Evaluating the Impact of Java Virtual Machines on Energy Consumption

Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, Joel Penhoat

▶ **To cite this version:**

Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, Joel Penhoat. Evaluating the Impact of Java Virtual Machines on Energy Consumption. 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Oct 2021, Bari, Italy. hal-03275286

HAL Id: hal-03275286
https://hal.inria.fr/hal-03275286

Submitted on 31 Aug 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Evaluating the Impact of Java Virtual Machines on Energy Consumption

Zakaria Ournani
Orange Labs / Inria / Univ. Lille
zakaria.ournani@inria.fr

Mohammed Chakib Belgaid
Inria / Univ. Lille
chakib.belgaid@inria.fr

Romain Rouvoy
Univ. Lille / Inria / IUF
romain.rouvoy@univ-lille.fr

Pierre Rust
Orange Labs
pierre.rust@orange.com

Joël Penhoat
Orange Labs
joel.penhoat@orange.com

ABSTRACT

Background. The *Java Virtual Machine* (JVM) platforms have known multiple evolutions along the last decades to enhance both the performance they exhibit and the features they offer. With regards to energy consumption, few studies have investigated the energy consumption of code and data structures. Yet, we keep missing an evaluation of the energy efficiency of existing JVM platforms and an identification of the configurations that minimize the energy consumption of software hosted on the JVM.

Aims. The purpose of this paper is to investigate the variations in energy consumption between different JVM distributions and parameters to help developers configure the least consuming environment for their Java application.

Method. We thus assess the energy consumption of some of the most popular and supported JVM platforms using 12 Java benchmarks that explore different performance objectives. Moreover, we investigate the impact of the different JVM parameters and configurations on the energy consumption of software.

Results. Our results show that some JVM platforms can exhibit up to 100% more energy consumption. JVM configurations can also play a substantial role to reduce the energy consumption during the software execution. Interestingly, the default configuration of the garbage collector was energy efficient in only 50% of our experiments.

Conclusion. Finally, we provide an Open source tool, named J-Referral that recommends an energy-efficient JVM distribution and configuration for any Java application.

KEYWORDS

JVM Energy Consumption; JVM Energy Efficiency

1 INTRODUCTION

Software services are widely deployed to support our daily activities, being mobile or hosted in the cloud. Yet, beyond this undeniable success, the environmental impact of Information and Communication Technology (ICT) is raising concerns and calls for solutions to reduce the energy footprint of software services [22].

Software developers often report that such solutions should come from more energy-efficient hardware components or optimized algorithms [15, 24] but, given the complexity of modern software environments, the composition of software layers makes this sustainability objective particularly challenging.

Given this context, this paper more specifically investigates the impact of one of these layers, the runtime environment and its settings, on the energy consumption of a hosted software service. More precisely, we aim at revealing the importance of carefully selecting and configuring the *Java Virtual Machine* (JVM) to reduce the energy consumption of any software service built from any language compatible with the Java ecosystem (Java, Kotlin, Scala, Groovy, Clojure, Jython, etc.).

The empirical study we conduct in this paper reports on the energy footprint of several versions of popular JVM distributions that are freely available for download. Beyond the choice of an appropriate runtime and its most energy-efficient version, we also consider the impact of exposed JVM settings to maximize the energy savings for a given software service. The observations of this study aim to quantify the role played by internal JVM mechanisms, like the *Just in Time* (JIT) compiler and the *Garbage Collector* (GC), in the reduction of the energy consumption of hosted applications. More formally, we formulate the following research questions:

RQ 1: *What is the impact of existing JVM distributions on the energy consumption of Java-based software services?*

RQ 2: *What are the relevant JVM settings that can reduce the energy consumption of a given software service?*

This paper comes with a set of contributions that can be summarized as:

(a) Reporting on the energy-efficiency of a large panel of JVM when running acknowledged benchmarks,
(b) Identifying and assessing the key JVM settings that can influence the energy consumption of a software service,
(c) Sharing guidelines and prerequisites that will help in configuring the most energy-efficiency environment before deployment,
(d) Providing a JVM benchmarking environment to evaluate the energy-efficiency of upcoming JVM distributions and their settings,
(e) Delivering an open-source software (OSS) tool, named J-Referral, to recommend the most energy efficient JVM distribution among 85 JVMs/versions and hundreds of configurations for Java applications.

The remainder of this paper is organized as follows. Section 3 introduces the experimental protocol and methodology (hardware, projects, tools, and methodology) we adopted in this study. Section 4 analyzes the results of our experiments on the energy consumption of the different JVM configurations. Section 5 discusses the related

works to reduce the energy consumption of Java-based Software Services. Finally, Section 6 covers our conclusions.

2 THE JAVA VIRTUAL MACHINE

Java was originally developed by James Gosling at Sun Microsystems and released in 1995, before being acquired by Oracle in 2010. One key design goal of Java is portability, which means that Java applications must run similarly on any combination of hardware and operating system with adequate runtime support. This is achieved by compiling the Java language code to an intermediate representation, called Java *bytecode*, instead of machine code. Java bytecode instructions are analogous to machine code, but executed by a *Java Virtual Machine* (JVM), which is specific to the host machine. For example, Oracle keeps offering the HotSpot JVM, while the official reference implementation is now the OpenJDK JVM—a free and open-source software used by most developers.

Programs written in Java have a reputation for being slower and requiring more memory than those written in C++, but *Just-in-Time* (JIT) compilation, embedded in the JVM, delivers a boost of performance by opportunistically compiling bytecode to machine code at runtime. The JIT combines two compilers, C1 and C2 (also known as Client & Server VM), which are triggered based on the activity of the hosted application. Additionally, Java uses an automatic *garbage collector* (GC) to manage memory in the object lifecycle and recovering the memory once objects are no longer in use. Each JVM usually includes multiple GC, each designed to satisfy different requirements.

By default, HotSpot uses both C1 and C2 as tiered compilers,[*] and the *Garbage-First* (G1) GC with a maximum number of GC threads limited by available CPU resources and heap size, whose initial size is set $1/64^{th}$ of physical memory and maximum size may reach up to $1/4^{th}$ of physical memory.[†]

At the time of writing this paper, the latest JVM version is Java 15, released in September 2020, while Java 11 is the current *Long-Term Support* (LTS) version. As Java 9, 10, 12, and 13 are no longer supported, Oracle advises developers to immediately transition to the latest version (currently Java 15), or an LTS release.

Beyond HotSpot, one can observe that the initial JVM design leads to numerous initiatives to improve the performances of Java applications, including new hotswapping strategies—with the *Dynamic Code Evolution Virtual Machine* (DCE VM) [25]—or alternative JIT—with GraalVM.[‡] This also includes alternative implementations, like IBM J9 JVM, which is currently distributed as part of the Eclipse foundation, and known as J9.[§] Given the wide diversity of distributions and related settings, this paper aims to study the impact of the features implemented by available JVM distributions on the energy consumption of the hosted Java software services.

3 EXPERIMENTAL PROTOCOL

To investigate the effect that could have the JVM distribution choice and/or parameters on software energy consumption, we conducted a wide set of experiments on a cluster of machines and using several established Java benchmarks and JVM configurations.

[*] http://www.ittc.ku.edu/~kulkarni/teaching/EECS768/19-Spring/Idhaya_Elango_JIT.pdf

[†] https://docs.oracle.com/en/java/javase/15/gctuning/ergonomics.html

[‡] https://www.graalvm.org

[§] https://www.eclipse.org/openj9

Table 1: List of selected JVM distributions.

Distribution	Provider	Support	Selected versions
HotSpot	**Adopt OpenJDK**	All	8.0.275, 11.0.9, 12.0.2, 13.0.2, 14.0.2, 15.0.1
HotSpot	**Oracle**	All	8.0.265, 9.0.4, 10.0.2, 11.0.2, 12.0.2, 13.0.2, 14.0.2, 15.0.1, 16.ea.24
Zulu	**Azul Systems**	All	8.0.272, 9.0.7, 10.0.2, 11.0.9, 12.0.2, 13.0.5, 14.0.2, 15.0.1
SapMachine	**SAP**	All	11.0.9, 12.0.2, 13.0.2, 14.0.2, 15.0.1
Librca	**BellSoft**	All	8.0.275, 11.0.9, 12.0.2, 13.0.2, 14.0.2, 15.0.1
Corretto	**Amazon**	Mjr	8.0.275, 11.0.9, 15.0.1
HotSpot	**Trava OpenJDK**	LTS	8.0.232, 11.0.9
Dragonwell	**Alibaba**	LTS	8.0.272, 11.0.8
OpenJ9	**Eclipse**	All	8.0.275, 11.0.9, 12.0.2, 13.0.2, 14.0.2, 15.0.1
GraalVM	**Oracle**	LTS	19.3.4.r8, 19.3.4.r11, 20.2.0.r8, 20.2.0.r11
Mandrel	**Redhat**	LTS	20.2.0.0

Hardware Settings. To report on reproducible measurements, we used the cluster Dahu of the G5K platform [2] for most of our experiments. This cluster is composed of 32 identical compute nodes, which are equipped with 2 Intel Xeon Gold 6130 and 192 GB of RAM. Our experimental protocol enforces that the software under test is the only process executed on the node configured with a very minimal Linux Debian 9 (4.9.0 kernel version). The minimal OS configuration ensures that only mandatory services and daemons are kept active to conduct robust experiments and reduce the factors that can affect the energy consumption measurements during our experiments [15].

Energy Measurements. We used Intel RAPL as a physical power meter to analyze the energy consumption of the CPU package and the DRAM. RAPL is one of the most accurate tools to report on the global energy consumption of a processor [6, 9]. We note that, due to CPU energy consumption variations issues [15], we used the same node for all our experiments. Moreover, we tried to be very careful, while running our experiments, not to fall in most common benchmarking "crimes" [23]. Every single experiment, therefore, reports on energy metrics obtained from at least 20 executions of 50 iterations per benchmark. All of our experiments are available for use/reproducibility from our anonymous repository.[¶]

Java Virtual Machines. We considered a set of 52 JVM distributions taken from 8 different providers/packagers mostly obtained from SDKMAN,[‖] as listed in Table 1. Depending on providers, either all the versions, majors, or LTS are made available by SDKMAN.

Java Benchmarks. We ran our experiments across 12 Java benchmarks we picked from OpenBenchmarking.org.[**] This includes 5 acknowledged benchmarks from the Dacapo benchmark suite v. 9.12 [3], namely Avrora, H2, Lusearch, Sunflow and PMD, that have been widely used in previous studies and proven to be accurate for memory management and computer architecture communities [8, 11]. It consists of open-source and real-world applications with non-trivial memory loads. Then, we also considered 7 additional benchmarks from the Renaissance benchmark suite [19, 20], namely ALS, Dotty, Fj-kmeans, Neo4j, Philosophers, Reaction and Scrabble, which offers a diversified set of benchmarks aimed at testing JIT, GC, profilers, analyzers, and other tools. The benchmarks we picked from both suites exercise a broad range of programming paradigms, including concurrent, parallel, functional, and object-oriented programming. Table 2 summarizes the selected benchmarks with a short description.

[¶] https://anonymous.4open.science/r/jvm-comparaison-213E/Readme.md

[‖] https://sdkman.io/

[**] https://openbenchmarking.org

Table 2: List of selected open-source Java benchmarks taken from DaCapo and Renaissance.

Benchmark	Description	Focus
ALS	Factorize a matrix using the alternating least square algorithm on spark	Data-parallel, compute-bound
Avrora	Simulates and analyses for AVR microcontrollers	Fine-grained multi-threading, events queue
Dotty	Uses the dotty Scala compiler to compile a Scala codebase	Data structure, synchronization
Fj-Kmeans	Runs K-means algorithm using a fork-join framework	Concurrent data structure, task parallel
H2	Simulates an SQL database by executing a TPC-C like benchmark written by Apache	Query processing, transactions
Lusearch	Searches keywords over a corpus of data comprising the works of Shakespeare and the King James bible	Externally multi-threaded
Neo4j	Runs analytical queries and transactions on the Neo4j database	Query Processing, Transactions
Philosophers	Solves dining philosophers problem	Atomic, guarded blocks
PMD	Analyzes a list of Java classes for a range of source code problems	Internally multi-threaded
Reactors	Runs a set of message-passing workloads based on the reactors framework	Message-passing, critical-sections
Scrabble	Solves a scrabble puzzle using Java streams	Data-parallel, memory-bound
Sunflow	Renders a classic Cornell box; a simple scene comprising two teapots and two glass spheres within an illuminated box	Compute-bound

4 EXPERIMENTS & RESULTS

4.1 Energy Impact of JVM Distributions

Job-oriented applications. To answer our first research question, we executed 62,400 experiments by combining the 52 JVM distributions with the 12 Java benchmarks, thus reasoning on 100 energy samples acquired for each of these combinations. Figure 1 first depicts the accumulated energy consumption of the 12 Java benchmarks per JVM distribution and major versions (or LTS when unavailable). Concretely, We measure the energy consumption of each of the benchmarks and compute the ratio of energy consumption compared to HotSpot-8, which we consider as the baseline in this experiment. Then, we sum the ratios of the 12 benchmarks and depict them as percentages in Figure 1.

One can observe that, along with time and versions, the energy efficiency of JVM distributions tends to improve (10% savings), thus demonstrating the benefits of optimizations delivered by the communities. Yet, one can also observe that energy consumption may differ from one distribution to another, thus showing that the choice of a JVM distribution may have a substantial impact on the energy consumption of the deployed software services. For example, one can note that J9 can exhibit up to 15% of energy consumption overhead, while other distributions seem to converge towards a lower energy footprint for the latest version of Java. As GraalVM adopts a different strategy focused on LTS support, one can observe that its recent releases provide the best energy efficiency for Java 11, but recent releases of other distributions seem to reach similar efficiency for Java 13 and above, which are recent versions not supported by GraalVM yet.

Interestingly, this convergence of distributions has been observed since Java 11 and coincides with the adoption of DCE VM by HotSpot. Ultimately, 3 clusters of JVMs that encompass JVMs with similar energy consumption can be seen through Figure 1: J9, the HotSpot and its variants, and GraalVM. Additional detailed figures to illustrate the evolution of energy consumption per benchmark/JVM are made available from the anonymous repository.[††]

Then, Figure 2 depicts the evolution of the energy consumption of the 12 benchmarks, when executed on the HotSpot JVM. Figure 2 reports on the energy consumption variation of individual benchmarks, using to HotSpot-8 as the baseline. Our results show that the JVM version can severely impact the energy consumption of the application. However, unlike Figure 1, one can observe that, depending on applications, latest JVM versions can consume less energy (60% less energy for Scrabble) or more energy (25% more energy for the Neo4J). It is worth noticing that the energy consumption of some benchmarks, such as Reactors, exhibit large variations across JVM versions due to experimental features and changes that are not always kept when releasing LTS versions (version 11 here). For example, the introduction of `VarHandle` to allow low-level access to the memory order modes available in JDK 9 and work along `Unsafe Classe` that was removed from from JVM 11.[‡‡]

Given that the wide set of distributions and versions seems to highlight 3 classes of energy behaviors, the remainder of this paper considers the following distributions as relevant samples of JVM to be further evaluated: 20.2.0.r11-grl (GraalVM), 15.0.1-open (HotSpot-15), 15.0.21.j9 (J9). We also decided to keep the 8.0.275-open (HotSpot-8) as a baseline JVM for some figures to highlight the evolution of energy consumption over time/versions.

Figure 3 further explores the comparison of energy efficiency of the JVM distributions per benchmark. One can observe that, depending on the benchmark's focus, the energy efficiency of JVM distributions may strongly vary. When considering individual benchmarks, J9 performs the worst for at least 6 out of 12 benchmarks—*i.e.*, worst ratio among the 4 tested distributions. Even though, J9 can still exhibit a significant energy saving for some benchmarks, such as Avrora, where it consumes 38% less energy than HotSpot and others.

Interestingly, GraalVM delivers good results overall, being among the distributions with a low energy consumption for all benchmarks, except for Reactors and Avrora. Yet, some differences still can be observed with HotSpot depending on applications. The newer version of HotSpot-15 was averagely good and, compared to HotSpot-8, it significantly enhances energy consumption for most scenarios.

[††]https://anonymous.4open.science/r/jvm-comparaison-213E/Readme.md

[‡‡]https://blogs.oracle.com/javamagazine/the-unsafe-class-unsafe-at-any-speed

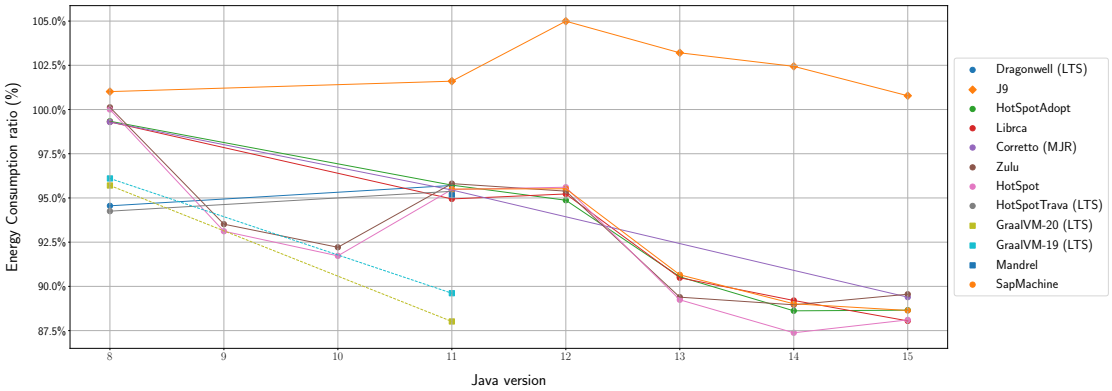

Figure 1: Energy consumption evolution of selected JVM distributions along versions.

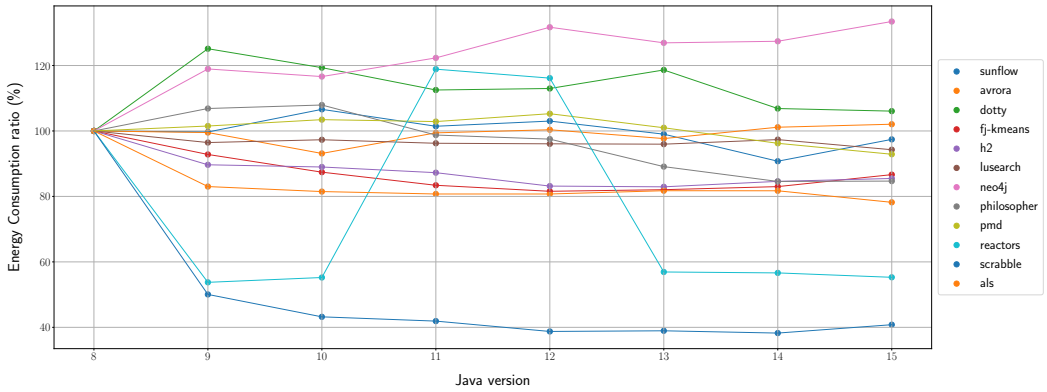

Figure 2: Energy consumption of the HotSpot JVM along versions.

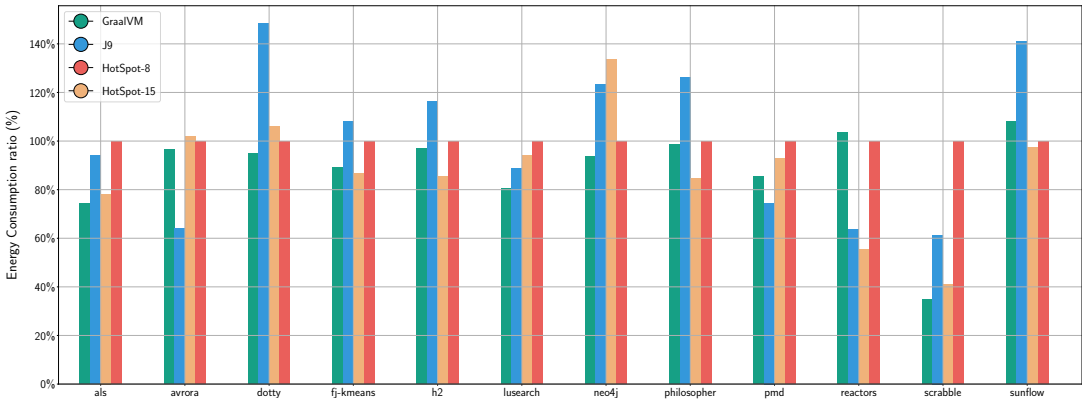

Figure 3: Energy consumption comparison across Java benchmarks for HotSpot, GraalVM & J9.

Table 3: Power per request for HotSpot, GraalVM & J9.

Benchmark	JVM	Power (P)	Requests (R)	$P/R \times 10^{-3}$
Scrabble	GraalVM	109 W	**5,336 req**	**20 mW**
	HotSpot	98 W	3,595 req	27 mW
	J9	**92 W**	2,603 req	35 mW
Dotty	GraalVM	**45 W**	510 req	88 mW
	HotSpot	**45 W**	**597 req**	**75 mW**
	J9	46 W	381 req	120 mW

Finally, Neo4J is the only selected benchmark where HotSpot-8 is more energy efficient than HotSpot-15.

Service-oriented applications. In this section, instead of considering bounded execution of benchmarks, we run the same benchmarks as services for 20 minutes, and we compare the average power and total requests processed by each of the 3 JVM distributions. Globally, the results showed that the average power when using GraalVM, HotSpot, and OpenJ9 is often equivalent and stable over time. This means that the energy efficiency observed for some JVM distributions with Job-oriented applications is mainly related to shorter execution times, which incidentally results in energy savings. Nonetheless, we can highlight two interesting observations for two benchmarks whose behaviors differ from others. First, the analysis of the Scrabble benchmark experiments showed that, in some scenarios, some JVMs can exhibit different power consumptions. Figure 4 depicts the power consumed by the 3 JVM distributions for the Scrabble benchmark. One can clearly see that GraalVM requires an average power of 109 W, which is 9 W higher than HotSpot-15 and 15 W higher than J9. When it comes to the number of requests processed by Scrabbles during that same amount of time, GraalVM completes 5,336 requests, against 3,595 for HotSpot and 2,603 for J9, as shown in Table 3. The higher power usage for GraalVM helped in achieving a high amount of requests, but also the fastest execution of every, request which was 40% faster on GraalVM. Thus, GraalVM was more energy efficient, even if it uses more power, which confirms the results observed in Figure 3 for this benchmark.

The second interesting situation was observed on the Dotty benchmark. More specifically, during the first 100 seconds of the execution of the Dotty benchmark on all evaluated JVMs. At the beginning of the execution, GraalVM has a slightly lower power consumption, is faster, and consumes 10% less energy. After about 150 seconds, the power differences between the 3 JVMs is barely noticeable. One can, however, notice the effect of the JIT, as HotSpot takes the advantage over GraalVM and becomes more energy efficient. In total, HotSpot completes 597 requests against 510 for GraalVM and 381 for J9, as shown in Table 3. HotSpot was thus the best choice on the long term, which explains why it is always necessary to consider a warm-up phase and wait for the JIT to be triggered before evaluating the effect of the JVM or the performance of an application. This is exactly what we did in our experiments, and why HotSpot was more energy efficient than GraalVM in Figure 3, thus ignoring the warm-up phase would have been misleading.

To answer **RQ 1**, we conclude that—while most of the JVM platforms perform similarly—we can cluster JVMs in 3 classes: HotSpot, J9, and GraalVM. The choice of one JVM of these classes can have a major impact on software energy consumption, that strongly depends on the application context. When it comes to the JVM version, latest releases tend to offer the lowest power consumption, but experimental features should be carefully configured, thus further questioning the impact of JVM parameters.

4.2 Energy Impact of JVM Settings

The purpose of our study is not only to investigate the impact of the JVM platform on the energy consumption, but also the different JVM parameters and configurations that might have a positive or negative effect, with a focus on 3 available settings: multi-threading, JIT, and GC.

4.2.1 Multithreading. The purpose behind this phase is to investigate the impact JVM thread management strategies on the energy consumption. This encompasses exploring if the management strategies of application-level parallelism (so called *threads*) results in different energy efficiencies, depending on JVM distributions.

Investigating such an hypothesis requires a selection of highly parallel and CPU-intensive benchmarks, which is one of the main criteria for our benchmark selection. As no tool can accurately monitor the energy consumption at a thread level, we monitor the global power consumption and CPU utilization during the execution using RAPL for the energy, and several Linux tools for the CPU-utilization (`htop`, `cpufreq`). Knowing that most of the benchmarks are multi-threaded jobs that use multiple cores, further analysis of thread management is required to understand the results of our previous experiments. We thus selected the benchmarks that highlighted the highest differences along JVM distributions from Figure 3, namely Avrora and Reactors. We studied their multi-threaded behavior to optimize their energy efficiency.

Figures 6 and 7 deliver a closer look to the thread allocation strategies adopted by JVM. First, Figure 6 illustrates the active threads count evolution over time (excluding the JVM-related threads, usually 1 or 2 extra threads depending on the execution phase) for Avrora. One can notice through the figure that J9 exploits the CPU more intensively by running much more parallel threads compared to other JVMs (an average of 5.1 threads per second for J9 while the other JVMs do not exceed 1.5 thread per second). Furthermore, the number of context switches is twice as big for J9, while the number of soft page faults is twice as small. The efficient J9 thread management explains why running the Avrora benchmark took much less time and consumed less energy, given that no other difference for the JIT or GC configuration was spotted between the JVMs. Another key reasons of the J9's efficiency for the Avrora benchmark is memory allocation, as OpenJ9 adopts a different policy for the heap allocation. It creates a non-collectable *thread local heap* (TLH) within the main heap for each active thread. The benefit of cloning a dedicated TLH is the fast memory access for independent threads: each thread has its own heap and no deadlock can occur.

The second example in Figure 7 depicts the active threads evolution over time of the Reactors benchmark. In this case, all the JVMs have a close average of threads per second. Nevertheless, one can still observe that HotSpot-15 and J9 keep running faster, which confirms the results of Figure 3, where both JVMs consume much less

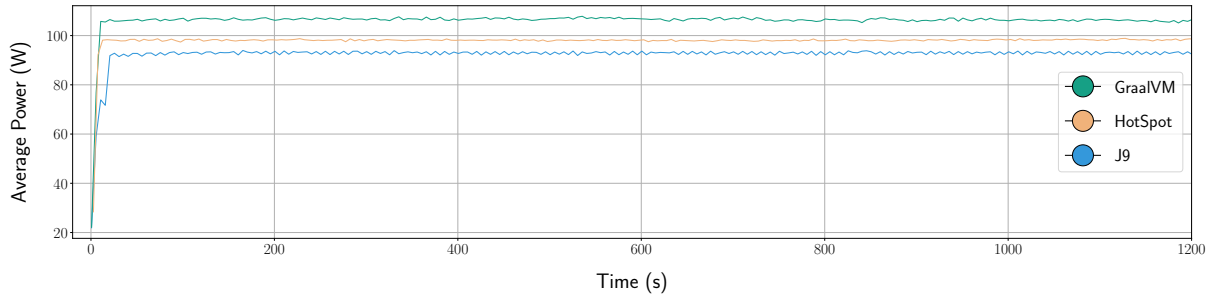

Figure 4: Power consumption of Scrabble as a service for HotSpot, GraalVM & J9.

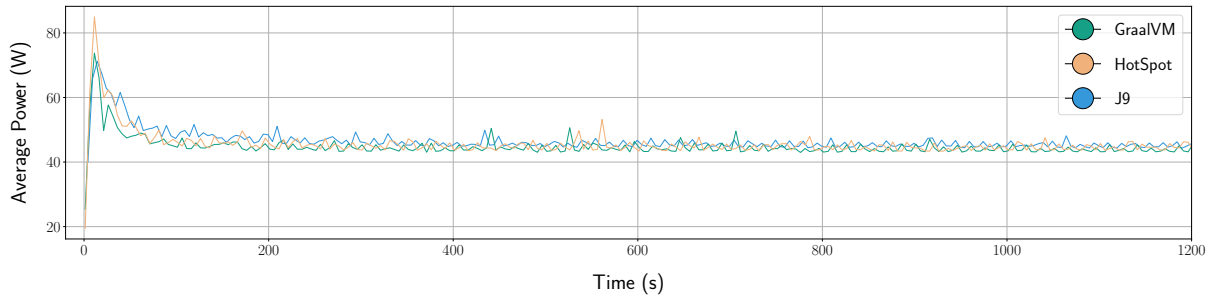

Figure 5: Power consumption of Dotty as a service for HotSpot, GraalVM & J9.

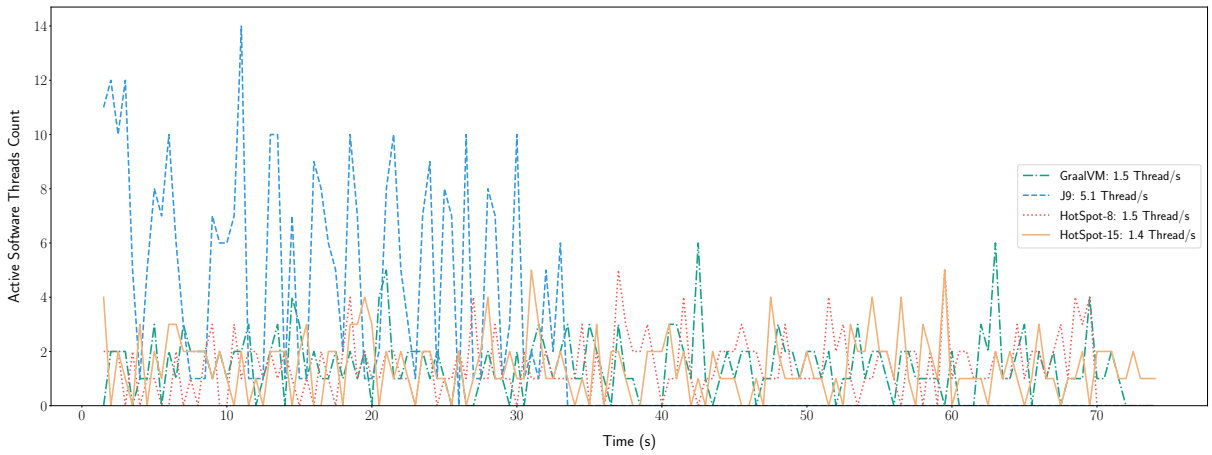

Figure 6: Active threads of Avrora when using HotSpot, GraalVM, or J9.

energy compared to GraalVM and HotSpot-8. This difference in energy consumption between benchmarks can be less likely caused by thread management for the Reactors benchmark, as HotSpot-8 reports on a higher average of active threads. However, the TLH mechanism was not as efficient as for the Avrora benchmark, as dedicating a heap for each thread can also cause some extra memory usage for data duplication and synchronization, especially if a lot of data is shared between threads.

In conclusion, JVMs thread management can sometimes constitute a key factor that impacts software energy consumption. However, we suggest to check and compare JVMs before deploying a software, especially if the target application is parallel and multi-threaded.

4.2.2 Just-in-Time Compilation. The purpose of experiments on JIT is to highlight the different strategies that can impact software energy consumption within a JVM and between JVMs. We identified a set of JIT compiler parameters for every JVM platform.

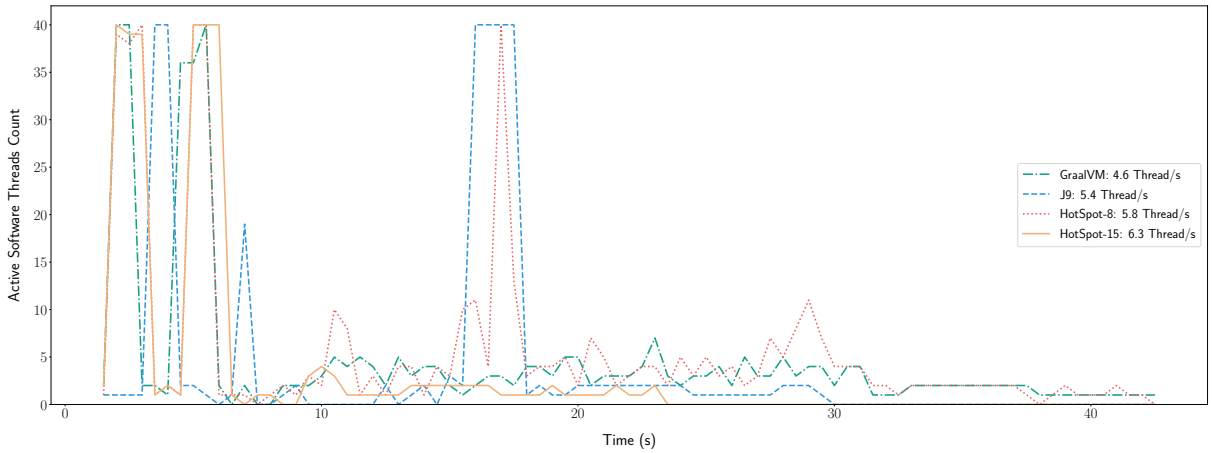

Figure 7: Active threads of Reactors when using HotSpot, GraalVM, or J9.

For J9, we considered fixing the intensity of the JIT compiler at multiple levels (cold, warm, hot, veryhot, and scorching).[§§] The hotter the JIT, the more code optimization to be triggered. We also varied the minimum count method calls before a JIT compilation occurs (10, 50, 100), and the number of JIT instances threads (from 1 to 7). For Hotspot-15, we conducted experiments while disabling the tiered complication (that generates compiled versions of methods that collect profiling information about themselves), and we also varied the JIT maximum compilation level from 0 to 4, we also tried out HotSpot with a basic GraalVM JIT. We note that the level 0 of JIT compilation only uses the interpreter, with no real JIT compilation. Levels 1, 2, and 3 use the C1 compiler (called client-side) with different amount of extra tuning. The JIT C2 (also called server-side JIT) compiler only kicks-in at level 4.

For GraalVM, we conducted experiments with and without the JVMCI (a Java-based JVM compiler interface enabling a compiler written in Java to be used by the JVM as a dynamic compiler). We also considered both the community and economy configurations (no enterprise). A JIT+AOT (*Ahead Of Time*) disabling experiment has also been considered for all of the 3 JVM platforms. Table 4 reports on the energy consumption of the experiments we conducted for most of the benchmarks and JIT configurations under study.

The p-values are computed with the mann-whitney test, with a null hypothesis of the energy consumption being equal to the default configuration. The p-values in bold show the values that are significantly different from the default configuration with a 95% confidence, where the values in green highlight the strategies that consumed significantly less energy than default (less energy and significant p-value).

For J9, we noticed that adopting the default JIT configuration is always better than specifying a custom JIT intensity. The warm configuration delivers the closest results to the best results observed with the default configuration. Moreover, choosing a low minimum count of method calls seems to have a negative effect on the execution time and the energy consumption. The only parameter that can give better performance than the default configuration in some cases is the number of parallel JIT threads—using 3 and 7 parallel threads—but is not statistically significant.

For GraalVM, the default community configuration is often the one that consumes the least energy. Disabling the JVMCI can—in some cases—have a benefit (16% of energy consumption reduction for the H2 benchmark), but still gave overall worst results (80% more energy consumption for the Neo4J benchmark). In addition, switching the economy version of the GraalVM JIT often results in consuming more energy and delaying the execution.

For HotSpot, keeping the default configuration of the JIT is also mostly good. In fact, the usage of the C2 JIT is often beneficial (JIT level 4) in most cases, while using the GraalVM JIT reported similar energy efficiency. Yet, some benchmarks showed that using only the C1 JIT (JIT level 1) is more efficient and even outperforms the usage of the C2 compiler. 10% on Avrora and 30% on Pmd are examples of energy savings observed by using the C1 compiler. However, being limited to the C1 compiler can also cause a huge degradation in energy consumption, such as 32% and 34% of additional energy consumed for the Dotty and FJ-kmeans benchmarks, respectively. Hence, if it is a matter of not using the C2 JIT, the experiments have shown that the level 1 JIT is always the best, compared to levels 2 or 3 that also use the C1 JIT, but with more options, such as code profiling that impacts negatively the performance and the energy efficiency. Level 0 JIT compilation should never be an option to consider. No p-value has been computed for Level 0, due to the limited amount of iterations executed with this mode (very high execution time, clearly much more consumed energy).

Globally, we conclude through these experiments that keeping the default JIT configuration was more energy efficient in 80% of our experiments and for the 3 classes of JVMs. This advocates that using the default JIT configuration that can often deliver near-optimal energy efficiency. Although, some other configurations, such as using only the C1 JIT or disabling the JVMCI could be advantageous in some cases.

4.2.3 Garbage Collection. Changing or tuning the GC strategy has been acknowledged to impact the JVM performances [12]. To investigate if this impact also benefits to energy consumption, we conducted

[§§] [https://www.eclipse.org/openj9/docs/jit/]

Table 4: Energy consumption when tuning JIT settings on HotSpot, GraalVM & J9

JVM	Mode	ALS		Avrora		Dotty		Fj-kmeans		H2		Neo4j		Pmd		Reactors		Scrabble		Sunflow	
GraalVM	*Default*	2848	*p-values*	3861	*p-values*	2271	*p-values*	948	*p-values*	1959	*p-values*	3313	*p-values*	297	*p-values*	23452	*p-values*	452	*p-values*	335	*p-values*
	DisableJVMCI	3099	**0.001**	4012	**0.041**	2694	**0.001**	**934**	**0.011**	**1771**	**0.005**	5086	**0.001**	353	**0.001**	25007	**0.007**	503	**0.002**	354	0.227
	Economy	4503	**0.001**	3895	0.793	3466	**0.001**	1306	**0.002**	2560	**0.001**	9525	**0.001**	**270**	**0.001**	30317	**0.001**	649	**0.002**	392	**0.002**
J9	*Default*	3792	*p-values*	2122	*p-values*	3515	*p-values*	1271	*p-values*	2426	*p-values*	4336	*p-values*	277	*p-values*	12705	*p-values*	734	*p-values*	476	*p-values*
	Thread 1	4157	**0.001**	2121	0.875	4749	**0.001**	1297	0.097	2597	0.066	4906	**0.001**	350	**0.001**	12800	0.713	948	**0.002**	626	**0.005**
	Thread 3	3849	0.018	2105	0.713	3574	0.104	1259	0.371	2450	0.637	4477	**0.005**	294	**0.004**	12647	0.875	795	0.021	457	0.27
	Thread 7	3843	0.041	2386	0.372	3511	0.875	1259	0.25	2424	0.637	4431	0.104	273	0.372	12600	0.875	808	0.055	463	0.372
	Count 0	8461	**0.001**	2425	**0.001**	4877	**0.001**	2289	**0.002**	3212	**0.001**	10565	**0.001**	744	**0.001**	18084	**0.001**	1476	**0.002**	922	**0.001**
	Count 1	4281	**0.001**	2150	0.431	3164	**0.001**	1841	**0.002**	2546	0.431	7166	**0.001**	272	0.128	14715	**0.001**	1005	**0.002**	514	0.052
	Count 10	3980	**0.001**	2431	0.713	3771	**0.001**	1312	**0.011**	2779	**0.003**	4979	**0.001**	299	**0.001**	12000	0.104	860	**0.005**	1182	**0.001**
	Count 100	3878	**0.007**	2141	0.713	3469	0.227	1363	0.523	2513	0.128	4547	**0.001**	262	0.031	12313	0.024	768	0.16	634	**0.004**
	Cold	6788	**0.001**	2134	0.637	4855	**0.001**	1636	**0.002**	2873	**0.001**	7250	**0.001**	275	0.372	20380	**0.001**	870	**0.005**	386	**0.001**
	Warm	4594	**0.001**	2112	0.713	4253	**0.001**	1244	0.055	2521	0.128	5305	**0.001**	411	**0.001**	13726	**0.001**	913	**0.002**	336	**0.001**
	Hot	7553	**0.001**	2310	**0.001**	12749	**0.001**	1452	**0.002**	3973	**0.001**	8979	**0.001**	857	**0.001**	36534	**0.001**	1180	**0.002**	506	0.128
	VeryHot	15113	**0.001**	3300	**0.001**	18235	**0.001**	2430	**0.002**	7205	**0.001**	19359	**0.001**	793	**0.001**	38303	**0.001**	5420	**0.002**	1692	**0.001**
	Schorching	18316	**0.001**	3541	**0.001**	21686	**0.001**	2514	**0.002**	7855	**0.001**	26409	**0.014**	808	**0.001**	43929	**0.001**	5583	**0.002**	1778	**0.001**
HotSpot	*Default*	2997	*p-values*	4014	*p-values*	2516	*p-values*	934	*p-values*	1796	*p-values*	4787	*p-values*	323	*p-values*	11685	*p-values*	530	*p-values*	325	*p-values*
	Graal	2999	0.637	3971	0.318	2512	0.318	929	0.609	**1662**	**0.007**	4750	0.372	327	0.189	11548	0.523	537	0.701	338	0.564
	Lvl 0	491443	/	14484	/	84395	/	/	/	52344	/	356287	/	1073	/	148381	/	/	/	14559	/
	Lvl 1	/	/	**3731**	**0.001**	3302	**0.001**	1256	**0.002**	2523	**0.001**	8304	**0.001**	**222**	**0.001**	22410	**0.002**	735	**0.002**	**277**	**0.007**
	Lvl 2	3079	**0.004**	4110	0.189	3723	**0.001**	22547	**0.002**	2840	**0.001**	19058	**0.001**	226	**0.001**	40701	**0.002**	2291	**0.002**	4131	**0.001**
	Lvl 3	16375	**0.001**	7729	**0.001**	6789	**0.001**	144914	**0.002**	4139	**0.001**	44594	**0.001**	330	**0.005**	190124	**0.002**	9070	**0.002**	10449	**0.001**
	NotTired	3254	**0.001**	3901	0.189	3110	**0.001**	**912**	**0.021**	1846	0.227	**3844**	**0.001**	933	**0.001**	**11256**	**0.041**	588	**0.003**	405	**0.001**

Table 5: The different J9 GC policies

Policy	Description
Balanced	Evens out pause times & reduces the overhead of the costlier operations associated with GC
Metronome	GC occurs in small interruptible steps to avoid stop-the-world pauses
Nogc	Handles only memory allocation & heap expansion, with no memory reclaim
Gencon (default)	Minimizes GC pause times without compromising throughput, best for short-lived objects
Concurrent Scavenge	Minimizes the time spent in stop-the-world pauses by collecting nursery garbage in parallel with running application threads
optthruput	Optimized for throughput, stopping applications for long pauses while GC takes place
Optavgpause	Sacrifices performance throughput to reduce pause times compared to optthruput

Table 6: The different HotSpot/GraalVM GC policies

Policy	Description
G1GC (default)	Uses concurrent & parallel phases to achieve low-pauses GC and maintain good throughput
SerialGC	Uses a single thread to perform all garbage collection work (no threads communication overhead)
ParallelGC	Known as throughput collector: similar to SerialGC, but uses multiple threads to speed up garbage collections for scavenges
parallelOldGC	Use parallel garbage collection for the full collections, enabling it automatically enables the ParallelGC

a set of experiments on the selected JVMs. We considered different garbage collector strategies with a limited memory quantity of 2 GB, and recorded the execution time and the energy consumption. The tested GC strategies options mainly vary between J9 and the other 2 JVMs, as detailed in Table 5.

For HotSpot and GraalVM, we also considered many GC policies, as described in Table 6. Furthermore, other GC settings have also been tested for all JVM platforms, such as the *pause time*, the *number of parallel threads* and *concurrent threads* and *tenure age*.

Table 7 summarizes the results of all the tested GC strategies with our selected benchmarks and the p-values of the mann-whitney test, with a null hypothesis of the energy consumption being equal to the default configuration with a 95% confidence. The p-values in bold show the values that are significantly different from the default configuration, where the values in green highlight the strategies that consumed significantly less energy than default. For GraalVM, one can see that the GC default configuration is efficient in most experiments, compared to other strategies. The main noticeable impact is related to the ParallelGC and ParallelOldGC. In fact, the ParallelGC can be 13% more energy efficient in some applications with a significant p-value, such as Reactors, compared to default. However, the same GC strategy can cause the software to consume twice times more, as for the Neo4j benchmark, due to the high communications between the GC threads, and the fragmentation of the memory.

For J9, the default Gencon GC causes the software to report an overall good energy efficiency among the tested benchmarks. However, other GC can cause better or worse energy consumption than Gencon depending on workloads. Using the Metronome GC consumes 35% less energy for the ALS benchmark and 17% less energy for the Sunflow benchmark, but it also consumes twice energy for the Neo4j benchmark and 28% more energy for Reactors. The reason is that Metronome occurs in small preemptible steps to reduce the GC cycles composed of many GC quanta. This suits well for real-time applications and can be very beneficial when long GC pauses are not desired, as observed for ALS. However, if the heap space is insufficient after a GC cycle, another cycle will be triggered with the same ID. As Metronome supports class unloading in the standard way, there might be pause time outliers during GC activities, inducing a negative impact on the Neo4j execution time and energy consumption.

The same goes for the Balanced GC that tries to reduce the maximum pause time on the heap by dividing it into individually managed regions. The Balanced strategy is preferred to reduce the pause times that are caused by global GC, but can also be disadvantageous due to the separate management of the heap regions, such as for ALS where

Table 7: Energy consumption when tuning GC settings on HotSpot, GraalVM & J9

JVM	Mode	ALS		Avrora		Dotty		H2		Neo4j		Pmd		Reactors		Scrabble		Sunflow	
	Default	2570	*p-values*	4153	*p-values*	2223	*p-values*	1870	*p-values*	5256	*p-values*	281	*p-values*	2611	*p-values*	410	*p-values*	353	*p-values*
	1Concurent	2567	0.403	**4007**	**0.023**	2220	1.000	1883	0.982	5368	1.000	286	0.182	2664	1.000	413	0.885	347	0.573
	1Parallel	2668	**0.012**	**3904**	**0.008**	2228	0.835	2022	**0.000**	5836	**0.012**	298	**0.000**	2869	0.144	561	**0.030**	**317**	**0.000**
	5Concurent	2570	0.676	4117	0.161	2215	0.210	1862	0.505	5259	1.000	282	0.980	2611	0.531	414	0.885	362	0.356
GraalVM	5Parallel	2561	0.676	**3863**	**0.012**	2237	1.000	1910	0.103	5223	0.403	282	0.538	2682	0.531	424	0.112	353	0.758
	DisableExplicitGC	2559	0.210	**3911**	**0.003**	2215	1.000	1978	**0.018**	5106	0.210	281	0.758	2704	0.676	400	0.312	**332**	**0.036**
	ParallelCG	2720	**0.012**	4016	0.206	2237	0.531	1945	**0.000**	13172	**0.037**	282	0.878	**2267**	**0.022**	545	**0.030**	**329**	**0.003**
	ParallelOldGC	2715	**0.012**	4032	0.103	2221	1.000	1925	**0.002**	13362	/	282	0.918	**2514**	**0.012**	535	**0.030**	**329**	**0.008**
	Default	3371	*p-values*	2243	*p-values*	3237	*p-values*	2107	*p-values*	6277	*p-values*	232	*p-values*	1644	*p-values*	589	*p-values*	510	*p-values*
	Balanced	9012	**0.012**	2232	0.597	3429	**0.012**	2247	**0.002**	8853	**0.012**	235	0.412	1902	**0.020**	661	0.061	519	0.505
	ConcurrentScavenge	3487	**0.012**	2270	0.280	3388	**0.012**	2319	**0.001**	6857	**0.012**	233	0.878	1705	0.903	639	0.194	546	**0.018**
	Metronome	**2098**	**0.012**	2265	0.505	3815	**0.012**	2717	**0.000**	12103	**0.012**	239	**0.022**	2089	**0.020**	758	**0.030**	**422**	**0.000**
J9	Nogc	3454	**0.022**	2239	0.872	3259	0.144	2207	0.031	61781	**0.012**	227	0.151	**1505**	**0.066**	711	**0.030**	499	0.720
	Optavgpause	3601	**0.012**	2431	0.370	3425	**0.012**	2169	0.297	7495	**0.012**	253	**0.000**	1772	0.391	1089	**0.030**	478	**0.046**
	Optthruput	3357	1.000	2432	0.241	3178	0.403	2194	0.139	6324	0.835	232	0.878	1554	0.111	640	0.194	**429**	**0.000**
	ScvNoAdaptiveTenure	3494	**0.012**	2253	0.800	3248	0.835	2161	0.103	8442	**0.012**	228	0.137	1908	**0.020**	618	0.665	528	0.218
	Default	2765	*p-values*	4115	*p-values*	2492	*p-values*	1673	*p-values*	8152	*p-values*	316	*p-values*	1546	*p-values*	484	*p-values*	347	*p-values*
	1Concurent	2775	0.060	4137	0.346	2493	0.676	1675	0.918	8062	0.531	316	0.383	1533	0.665	478	0.470	334	**0.218**
	1Parallel	2863	**0.012**	4142	0.800	2526	**0.037**	1853	**0.001**	8270	0.676	334	**0.000**	1747	**0.030**	592	**0.030**	**320**	**0.002**
	5Concurent	2758	0.676	4091	0.872	2485	0.296	1681	0.608	8087	0.835	314	0.330	1497	0.665	**469**	**0.030**	336	0.259
	5Parallel	2767	0.144	4176	0.077	2473	0.060	1654	0.720	8046	0.835	316	0.573	1546	0.470	489	0.470	342	0.573
HotSpot	DisableExplicitGC	**2734**	**0.012**	4062	0.448	2483	0.835	1702	0.248	**7710**	**0.037**	312	0.200	1545	0.470	470	0.061	**325**	**0.014**
	ParallelCG	**2653**	**0.012**	4064	0.629	**2356**	**0.012**	**1602**	**0.008**	8953	0.060	**300**	**0.000**	1476	0.885	579	**0.030**	336	0.081
	ParallelOldGC	2764	0.531	4070	0.872	2525	0.802	1675	0.959	7963	0.403	314	0.720	1582	0.194	475	0.470	333	0.151
	SerialGC	**2593**	**0.012**	4083	0.395	**2378**	**0.012**	**1620**	**0.046**	**5745**	**0.012**	**307**	**0.002**	1672	0.061	601	**0.030**	352	0.473

it consumed about three times the energy consumption, compared to the default Gencon GC. On the other hand, the Optthruput GC, which stops the application longer and less frequently, gave very good overall results and sometimes even outperformed the Gencon GC by a small margin. Other JVM parameters, such as the ConcurrentScavenge or noAdaptiveTenure did not have a substantial impact during our experiments.

Finally, the results of HotSpot shared similarities with GraalVM. The ParallelGC happened to give better (6% for Dotty) or worst (10% for Neo4j) energy efficiency compared to the default GC. On the other hand, ParallelOldGC and Serial GC gave better results than the default G1 GC. More specifically, the second one consumed 30% and 6% less energy than default GC for the Neo4j and Dotty benchmarks, respectively. The most interesting result for HotSpot is the 30% energy reduction obtained with the Serial GC. This last was also more efficient on ALS (6% less energy), compared to the default G1 GC, due to its single-threaded GC that only uses one CPU core.

Therefore, we noticed during our experiments that, even if using the default GC configuration ensures an overall steady and correct energy consumption, we still found other settings that reduce that energy consumption in 50% of our experiments. Tuning the GC according to the hosted app/benchmark is thus critical to reduce the energy consumption.

To answer **RQ 2**, we conclude that users should be careful while choosing and configuring the garbage collector as substantial energy enhancements can be recorded from a configuration to another. The default GC consumed more energy than other strategies in most of the situations. However, keeping the default JIT parameters often delivers near-optimal energy efficiency. In addition, the JVM platforms can handle differently multi-threaded applications and thus consume a different amount of time/energy. Dedicated performance tuning evaluations should therefore be conducted on such software to identify the most energy-efficient platform and settings.

4.2.4 J-Referral. To help developers and practitioners choosing an energy efficient JVM distribution and configuration, we propose a tool named J-Referral. This tool takes a Java application as an input and a launch script to tune the JVM configuration. The recommendation is is based on the 85 JVM distributions and versions made available by SDKMAN. Beyond the JVM distribution and version recommendation, J-Referral automatically explores many GC and JIT options for each distribution. Hundreds of combinations are thus compared to recommend an energy efficient JVM distribution and the associated configuration parameters.

Table 8 illustrates an example of the final report of J-Referral. The tool was tested for 2 real Java projects: Zip4J and[¶¶] and K-nucleotide.[***]. Zip4J runs a large file compression, while K-nucleotide extracts a DNA sequence, and updates a hashtable of k-nucleotide keys to count specific values. The short report presented in Table 8 shows the ratio of potential energy saving between the most and least energy consuming tested JVM (40% and 70% energy savings for Zip4J and K-nucleotide respectively). Options are available for J-Referral to obtain much more detailed reports including execution time, DRAM usage, split DRAM vs. CPU consumption, etc. The tool is available as *open-source software* (OSS) from our anonymous repository.[†††]

5 RELATED WORK

In this section, we review the state of the art of studies related to JVM energy consumption.

At code level. Many works investigated software energy consumption efficiency through source code changes and optimizations. For

[¶¶] https://github.com/srikanth-lingala/zip4j

[***] https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/knucleotide.html

[†††] https://anonymous.4open.science/r/jreferral/Readme.md

Table 8: J-Referral recommendations.

Project	Metric	Energy	JVM	Execution flags
Zip4J	Least energy	**2210 J**	**16-sapmchn**	**default**
	Most energy	3680 J	8.0.292-J9	default
K-nucl	Least energy	**1296 J**	**21.1.r16-grl**	**default**
	Most energy	4433 J	15.0.1-J9	-Xjit:optlevel=cold

example, [7, 18] studied the effect of Java collections on energy consumption, with regards to the collection size and/or the most executed tasks on the collection (insertion, removal, search). In [21], Hasan *et al.* compared the energy consumption of several Java data structures, analyzing the bytecode using the Wala framework[‡‡‡] and assessing the evolution of the energy consumption in different scenarios (insertion at the beginning, iteration, etc.). They also used some automated replacement of `LinkedList` and `ArrayList` to simulate best- and worst-case energy consumption scenarios on real production applications. Their study showed that using inappropriate collection can cause an energy consumption inefficiency of 300% for the worst-case scenario.

SEEDS and SEEDS-API is a fully automated framework, proposed in [13], to analyze source code (at bytecode level for Java) and autotunes apps to have reduce their energy consumption, with a focus on Java collection tuning. The authors reported on an improvement up to 17%. In [16], the same authors presented SPELL, the energy leaks detector tool. The tool uses JRapl [1, 18] to detect energy-inefficient code fragments by using a statistical spectrum-based energy red spots localization. Their evaluation reported up to 18% energy savings on Java applications.

At JVM level. In [17], Pereira *et al.* investigated several programming languages, including JVM-based languages like Java and Scala, and compared them on different dimensions (energy, memory, and time) using CLBG benchmarks. Their observation reported that compiled languages are more energy-efficient than interpreted ones, overall. More advanced results expose how the selected languages may satisfy one, two, or three dimensions of the equation. The authors of [14] conducted a performance analysis and comparison between HotSpot and J9. They claimed in their results that the relative performance of HotSpot ranges from 44% to 289% of J9, while the dynamic power consumption varies from 2.7W to 7.2W using the SPECjvm2008 benchmarks.

The difference between HotSpot and J9 was also reported in other studies. Chiba *et al.* [5] evaluated the effect that could have those 2 JVM platforms on the performance of a combination of big data query engines (Spark and Tez) using TPC-DS benchmark. They reported on a 3-fold drawback that can exhibit one JVM compared to the other.

On another note, the authors of [10] attempted to assess and assign an energy cost to atomic bytecode instructions together with a constant overhead of using the JVM. Their claims are rather surprising, as the energy cost should be impacted by the execution environment, workload, core frequencies, and can hardly be fixed. A similar idea was used in [4] to design a model for JVM-based software energy consumption, using a bytecode-level model. The authors described their tool, named OPACITOR, as being deterministic, accurate, and robust to the surrounding noise. However, they disable the JIT in their experiments to maintain the deterministic behavior of their tool, which does not reflect a real software execution given all the optimizations that the JVM triggers to optimize the performances.

[‡‡‡] http://wala.sourceforge.net/wiki/index.php

6 CONCLUSION

This paper reports on an empirical investigation of the key differences in energy consumption that some of the most famous and supported JVM platforms can exhibit, in addition to the key settings that can impact this energy consumption positively or negatively. During our experiments, we considered a total of 12 well-known and diversified-purposes Java benchmarks together with a total of 52 JVMs, including many versions of 11 different distributions. The results of our investigations showed that many JVMs share energy efficiencies and can grouped into 3 classes: HotSpot, J9, and GraalVM. The 3 selected JVM classes can however report a different energy efficiency for different software and/or workloads, sometimes by a large margin. While we did not observed a unique champion when it comes to energy consumption, GraalVM reported the best energy efficiency for a majority of benchmarks. Nonetheless, each JVM can achieve the best or the worst depending on the hosted application. One cause can be thread management strategies, as observed with J9 when advantageously running Avrora. Moreover, some JVM settings can cause energy consumption variations. Our experiments showed that the default JIT compiler of the JVM is often near-optimal, in at least 80% of our experiments. The default GC, however, was outperforming alternative strategies in half of our experiments, with some large gains observed when using some alternative GC depending on the application characteristics.

To ease the integration of the above guidelines, we propose a tool, named J-Referral, to recommend the most energy-efficient JVM distribution and configuration among more than a hundred considered possibilities. It establishes a full report on the energy consumption of both CPU and DRAM components for each JVM distribution and/or configuration to help the user to choose the one with the least consumption for a Java Software.

REFERENCES

[1] Danilo S Alves, Lucio M Duarte, Davi Silva, and Paulo H M Maia. 2020. Experiments on Model-Based Software Energy Consumption Analysis. (2020), 8.

[2] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. 2013. Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science*, Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan (Eds.). Communications in Computer and Information Science, Vol. 367. Springer International Publishing, 3–20. https://doi.org/10.1007/978-3-319-04519-1_1

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Alexander Edward Ian Brownlee, Nathan Burles, and Jerry Swan. 2017. Search-Based Energy Optimization of Some Ubiquitous Algorithms. *IEEE Transactions on Emerging Topics in Computational Intelligence* 1, 3 (June 2017), 188–201. https://doi.org/10.1109/TETCI.2017.2699193

[5] T. Chiba, T. Yoshimura, M. Horie, and H. Horii. 2018. Towards Selecting Best Combination of SQL-on-Hadoop Systems and JVMs. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 245–252. https://doi.org/10.1109/CLOUD.2018.00038

[6] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. 2016. A Validation of DRAM RAPL Power Measurements. In *Proceedings of the Second International Symposium on Memory Systems (MEMSYS '16)*. Association for Computing Machinery, New York, NY, USA, 455âĂŞ470. https://doi.org/10.1145/2989081.2989088

[7] Benito Fernandes, Gustavo Pinto, and Fernando Castor. 2017. Assisting Non-Specialist Developers to Build Energy-Efficient Software. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, Buenos Aires, Argentina, 158–160. https://doi.org/10.1109/ICSE-C.2017.133

[8] Tomas Kalibera, Matthew Mole, Richard E. Jones, and Jan Vitek. 2012. A black-box approach to understanding concurrency in DaCapo. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 335–354. https://doi.org/10.1145/2384616.2384641

[9] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3, 2, Article 9 (March 2018).

[10] Sébastien Lafond and Johan Lilius. 2006. An Energy Consumption Model for an Embedded Java Virtual Machine. In *Architecture of Computing Systems - ARCS 2006*, Werner Grass, Bernhard Sick, and Klaus Waldschmidt (Eds.). Vol. 3894. Springer Berlin Heidelberg, Berlin, Heidelberg, 311–325. https://doi.org/10.1007/11682127_22

[11] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, Walter Binder, Vittorio Cortellessa, Anne Koziolek, Evgenia Smirni, and Meikel Poess (Eds.). ACM, 3–14. https://doi.org/10.1145/3030207.3030211

[12] Peter Libič, Lubomír Bulej, Vojtěch Horky, and Petr Tůma. 2014. On the Limits of Modeling Generational Garbage Collector Performance. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE '14)*. Association for Computing Machinery, New York, NY, USA, 15âĂŞ26. https://doi.org/10.1145/2568088.2568097

[13] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: A Software Engineer's Energy-Optimization Decision Support Framework. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, Hyderabad, India, 503–514. https://doi.org/10.1145/2568225.2568297

[14] H. Oi. 2011. Power-performance analysis of JVM implementations. In *ICIMU 2011 : Proceedings of the 5th international Conference on Information Technology Multimedia*. 1–7. https://doi.org/10.1109/ICIMU.2011.6122743

[15] Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, Joel Penhoat, and Lionel Seinturier. 2020. Taming Energy Consumption Variations In Systems Benchmarking. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 36âĂŞ47. https://doi.org/10.1145/3358960.3379142

[16] Rui Pereira, Tiago Carcao, Marco Couto, Jacome Cunha, Joao Paulo Fernandes, and Joao Saraiva. 2017. Helping Programmers Improve the Energy Efficiency of Source Code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, Buenos Aires, Argentina, 238–240. https://doi.org/10.1109/ICSE-C.2017.80

[17] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate? ACM Press, 256–267. https://doi.org/10.1145/3136014.3136031

[18] Gustavo Pinto, Kenan Liu, Fernando Castor, and Yu David Liu. 2016. A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Raleigh, NC, USA, 20–31. https://doi.org/10.1109/ICSME.2016.34

[19] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31âĂŞ47. https://doi.org/10.1145/3314221.3314637

[20] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *PLDI*. ACM, 31–47.

[21] Samir Hasan, Rachary King, and Munawar Hafiz. 2016. Energy Profiles of Java Collections Classes. In *ICSE*.

[22] The shift Project. 2019. *Lean ICT - Towards Digital Sobriety*. Technical Report. 90 pages.

[23] Erik van der Kouwe, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Gernot Heiser. 2018. Benchmarking Crimes: An Emerging Threat in Systems Security. *CoRR* abs/1801.02381 (2018). arXiv:1801.02381

[24] Yewan Wang, David Nörtershäuser, Stéphane Le Masson, and Jean-Marc Menaud. 2018. Potential Effects on Server Power Metering and Modeling. *Wireless Networks* (Nov. 2018). https://doi.org/10.1007/s11276-018-1882-1

[25] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. 2013. Unrestricted and safe dynamic code evolution for Java. *Sci. Comput. Program.* 78, 5 (2013), 481–498.