Energy Consumption Analysis of Java Command-line Options

Mohit Kumar
Mobile and Internet Systems Laboratory
Wayne State University
Detroit, USA
Email: mohitkumar@wayne.edu

Weisong Shi
Mobile and Internet Systems Laboratory
Wayne State University
Detroit, USA
Email: weisong@wayne.edu

***Abstract*—In 2018 Turing Award lecture, John L. Hennessy discusses software-centric opportunities to save Moore's law. The software is a major consumer of energy in ICT, IoT, and edge systems, but even then the research to make it energy efficient remains fractional. Java is one of the most commonly used languages to develop software for these systems. Java has various command-line options that an application user can use for JVM tuning to enhance the performance of an application. However, there is no study about how these Java command-line options impact the energy consumption of an application. In this work, we explore the impact of various Java command-line options on SPECjvm2008 benchmarks in terms of energy consumption and execution time using different JDKs. Our key findings are: 1) Oracle JDK is more energy efficient than Open JDK, 2) `Xint` command-line option is the least energy efficient, 3) `UseG1GC` command-line option is the most energy efficient, and 4) Active energy and execution time show a high correlation.**

***Index Terms*—Java, JDK, Command-line, Energy-efficiency**

I. Introduction

Information and Communications Technologies (ICT) amounts for 10% of the world energy which will keep on growing in future [1] and 3% of the overall carbon footprint which is now more than the level of CO_2 emission as that of aviation industry [2]. Most of the green IT initiative in ICT concentrate on the hardware part resulting in significant reduction of hardware energy consumption. However, a lot needs to be done to improve the energy efficiency of the software. An energy-aware software that can optimize execution time can help to make ICT systems more energy efficient. Software power savings are considered to be greater than the power saving in hardware, but they are harder to achieve [3].

Java is one of the most commonly-used languages in ICT systems. 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, in this paper, we conduct a comprehensive study to evaluate the energy efficiency of Java command-line options. We use Intel Running Average Power Limit (RAPL) technology to log the energy consumption values. We first optimize 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`. Our work answers the following questions:

- Do same versions of Open and Oracle JDK have same energy efficiency?
- Which command-line option has the lowest energy efficiency of JVM?
- Which command-line option has the highest energy efficiency of JVM?
- What is the relation between active energy and execution time?

Answers to these questions will help software users to tune the JVM for energy efficiency. To the best of our knowledge, no research optimized the idle energy consumption of a system while evaluating the energy efficiency of different JDKs and Java command-line options. Not optimizing the idle energy consumption results in outliers which cause inaccurate measurements. We optimize the idle energy consumption and conduct statistical tests - independent sample t-test and one way ANOVA [4]- to compare energy consumption in cases where it is hard to decide whether the values are the same or not. 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.
- Active energy and execution time show a high correlation with a maximum value of 0.98 and a minimum value of

TABLE I
RAPL DOMAINS

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

0.94.

The rest of the paper is organized as follows. Section 2, provides the background for energy measurement, optimization and SPECjvm2008 benchmarks. Section 3, describes the energy consumption measurement setup. Section 4, investigates the energy consumption traits of Java command-line options. Section 5, analyzes the relation between active energy and execution time. Section 6, discusses related work in the field. Section 7, concludes the paper and discusses the future work in energy efficient software.

II. BACKGROUND

In this section, we first introduce energy-related terms, `perf`, `systemd`, and `systemctl`. Then, we describe SPECjvm2008 benchmarks in detail.

The *idle power* is defined as the amount of power consumed by a system when it is not performing any task. As defined in [5], it is the sum of static and dynamic power, as systems have a different number of background processes running all the time. In this work, we try to reduce the dynamic power to stabilize the idle power. The *active* power is defined as the amount of power 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 CPU-level components as listed in Table I. It uses a software power model which estimates the energy consumption by leveraging hardware performance counters. A user can configure and read RAPL information through Mode Specific Registers in privileged kernel mode. We use Linux `perf` tool which leverages RAPL technology to measure energy consumption.

`systemd` is a critical suite of software for the Linux operating system that manages and operates various units like service, target, path, mount etc. Some of the units can trigger other units and work together to add functionality. In this work, we utilize the service unit to optimize the idle energy consumption of the whole system as it is the most commonly utilized service by system administrators. We use `systemctl` command to start or stop a service.

SPECjvm2008 consists of 11 benchmarks which are split into sub-benchmarks as shown in Table II. 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 FileManger 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 docu-

TABLE II
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

TABLE III
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
Number of cores	4	2
Number of threads	8	4
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
JDK	OpenJDK 64-Bit Server VM	OpenJDK 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

ments. `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.

III. SET UP

We leverage two different ICT systems to conduct our experiments: Intel Fog Node(IFN) and Laptop. The configuration of these two systems is presented in Table III. We use the same versions of Open and Oracle JDK for this study. Oracle JDK is expected to consume lesser energy as it is maintained by the same group of coders and is more consistent. 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.

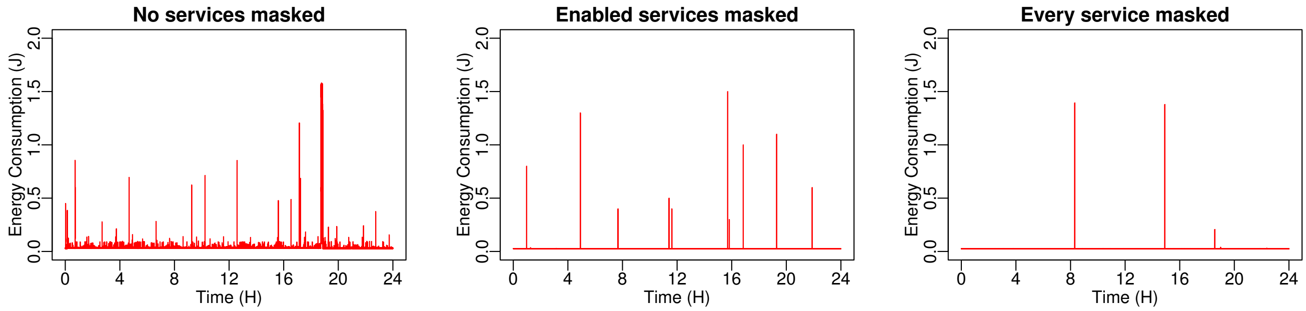
The total energy consumption is the sum of active and idle energy. As `perf` reports the total energy, one can subtract the idle energy out of the total energy to find the active energy of an application. However, the idle energy of a system can vary a lot due to the background services running on an operating system. These variations make it hard to measure an accurate idle energy of a system. To measure the idle energy of both systems considered here, we conduct an experiment in which we first optimize 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 optimization for 24 hours with a sampling rate of 10Hz to determine how the idle energy varies. In Fig. 1a and 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.

Next, we stop the background services using `systemctl` command to optimize 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 optimize 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 were able to optimize the idle energy with a very few outliers as shown in Fig. 1b and 2b. We go one step further and mask all the disabled services and then obtain fewer outliers, as shown in Fig. 1c and 2c. However, outliers were still there as we can't mask some of the services like log in, user manager and dbus.

The next step is to remove the outliers from the 24-hour dataset shown in Fig. 1c and 2c and calculate the mean of the values. We use Tukey's method to remove the outliers [6]. For the IFN and the 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 IFN are shown in Fig. 3a and 3b, and for the Laptop in Fig. 4a and 4b. The standard deviation for the IFN is 0.0006, and for the Laptop is 0.0007. The standard deviation indicates that both datasets have very low variation. The mean idle energy consumed per one-tenth of a second for the IFN and the Laptop was 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.

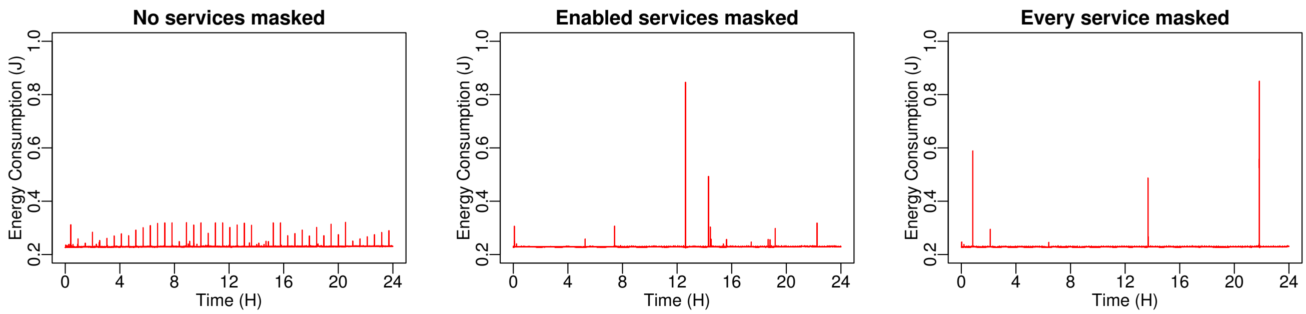
IV. ENERGY CONSUMPTION ANALYSIS

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 the outliers measurements with new measurements and again check for outliers. We repeat this process until no outlier is left. Next, we subtract the idle energy from the total energy consumption



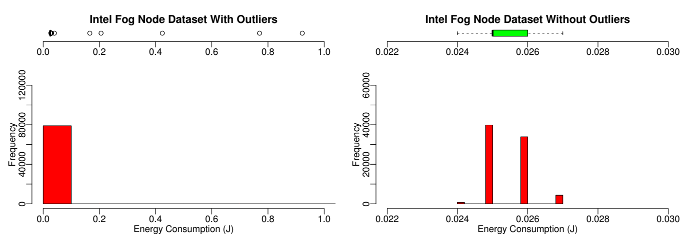
(a) (b) (c)

Fig. 1. Intel Fog Node package idle energy consumption: (a) with all services unmasked; (b) with enabled services masked, and (c) with enabled and disabled services masked.



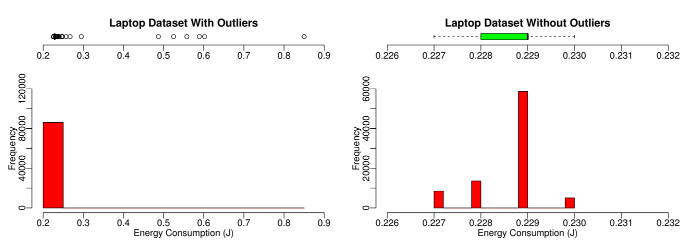
(a) (b) (c)

Fig. 2. Laptop package idle energy consumption: (a) with all services unmasked; (b) with enabled services masked, and (c) with enabled and disabled services masked.



(a) (b)

Fig. 3. Intel Fog Node dataset: (a) with outliers, and (b) without outliers.



(a) (b)

Fig. 4. Laptop dataset: (a) with outliers, and (b) without outliers.

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. 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 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 the paper 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) in Joule and execution time (T) in second. `server` command-line option refers to default mode as both systems use server JVM by default. Tables for only specific Java command-line options are included due to space limit.

A. *`-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 IV and V. For the `client`, we can see that 20 benchmarks on the IFN and 19 benchmarks on the Laptop have lower

TABLE IV
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	243.99
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	21.99	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	0.99	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

TABLE V
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	242.99	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	10948.76	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	2365.99	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	2786.99	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	37.99	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	0.99	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

energy consumption when executed on Oracle JDK instead of Open JDK. For the `server`, these number jump to 25 and 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 consumption for the smaller dataset instead of the larger dataset. This happens because small dataset results in up to five times higher ops/m than large 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.

B. `-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. 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. `Xcomp` forces the compilation of a method on the first invocation instead of doing that after a set threshold of interpreted method invocations. 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 VI, 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 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. 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 energy efficiency of most benchmarks for

TABLE VI
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

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. 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 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.

C. `-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. 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 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.

D. `-XX:+AggressiveOpts` and `-XX:+AggressiveHeap`

`AggressiveOpts` option enables the use of aggressive performance optimization features. 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 benchmarks than the default server option for both JDKs. `AggressiveHeap` option enables Java heap optimization which is optimal for long-running computation-intensive jobs. 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.

E. `-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. 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.

F. `-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. 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.

G. `-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. 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 garbage collection. 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 Open JDK for 16 benchmarks on the IFN and 21 benchmarks on the Laptop. 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 VII. For both systems, `UseG1GC` consumes up to 14% lesser energy than the default mode for 17 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.

V. Active Energy & Execution Time

In this section, we analyze the correlation between active energy and execution time for each command-line option for each JDK and system. The values for the correlation are shown in Fig. 5. The first thing we notice is that active energy and execution 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 Open JDK on the Laptop. The high correlation is expected because we optimize 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.

TABLE VII
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	2854.99	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	18.99	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

VI. Related Work

Software energy efficiency research has escalated in the past few years. The energy consumption of sorting algorithms in embedded and mobile environments was examined in [7]. Quality contracts that express dependencies between software and hardware components for energy efficiency of software systems were used in [8] and [9]. The impact of languages, compiler optimization, and implementation choices on Fast Fourier Transform, Linked List Insertion/Deletion, and Quicksort was examined in [10]. SEEDS and Chameleon frameworks for automating code-level changes and optimizing Java applications were introduced in [11] and [12]. Java thread management constructs - explicit thread creation, fixed-size thread pooling, and work stealing - relation to energy consumption was explored in [13]. Application programmers were shown to be aware of software energy consumption problems in [14]. Energy efficient multithreaded program runtimes are shown to save 11-12% of energy in [15]. The change in the energy efficiency of software by using different classes that implement the same interface was investigated in [16]. Software energy efficiency research challenges are discussed in [17]. Java collections were studied in terms of energy efficiency in [18] and [19]. In [20] and [21], the authors investigated energy consumption of Java's data types, operators, control statements and, exception levels but did not consider inaccuracies due to variation in idle energy. In this paper, we handle such inaccuracies by optimizing the idle energy consumption and then subtracting it from the total energy consumption to calculate the active energy consump-

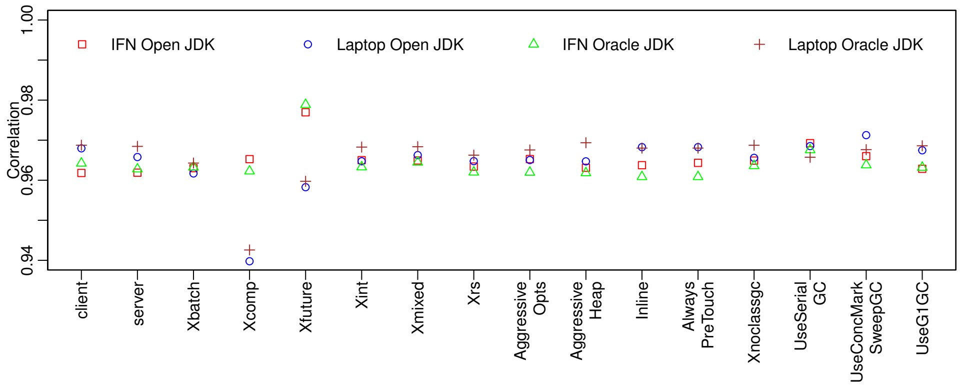


Fig. 5. Active energy & execution time correlation.

tion. OpenJDK and IBM I9 performance-power analysis is presented in [22] using SPECjvm2008 Base run category. It is the closest work we can find, however, it neither optimize the idle energy and nor analyze Java command-line options in terms of energy efficiency.

VII. Conclusion & Future Work

In this paper, 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 optimize 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.

Acknowledgment

This work is supported in part by National Science Foundation (NSF) grant CNS-1561216.

References

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

[2] 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.

[3] 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*. IEEE Press, 1996, p. 175.

[4] H. M. Park, "Comparing group means: T-tests and one-way anova using stata, sas, r, and spss," *The University Information Technology Services (UITS) Center for Statistical and Mathematical Computing, Indiana University*, 2009.

[5] 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*. IEEE, 2011, pp. 1–6.

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

[7] 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*. IEEE, 2009, pp. 600–607.

[8] 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*. IEEE Press, 2012, pp. 8–14.

[9] 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.

[10] 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*. IEEE, 2014, pp. 1–6.

[11] 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*. ACM, 2014, pp. 503–514.

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

[13] 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.

[14] ——, "Mining questions about software energy consumption," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 22–31.

[15] 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*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 513–528. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541971

[16] 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*. IEEE, 2015, pp. 1–8.

[17] 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*. IEEE Press, 2015, pp. 941–942.

[18] 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.

[19] 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*. ACM, 2016, pp. 225–236.

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

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

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