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 CO₂ emission as that of
aviation industry [2]. Most of the green IT initiative in ICT
concentrate on the hardware part resulting in significant reduc-
ion 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
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 FileManager 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 MD5 with RSA, SHA1 with RSA, SHA1 with DSA and SHA256 with RSA 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 consists 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 III  
SYSTEM SPECIFICATION

<table>
<thead>
<tr>
<th>System Component</th>
<th>Intel Fog Node Configuration</th>
<th>Laptop Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel(R) Xeon(R) E3-1225 v5</td>
<td>Intel(R) Core(TM) i5-3317U v5</td>
</tr>
<tr>
<td>Number of cores</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Number of threads</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Kernel</td>
<td>4.13.0-37-generic</td>
<td>4.4.0-116-generic</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu Server 16.04.4 LTS</td>
<td>Ubuntu Server 16.04.3 LTS</td>
</tr>
<tr>
<td>CPU governor</td>
<td>powersave</td>
<td>powersave</td>
</tr>
<tr>
<td>Memory</td>
<td>32GB SODIMM 2133 MHz</td>
<td>4GB DDR3 1600 MHz</td>
</tr>
<tr>
<td>JDK</td>
<td>OpenJDK 64-Bit Server VM</td>
<td>OpenJDK 64-Bit Server VM</td>
</tr>
<tr>
<td>JDK build</td>
<td>251519152</td>
<td>251519152</td>
</tr>
<tr>
<td>JDK version</td>
<td>1.8.0.151</td>
<td>1.8.0.151</td>
</tr>
<tr>
<td>Initial Heap Size</td>
<td>220MB</td>
<td>0MB</td>
</tr>
<tr>
<td>Maximum Heap Size</td>
<td>8.0GB</td>
<td>1GB</td>
</tr>
</tbody>
</table>

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.

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:

```bash
$ sudo perf stat -a -r 1 -I 100\n  -e 'power/energy-pkg/\n  -o pack.txt\n  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.
to determine 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 JVM 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...
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 - start 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 efficiency variation of different benchmarks.

B. -Xbatch, -Xcomp, -Xint, -Xfuture, and -Xmixed

JVM runs a method in interpreted mode until the 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 also results in stricter class-file format checks. For crypto.aes and derby.java, Xint results in significant decrease in energy consumption than the default mode for most benchmarks. Xint causes the highest variation among the benchmarks with up to 125% increase in the energy than the default. Interestingly, Xint consumes up to 28% lesser energy than the default mode for lua.small.

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 for 24 benchmarks.
### TABLE VI
ENERGY CONSUMPTION FOR XINT option

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Open</th>
<th>Xint</th>
<th>Xint</th>
<th>Laptop</th>
<th>Xint</th>
<th>Laptop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IFN</td>
<td>IFN</td>
<td>XINT</td>
<td>IFN</td>
<td>XINT</td>
<td>IFN</td>
</tr>
<tr>
<td></td>
<td>in</td>
<td>out</td>
<td>in</td>
<td>out</td>
<td>in</td>
<td>out</td>
</tr>
<tr>
<td>compress</td>
<td>10026.54</td>
<td>300.11</td>
<td>382.80</td>
<td>344.55</td>
<td>10044.47</td>
<td>300.76</td>
</tr>
<tr>
<td>crypto.jar</td>
<td>1328.14</td>
<td>731.08</td>
<td>5904.80</td>
<td>385.85</td>
<td>12820.80</td>
<td>335.64</td>
</tr>
<tr>
<td>crypto.jar</td>
<td>11158.43</td>
<td>280.93</td>
<td>3221.55</td>
<td>326.39</td>
<td>2255.85</td>
<td>286.66</td>
</tr>
<tr>
<td>crypto.jar</td>
<td>10556.55</td>
<td>280.93</td>
<td>3221.55</td>
<td>326.39</td>
<td>2255.85</td>
<td>286.66</td>
</tr>
<tr>
<td>dell</td>
<td>18794.08</td>
<td>3131.60</td>
<td>5915.05</td>
<td>396.79</td>
<td>14301.37</td>
<td>314.11</td>
</tr>
<tr>
<td>mpegand</td>
<td>11065.41</td>
<td>283.70</td>
<td>3288.16</td>
<td>328.00</td>
<td>11439.94</td>
<td>282.00</td>
</tr>
<tr>
<td>mark</td>
<td>11158.55</td>
<td>285.44</td>
<td>3283.84</td>
<td>272.29</td>
<td>12361.74</td>
<td>282.42</td>
</tr>
<tr>
<td>ln</td>
<td>10480.18</td>
<td>288.97</td>
<td>2925.48</td>
<td>288.11</td>
<td>10462.62</td>
<td>288.74</td>
</tr>
<tr>
<td>mke</td>
<td>12373.38</td>
<td>291.06</td>
<td>2929.25</td>
<td>313.89</td>
<td>12373.51</td>
<td>292.36</td>
</tr>
<tr>
<td>mke</td>
<td>10991.32</td>
<td>334.18</td>
<td>3531.03</td>
<td>364.24</td>
<td>10995.95</td>
<td>334.69</td>
</tr>
<tr>
<td>mke</td>
<td>10427.10</td>
<td>285.26</td>
<td>3071.74</td>
<td>284.22</td>
<td>10391.35</td>
<td>285.55</td>
</tr>
<tr>
<td>mke</td>
<td>13592.80</td>
<td>343.62</td>
<td>3459.25</td>
<td>349.52</td>
<td>13439.79</td>
<td>344.92</td>
</tr>
<tr>
<td>mke</td>
<td>12068.71</td>
<td>291.05</td>
<td>3122.02</td>
<td>300.26</td>
<td>12028.71</td>
<td>291.34</td>
</tr>
<tr>
<td>mke</td>
<td>12284.09</td>
<td>291.22</td>
<td>3141.33</td>
<td>375.06</td>
<td>13150.00</td>
<td>291.43</td>
</tr>
<tr>
<td>mke</td>
<td>12301.92</td>
<td>290.03</td>
<td>3150.53</td>
<td>280.03</td>
<td>12306.04</td>
<td>290.78</td>
</tr>
</tbody>
</table>

**Note:**
- 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, except startup benchmark where the default mode is more energy efficient for each sub-benchmark. 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 option results in the better energy efficiency of 20 benchmarks than the default mode for both systems. For the Laptop, Xmixed option results in the better energy efficiency 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 20 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 shows the better energy efficiency than Open JDK. 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 tracing 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, 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 as 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.

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 dependences 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 - relating 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-
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