Analysing and Predicting Energy Consumption of Garbage Collectors in OpenJDK

Marina Shimchenko  
marina.shimchenko@it.uu.se  
Uppsala University  
Sweden

Mihail Popov  
mihail.popov@inria.fr  
Inria  
France

Tobias Wrigstad  
tobias.wrigstad@it.uu.se  
Uppsala University  
Sweden

ABSTRACT
Sustainable computing needs energy-efficient software. This paper explores the potential of leveraging the nature of software written in managed languages: increasing energy efficiency by changing a program’s memory management strategy without altering a single line of code. To this end, we perform comprehensive energy profiling of 35 Java applications across four benchmarks. In many cases, we find that it is possible to save energy by replacing the default G1 collector with another without sacrificing performance. Furthermore, potential energy savings can be even higher if performance regressions are permitted. Inspired by these results, we study what the most energy-efficient GCs are to help developers prune the search space for energy profiling at a low cost. Finally, we show that machine learning can be successfully applied to the problem of finding an energy-efficient GC configuration for an application, reducing the cost even further.

CCS CONCEPTS
• Hardware → Power estimation and optimization: • Software and its engineering → Garbage collection; • Computing methodologies → Machine learning.

KEYWORDS
Energy, GC, Java, Machine Learning

ACM Reference Format:

1 INTRODUCTION
The Information and Communication Technology (ICT) sector uses ≈3% (805TWh) of the global electricity [12], enough to rank it 6th on the list of the most energy-consuming countries in the world [1], and is predicted to continue to grow [2, 4]. Tremendous advances have been made in power-efficient hardware, but potential energy savings are wasted by poor resource and task management and unoptimised software. As Roy and Johnson [30] stated already in 1997, to improve overall energy, software design has to align with energy efficiency goals. In other words, to reduce the energy consumption of a system, one has to optimise every layer of the stack—including both hardware and software.

In this work, we take a first modest step towards energy-efficient deployment by analysing the energy impact of the choice of garbage collector (GC) used to manage memory on a range of Java applications. Based on these, we show that it is possible to train a machine-learning model which is able to predict a suitable GC for a program that reduces its energy usage while still meets service-level agreements expressed in terms of throughput or latency.

The potential impact, in terms of performance and energy, of carefully selecting a GC is great. As Java programs cannot directly access memory, switching between garbage collectors is as easy as changing a command-line flag on start-up. Given the wide base of Java Virtual Machines (JVM) in deployment (estimated by Oracle at over 38 billion in 2017 [22]), even a small energy savings on average has the potential of having an impact on a global scale.

Recently, Ournani et al. [23] investigated the energy impact of choosing different JVMs for deployment. In addition, they looked at the energy impact of GC and found that carefully selecting a GC can reduce energy consumption. Focusing on a single JVM—OpenJDK—we expand their work on comparing GCs by including a more comprehensive range of collectors, as well as by considering the impact on energy when changing heap sizes, Java versions, number of GC worker threads, level of inlining, and compiler optimisations. The observations of this study aim to quantify the role played by GCs in energy consumption and understand the connection between application characteristics and the energy efficiency of GCs.

We formulate the following research questions: RQ1: What is the impact of existing GCs on the energy consumption of Java applications? RQ2: Is there a single most energy-efficient GC? RQ3: How much energy can we save by carefully choosing a GC? RQ4: Is the best way to optimise energy optimising for performance? RQ5: Knowing some characteristics of an application, can we predict the most energy-efficient GC for it? In addressing these, we make the following contributions:

- We explain in detail a method to measure the energy consumption of GCs (§3.2);
- We report on the energy efficiency of six different GCs running established throughput-oriented and latency-oriented benchmarks (§4.2);
- We share guidelines and prerequisites that will help in choosing the most energy-efficiency GC before deployment in case of no other constraints (§4.4);
According to the table provided, the Java GCs used in this work are Serial, Parallel, CMS, ZGC, Shenandoah, and G1. Serial and Parallel are simple mark-compact collectors, while CMS and ZGC are low-latency, parallel, concurrent collectors. Shenandoah and ZGC are low-latency, parallel, concurrent, and non-generational GCs.

We identify a correlation between the energy of GCs and other performance characteristics (§4.6); and provide a machine learning model to predict the most energy-efficient GC for a certain application in case of service level agreement constraints (§5).

Outline: §2 contains a brief overview of Java GC. §3 introduces the experimental protocol and methodology (hardware, projects, tools) we adopted in this study. §4 analyses the results of our experiments on the energy consumption of different GCs. §5 describes how we apply machine learning to predict the most energy-efficient GC. §6 discusses related work.

2 OVERVIEW OF GARBAGE COLLECTION IN JAVA

Java is a memory-safe programming language whose objects’ life cycles are managed by automatic GC. Programs are unaware of the GC algorithm used to manage memory. The Java Virtual Machine (JVM) supports several forms of tracing GCs, which can be controlled at VM start by a flag, e.g., Serial, Parallel, G1, CMS, ZGC, and Shenandoah.

Different GCs exhibit different performance characteristics. For example, Serial and Parallel suspend a program as part of their operation and are good fits for applications that do not suffer from sudden uncontrollable pauses. In contrast, the ZGC and Shenandoah GCs try to avoid suspending the program and carry out their operation concurrently with an executing application. It adds additional costs to running a program to coordinate garbage collection work with program work. Suspending the program for GC work is called a “stop-the-world pause” (STW). Collectors that minimize STW are called concurrent collectors.

Tracing garbage collectors typically carry out their work in a few discrete phases. During the mark phase, the GC identifies which objects are still live, i.e., accessible to the program. This operation typically involves traversing the entire live object graph and suffers from poor cache performance, leading to cache pollution. During the sweep phase, the GC removes all objects which were found not to be live. Serial and Parallel, carry out both these operations in a STW pause. CMS and G1 perform marking concurrently with the application but sweep during a STW pause. Finally, ZGC and Shenandoah carry out both marking and sweeping concurrently and require only extremely short STW pauses, which do not grow with the size of the heap. For efficiency, it is often a good strategy to divide the heap into a number of regions based on objects’ ages and only GC in one region. Because “most objects die young”, GC in the region containing the youngest objects tends to give the most return on investment. Collections of young generations are referred to as minor collections, and collection in old generations is referred to as a major collection.

For fast allocation with good memory locality, many GCs use bump-pointer allocation where allocation moves a pointer forward a number of bytes. It requires free space to be contiguous, which is achieved by moving live objects in memory. Such GCs are called moving/compacting GCs.

In the remainder, we quickly introduce the Java GCs used in this work to facilitate an understanding of our measurements. §1 shows an overview.

2.1 STW Collectors: Serial and Parallel

Serial and Parallel are relatively simple GCs running a mark-compact algorithm in a STW pause (single-threaded in the case of serial, with multiple threads in the case of parallel). Reclamation focuses on the young generation where it is most efficient, with occasional reclamation in the old generation.

2.2 Mostly Concurrent Collectors: CMS & G1

Mostly concurrent collectors perform some but not all of their work concurrently with program threads. Both CMS (Concurrent Mark-Sweep) and G1 (Garbage First, [9]) perform marking concurrently with the application. CMS avoids long pauses by not moving objects around during reclamation, meaning CMS is sensitive to fragmentation. CMS uses brief STW pauses to coordinate work. Still, there are circumstances during which the algorithm will incur longer STW pauses, in particular when a minor collection happens while a major collection is running. CMS was deprecated in Java 9 and removed in Java 14, but remains popular [29].

G1 partitions the memory into regions, and each region can belong to the young or old generation. G1 performs reclamation in a STW pause. G1 selects only a few young regions (and optionally old regions) and reclaims them in each GC cycle to ensure low pause times. Thus G1 incrementally collects garbage during multiple pauses instead of a single large pause like Serial or Parallel.

2.3 Concurrent Collectors: Shenandoah & ZGC

Shenandoah [11] and ZGC [20] are low-latency, parallel, concurrent, and non-generational GCs. (Generational ZGC is under development at the time of writing this.) They both implement algorithms whose STW pause times do not increase with the size of the heap, including concurrent evacuation of pages during reclamation (meaning regions of memory are freed by moving all live objects away from the page).

Both Shenandoah and ZGC use load barriers to trap accesses to relocated objects and remap dangling pointers to point to their updated location before going ahead. Shenandoah uses write barriers to coordinating GC threads and application threads during marking. The ZGC algorithm forces all pointers on the heap to be remapped once per GC cycle, and makes frequent use of atomic
operations such as compare and set to coordinate GC workers and program threads operating concurrently on the same objects.

3 METHODOLOGY

In this section, we present our benchmarks, the methodology we used to measure energy consumption, and the techniques to change GC configurations.

3.1 Java Benchmarks

To the best of our knowledge, there is no standard benchmark in the community for measuring energy for Java server workloads. Thus, we perform our measurements on multiple sets of benchmarks (BMs) to cover a wider range of real-world applications to get representative energy measurements. In particular, we use DaCapo [6], Renaissance [27], Hazelcast [13], and Specjbb [35]. We divide these benchmarks into two categories: "throughput-oriented" and "latency-oriented". Benchmarks that explicitly measure the response time of an application (Hazelcast and Specjbb) fall into the latency-oriented category of BMs. Benchmarks where the original performance metric is execution time without considering jitter or stalls (§3.1) are throughput-oriented (all programs in DaCapo and Renaissance).

3.1.1 Throughput-Oriented BMs

In this section, we present our benchmarks, the methodology we used to measure energy consumption, and the techniques to change GC configurations.

3.1.2 Latency-Oriented BMs

In contrast to the throughput-oriented benchmarks, Hazelcast and Specjbb does not use fixed workloads but instead adjust them dynamically to saturate the machine, and having very few long-living objects. DaCapo consists of ten applications, some of which have different working set sizes options (see §2). We omitted tradesoap due to a known concurrency bug [5].

Renaissance is a benchmark suite of concurrent and object-oriented workloads that exercise various concurrency primitives of the JVM. Focusing on concurrency and parallelism, the Renaissance benchmark mirrors a part of modern parallel workloads which DaCapo does not. The suite consists of twenty-five programs, but we excluded db-shootout and neo4j-analytics due to unsupported Java versions and rx-scarbel due to absence of GC work.

3.1.3 Energy Consumption

Due to the lack of latency-oriented benchmarks in the community, we opted to use GC pause times in throughput-oriented BMs as a proxy for latency scores. While approximate, this is reasonable as GC pauses lower bound application response times. This approximation can also be corroborated experimentally. For example, Specjbb reports so-called critical-jOPS, the rate of upcoming requests when an application latency becomes longer than a chosen interval (10ms, 25ms, 50ms, 75ms, and 100ms). Thus, longer latency leads to lower critical-jOPS. §1a shows the critical-jOPS for different GCs. STW collectors, with the longest GC pauses, Serial and Parallel, have the lowest critical-jOPS. Semi-concurrent collectors, G1 and CMS that only incur STW pauses for certain GC activities are in the middle. Fully concurrent collectors, whose STW pauses are short and do not scale with the size of the heap, are on top. §1b shows the same trend for the Hazelcast BM. The latency metric in Hazelcast is different from Specjbb. It reports the time from issuing a request to its completion, meaning lower is better. Still, we observe the same trend: fully concurrent collectors to the right, primarily concurrent collectors in the middle, and STW collectors to the left. (As one would expect.)

Thus, for this work, we define throughput and latency as:

Definition 3.1 (Throughput). The amount of work produced over a given period. Higher throughput leads to lower execution time. Thus, in this work we consider execution time as a performance metric for throughput.

Definition 3.2 (Latency). In this work, we defined latency as waiting time introduced due to GC work. This metric is not perfect, and is due to the lack of standard benchmarks that provide latency scores.

3.1.3.1 Throughput-Oriented BMs

In contrast to the throughput-oriented benchmarks, Hazelcast and Specjbb does not use fixed workloads but instead adjust them dynamically to saturate the machine entirely. Thus, changing the GC impacts the amount of work carried out in the benchmark, making it impossible to make energy comparisons. To counter this, we set Specjbb to run at a fixed load (injection rate) below the machine’s capacity (which is representative of an actual deployment). We used the same approach for Hazelcast (details follow shortly). It makes our Specjbb results non-compliant, which is not relevant to us since we are not using the Specjbb performance metrics (max and critical j-OPS). Specjbb simulates the work of an online store. We used the max-jOPS parameter from the Specjbb report to establish the maximum injection rate (IR) for each GC to identify the appropriate level for a fixed workload (same for all GCs). For example, the highest IR we measured was 37007 for j13P with a default number
Table 2: Benchmark Parameters. # = iterations per run. We run each benchmark 10 times for stability.

<table>
<thead>
<tr>
<th>BMs</th>
<th>Suite</th>
<th>Size</th>
<th>#</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>hazelcast</td>
<td>DaCapo</td>
<td>small</td>
<td>50</td>
<td>300m</td>
</tr>
<tr>
<td>hazelcast</td>
<td>DaCapo</td>
<td>large</td>
<td>30</td>
<td>1200m</td>
</tr>
<tr>
<td>avrorra</td>
<td>DaCapo</td>
<td>large</td>
<td>17</td>
<td>45m</td>
</tr>
<tr>
<td>flop</td>
<td>DaCapo</td>
<td>default</td>
<td>50</td>
<td>135m</td>
</tr>
<tr>
<td>jython</td>
<td>DaCapo</td>
<td>large</td>
<td>20</td>
<td>135m</td>
</tr>
<tr>
<td>luindex</td>
<td>DaCapo</td>
<td>default</td>
<td>30</td>
<td>21m</td>
</tr>
<tr>
<td>lusearch</td>
<td>DaCapo</td>
<td>large</td>
<td>20</td>
<td>21m</td>
</tr>
<tr>
<td>pmid</td>
<td>DaCapo</td>
<td>large</td>
<td>30</td>
<td>150m</td>
</tr>
<tr>
<td>sunflow</td>
<td>DaCapo</td>
<td>large</td>
<td>20</td>
<td>60m</td>
</tr>
<tr>
<td>xalan</td>
<td>DaCapo</td>
<td>large</td>
<td>20</td>
<td>35m</td>
</tr>
<tr>
<td>biojava</td>
<td>Renaissance</td>
<td>default</td>
<td>25</td>
<td>525m</td>
</tr>
<tr>
<td>als</td>
<td>Renaissance</td>
<td>default</td>
<td>30</td>
<td>1455m</td>
</tr>
<tr>
<td>chi-square</td>
<td>Renaissance</td>
<td>default</td>
<td>60</td>
<td>1455m</td>
</tr>
<tr>
<td>dec-tree</td>
<td>Renaissance</td>
<td>default</td>
<td>40</td>
<td>1455m</td>
</tr>
<tr>
<td>gauss-mix</td>
<td>Renaissance</td>
<td>default</td>
<td>40</td>
<td>1455m</td>
</tr>
<tr>
<td>log-regression</td>
<td>Renaissance</td>
<td>default</td>
<td>20</td>
<td>1455m</td>
</tr>
<tr>
<td>movie-lens</td>
<td>Renaissance</td>
<td>default</td>
<td>20</td>
<td>1455m</td>
</tr>
<tr>
<td>naive-bayes</td>
<td>Renaissance</td>
<td>default</td>
<td>30</td>
<td>3825m</td>
</tr>
<tr>
<td>page-rank</td>
<td>Renaissance</td>
<td>default</td>
<td>20</td>
<td>1875m</td>
</tr>
<tr>
<td>akka-uct</td>
<td>Renaissance</td>
<td>default</td>
<td>24</td>
<td>705m</td>
</tr>
<tr>
<td>fj-kmeans</td>
<td>Renaissance</td>
<td>default</td>
<td>30</td>
<td>285m</td>
</tr>
<tr>
<td>reactors</td>
<td>Renaissance</td>
<td>default</td>
<td>10</td>
<td>1500m</td>
</tr>
<tr>
<td>future-genetic</td>
<td>Renaissance</td>
<td>default</td>
<td>50</td>
<td>30m</td>
</tr>
<tr>
<td>mnemonics</td>
<td>Renaissance</td>
<td>default</td>
<td>16</td>
<td>180m</td>
</tr>
<tr>
<td>par-mnemonics</td>
<td>Renaissance</td>
<td>default</td>
<td>16</td>
<td>180m</td>
</tr>
<tr>
<td>rx-scrabble</td>
<td>Renaissance</td>
<td>default</td>
<td>80</td>
<td>35m</td>
</tr>
<tr>
<td>scrabble</td>
<td>Renaissance</td>
<td>default</td>
<td>50</td>
<td>30m</td>
</tr>
<tr>
<td>dotty</td>
<td>Renaissance</td>
<td>default</td>
<td>50</td>
<td>180m</td>
</tr>
<tr>
<td>philosophers</td>
<td>Renaissance</td>
<td>default</td>
<td>30</td>
<td>30m</td>
</tr>
<tr>
<td>scala-doku</td>
<td>Renaissance</td>
<td>default</td>
<td>20</td>
<td>105m</td>
</tr>
<tr>
<td>scala-kmeans</td>
<td>Renaissance</td>
<td>default</td>
<td>50</td>
<td>105m</td>
</tr>
<tr>
<td>scala-stm-bench7</td>
<td>Renaissance</td>
<td>default</td>
<td>60</td>
<td>930m</td>
</tr>
<tr>
<td>finagle-chirper</td>
<td>Renaissance</td>
<td>default</td>
<td>90</td>
<td>180m</td>
</tr>
<tr>
<td>finagle-http</td>
<td>Renaissance</td>
<td>default</td>
<td>12</td>
<td>105m</td>
</tr>
<tr>
<td>specjbb</td>
<td>SpecJbb2015</td>
<td>default</td>
<td>1</td>
<td>2.7g</td>
</tr>
<tr>
<td>specjbb</td>
<td>SpecJbb2015</td>
<td>default</td>
<td>1</td>
<td>32g</td>
</tr>
</tbody>
</table>

If we keep the same allocation rate for the same amount of time, we get the same workload regardless of a GC.

In addition, we decide to experiment with multiple keyset sizes similar to multiple IRs in Specjbb15: 100–80, 80–60, 60–40, 40–20. Thus, we report the results of four different configurations of Hazelcast.

3.1.3 Approximating Latency Scores. In addition to execution time, we measured the GC pause times incurred by different GCs as an approximation for a latency score. We calculate the maximum (L_max) latency among all the measured pauses, which specifies the system’s hard upper bound latency requirement.

As mentioned above, Specjbb15 uses a latency metric called critical-jOPS. We do not use it for two reasons. First, the metric does not consider latencies shorter than 10ms. Therefore, it is unfair to low-latency GCs (which were not around when the metric was defined). Second, using this metric would complicate comparison with the approximated latency scores in the programs in DaCaPo and Renaissance, which are GC pause times. Thus, we use GC pause times also for Specjbb15.

Hazelcast includes its latency metric similar to Specjbb15. But, for the reasons above, we follow the same protocol of measuring latency in Hazelcast as in all other BMs.

3.2 Measuring Energy

In this section, we explain how we measure the energy consumption of GCs in Java. A naive solution consists of only profiling the GC activity to estimate its energy consumption. However, the GC activity directly impacts the application behavior. GCs cause memory to be rearranged, which influences the energy consumption in other parts of the application due to cache effects. Therefore, to consider the full impact of the GCs activity, we profile GCs together with the application on the system.

Recent Intel systems support RAPL [16]: an interface to read model-specific registers (MSRs). The MSRs store data and settings related to the processor itself and can be used to estimate the energy consumption across the different domains of the system. RAPL considers four domains: 1) PKG: the entire package composed of cores, shared caches, memory controllers, and optional uncore devices, 2) PP0: the cores, 3) PP1: the uncore device, usually a GPU (which was not used in this study), and 4) DRAM: the memory. To evaluate the energy consumption across the entire system, we profile both PKG and DRAM, using jRAPL [21]. jRAPL enables us to insert probes to monitor regions of interest in the applications.

This is especially useful as DaCaPo and Renaissance benchmarks have a relatively short execution time: they are susceptible to the cold cache bias effects. To address this issue and avoid profiling the JVM’s start-up phase, we define a warm-up phase for each benchmark and only start recording after the iterations’ behaviors stabilize. We use the built-in support to run monitoring code around application iterations for DaCaPo and Renaissance. The total number of profiled iterations per benchmark is reported in §2. We did not use any warm-up mechanism for Hazelcast and Specjbb as they have a long execution time which naturally makes the impact of the warm-up phase negligible.

To check and improve the quality of our measurements, we repeated our executions: we profiled each application at least 10
times and reported the arithmetic average across repetitions. To preserve a consistent system state across repetitions, we executed cache flushes between each profiling run and set the CPU frequency governors to performance mode to prevent the OS from lowering the CPU frequency between executions. We calculated the relative standard deviation per benchmark across our repetitions and found it to be consistently below 5%, suggesting that our results are stable. This guarantees that the ML models from §5, which require a single profiling run to predict a GC configuration, can safely be deployed without concerns regarding execution stability. Changing the CPU governor is an additional method to save energy: the complex interaction between GCs and frequency scaling is a potential future direction to investigate.

3.3 Heap Size Matters

As noted by Blackburn et al. [7], managed languages add a degree of freedom to experimental evaluation, a space-time trade-off due to garbage collection, in which heap size is a control variable. Therefore, our experiments would not be complete without estimating the impact of heap size on the energy consumption of different GCs. We use a standard approach [19] to pick a default heap size for an application. We calculate our default so that it is $3 \times$ larger than the minimum heap size required for the application to run to completion with Serial GC (see §2).

To further explore a larger spectrum of behaviors, we also consider heap sizes $4.5 \times$ and $6 \times$ the minimum heap size required by Serial. Our main motivation for this is that concurrent collectors such as ZGC need larger heap sizes to avoid allocation stalls. However, on the opposite, smaller heap sizes remain more efficient for STW GCs.

3.4 Changing Java Configurations

We perform our measurements on Java 16 and Java 13. The former is a recent version, while the latter is the last version that supports CMS. Even though it is deprecated, CMS is popular as it is presumed to consume low energy\(^1\). As JVM, we use OpenJDK as one of the most common, open-sourced, and well-established JVMs.

For JVM invocations, we used the parameters $-Xlog:gc* \text{ and } -XX:+DisableExplicitGC$. The first parameter dumps GC log information which allows us to determine the number of GC cycles, distinguish minor collections from major collections, etc. We disabled explicit GC because some BMs call explicit GCs inside their code, which we believe interferes with the default heuristics of GCs. For example, each GC has a different trigger for starting a cycle. G1 tests how many regions the young generation can consist of and still be collected within a given GC pause time target. Shenandoah initiates a garbage collection cycle based on observations of time, heap occupancy, and allocation pressure from previous garbage collection cycles. Therefore, the number of cycles for each GC will differ, which we expect. However, inserting explicit GC cycles will change the “natural” behavior of GCs. In this case, it is not clear if what we measure is ever representative of a GC or a result of artificial manual policies. Finally, we used $-XX:+<\text{Name}>GC$ to select a GC for the the whole program (e.g., Parallel1GC, or G1GC).

4 RESULTS

In this section, we present our results by profiling energy over a range of BMs (§3.1) with different GCs configurations.

4.1 Experimental Setup

To conduct our experiments, we used an Intel Xeon SandyBridge EN/EP server machine running Oracle Linux Server 8.4. The machine has 32 identical CPUs, which can be configured into a two NUMA node configuration or used as a homogeneous single node machine. In our case, we use only one NUMA node for all the experiments, hence 16 cores. The CPU model is Intel® Xeon® CPU E5-2680 with 64K L1 cache, 256K L2 cache, and 20 M L3 cache.

4.2 Energy Impact

We start by answering RQ1 by reporting the impact of GCs on a program’s energy consumption. §2 is a heat map that shows energy consumption for all the BMs (vertical labels) and tested configurations (horizontal labels).

Horizontal labels represent different GC configurations. The first three letters indicate the Java version: j13 for Java 13 and j16 for Java 16. Then we specify a GC—Ser(ial), Shen(andoah), P(arallel), Z, G1, and CMS—and the heap size ($3 \times$, $4.5 \times$, $6 \times$). The part after the ' ' adds particular parameters: $n1$, $n2$ and $n4$ for different number of GC worker threads; $M19$ for maximum inlining level. All values are normalized to the default GC, of the most modern OpenJDK in our study: j16G13x.

We applied a hierarchical clustering and ordered the BMs according to the resulting dendrogram. To calculate distances between applications, we aggregated their energy consumptions across the different GCs into vectors. We expect that applications in similar clusters (i.e., generally close to each other in the order) will be affected in a similar way by different GC configurations. Therefore, if we know to what cluster an application belongs, we can predict its most energy-efficient GC based on the behavior of the applications in the same cluster. We develop a similar approach in §5.

As a trend within our observations, energy decreases with the increased heap size due to reduced GC activity. However, increasing memory too much might e.g., worsen locality, so this trend should not be extrapolated to the general case.

We observe that ZGC and Shenandoah, the fully concurrent collectors, are less energy-efficient than all other collectors. There are multiple reasons for that:

- The ZGC and Shenandoah implementations under test are single-generation. Thus each GC cycle, they go through a more extensive chunk of memory.
- Concurrency costs. Concurrent GCs must carry out additional work to synchronize worker threads with mutators, which is not needed in less concurrent GCs. For example, the remapping of all pointers in a GC cycle in ZGC requires atomic operations.

Reading the Heatmaps. Energy values are normalized to the default GC for Java 16 with the default heap size, G1 with heap size 3\times. Thus, j16G13x values are equal to 1 for all the BMs, and its column is white. The lower (higher) the energy consumption, the greener (pink) is a cell. 6 means the energy consumption is the 6\times more.

\(^1\)Members of the Oracle GC team, personal communication.
4.3 Differences in Energy Efficiency Between Java Versions

§2 shows that energy consumption varies between Java versions. For example, Java 13 GCs (Serial, Parallel, and CMS) are more energy-efficient than their Java 16 analogs while keeping the same performance. Notably, Ournani et al. [23] compared Java 15 and Java 8 using a different set of BMs and observed the opposite trend, i.e., Java 15 was more energy-efficient than Java 8. It confirms that Java releases vary in energy efficiency. Also, there is no clear rule (yet) for which version to use for energy efficiency.

We noticed that only Renaissance was susceptible to changing the Java version. According to Oracle developers, this BM is sensitive to inlining. By investigating further, we realized that the main difference between Java 13 and 16 (with respect to this work) is in the default flag -XX:MaxInlineLevel=9. We conclude that inlining indeed affects the energy efficiency of the Renaissance BM. Java 16 with the adjusted inlining level is almost as energy-efficient as Java 13.

Following this insight, we decided to extend our observation to CMS. Our analysis suggests that the apparent energy efficiency of CMS in Java 13 is not due to something fundamentally different in the CMS algorithm but rather from unrelated changes in inlining. We base this claim on comparing two versions of GCs that have not changed across the versions, Serial and Parallel. Both GCs show a significant difference in energy use between Java releases when run with default flags, but using consistent inline across versions removes these differences almost entirely.

To conclude, energy impact varies not only with implementations of different GC algorithms but also with different versions of OpenJDK using the same GC algorithm. In the latter case, differences may stem from changes unrelated to GC, such as changes to parameter defaults in different versions of OpenJDK. We also see that the behavior of the same GCs varies between applications, and if for some programs, Parallel GC is the most energy-efficient choice (e.g., naive-bayes), for others, it is the most energy-hungry (e.g., lucene_default) as represented by changing horizontal color patterns in §2 (RQ1).

4.4 Choosing the Most Energy-Efficient GC

Having established a clear difference in energy-efficiency between different GC’s, we embark on RQ2—identifying the most energy-efficient GC.

Picking the most energy-efficient GC for a particular application requires adequate profiling with all GCs of interest and multiple heap sizes. Furthermore, ensuring statistically verifiable results requires multiple runs with multiple iterations in each. In §4a, we show the return on investment of including multiple GCs in addition to Java’s default in the search space for energy efficiency. The order of the GCs on the x-axis is based on the measured energy than the energy needed by j16G13x. 0.3 represents (≈ 1/3) of the energy required by j16G13x.

Figure 2: Relative energy consumption for all BMs and GC configurations. Each GC configuration has 3 columns for 3×, 4.5×, and 6× of the minimum heap size (see §3.3 for more information on heap sizes.)

Figure 3: Comparing energy consumption of Java 13 and Java 16 with -XX:MaxInlineLevel=9 for Serial and Parallel GCs in the Renaissance BMs. Columns as in §2.
efficiency of our benchmarks. For example, profiling the default configuration and Java 13’s ParallelGC with 6x heap size and a single worker thread can save a bit more than 40% of energy (attainable by exploring the entire search space and picking the best). By including SerialGC (j13Ser6x) and ParallelGC with four worker threads (j13P6x_n4), we can save around 45%, i.e., run the benchmarks using only 55% of the energy needed by the default configuration. There seems to be a law of diminishing returns, which is unknown to the deployer unless one first performs the work we are trying to prun.

By trying only three out of many GC configurations (j13P6x_n1, j13Ser6x, j13P6x_n4), we can obtain reasonably good energy saving while keeping the cost of exploration low. Our results relate to the ones presented by Ournani et al. [23]. They conclude that even though G1 is stable and efficient in general, there are BMs that significantly benefit from using Parallel or Serial GCs.

But how much energy can one save without sacrificing performance? For example, suppose we do not permit performance to drop more than 5% or 10% below the default configuration (j16G13x) in order to save energy. Fig 4b and 4c show the two cases, respectively. The first observation is that maximum energy savings decrease as performance constraints increase. This is because fewer GCs satisfy the requirements. So, with these constraints (5% and 10%), the maximum energy savings drop to 40%, compared to 47% without constraints.

The second observation is that the order of GCs (see §4a, 4b and 4c) differ even with the slightest change of constraints. Thus, a new manual exploration is needed if Service-Level Agreement requirements (SLA) change. It increases the cost of manually choosing the right GC for energy-efficient deployment.

As noted above, where there are no SLA constraints, the value of adding more GCs to the search space decreases quickly. These results hold up also when considering performance constraints (§ 4b and 4c). For example, with only two GC configurations (j16P4.5x_n1 and j16G16x), we can get a significant portion (~97%) of the maximum energy savings (rightmost bar) in both cases.

We summarize these findings in §3 that show the necessary GC configurations worth exploring to get at least 95% of the maximum energy savings, which is typically two and at most three. It is one of the most important contributions of this work—providing developers with practical advice on the low-cost improvement of the energy efficiency of their applications.

In conclusion, it can be costly to identify the most energy-efficient GC for each particular BM and SLA. However, we determined that the search space can be decreased to at most three configurations in each case (RQ2).

### 4.5 Energy Savings Under Service-Level Agreement Constraints

In the previous section, we have already started answering RQ3: How much energy can we save by carefully choosing a GC? If energy is the only concern, we can get the same work done with around 47% less energy by explicitly selecting a GC and not using the default GC choice. However, as mentioned previously, energy is typically not the only concern. Most deployments have SLA constraints on throughput and/or latency. An typical example is a real-time system with a large heap. Using a collector like Parallel, whose pause-time is proportional to the size of the heap, might lead to GC pauses that, in turn, lead to SLA violations. To this end, this section focuses on the possible average energy savings of different SLA scenarios if choosing a GC carefully.

We measured the energy consumption for two types of constraints: max_1 and execution time. The restriction on the latency means that we only chose from the GCs, which have a latency below a required value, such as 10 ms (max_1_10), 50 ms (max_1_50), 100 ms (max_1_100), 500 ms (max_1_500), 1000 ms (max_1_1000), and 2000 ms (max_1_2000). The execution time constraint weakens the performance criteria—we allow performance to be X% lower than for G13x, where X is 5% (perf_5%), 10% (perf_10%), 50% (perf_50), and 100% (perf_100). Fig 5 shows the average energy savings across all the BMs for each scenario above.

We normalize against a GC that we believe is a sensible “safe choice” for the SLA constraint in point, following our table from §2. Thus, we normalize energy with our strictest latency constrain (<100 ms) to j16Z6x (which typically achieves sub-millisecond pauses), and >100 ms to j16G13x (as low pause times clearly are not needed, and G1 typically has better throughput than ZGC, which manifests itself as lower energy consumption, as per §2). Following this reasoning, we consider j13P6x_n1 the most sensible safe choice for performance. According to our findings, this GC is the most energy-efficient GC on average (§4a). In addition, the Parallel collector is suitable for optimising throughput when latency is not an issue.

As shown in §5, our previous observations that savings increase when constraints relax holds for latency as well as execution time. Picking a collector is more crucial for energy in case of latency SLA than performance as we can save 45% for a few latency scenarios vs. 25% for perf_100.

Now, consider two scenarios, one for latency (max_1_100) and one for performance (perf_50). These scenarios are in the middle
### 4.6 Correlation Between Energy and Performance

A “folklore truth” is that optimising for execution time also optimises energy consumption as a side-effect, *i.e.*, the faster we can finish performing some work $W$, the lower is the total energy cost of performing $W$. In RQ4, we ask whether this correlation between increased performance and decreased energy consumption indeed holds, or whether there is a case for optimising energy separately.

For throughput-oriented BMs, we define performance as throughput divided (§3.1) by execution time ($T$) in our search for correlation between performance and energy ($E$). §7 shows the dependency between execution time and energy separately for each GC. $T$ has a linear correlation to $E$ varying in strength between GCs. For example, ZGC has a 0.95 correlation, and CMS only 0.29. Also, Java 13 collectors show the weakest correlation (§7), which indicates that there might be variability between all Java versions.

For latency-oriented BMs, we plot maximum latency ($L_{\text{max}}$) to $E$ for all BM–GC combinations in §6b. As one can see, $L_{\text{max}}$ is independent—the pause-time induced by a particular GC does not correlate with energy efficiency.

Overall, we conclude that the folklore truth (RQ4) holds for some GCs: optimising execution time leads to lower $E$ for Java 16 ZGC, Shenandoah, Serial, and Parallel with multiple worker threads. Meanwhile, the same is not true for latency and Java 13 collectors. Our findings add additional depth to the conclusion of Khokhriakoc et al. [17]. They studied energy proportionality to performance in modern multicores processors. Their results show that decreased performance does not lead to a proportional decrease in energy and may even have an opposite effect.

Therefore, it might be counterproductive to optimise performance to get better energy in many cases. Thus, either energy should be the main focus of optimizations or be tangled with performance optimization (RQ4).

### 5 USING MACHINE LEARNING TO PICK THE MOST ENERGY-EFFICIENT GC

The results from the previous sections show that for a given performance or latency constraint, picking “the right” garbage collector can lead to considerable energy savings. However, for each new application and additional constraint on latency or performance, there is a need to profile many GCs to select the most efficient one. This section explores how we can reduce this profiling cost using Machine Learning (ML).

---

**Figure 4:** [higher is better] Possible energy savings from profiling additional GCs (i.e., 0.45 means a 45% reduction in energy). Note that the energy savings decrease with more strict constraints.

**Figure 5:** [higher is better] The maximum average energy savings achieved by choosing the most energy-efficient GC for each BM. The numbers are normalized to ZGC6x for max$_l_{(10,50,100)}$ (as ZGC leads in low-latency), j16G13x for max$_l_{(500,1000,2000)}$ (as G1 typically has higher throughput than ZGC) and j13P6x$_n_1$ for performance (as ParallelGC avoids paying any overhead for running concurrently with mutators).
To train our model, we assess for each application a large set of
application characteristics (called features). We then group both labels and
features in §5.2. Finally, we show our results in §5.3.

5.1 Training and Validating ML Models
Figure 8 presents the workflow of the ML models that we train and
evaluate. We start by profiling a set of training applications. We
collect features that we subset (detailed in §5.2). In parallel, we
also evaluate all the GC configurations and return the most effi-
cient ones per application as labels. We then group both labels and
features and provide them to machine learning algorithms that
generate prediction models. Different algorithms operate more or less
efficiently depending on the available data. Therefore, we empiri-
cally evaluated different ML algorithms, including Support Vector
Machine (SVM), Neuronal Networks (NN), Decision Tree (DT), and
Linear Regression (LR) using model bucketing [38]. We implement
the different learning algorithms with the default Scikit [25] learn
setup for DT and LR and just increased the iterations count for the
other methods (i.e., 2000 and 4000 for LR and NN respectively).
To evaluate and validate our models, we deploy them over new un-
seen applications only based on their features. For the rest of
this section, we describe the training and validation of our machine
learning algorithms in §5.1. We then present how we select our
features in §5.2. Finally, we show our results in §5.3.

5.2 Application Characteristics (Features)
ML models use features to predict what configuration to use at
deployment. As previously described in §4, optimising GC energy
requires exploring many different configurations. On the other
hand, an application’s features can be cheaply collected through

We develop a ML algorithm that predicts a new application’s
most efficient GC configuration on a given execution scenario. In-
stead of measuring the energy/efficiency of an application multiple
times with different GCs each time, our goal is to make profiling
a single configuration enough to generate input to the ML model.
To train our model, we assess for each application a large set of
application characteristics (called features) along with the different
GC configurations (called labels). Features and labels are provided
together to a ML algorithm to create a model that predicts labels for
new unseen applications only based on their features.
light profiling. We investigate features that maximize information to make accurate predictions while minimising collection overhead to reduce deployment costs. Performance counters are a standard feature candidate for performance and energy tuning that we explore in this work. They are available across many systems and characterize the execution of an application.

To collect performance counters as features, we use the Java marker API for likwid [18]. Likwid [36] provides an interface to collect counters aggregated in groups sharing similar events. To broadly characterize the applications’ execution, we profile the following diverse groups: BRANCH, CYCLESTALLS, DATA, ICACHE, L2CACHE, L3CACHE, MEM_DP, NUMA, and TLB_DATA. To get consistent measurements with jRAPL and collect fine-grained measurements, we directly profile the regions of interest using the likwid probes API. We collect all counters from the applications using the default GC configuration j16G13x. Finally, to preserve consistent values across features, we normalize each counter by the highest value recorded for that counter across all the applications.

We can collect performance counters by either executing the application over a single configuration or across different ones. Collecting counters from a single configuration is more straightforward as we do not need to change the GC across executions. However, collecting counters across configurations has higher tuning potential in the context of compiler [39] or runtime [32] optimization. By profiling the same counter across different configurations, we measure its reaction to various contexts. This reaction is valuable information to guide the optimization, a so-called reaction-based profiling [32, 39]. We further optimise our models by collecting features using reaction-based profiling of performance and energy. First, we collect them across all the GC configurations and then normalize their values against j16G13x. The performance or energy changes between j16G13x and all the other GC configurations are reaction-based features that we use to train our models. We profile 42 performance counters and evaluate 36 different GC configurations resulting in 72 reaction-based performance and energy measurements. We, therefore, consider 114 various features.

Using multiple features for a model increases its information and prediction potential. However, it creates noise as some information might not be relevant to the optimization. More importantly, it increases the profiling cost at deployment. To reduce deployment cost, we use feature selection: we train different models using only one feature at a time.

Training models using one feature at a time significantly complicates the learning process. Indeed, we evaluate ten models per learning algorithm for each execution scenario because of cross-validation. In addition, we consider four learning algorithms and 114 features. It results in a total of 4560 models per scenario. However, this enables us to deploy learning algorithms that predict the best GC configuration using only a single feature. In other words, to optimise an application, we only need to profile a single feature.

Collecting one feature at deployment requires profiling the application once or twice. If the feature is a performance counter, we can collect it with one run and then normalize it with the highest recorded value for that counter. Otherwise, the feature is reaction-based, and we need to profile the application twice: with the configuration of the feature and with j16G13x to normalize it.

5.3 Prediction Results

We now present the prediction savings of the previously described models. We validated them for each execution scenario with cross-validation. Our method predicts with 100% accuracy the most energy-efficient GC configuration across all the 36 different GC configurations with a single profiling run over all the execution scenarios except perf_50 (which is predicted with 98.8% accuracy).

Two aspects were essential to achieve this accuracy: the machine learning algorithm and the features selection. Figure 10 presents the different machine learning algorithms that we evaluated. For each learning algorithm, we select its most efficient feature and report the resulting average energy savings across all applications from the validation folds. We observe how DT significantly outperforms other learning algorithms. Therefore, selecting the learning algorithm is essential for efficient GC energy tuning. Similar results were observed in the context of thread and data mapping [32].

Figure 9 shows the impact of feature selection on the prediction models. We present how different features affect the prediction accuracy of our most efficient learning algorithm, DT. We replicate the experiment with two cross-validation fold distributions to validate these results. Observing similar results across fold distributions further justifies that our approach can be applied consistently over new cases.

Features have a significant impact on prediction accuracy. Depending on the feature, the models either perfectly predict the GCs or almost always miss (i.e., 80% error). Interestingly, some features are consistently efficient for GC prediction across different fold distributions or execution scenarios (i.e., a given feature provides a relatively constant prediction accuracy across scenarios and fold distributions). Our intuition is that some features carry intrinsic information agnostic to the execution scenario or the fold distribution,
We replicated the experiment two times with different cross validation folds distributions, illustrated by validation code distribution 1 (V1) and 2 (V2). We observe that each feature provides mostly similar accuracy predictions across the different execution scenarios and fold distributions. Therefore, we do not need to specialize features per execution scenario as accurate features on one scenario are likely to be accurate on others.

Horizontal lines separate 100% accurate features from other features for each type of constraint and validation folds.

**X-axis:** feature used for training the models.

**Y-axis:** average prediction error across all benchmarks.

**Colors:** Different execution constraints.

Figure 9: DT prediction using different features. [lower is better] (Explanation above.)
which the DT learning algorithms can harness. Such features (e.g., L1 miss rate, branch information, or energy profiled with j13p1_n4) include both counters and reaction-based profiling. Using a counter feature is attractive as we can deploy the prediction model with a single profiling run. However, as reaction-based profiling manages to predict the GCs only from reaction-based energy profiling, we can deploy our method over different systems where some performance counters might not be available. As future work, we can further investigate the correlations between the features and the models decisions to better understand the GCs behaviors.

Our results suggest that building an ML tool that can efficiently predict energy-efficient GC configurations is practical. Furthermore, such a tool would avoid costly tuning and only need profiling of a specific performance counter to select the most efficient configuration. Therefore, we conclude that it is indeed possible to choose the most energy-efficient GC from “some” application characteristics (RQ5).

6 RELATED WORK

Likely due to the energy constraints of embedded systems, there has been a long-standing interest in the energy consumption of garbage collectors for embedded systems. For instance, Velasco et al. [37] investigated the energy consumption of different garbage collection techniques on embedded devices. Their approach is based on JikesRVM running in a simulator, and they are therefore able to provide a very fine-grained breakdown of energy consumption for different GC techniques. Taking a more constructive approach, Griffin et al. [14] designed a new energy-efficient GC for embedded systems and evaluated it in a hardware simulator. Due to a combination of almost two decades of technical progress, hardware development, different focus, and differences in methodology, comparing their results with ours is not very useful.

More recently, researchers looked into the energy consumption of Java for desktop and server workloads. Ournani et al. [23] experimented with energy for different JVM platforms as well as performed tests with various GCs. Our research complements this work. We give a more comprehensive study with a broader range of GCs for throughput-oriented benchmarks and latency-oriented ones. Additionally, we make a prediction model for the best energy-efficient GC based on the collected features of the applications. We discuss similarities and differences in our findings at a greater length throughout the paper.

Fernandes et al. [10], Pinto et al. [26], and Hasan et al. [15] study the effect of Java collections on energy consumption with a focus on different data structures and algorithms’ implementations. These studies show that careful choice of GC implementation can yield 300% energy savings. Even though the average savings that we report go up to 55% only, we can achieve them without substantial changes to the program. For example, we propose energy optimization—selecting a GC—that can be done by changing one VM parameter. Still, the savings are substantial enough to make a difference.

ML is used to optimise the energy consumption of the runtime or the Operating System. Roy et al. [31] provided energy savings by auto-tuning OpenMP parameters and frequency scaling: a pool of Bayesian Optimization models smartly navigates the tuning space and prune sub-optimal configurations. Curtis-Maury et al. [8] collected performance counters and used them with NNs to optimise Concurrency Throttling. Su et al. [33] also used NNs to predict energy delay product values on NUMA systems while Bailey et al. [3] predicted power consumption values through LR and clustering. ML techniques are also optimising the compiler. Wang and O’Boyle [39] reviewed different ML strategies to optimise the compiler phase ordering problem. In contrast, [24] quantified the gap between performance and energy optimization in an iterative compilation. To the best of our knowledge, this paper is the first that uses ML to optimise GC energy consumption.

7 CONCLUSION

In this paper, we analysed energy on a wide range of garbage collectors and benchmarks. We discovered that considerable energy savings could be achieved without invasive changes to the program by carefully choosing a collector. Moreover, we identified the most energy-efficient GCs for multiple scenarios with different performance constraints to help developers save energy at the cost of 2–3 runs. In addition, we explored machine learning possibilities to reduce the cost of finding the most energy-efficient GC for each application to one profiling run for a single configuration (given that the variance is sufficiently low).

REFERENCES
