

Next, we make changes to the dependent classes as per JEPO suggestions and evaluated various classifiers using stratified 10-fold cross-validation. We first run each classifier 10 times to measure Package energy, CPU energy, and execution time using `perf` Linux tool. After that, we detect outliers using Tukey's method [60] from each metric, replace the outliers measurements with new measurements and again check for outliers. We repeat this process until no outlier is left. When no outlier is left, we calculated the mean of values. The final values are shown in Table 6.4. As expected, the changes made are almost same due to the same number of dependencies. However, other metrics do not agree with the number of changes. For Package energy consumption, CPU energy consumption and execution time, Random Forest shows the highest improvement of 14.46%, 14.19% and 12.93%, respectively. Random Tree shows the most amount of accuracy drop of 0.48%.

Table 6.4: WEKA evaluation

Classifiers	Changes	Package Improvement (%)	CPU Improvement (%)	Execution Time Improvement (%)	Accuracy Drop (%)
J48	877	4.44	4.68	3.96	0.00
Random Tree	709	0.02	0.01	0.01	0.48
Random Forest	719	14.46	14.19	12.93	0.00
REP Tree	723	3.70	3.49	2.01	0.00
Naive Bayes	711	3.58	3.82	0.00	0.00
Logistic	711	0.10	0.10	0.00	0.00
SMO	713	0.05	0.08	0.04	0.17
SGD	713	7.48	5.76	5.56	0.05
KStar	711	6.82	5.31	0.00	0.00
IBk	711	5.50	5.34	6.01	0.00

These results shows increase in metrics improvement when we increase the number of instances of MOA data to 20,000. For autonomous vehicles, data centers, and supercomputers, where huge amount of data is analyzed in short time, JEPO can help to significantly reduce the energy consumption of software.

6.3 Related Work

Software energy measurement challenges like lack of energy measurement tools, lack of instrumentation to estimate energy consumption for various OS and processors is presented in [99]. SEEDS and Chameleon frameworks for automating code-level changes and optimizing Java applications were introduced in [100] and [101]. These frameworks can select the most efficient collection for improving the energy efficiency of an application. SEEDS resulted in 17% energy consumption improvement. Software change-impact analysis tool, GreenAdvisor, help software developers to estimate the change in energy profiles due to change in an application system calls [102]. Eco, a programming model, is introduced in [103] to provide support for energy-aware applications. A similar tool, EnSights, provides energy change information by analyzing the change in the structure of a code [104]. It can estimate the change in energy consumption with F-scores of up to 86%. jStanley - an Eclipse plugin - provides a suggestion for energy consumption usage of collections in Java [16]. The software developers can use the suggestion to replace a collection

with a better collection. It shows to improve energy consumption between 2% and 17%. In this chapter, we present an Eclipse plugin which gives suggestions about data types, operators, control statements, String, exceptions, objects, and Arrays. It can also help software developers in measuring the energy consumption of software applications automatically at method granularity.

6.4 Summary

Software energy research has been around for some years. Researchers have performed analysis on various languages. However, there is a lack of tool which can disseminate these software power savings findings to software developers. Therefore, in this work, we present an Eclipse plugin JEPO to help software developers write energy-efficient code, dynamically and statically. JEPO can also provide measurements for energy consumption automatically at method granularity. Using JEPO we were able to achieve up to 14.46% improvement in package energy consumption, up to 14.19% improvement in CPU energy consumption, up to 12.93% in execution time and with only 0.48% drop in accuracy.

CHAPTER 7 CONCLUSION

Software energy efficiency research still has a long way to go as most of the energy efficiency research concentrate on hardware. Software developers have been oblivious to application energy efficiency for years. They do not pay much attention to the energy consumption of the software they develop. There are no well-established guidelines that they can follow to write energy-efficient code. We try to stimulate the development of such guidelines for developing energy-efficient software. We measure and compare the energy efficiency of data types, operators, control statements, String, exceptions, objects, and Arrays in Java. We find a strong linear relationship between energy consumption and execution time for different JDKs. These results will help software developers to build more energy-efficient Java applications in the future.

Java command-line options cause Java applications to consume different energy. We evaluate these command-line options for active energy efficiency on two different ICT systems using the SPECjvm2008 benchmarks for Open and Oracle JDK. We stabilize the idle energy to get an accurate measurement of the active energy. For each command-line options, we check which JDK performs better. Oracle JDK results in better energy efficiency for most of the command-line options. Next, we compare each command-line option to default JVM settings or server command-line option. We show that `Xint` causes the lowest energy efficiency and `UseG1GC` causes the highest energy efficiency. We find a strong linear relationship between active energy and execution time. These results will help software users to choose between command-line options for a better energy efficiency of Java applications.

Researchers have performed analyses on various languages. However, there is a lack of tools that can disseminate these software power savings findings to software developers. Therefore, we present an Eclipse plugin JEPO to suggest software developers in writing energy-efficient code, dynamically and statically. JEPO can also provide energy consumption measurements automatically at method granularity. Using JEPO we were able to achieve up to 14.46% improvement in package energy consumption, up to 14.19% improvement in CPU energy consumption, up to 12.93% in execution time and with only 0.48% loss in accuracy. We hope software developers can

use JEPO to make their software energy-efficient.

CHAPTER 8 FUTURE WORK

Java programming language is one of the many languages used today for software development. In the future, we can evaluate other programming languages at the code level and command-line options level. We can also work on improving JEPO to add more suggestions.

For other programming languages, we can evaluate components that are same as Java programming language like primitive data types, operators, conditional statements and threads. We can also evaluate methods that have different names but can perform the same functionalities. Like Java, every programming language has several versions that we can compare for consistency of energy consumption and execution time.

Programming languages lack tools like JEPO which can provide suggestions to software developers for improving the energy consumption of software. Such tools can also help in measuring the energy consumption of software. Software developers can use such tools to not only find energy-hungry locations at different granularity like methods and classes but also to apply energy-savings suggestions automatically to such locations. We hope, we can provide JEPO like tools for other programming languages in the future.

REFERENCES

[1] B. Steigerwald, C. Lucero, C. Akella, and A. Agrawal, *Energy Aware Computing: Powerful Approaches for Green System Design*. Intel Press, 2012.

[2] J. Koomey, S. Berard, M. Sanchez, and H. Wong, "Implications of historical trends in the electrical efficiency of computing," *IEEE Annals of the History of Computing*, vol. 33, no. 3, pp. 46–54, 2011.

[3] ITU, "Ict facts and figures 2016." `http://www.itu.int/en/ITU-D/Statistics/Pages/facts/default.aspx`.

[4] M. Mills, "The cloud begins with coal. digital power group," 2013.

[5] S. Mittal, "A survey of techniques for improving energy efficiency in embedded computing systems," *Int. Journal of Computer Aided Engineering and Technology*, vol. 6, no. 4, pp. 440–459, 2014.

[6] S. Murugesan and G. Gangadharan, *Harnessing green IT: Principles and practices*. Wiley Publishing, 2012.

[7] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEICE Transactions on Electronics*, vol. 75, no. 4, pp. 371–382, 1992.

[8] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada, "1-v power supply high-speed digital circuit technology with multithreshold-voltage cmos," *IEEE Journal of Solid-state circuits*, vol. 30, no. 8, pp. 847–854, 1995.

[9] T. D. Burd and R. W. Brodersen, "Energy efficient cmos microprocessor design," in *Proc. 28th Hawaii Int. Conf. on System Sciences*, vol. 1, pp. 288–297, IEEE, 1995.

[10] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen, "A dynamic voltage scaled microprocessor system," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1571–1580, 2000.

[11] J. B. Burr and A. M. Peterson, "Energy considerations in multichip-module based multiprocessors," in *Proc., IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors*, pp. 593–600, IEEE, 1991.

[12] J. Burr, L. Gal, R. Haddad, J. Rabaey, and B. Wooley, "Which has greater potential power impact: High-level design and algorithms or innovative low power technology?(panel)," in *Proc. Int. Symp. on Low Power Electronics and Design*, p. 175, IEEE Press, 1996.

[13] H. Chen, S. Wang, and W. Shi, "Where does the power go in a computer system: Experimental analysis and implications," in *Green Computing Conference and Workshops (IGCC), 2011 International*, pp. 1–6, IEEE, 2011.

[14] "systemd." `https://www.freedesktop.org/wiki/Software/systemd/`. Accessed: 2018-08-04.

[15] "systemctl." `https://www.freedesktop.org/software/systemd/man/systemctl.html`. Accessed: 2018-08-04.

[16] R. Pereira, P. Simão, J. Cunha, and J. Saraiva, "jstanley: placing a green thumb on java collections," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 856–859, ACM, 2018.

[17] S. Chiba, "Javassista reflection-based programming wizard for java," in *Proceedings of OOPSLA98 Workshop on Reflective Programming in C++ and Java*, vol. 174, p. 21, 1998.

[18] E. Oró, V. Depoorter, A. Garcia, and J. Salom, "Energy efficiency and renewable energy integration in data centres. strategies and modelling review," *Renewable and Sustainable Energy Reviews*, vol. 42, pp. 429–445, 2015.

[19] S. Heath, *Embedded systems design*. Elsevier, 2002.

[20] M. Barr, "Real men program in c," *Embedded systems design*, vol. 22, no. 7, p. 3, 2009.

[21] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," 2007.

[22] D. Anderson, J. Dykes, and E. Riedel, "More than an interface-scsi vs. ata.," in *FAST*, vol. 2, p. 3, 2003.

[23] T. Murphy, "40 years after the first cell phone call: Who is inventing tomorrow? s future?," *IEEE Consumer Electronics Magazine*, vol. 2, no. 4, pp. 44–46, 2013.

[24] A. Fehske, G. Fettweis, J. Malmodin, and G. Biczok, "The global footprint of mobile communications: The ecological and economic perspective," *IEEE communications magazine*, vol. 49, no. 8, pp. 55–62, 2011.

[25] "Global tablet market falls 8 percent in 2015," 2019. Accessed: 2019-17-09.

[26] "Computers sold in the world this year - worldometers. 2019," 2019. Accessed: 2019-17-09.

[27] B. GeSI, "2020: Enabling the low carbon economy in the information age,[london, uk, 2008."

[28] "Environmental responsibility report. 2019," 2019. Accessed: 2019-17-09.

[29] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pp. 273–286, USENIX Association, 2005.

[30] A. Kwasinski and A. Kwasinski, "Resiliency in the sustainability of distributed green data centers," in *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*, pp. 1–6, IEEE, 2015.

[31] W.-c. Feng, X. Feng, and R. Ge, "Green supercomputing comes of age," *IT professional*, vol. 10, no. 1, pp. 17–23, 2008.

[32] W.-c. Feng and K. Cameron, "The green500 list: Encouraging sustainable supercomputing," *Computer*, vol. 40, no. 12, pp. 50–55, 2007.

[33] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, *et al.*, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[34] J. H. Laros III, K. T. Pedretti, S. M. Kelly, W. Shu, and C. T. Vaughan, "Energy based performance tuning for large scale high performance computing systems," in *Proceedings of the 2012 Symposium on High Performance Computing*, p. 6, Society for Computer Simulation International, 2012.

[35] S. Bandyopadhyay, "A study on performance monitoring counters in x86-architecture," *Indian Statistical Institute*, vol. 43, 2004.

[36] R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu, "A study on the use of performance counters to estimate power in microprocessors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 12, pp. 882–886, 2013.

[37] C. Gilberto and M. Margaret, "Power prediction for intel xscale processors using performance monitoring unit events power prediction for intel xscale processors using performance monitoring unit events," in *ISLPED*, vol. 5, pp. 8–10, 2005.

[38] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, pp. 46–55, 2009.

[39] T. Li and L. K. John, "Run-time modeling and estimation of operating system power consumption," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, pp. 160–171, ACM, 2003.

[40] I. Lopez, S. Moore, and V. Weaver, "A prototype sampling interface for papi," in *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, p. 27, ACM, 2015.

[41] M. Poess, R. O. Nambiar, K. Vaid, J. M. Stephens Jr, K. Huppler, and E. Haines, "Energy benchmarks: a detailed analysis," in *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, pp. 131–140, ACM, 2010.

[42] G. Pinto, F. Castor, and Y. D. Liu, "Mining questions about software energy consumption," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 22–31, ACM, 2014.

[43] R. Joseph and M. Martonosi, "Run-time power estimation in high performance microprocessors," in *ISLPED'01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (IEEE Cat. No. 01TH8581)*, pp. 135–140, IEEE, 2001.

[44] S. Kamil, J. Shalf, and E. Strohmaier, "Power efficiency in high performance computing," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, IEEE, 2008.

[45] D. Brooks, V. Tiwari, and M. Martonosi, *Wattch: A framework for architectural-level power analysis and optimizations*, vol. 28. ACM, 2000.

[46] D. Liu and C. Svensson, "Power consumption estimation in cmos vlsi chips," *IEEE Journal of Solid-State Circuits*, vol. 29, no. 6, pp. 663–670, 1994.

[47] D. Marculescu, R. Marculescu, and M. Pedram, "Information theoretic measures of energy consumption at register transfer level," in *Proceedings of the 1995 international symposium on Low power design*, pp. 81–86, ACM, 1995.

[48] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of simplepower: a cycle-accurate energy estimation tool," in *Proceedings of the 37th Annual Design Automation Conference*, pp. 340–345, ACM, 2000.

[49] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, and M. Kandemir, "Using complete machine simulation for software power estimation: The softwatt approach," in *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pp. 141–150, IEEE, 2002.

[50] W. L. Bircher, M. Valluri, J. Law, and L. K. John, "Runtime identification of microprocessor energy saving opportunities," in *ISLPED'05. Proceedings of the 2005 International Symposium on Low Power Electronics and Design, 2005.*, pp. 275–280, IEEE, 2005.

[51] W. Wu, L. Jin, J. Yang, P. Liu, and S.-D. Tan, "A systematic method for functional unit power estimation in microprocessors," in *2006 43rd ACM/IEEE Design Automation Conference*, pp. 554–557, IEEE, 2006.

[52] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya, "Virtual machine power metering and provisioning," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 39–50, ACM, 2010.

[53] G. Dhiman, K. Mihic, and T. Rosing, "A system for online power prediction in virtualized environments using gaussian mixture models," in *Proceedings of the 47th Design Automation Conference*, pp. 807–812, ACM, 2010.

[54] C. Wilke, S. Götz, and S. Richly, "Jouleunit: a generic framework for software energy profiling and testing," in *Proceedings of the 2013 workshop on Green in/by software engineering*, pp. 9–14, ACM, 2013.

[55] S. Wang, H. Chen, and W. Shi, "Span: A software power analyzer for multicore computer systems," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 1, pp. 23–34, 2011.

[56] J. Mair, D. Eyers, Z. Huang, and H. Zhang, "Myths in power estimation with performance monitoring counters," *Sustainable Computing: Informatics and Systems*, vol. 4, no. 2, pp. 83–93, 2014.

[57] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*, pp. 2–10, IEEE, 1999.

[58] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 93, IEEE Computer Society, 2003.

[59] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. W. Cameron, "Powerpack: Energy profiling and analysis of high-performance systems and applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 5, pp. 658–671, 2009.

[60] J. W. Tukey, *Exploratory data analysis*, vol. 2. Reading, Mass., 1977.

[61] K. Liu, G. Pinto, and Y. D. Liu, "Data-oriented characterization of application-level energy optimization," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 316–331, Springer, 2015.

[62] M. Kumar, Y. Li, and W. Shi, "Energy consumption in java: An early experience," in *Green and Sustainable Computing Conference (IGSC), 2017 Eighth International*, pp. 1–8, IEEE, 2017.

[63] M. Kumar, *Energy Efficiency of Java Programming Language*. Wayne State University, 2018.

[64] A. Rodriguez, C. Mateos, and A. Zunino, "Mobile devices-aware refactorings for scientific computational kernels," in *13Th Argentine Symposium on Technology, AST*, 2012.

[65] R. A. A. Pereira, *Energyware engineering: techniques and tools for green software development*. PhD thesis, Universidade do Minho, 2018.

[66] J. Shirazi, *Java performance tuning*. " O'Reilly Media, Inc. ", 2003.

[67] H. Malik, P. Zhao, and M. Godfrey, "Going green: An exploratory analysis of energy-related questions," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, pp. 418–421, IEEE Press, 2015.

[68] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?," *IEEE Software*, vol. 33, no. 3, pp. 83–89, 2016.

[69] G. Procaccianti, P. Lago, A. Vetrò, D. M. Fernández, and R. Wieringa, "The green lab: Experimentation in software energy efficiency," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pp. 941–942, IEEE Press, 2015.

[70] C. Bunse, H. Höpfner, E. Mansour, and S. Roychoudhury, "Exploring the energy consumption of data sorting algorithms in embedded and mobile environments," in *Tenth Int. Conf. on Mobile Data Management: Systems, Services and Middleware*, pp. 600–607, IEEE, 2009.

[71] S. Götz, C. Wilke, S. Richly, and U. Aßmann, "Approximating quality contracts for energy auto-tuning software," in *Proceedings of the First International Workshop on Green and Sustainable Software*, pp. 8–14, IEEE Press, 2012.

[72] S. Götz, C. Wilke, M. Schmidt, S. Cech, and U. Aßmann, "Towards energy auto tuning," in *Proceedings of the 1st Annual International Conference on Green Information Technology*, 2010.

[73] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong, "Program energy efficiency: The impact of language, compiler and implementation choices," in *Green Computing Conference (IGCC), 2014 International*, pp. 1–6, IEEE, 2014.

[74] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 345–360, 2014.

[75] H. Ribic and Y. D. Liu, "Energy-efficient work-stealing language runtimes," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 513–528, ACM, 2014.

[76] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, (New York, NY, USA), pp. 36:1–36:10, ACM, 2014.

[77] J. Michanan, R. Dewri, and M. J. Rutherford, "Predicting data structures for energy efficient computing," in *Green Computing Conference and Sustainable Computing Conference (IGSC), 2015 Sixth International*, pp. 1–8, IEEE, 2015.

[78] G. Pinto, K. Liu, F. Castor, and Y. D. Liu, "A comprehensive study on the energy efficiency of java thread-safe collections," *Journal of Systems and Software*, 2016.

[79] R. Pereira, M. Couto, J. a. Saraiva, J. Cunha, and J. a. P. Fernandes, "The influence of the java collection framework on overall energy consumption," in *Proceedings of the 5th International Workshop on Green and Sustainable Software*, GREENS '16, (New York, NY, USA), pp. 15–21, ACM, 2016.

[80] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *Proc. 38th Int. Conf. on Software Engineering*, pp. 225–236, ACM, 2016.

[81] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 615–629, 2017.

[82] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes, "Haskell in green land: Analyzing the energy behavior of a purely functional language," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1, pp. 517–528, IEEE, 2016.

[83] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, pp. 93–95, 2008.

[84] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *European conference on object-oriented programming*, pp. 220–242, Springer, 1997.

[85] J. M. Horcas, M. Pinto, L. Fuentes, and N. Gámez, "Self-adaptive energy-efficent applications: the hadas developing approach," in *Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence & Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2017 IEEE 15th Intl*, pp. 828–835, IEEE, 2017.

[86] L. Cruz and R. Abreu, "Performance-based guidelines for energy efficient mobile applications," in *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*, pp. 46–57, IEEE, 2017.

[87] S. Maleki, C. Fu, A. Banotra, and Z. Zong, "Understanding the impact of object oriented programming and design patterns on energy efficiency," in *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*, pp. 1–6, IEEE, 2017.

[88] M. Kumar and W. Shi, "Energy consumption analysis of java command-line options," in *Green and Sustainable Computing Conference (IGSC), 2019 Tenth International*, IEEE, 2019.

[89] S. Oaks, *Java Performance: The Definitive Guide: Getting the Most Out of Your Code*. " O'Reilly Media, Inc.", 2014.

[90] P. Maldikar, S.-H. Li, and K. Chow, "Java performance mysteries," in *ITM Web of Conferences*, vol. 7, p. 09015, EDP Sciences, 2016.

[91] S. B. Joshi, "Apache hadoop performance-tuning methodologies and best practices," in *Proceedings of the 3rd acm/spec international conference on performance engineering*, pp. 241–242, ACM, 2012.

[92] H. Oi, "Power-performance analysis of jvm implementations," in *Information Technology and Multimedia (ICIM), 2011 International Conference on*, pp. 1–7, IEEE, 2011.

[93] "Eclipse metrics plug-in 1.3.6," 2019. Accessed: 2019-17-09.

[94] "Class dependency analyzer," 2019. Accessed: 2019-17-09.

[95] S. Le Cessie and J. C. Van Houwelingen, "Ridge estimators in logistic regression," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 41, no. 1, pp. 191–201, 1992.

[96] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," 1998.

[97] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy, "Improvements to platt's smo algorithm for svm classifier design," *Neural computation*, vol. 13, no. 3, pp. 637–649, 2001.

[98] "Moa," 2019. Accessed: 2019-17-09.

[99] A. Hindle, "Green software engineering: the curse of methodology," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 5, pp. 46–55, IEEE, 2016.

[100] I. Manotas, L. Pollock, and J. Clause, "Seeds: a software engineer's energy-optimization decision support framework," in *Proc. 36th Int. Conf. on Software Engineering*, pp. 503–514, ACM, 2014.

[101] O. Shacham, M. Vechev, and E. Yahav, "Chameleon: adaptive selection of collections," in *ACM Sigplan Notices*, vol. 44, pp. 408–418, ACM, 2009.

[102] K. Aggarwal, A. Hindle, and E. Stroulia, "Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pp. 311–320, IEEE, 2015.

[103] H. S. Zhu, C. Lin, and Y. D. Liu, "A programming model for sustainable software," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, pp. 767–777, IEEE, 2015.

[104] H. M. Alvi, H. Sahar, A. A. Bangash, and M. O. Beg, "Ensights: A tool for energy aware software development," in *Emerging Technologies (ICET), 2017 13th International Conference on*, pp. 1–6, IEEE, 2017.

ABSTRACT

IMPROVING ENERGY CONSUMPTION OF JAVA PROGRAMS

by

MOHIT KUMAR

December 2019

Advisor: Dr. Weisong Shi

Major: Computer Science

Degree: Doctor of Philosophy

Information and Communications Technologies (ICT) amounts for 10% of the world energy which will keep on growing in the future and 3% of the overall carbon footprint which is now more than the level of CO_2 emission as that of the aviation industry. For many past years, the focus was on hardware to optimize the energy consumption of ICT systems. This includes dynamic adaptation of hardware techniques such as fine-grain clock gating, power gating, and dynamic voltage/frequency scaling. However, recent demands of exascale computation, as well as the increasing carbon footprint, require new breakthroughs to make ICT systems more energy-efficient. This is not possible by only making the hardware energy-efficient. As a result, the focus is shifting on software now. Software is one of the most critical bottlenecks while trying to optimize the energy consumption of any ICT system.

Software energy consumption can be optimized in several ways like choosing the energy-efficient option in a programming language, using an energy-efficient programming language or choosing an energy-efficient compiling option. In this work, we concentrate on the energy-efficient options and command-line options to optimize software energy consumption. Today's programming languages provide software developers with several options to perform the same task. For example, in Java, an Array can be copied to other Array either manually or using Java methods. However, not every option available is energy-efficient and the software developers lack the knowledge to choose the best energy-efficient option. We perform various analyses to decide on choos-

ing the best option for different components of Java programming language. These components include data types, operators, control statements, String, exceptions, objects, and Arrays.

Java has different command-line options that can be used to tune the JVM. These options can significantly affect the energy behavior of Java applications. We conduct a comprehensive study to evaluate the energy efficiency of Java command-line options. We first stabilize the idle energy consumption of two ICT systems and then evaluate the active energy consumption of SPECjvm2008 benchmarks using different JDKs (Open and Oracle) and Java command-line options. The Java command-line options include `client`, `server`, `Xbatch`, `Xcomp`, `Xfuture`, `Xint`, `Xmixed`, `Xrs`, `AggressiveOpts`, `AggressiveHeap`, `Inline`, `AlwaysPreTouch`, `Xnoclassgc`, `UseSerialGC`, `UseParallelGC`, `UseConcMarkSweepGC`, and `UseG1GC`.

Next, we present Java Energy Profiler and Optimizer (JEPO) tool to help software developers to write energy-efficient code. This tool is an Eclipse IDE plugin and provides energy efficiency suggestions for Java programming language. It can provide suggestions dynamically while writing code or statically to refactor already written code. For providing suggestions, it analyzes each line of Java file and matches it to the pool of suggestions. JEPO can also help the software developers to automatically measure energy consumption at method granularity to determine the energy-hungry Java methods in software. We hope our findings and tool can help software developers to write energy-efficient code in the future.

AUTOBIOGRAPHICAL STATEMENT

MOHIT KUMAR

Mohit Kumar received his Bachelor degree in Computer Science and Engineering from Maharshi Dayanand University, Rohtak, India. He received his MS in Computer Science from Wayne State University, Detroit. He is currently a Ph.D. candidate in the Computer Science department at Wayne State University, Detroit. His research interests include energy efficiency, software engineering, and high-performance computing.