Digital Comprehensive Summaries of Uppsala Dissertations
from the Faculty of Science and Technology 2412

Optimizing Energy Efficiency of Concurrent Garbage Collection

MARINA SHIMCHENKO

ACTA UNIVERSITATIS
UPSALIENSIS
2024

ISSN 1651-6214
ISBN 978-91-513-2155-4
urn:nbn:se:uu:diva-527713

UPPSALA
UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in 101195, Heinz-Otto Kreiss, Ångström, Lägerhyddsvägen 1, Uppsala, Friday, 30 August 2024 at 13:19 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Yu David Liu (Department of Computer Science, State University of New York at Binghamton).

Abstract
Shimchenko, M. 2024. Optimizing Energy Efficiency of Concurrent Garbage Collection. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 2412. 76 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-2155-4.

The increasing energy consumption of the Information and Communication Technology sector amid climate change concerns underscores the urgency for energy efficiency improvements in computing. This thesis focuses on optimizing the energy efficiency of Java, a widely used programming language, and its implementation in OpenJDK. Specifically, our focus is on enhancing the energy efficiency of concurrent garbage collection.

As a starting point for our work, we assessed the energy consumption of various garbage collection algorithms within OpenJDK, establishing concurrent garbage collectors as the least energy-efficient. This prompted further investigation into methods to enhance their energy consumption. We investigated methods like dynamically adjusting the memory size required by an application based on how much of the computer's processors one wants to use for garbage collection. We also looked into scheduling garbage collection tasks to run on specific types of computer cores that use less energy and running these tasks when the computer is not being actively used.

We implemented all the abovementioned strategies in one of Java's concurrent garbage collectors, ZGC. Through our experiments, we showed that these techniques can significantly reduce the amount of energy used by garbage collection without slowing down the performance of the programs running on the computer. Overall, our research contributes to making computing more environmentally friendly by finding ways to use less energy while still getting the same results.

Keywords: Energy Efficiency, Garbage Collection, Java, Runtimes.

Marina Shimchenko, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-2155-4
URN urn:nbn:se:uu:diva-527713 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-527713)

To all Russian people who stay sane.

List of Papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I **Marina Shimchenko**, Mihail Popov, and Tobias Wrigstad. 2022. *Analysing and Predicting Energy Consumption of Garbage Collectors in OpenJDK*. In Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR 2022). Association for Computing Machinery, New York, NY, USA, 3–15.

II Sanaz Tavakolisomeh, **Marina Shimchenko**, Erik Österlund, Rodrigo Bruno, Paulo Ferreira, and Tobias Wrigstad. 2023. *Heap Size Adjustment with CPU Control*. In Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2023). Association for Computing Machinery, New York, NY, USA, 114–128.

III **Marina Shimchenko**, Erik Österlund, Tobias Wrigstad. 2024. *Scheduling Garbage Collection for Energy Efficiency on Asymmetric Multicore Processors*. Programming Journal: The Art, Science, and Engineering of Programming, 2024, Vol. 8, Issue 3, Article 10.

IV **Marina Shimchenko**, Erik Österlund, Tobias Wrigstad. (n.d.). *Monk: Opportunistic Scheduling to Delay Horizontal Scaling* (Under submission).

Reprints were made with permission from the publishers.

Other publications by the defendant that are not a part of this thesis:

> **Marina Shimchenko**, J. Rubén Titos Gil, Ricardo Fernández Pascual, Manuel E. Acacio, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean: *Analysing software prefetching opportunities in hardware transactional memory*. Journal of Supercomputing 78(1): 919-944 (2022).

Contents

1 Introduction ... 9

2 Background ... 11
2.1 Main Concepts ... 11
2.1.1 Memory Management ... 11
2.1.2 Garbage Collection ... 12
2.1.3 Z Garbage Collector (ZGC) ... 15
2.2 State of the Art of Energy Efficiency for Java Memory Management and ICT ... 17
2.2.1 Energy Efficiency for Java Memory Management ... 18
2.2.2 Energy Efficiency Broad Perspective ... 24
2.2.3 Concluding Remarks ... 33

3 Methods ... 35
3.1 Software and Runtime Environment ... 35
3.2 Java Benchmarks ... 35
3.2.1 Throughput BMs ... 36
3.2.2 Latency Benchmarks ... 36
3.3 Heap Size ... 37
3.4 Measuring Steady-State Performance ... 38
3.5 Measuring Energy Consumption ... 39
3.6 Statistical Analysis ... 40

4 Research summary ... 42
4.1 Energy Consumption of Fully Concurrent GCs ... 42
4.2 Improving Heap Sizing ... 44
4.3 GC Scheduling for Energy Efficiency ... 46
4.3.1 Energy Impact of Scheduling GC on Energy-Efficient Cores ... 46
4.3.2 Scheduling Concurrent GC on Idle Cores to Improve Hardware Utilization. ... 48

5 Future Work ... 50

6 Conclusions ... 52

7 Popular summary in English ... 53

8 Popular summary in Swedish ... 56

9 Acknowledgments 59
References 61

1. Introduction

In recent years, the Information and Communication Technology (ICT) sector has emerged as a significant consumer of electricity, constituting around 3% (805TWh) of global energy consumption in 2019, according to [51]. Projections suggest that this trend is poised to continue its upward trajectory, as highlighted by Belkhir and Elmeligi [11], Andrae and Edler [5]. Addressing the environmental impact of this sector, particularly in terms of carbon emissions, is paramount. If the rate of carbon emissions growth mirrors that of the period between 2007 and 2020, global greenhouse gas emissions associated with the ICT sector could surge from 3.0–3.6% in 2020 to 14% by 2040 [136]. This forecast underscores the urgent necessity for strategies aimed at reducing the sector's carbon footprint. The escalating costs of energy and the global imperative for low-carbon emission policies further heighten the urgency for companies and individuals to adopt and prioritize energy-efficient measures.

Java is one of the most popular programming languages and there is evidence of its energy inefficiency [56]. It runs on Virtual Machines (VMs/JVMs), for example, OpenJDK, estimated by Oracle at over 60 billion instances in 2024 [97]. Garbage collection, done by garbage collectors (GCs), is present in every running instance of JVM and manages the allocation and deallocation of memory. If the energy consumption of garbage collection is reduced even by 1%, it would be equivalent to reducing the environmental impact of 600 million JVM instances.

There are many GC algorithms in OpenJDK, from simple stop-the-world to complex fully-concurrent[1], meaning running at the same time with application threads. These collectors are critical for a large number of latency-sensitive workloads, such as e-commerce platforms, content management systems, social media platforms, and online banking portals. In this thesis, we first investigated the energy impact of different GC algorithms, concluding that fully concurrent collectors use more energy than other alternatives. So, in further research we focused on fully concurrent GCs, in particular, ZGC, leaving other GCs for future work.

Thus, the **aim** of this thesis is the following. In light of the current climate and energy crises, it is important to focus research efforts on enhancing the environmental sustainability of the ICT sector. While this problem certainly

[1] In this thesis, we refer to collectors as fully concurrent if they do not stop the entire program to do garbage collection work that is proportional to the size of the heap. The garbage collector we use as a basis for our proofs-of-concept, ZGC, has several short stop-the-world pauses in every GC cycle, but these have essentially a fixed, short duration.

cannot be addressed in full in a single PhD thesis, every contribution counts. The objective of this research is to enhance the energy efficiency of managed programming languages such as Java, primarily through optimizing garbage collection, with a specific emphasis on concurrent GCs. This approach enables us to reduce energy consumption without necessitating intervention from end-users. Given the extensive usage of managed programming languages like Java, our optimizations hold the potential to impact billions of instances globally. By focusing on optimizing GC, particularly with concurrent garbage collectors like ZGC, we ensure that energy-efficient practices seamlessly propagate to all users employing these technologies. This automatic dissemination of energy-efficient techniques is crucial, as it addresses barriers that often impede the widespread adoption of proposed solutions, thus representing a significant step towards enhancing sustainability within the ICT domain.

The goal of the following chapters is to introduce a reader to the main concepts used in the thesis (section 2.1), describe the state of the art in energy-efficiency for Java memory management (section 2.2.1), and then outline broader context (section 2.2.2). Chapter 4 summarises each paper and explains the main findings.

This research was supported by the Swedish Research Council through the projects Accelerating Managed Languages (2020-05346_VR), Optimizing for performance and energy efficiency with speculative compilers and co-designed hardware (2016-05086_VR), and by the Swedish Foundation for Strategic Research through the project Deploying Memory Management Research in the Mainstream (SM19-0059), and by donations from Oracle Corporation.

2. Background

This chapter aims to prepare readers to understand the practical aspects of this thesis. We start by explaining the main concepts and overviewing the state of the art, starting with the more specific topic of energy efficiency in memory management before going into a broader angle of energy efficiency in ICT. Understanding the current state of the art is crucial for contextualizing the thesis and its findings. We will finish with some conclusion remarks on expected outcomes and success metrics.

2.1 Main Concepts

To understand this thesis, it is useful to be familiar with several fundamental concepts used throughout the underlying work. Memory management and the idea of garbage collection are at the forefront, a critical aspect of memory management. In particular, we will go into additional details of ZGC (the Z Garbage Collector), which is used extensively in the experimental part of the work.

2.1.1 Memory Management

Memory management is a fundamental aspect of computer systems that involves organizing and controlling the allocation and deallocation of memory resources [101]. It is a critical function that ensures efficient memory utilization and is crucial in a computing system's overall performance and stability [16].

There are two main ways to manage memory: manual–explicit and automatic–dynamic. Manual memory management means that a programmer explicitly allocates and deallocates memory in a program. Manual memory management offers several benefits. For example, programmers having direct control over memory allocation and deallocation allows for fine-tuning of memory usage, which can be critical in resource-constrained environments or performance-sensitive applications. On the other hand, manual memory management adds complexity to programming tasks. This complexity increases the likelihood of errors [39] such as memory leaks (failure to deallocate memory), dangling pointers (accessing memory after deallocation), and buffer overflows (writing

beyond the allocated memory boundaries), which can lead to program crashes and bugs [52, 87, 53] as well as security vulnerabilities [58, 129].

Automatic memory management (including garbage collection) streamlines programming tasks by relieving programmers from the burden of explicit memory allocation and deallocation. This simplification reduces the occurrence of memory-related errors and enhances code readability, writability, and maintainability [76].

The choice between the two approaches is unclear as both have advantages and disadvantages [149, 68, 65, 17]. Recently, Sareen and Blackburn [115] looked into the costs and benefits of automatic memory management compared to a manual approach. Specifically, they highlight the memory overhead of automatic memory management and its potential negative effects on performance.

An advantage of encapsulating memory management within a dedicated automatic module is the potential for global reduction in energy consumption without requiring individual developers to invest additional effort. For instance, even a marginal enhancement, such as a 1% increase in the energy efficiency of a garbage collector, can yield substantial benefits when scaled across all applications utilizing automatic memory management (as discussed in chapter 1).

Thus, automatic memory management stands out as a compelling target for optimization for its potential to deliver salable energy efficiency.

2.1.2 Garbage Collection

Garbage collection performed by GC is an automatic memory management technique used in programming languages and runtime environments to reclaim memory that is no longer used by the program. GC is commonly associated with imperative programming languages like Java, C#, and JavaScript and functional languages like LISP, ML, Haskell, and Miranda. These languages have built-in GCs that automatically manage memory, eliminating developers' need to allocate and deallocate memory for objects manually. To automate deallocation, GC implements algorithms for identifying memory that is no longer reachable by the program and for releasing it.

Approaches to Track Liveness of Objects

A common way to identify unreachable objects is the reachability analysis used in tracing GCs. The basic idea is to walk the object graph reachable from a root set (*e.g.,* variables on the stack and global variables). If an object cannot be reached from any root, it is considered garbage, and the memory it occupies will be available after a memory reclamation process, which will be discussed in the next section.

Another way to track the liveness of objects is through reference counting. When an object's reference count drops to zero, it means the object is no longer accessible and can be safely reclaimed. Reference counting is straightforward to implement but has limitations, particularly in handling cyclic object structures (reference cycles), where objects reference each other in a loop, and in incurring overhead when reference counts change frequently. An advantage of reference counting over tracing GC is the promptness of reclamation: memory can be reclaimed as soon as it becomes unreachable, without waiting for a garbage collection cycle (which *e.g.,* involves walking the reachable object graph). Another benefit of reference counting is that its performance impact is deterministic as it does not depend on external factors such as heap size, allocation patterns, or execution time [76]. This can make it easier to reason about a program's performance and memory requirements and debug memory issues.

Limited-field reference counting, introduced by Deutsch and Bobrow [41], is a memory management technique that extends the traditional reference counting approach with certain optimizations. This technique was specifically developed to address some of the limitations and performance issues associated with reference counting. In traditional reference counting systems, every time a reference to an object is created or destroyed, the reference count for that object needs to be updated. This operation can become a performance bottleneck, especially in multi-threaded environments where multiple threads may simultaneously create or destroy references to objects. Each object has a reference count limit associated with it in limited-field reference counting. Instead of maintaining an exact reference count, the system only tracks the reference count until it reaches the limit. When the reference count exceeds the limit, it is treated as an overflow event. Overflow events can be detected and handled by the GC. Limiting the number of bits used to represent reference counts reduces memory overhead compared to traditional reference counting, where each reference count typically requires an entire machine word. Another benefit is reduced performance overhead associated with frequent reference count updates, especially in multi-threaded environments. This is because limited-field reference counting addresses this issue by reducing the frequency of reference count updates. Once the reference count exceeds the limit, it is treated as an overflow event, which can be less frequent than updating the count for every individual reference.

Common GC Phases

There are many GC algorithms, and delving into each of them in detail would require considerable space. However, it is crucial to understand three primary phases: *marking*, *sweeping*, and *compacting*, as they play fundamental roles and are frequently utilized [76].

Marking is the process of walking the object graph to identify which objects in memory are still reachable. Collectors that use this technique are called tracing GCs.

After marking, **sweeping** involves identifying unreachable objects and reclaiming the memory they occupy. Not all objects marked as live during the marking phase are unreachable.

In addition to marking and sweeping, some GCs include a **compacting phase**. Compacting is an optional step that aims to reduce memory fragmentation. During this phase, live objects are rearranged in memory to be densely packed together, leaving fewer fragmented gaps between objects. This process helps optimize the system's memory usage. Compaction is often implemented to avoid the need for sweeping by copying all live objects off of a region in memory that can subsequently be used for new allocations. This can be very efficient as the number of live objects is typically smaller than the number of unreachable ones.

Generational vs. Non-Generational GCs

To make marking more efficient, many GCs partition the heap into multiple regions based on the age of objects and perform GC in one region at a time [113]. This strategy aligns with the "most objects die young" principle, known as the weak generational hypothesis [85, 128]. Concentrating GC efforts on the region containing the youngest objects typically yields the greatest return on investment regarding memory reclamation, making GC more efficient.

Collections that target the younger generations of objects are commonly referred to as minor collections. In contrast, collections focusing on older generations are known as major collections. When the memory management system lacks this division of objects into age-based regions, it is referred to as a non-generational collector [119].

Heap Size

Regardless of the algorithm in terms of concurrency or the approach to identifying live (reachable) objects, all GCs have a notion of a heap size–the size of memory that an application has at its disposal. OpenJDK is unique in its requirement to set a maximum heap size, which can be set manually or set by the GC to a default on start-up that depends on a collector. The connection between heap size and performance is crucial [140]. If the memory is insufficient, an application might crash or perform terribly as GC runs frequently.

On the other hand, if too much heap memory is available, an application might experience more cache misses as object locality deteriorates [143, 1] or pause times increase significantly as there are more objects in memory to trace.

Determining the heap size is crucial but not intuitive. In the upcoming sections, we delve into the challenges of choosing a suitable heap size (section 3.3)

and explore potential solutions (section 4.2). We discuss various methods, including the common practice of selecting default heap sizes around three times larger than the minimum required for application execution without out-of-memory errors. Additionally, we highlight the dynamic nature of heap size selection, influenced by factors like allocation rate and live object sizes during runtime. Ultimately, our discussion will lead to introducing a dynamic approach in Paper II.

Stop-the-World, Mostly Concurrent, and Fully Concurrent GCs

GC algorithms can be categorized into three primary groups based on the level of concurrency they exhibit – stop-the-world, mostly concurrent, and fully concurrent. In stop-the-world GC, application threads are paused to perform GC. Although all these algorithms can be parallel in the sense of using multiple threads to perform GC work, their advantage lies in their implementation simplicity, which introduces latency due to these pause times.

Mostly concurrent GCs combine stop-the-world pauses with concurrent execution of most GC phases alongside application threads (mutators). On the other hand, fully concurrent collectors aim to operate concurrently with mutators without any stop-the-world pauses. In practice, it is challenging to eliminate short synchronization pauses since GCs manipulate the same objects that the mutators operate on. For scalability, synchronization pauses mustn't increase as the heap size grows. Long synchronization pauses can lead to undesirable consequences, such as degraded user experience, increased latency, and potential disruptions in real-time or interactive applications. For example, in a server application handling user requests, long pauses during garbage collection can result in delayed responses, leading to poor overall service quality and user dissatisfaction.

The key distinction among the three categories is their performance, either in terms of execution time (throughput) or latency. Stop-the-world pauses can significantly impact latency since they disrupt mutators, making fully concurrent collectors more suitable for achieving low latency [21, 98]. In contrast, stop-the-world collectors typically achieve high throughput, although they may not offer the best latency performance [57, 133].

2.1.3 Z Garbage Collector (ZGC)

Our work is specifically centered around fully concurrent GCs. To this end, this section overviews one such collector, ZGC. Other fully concurrent collectors exist. In OpenJDK, there is Shenandoah [49]. Outside of OpenJDK there is *e.g.,* C4 [125] and Pauseless [31], which inspired ZGC.

ZGC is a concurrent collector in OpenJDK. Initially designed as a non-generational collector, it has since evolved into a generational collector. ZGC is a low-latency, parallel, concurrent, compacting, generational GC. Yang and

Wrigstad [142] describe the non-generational ZGC algorithm in more detail. With the introduction of the generations, ZGC distinguishes between minor cycles, which collect garbage in the young generations, and major collections, which collect garbage both in the young and old generations. While a major collection always starts with a minor collection, minor and major collections can operate in parallel.

To facilitate concurrent compaction, ZGC employs load barriers and write barriers (the latter only in the generational version) to intercept accesses to relocated objects and remap dangling pointers to their updated locations before any further accesses can proceed. Barriers are an additional piece of code executed when references are loaded from or written to the heap. When an application thread encounters such a barrier, it checks whether the reference it is about to access has been validated in the current GC cycle. If not, the load triggers the necessary actions to update the reference to its new location, if necessary, and marks the reference as validated in this cycle. For additional actions in a barrier, threads take a so-called slow path. In the context of barriers, we distinguish between fast paths and slow paths. The former is the case when a barrier check does not result in any additional work; otherwise, it is the slow path. In essence, barriers ensure that references to objects remain valid even in the presence of concurrent garbage collection activity, which may move objects around.

The ZGC algorithm mandates that all pointers within the heap are remapped once during each GC cycle, which validates the references. The remapping process involves using atomic operations like compare and set to synchronize the actions of GC workers and program threads that are concurrently operating on the same objects. Consequently, ZGC engages in a complex interaction with mutators.

As long as the reclamation rate of the concurrent worker threads can match the pace of the memory allocation in the application threads, ZGC's operations do not result in mutator threads blocking, except for brief stop-the-world pauses where no substantial GC work is conducted.

After each phase transition, such as moving from the marking phase to the relocation phase, mutator threads are compelled to verify the validity of all pointers on their first subsequent access. This verification process triggers a wave of mutator threads to encounter a slow path, potentially leading to a temporary reduction in application throughput [67].

One pivotal outcome of executing all GC activities concurrently with the application, in contrast to stop-the-world or mostly concurrent GC approaches, is that the speed of GC worker threads should not affect application latency,

provided that the system has enough headroom. This is because mutator threads are never subjected to blocking due to GC operations[1].

The primary objective of a concurrent GC is to keep up the rate of memory reclamation with the application's allocation rate while minimizing the impact on overall performance. In pursuit of this goal, ZGC employs sophisticated heuristics to make critical decisions, including when to initiate a GC cycle to prevent Out-Of-Memory (OOM) errors and how many threads should be used during marking and compaction.

For instance, ZGC dynamically determines the optimal number of GC worker threads needed to avert OOM errors by analyzing the durations of past GC cycles and adapting the worker count to the available hardware resources. To control the reclamation speed, ZGC predicts the anticipated duration of the upcoming GC cycle based on the configured number of GC workers and computes the appropriate start time for that cycle.

Various triggers prompt GC cycles in ZGC, such as a high rate of memory allocation, elevated heap usage, or a lack of collection activity for a specified period, typically around 5 minutes. Additionally, ZGC may periodically initiate the collection of objects in the old generation, even without specific triggers. These heuristics carefully consider factors like available free memory and estimated time until a potential OOM error based on historical allocation rates and adapt to unforeseen changes in the application's behavior, ensuring proactive memory management.

2.2 State of the Art of Energy Efficiency for Java Memory Management and ICT

This section is split into two parts. In the first part, we examine the current research on energy efficiency in Java memory management, a key focus of this thesis. We start by closely examining existing literature to spot gaps or areas needing further exploration. By doing so, we hope to show how our thesis work can fill these gaps and contribute to advancing knowledge in this field.

In the second part, we broaden our perspective to examine energy efficiency research from a wider angle. We explore various methods, frameworks, and approaches used across different domains. Through this exploration, we aim to highlight the many energy efficiency challenges and stress the importance of taking an interdisciplinary approach to tackling them effectively.

[1]Sufficient headroom implies the presence of ample CPU and RAM resources. A concurrent collector risks frequent stalls without these resources and cannot adapt to the application's allocation patterns.

2.2.1 Energy Efficiency for Java Memory Management

This section aims to overview what has been done in the field of saving energy for Java memory management. This overview will help to position the thesis in the general context.

The primary objective of this section is to address two research questions:

1. What are the existing techniques described in the literature for reducing the energy consumption of Java memory management?
2. What are the gaps or unexplored areas in this domain?

These research questions aim to identify and analyze various techniques that could be further explored and investigated, filling existing gaps in knowledge.

The search for candidate papers was conducted in the IEEE Digital Library, ScienceDirect, the digital library of the ACM, and Scopus using the search string — *energy AND java AND ("memory manag*" OR "garbage collect*")*. The search string includes relevant terms and was formulated to identify a wide selection of candidate publications.

After the primary selection of candidate papers by applying the search strings to the digital libraries, duplicates were removed and the candidate publications were first analyzed by inspecting the titles then filtered based on a set of eligibility criteria to construct a final selection of relevant studies. Papers were selected based on the following inclusion criteria: studies that (1) suggest a technique for reducing energy consumption and (2) focus on the memory management aspect of Java. Exclusion criteria were any of the following: (1) publications that were not original research papers (*e.g.*, index, abstract, or poster); (2) studies that did not focus on memory management optimizations for energy efficiency; (3) studies that did not use Java for the empirical evaluation.

A total of 130 candidate publications were identified; however, only 32 studies met the eligibility criteria and were included in the final selection in this section; 15 were published from 2011 to 2021, and 17 were published before that. Many of the excluded papers are discussed in the next chapter, in which we discuss energy efficiency in a broader context. The chosen studies focused on diverse application domains, including embedded systems, Java for Android, and general-purpose systems. Furthermore, these studies can be categorized based on the stack level where the solutions were implemented, distinguishing between hardware-level and software-level solutions table 2.1.

Embedded Systems

In early 2000, energy efficiency research focused on embedded systems and mobile devices. The most prominent GC type was stop-the-world. This era laid the foundation for exploring energy consumption in Java programs, with particular attention to GC, as evidenced by pioneering studies such as Vijaykrishnan et al. [134]. Their findings shed light on the inherent challenges of GC, notably its tendency to exhibit poor data locality due to the traversal of

Table 2.1. *Classification of published research on energy efficiency.*

	Embedded/Android	General Purpose
Hardware	[23, 24, 26, 80, 66, 30, 93, 120, 131, 132, 122]	[116, 88, 109, 73]
Software	[22, 59, 130, 60, 144, 70, 100]	[20, 2, 96, 3, 137, 27]
Energy analysis	[134, 33, 122, 71]	[33]

non-contiguous data structures within the heap. Consequently, its energy consumption attributed to data accesses surpasses that attributed to instruction accesses. A subsequent study by Velasco et al. [130] extended this analysis by comparing various GC algorithms commonly used at the time: the classic mark-and-sweep algorithm [69]; copying compacting semi-space collection, *i.e.,* using half of the memory which is copied and compacted into the other half when full [114, 112]; one generational hybrid and one non-generational hybrid of copying compacting semi-space collection in the young generation and mark-and-sweep in the old generation [84, 29, 47]; generational collectors using copying compacting semi-space collection in both young and old generations [148].

A key observation from work by Velasco et al. is that in scenarios where benchmarks allocate primarily large objects, the percentage of energy used during the collection phase ranges from 25% to 50% of the total JVM energy consumption. Comparing hybrid GCs (use both copying and mark-and-sweep) to pure copying GCs (use only copying), they found that hybrid GCs consume up to 40% less energy during garbage collections, despite applications consuming less energy when using only copying. As a result, hybrid GCs achieve an energy reduction of 5% compared to pure copying GCs. The study also identified the most energy-efficient collector as a hybrid of copying and mark-and-sweep [29, 47]. Furthermore, variations in energy consumption between GCs can be influenced significantly by the memory hierarchy, *e.g.,* L1 associativity, with differences of up to 50% observed in the collection phase and an additional 20% variation in energy consumption during the mutator phase. Consequently, the overall variation in energy consumption can reach up to 40%. All of this indicates that GC can consume a significant portion of total energy, which undoubtedly inspired further research on GC energy efficiency.

A prevalent technique in embedded systems for minimizing energy consumption through memory management involves leveraging energy-efficient memory types, such as scratchpad memory (SPM), specifically for garbage collection instructions or metadata storage [132, 131], for frequently accessed long-lived objects [30] or for unmodified Java bytecode [93]. SPM is a type of on-chip memory manually managed by software rather than automatically

managed by hardware like caches. It is known for offering lower access times and energy consumption compared to traditional cached memory, making it particularly beneficial for real-time and energy-efficient computing applications. Developers or compilers specify which data or code segments to store in SPM, enabling more predictable performance and efficient use of memory resources, especially in systems with limited power or processing capabilities. For example, Nguyen et al. [93] demonstrates a 21.8% reduction in energy consumption by placing unmodified Java bytecode to SPM, implemented entirely within a JVM instead of relying on compiler support.

Non-volatile memory (NVM) is another special memory type widely used in embedded and general-purpose computers. NVM retains data without power, contrasting with volatile memory like RAM, which loses data when powered off. While using NVM in Big Data systems or critical embedded systems makes sense for data preservation, it is also challenging due to its higher access latency and lower bandwidth than DRAM. For instance, NVM read latency is about 2–4 times longer than DRAM, and its bandwidth is roughly 1/8 to 1/3 of DRAM's, posing significant performance implications. Thus, making GC aware of NVM in addition to DRAM is crucial [137, 27]. In addition, GC generates many extra writes to NVM when NVM has limited write endurance, and writes are expensive operations in terms of both time and energy. Thus, Pan et al. [100] proposed various techniques to reduce the number of writes. For example, instead of physically moving objects during GC, live objects can be remapped, meaning dynamically reassigning addresses from one location to another without needing to rewrite the data. Akram et al. [3] also tried to tackle the issue of extensive writing to NVM. By modifying a GC, they improved the algorithm for placing highly mutated objects in DRAM and read-mostly objects in NVM.

Alternatively, instead of using special types of memory, developers may opt to utilize a portion of on-chip memory as local memory for similar efficiency-enhancing purposes [80]. This approach involves implementing the object allocation strategy using an annotation-based method, effectively reducing memory system energy consumption by up to 39%. Annotations are inserted into the bytecodes, allowing exploitation by the JVM implementation to facilitate allocations in the local memory.

Another approach involves leveraging lower power modes of memory, such as shutting off SDRAM banks [24, 23, 131] or cache lines, particularly when they are not actively storing live objects [26, 144]. For instance, a study by Chen et al. [26] highlighted that a significant portion of leakage energy is squandered on retaining objects beyond their last usage. Unlike dynamic energy consumption, which depends on component access, static power, including leakage energy, is consumed regardless of access frequency. To mitigate leakage, one approach involves completely gating the supply voltage of cache lines. However, this action results in the loss of data stored in the affected cache lines. The researchers implemented three strategies to address

this: first, they switched off cache lines with collected garbage. Second, they utilized escape analysis to identify method-local objects and switched off corresponding cache lines upon method completion. Third, they pinpointed the last use of an object by identifying the instruction preceding its disposal, allowing for immediate cache line deactivation. Lastly, they focused on objects with extended intervals between successive accesses, turning off cache lines containing such objects. By combining these methods, they achieved a notable 21% reduction in data cache leakage energy. Chen et al. [23] advocated for a more frequent invocation of GC based on the dynamic runtime behavior of applications. This approach aimed to promptly identify and dispose of garbage, allowing for a more aggressive power supply shutdown of memory banks. Their research demonstrated a noteworthy reduction of 28.4% in average heap energy consumption due to this technique.

Another trend of the time for energy reduction was adaptive micro architectures where resources can be dynamically tuned to match program runtime requirements [4, 50, 107]. One feature of adaptive microarchitecture can be selective cache ways. It allows disabling a subset of the ways in a set associative cache during periods of modest cache activity, while the full cache may remain operational for more cache-intensive periods [4]. Hu and John [66] explored the impact of GC on such a system. They concluded that GC benefits from simple cores with small caches. The results were later supported and utilized by Cao et al. [20], who used heterogeneous general-purpose architectures composed of simple in-order cores and complex out-of-order cores (see section 2.2.1 for details.)

Software

So far, we have explored strategies to use less energy by adjusting an environment (using special memory or architecture) rather than changing GC algorithms. Efforts to discover more energy-efficient memory management algorithms are also prevalent.

For instance, Chen et al. [22] explored the energy impact of compressing memory in embedded devices and concluded that it is an efficient technique even considering runtime overheads, leading to approx. 20.9% energy reduction on average. Griffin et al. [59, 60] updated a classical Mark-Sweep-Compact algorithm with deferred limited-field reference counting introduced by Deutsch and Bobrow [41] (see section 2.1). The proposed scheme was implemented into Sun KVM[121] and showed to reduce power consumption by as much as 27% compared to the default Sun KVM. The K Virtual Machine (KVM) originates from the JVM specification and is tailored for compact devices possessing 128K to 256K of memory. Its primary focus is on minimizing memory usage, hence the 'K' in KVM symbolizing kilobytes, highlighting its operation within kilobytes of memory rather than megabytes.

Olson et al. [96] extended the HotSpot VM to divide the application's heap into separate regions for objects with different (expected) usage patterns, such as hot and cold. Then, this information is communicated to the operating system, which uses it to map the logical application pages to the appropriate DRAM modules according to user-defined provisioning goals. The idea is that hot and cold objects if co-located, can use similar power states. This technique achieves a best-case total energy reduction of 8.8%. Hussein et al. [71] explored the effects of generational garbage collection and concurrency policies on Android devices' energy consumption and application performance. They drew multiple conclusions, *e.g.,* that increased throughput (larger heap) does not always correspond to better energy consumption. This is because GC work is memory-bound and, therefore, limited by the memory access speed. Thus, choosing a higher CPU frequency to perform the work does not necessarily improve throughput; a lower frequency can get the same work done in the same amount of time at lower energy.

Hardware

In this section, we delve into a spectrum of hardware-based strategies tailored to enhance the energy efficiency of Java memory management. Ranging from hardware-level optimizations within existing architectures to dynamic voltage and frequency scaling and leveraging heterogeneous computing paradigms, these approaches offer diverse avenues for improving the efficiency of Java runtime environments.

As the exploration of Java's energy efficiency expanded to encompass high-performance systems, researchers, such as Contreras and Martonosi [33], devised methodologies tailored to real-world hardware environments. Their work established a methodology for rigorous energy measurements using specialized equipment like the DBPXA255 development board [35]. However, with the evolution of hardware-integrated solutions, our approach diverges from traditional methodologies. The emergence of novel techniques directly integrated into existing hardware simplifies energy analysis and enhances accessibility, rendering previous methodologies obsolete.

Sun and Zhang [120] delve into three hardware-based code caching strategies to accelerate the writing and reading speed of dynamically generated code while consuming less energy than conventional JVMs. Unlike traditional JVMs, which typically write binary code into the data cache and then load it into the instruction cache through the shared L2 cache or memory–inefficient in both time and energy consumption–directly writing code into the instruction cache can significantly boost the performance of various Java applications. On average, these enhancements amount to a 9.6% improvement, with potential gains peaking at 42.9%. Moreover, this approach can achieve an average reduction of 6% in the overall energy dissipation of these Java programs. Reducing memory traffic was also explored by Sartor et al. [116]. In

their work, they reduced DRAM traffic by 59% and thus total DRAM energy by 14% by creating a hardware-software solution to communicate to hardware the information about dirty cache lines with dead objects after garbage collection to reduce the number of write-backs. Rodchenko et al. [109] reduced energy in memory hierarchy by proposing a software-hardware co-design to efficiently handle type information in objects. The focus is on using tagged pointers in 64-bit architectures to eliminate the need for storing a pointer to type information within each object. Instead, it proposes hardware extensions for efficient type information retrieval and compression-decompression of tagged object pointers. This technique was shown to reduce dynamic DRAM energy up to 50%.

Tang and Liu [122] studied the correlation between energy and heap size. The results show that with a very small heap/memory, GC would occur often, leading to massive CPU activities and energy inefficiency. On the other hand, with a large heap/memory, although GC frequency and, consequently, CPU cost are reduced, memory cost increases. For example, they compared 3 different heap sizes: 30Kb, 50Kb and 70Kb. While the CPU energy reduces from 38% to 30% with increased memory, in 70kB heap size configuration, memory costs as much as 38% total energy consumption. To mitigate this, the authors recommend using a hardware accelerator [124, 123] to decrease GC overhead, providing at least 20% total energy reduction.

In the context of accelerators, Maas et al. [88] proposed a versatile hardware GC accelerator compatible with both stop-the-world and pause-free collectors. Their prototype performs the mark phase of a tracing GC at 4.2× the performance of an in-order CPU, and the estimated energy improvement is 14.5%. Jang et al. [73] create the first 3D stacked memory-based hardware accelerator. 3D stacked memory refers to a type of memory architecture where multiple layers of DRAM chips are vertically stacked on top of each other and interconnected using through-silicon vias [77]. This type of memory enables efficient processing on near-memory logic and provides ample memory bandwidth, which is so crucial for GC work and is often limited by memory bandwidth as claimed by Jang et al. [73].

Dynamic Voltage/Frequency Scaling (DVFS) could be done in addition to other methods described above. DFVS creates energy-efficient modes by lowering the voltage and frequency. Since GC is a canonical example of the memory-bound workload that best responds to such scaling, Hussein et al. [70] explores the impact of frequency scaling for GC in a real mobile device running Android's Dalvik VM, which uses a concurrent collector. They can reduce power significantly by controlling the frequency of the core on which the concurrent collector thread runs. Running established multi-threaded benchmarks shows that total processor energy can be reduced up to 30%, with end-to-end performance loss of at most 10%.

Like DVFS, heterogeneous architectures with energy-efficient and performance cores, aka AMPs, can take advantage of running GC at a lower fre-

quency on more energy-efficient cores. Cao et al. [20] explored the energy impacts of running mostly concurrent GC on small energy-efficient cores. This study concluded that in terms of energy efficiency, GC threads are best suited for small clocked-down in-order cores as they are bound to wait for memory most of the time (see also work by Maas et al. [88]). On the other hand, when GC is on a critical path, which happens during stop-the-world pauses or critical sections, GC would benefit from boosting its priority as showcased by Akram et al. [2].

In this review (table 2.1), we classified the papers into distinct categories based on their focus areas: Embedded Systems and Android vs. General Purpose, and their approaches: Hardware Support vs. Software Solutions. This classification provides an overview of the current research landscape in energy-efficient memory management in Java. Further analysis, especially within the Software Solutions for general-purpose applications, reveals a gap:

Knowledge Gap. *Current research lacks systematic studies on energy for fully concurrent GCs (see section 2.1).*

The gap sets a basis for this thesis.

2.2.2 Energy Efficiency Broad Perspective

Energy optimization can and should happen at the entire system stack, from application software and algorithms to programming languages, compilers, instruction sets, and microarchitectures to hardware design. This is because energy is consumed by the hardware performing computations, but the control over the computation ultimately lies within the applications running on the hardware [43].

This section provides a non-exhaustive list of techniques for energy efficiency at different levels of the system stack to indicate possible angles for examining the energy efficiency problem. The papers are divided into the following categories: hardware, compilers, OS, runtimes, programming languages, and development frameworks. Most of the papers included in this section are byproducts of the literature review described in the previous section.

Hardware

Hardware optimizations for energy have a rich spectrum and grand possibilities. This section gives examples of the following optimization techniques:

- Specialized hardware and accelerators [10, 74, 102, 28, 79].
- Optimizing memory transfers [118, 110, 62].

Specialized Hardware and Accelerators Energy efficiency can be increased using an architecture that is well-suited for an application.

Beck et al. [10] examine Java processors' performance and energy efficiency compared to RISC ones, particularly in embedded systems. A Java processor is a type of microprocessor designed to execute Java bytecode directly from the hardware without requiring a JVM. This approach leads to fewer memory accesses and cache misses, especially in instruction memory, thereby reducing energy consumption by up to around 80% for instruction memory.

Similarly, the DianNao neural network accelerator demonstrates considerable advancements in energy efficiency for on-device deep neural network processing. Chen et al. [28] detail an experimental setup where the DianNao architecture achieves a remarkable reduction in energy consumption compared to conventional SIMD and GPU configurations. Specifically, DianNao utilizes a specialized architecture with 256 16-bit truncated multipliers and adder trees that operate every cycle, facilitating a throughput of 496 fixed-point operations per cycle at a frequency of 0.98GHz. This design allows DianNao to consume 21.08 times less energy than SIMD, largely due to its efficient handling of computations and memory accesses. The energy savings are attributed to the use of compact, custom storage closely integrated with the computational units, significantly minimizing the energy costs associated with memory access.

Building upon the concept of specialized architectures for enhanced energy efficiency, the MAHA (Memory Architecture for Hierarchical Access) framework introduces a novel approach by integrating in-memory computing within NAND Flash storage. NAND memory storage is a type of non-volatile flash memory, meaning it can retain data without needing a continuous power supply. It is a popular form of storage for a wide range of electronic devices, including smartphones, SSDs (solid-state drives), USB flash drives, and memory cards. Paul et al. [102] describe an architecture that reconfigures the conventional NAND Flash memory into a computational powerhouse by organizing it into Memory Logic Blocks, significantly reducing off-chip data transfer. The simulation results demonstrate that MAHA leads to substantial improvements in both energy efficiency and performance. Specifically, MAHA achieves up to 91.2 times higher energy efficiency and a 75% reduction in execution time for data-intensive applications compared to a traditional CPU setup with separate memory and compute units.

Architectural support for inefficient but common operations can significantly improve energy efficiency. For example, Kim et al. [79] propose architectural enhancements for efficient execution of dynamic scripting languages like JavaScript and Lua on low-power, resource-constrained devices. The focus is on extending the ISA to include typed instructions that reduce the overhead of dynamic type checks, a common source of inefficiency in scripting languages. By embedding high-level type information within the ISA and performing type checks in hardware, this approach significantly reduces instruction count and memory footprint, leading to energy savings. The eval-

uation shows Typed Architectures improve the energy-delay product (EDP) by 19.3% for JavaScript and 16.5% for Lua, with a minimal area overhead of 1.6%, which is important in IoT devices.

Other common operations are object serialization and deserialization (S/D). Object serialization is the process of converting an object's state into a format that can be stored or transmitted (*e.g.,* into a byte stream or string format) and later reconstructed. Deserialization is the reverse process, where the byte stream or string is used to recreate the original object in memory. This is crucial for saving object states to a file, sending objects over a network between different components of a distributed system, or for other forms of object persistence and communication. Optimizing S/D operations is crucial for big data analytics frameworks where S/D operations can be a significant overhead. Jang et al. [74] introduce "Cereal," a specialized hardware accelerator designed to improve the efficiency of object S/D operations. Cereal co-designs the serialization format with hardware architecture to exploit parallelism in the S/D process, delivering high throughput and efficient object packing to compress metadata. The evaluation demonstrates that Cereal significantly outperforms existing Java S/D libraries, offering up to 43.4× higher average S/D throughput and achieving considerable speedups and energy savings by up to 227.75× and 136.28× respectively in real-world Spark applications.

Memory One way to optimize energy at the hardware level is to reduce data transfers. For example, Singh et al. [118] introduce a time-based coherence framework for GPUs named Temporal Coherence (TC), which leverages synchronized counters in single-chip systems to develop a streamlined GPU coherence protocol. This approach eliminates all coherence traffic and protocol races by enabling coherence transitions, like invalidation of cache blocks, to happen synchronously. The implementation of TC called TC-Weak significantly improves the performance of GPU applications with inter-workgroup communication by 85% over disabling the non-coherent L1 caches in the baseline GPU and reduces interconnect energy usage by up to 40% across different communication patterns.

Ros and Kaxiras [110] introduces Racer, a protocol designed to simplify coherence in multicore systems. Racer is particularly effective for the Total Store Order (TSO) memory model, common in x86 processors, and it operates without needing explicit software annotations for synchronization. This approach offers a 26.4% improvement in energy consumption over traditional directory-based protocols by dynamically detecting read-after-write (RAW) races. When such a race is detected, Racer triggers self-invalidation of the affected cache lines, ensuring that memory reads always fetch the most up-to-date data. When a RAW situation is detected in a directory-based protocol, the directory maintains a record of which cores have a copy of the data. If a core tries to read data that another core has recently written, the directory ensures the reading core gets the updated value. This might involve invalidating the

old copy in the reading core's cache and providing it with the new data from the writing core or the main memory, ensuring coherence and consistency across the system. Races's approach eliminates the need for directory-based coherence, reducing the complexity and thus energy.

Another way to reduce data transfers is to optimize prefetching. Traditional aggressive prefetching methods often increase energy usage. Guo et al. [62] examines the impact of various hardware prefetching techniques on energy consumption and performance and introduces novel compiler and hardware strategies that significantly mitigate this overhead. Through methods like hardware-based filtering and compiler-assisted analysis, the study reduces energy in the data memory system by up to 40% with only a minimal impact on speed. Hardware-based filtering is designed to reduce unnecessary memory accesses. It uses Prefetch Filter Buffers (PFB), which are small hardware units that track recently prefetched addresses. Before issuing a new prefetch request, the system checks if the address is already in the PFB. If it is, this indicates that the data is likely still in the cache or has been recently requested, and the new prefetch request can be skipped, reducing redundant operations. Hardware-based filtering also uses tag arrays to quickly determine if prefetched data already resides in the cache. If a prefetch request hits this tag array, the prefetch can be canceled to avoid redundant data loading.

Compilers

Compilers can help to unlock energy-efficiency optimizations. For example, Chen et al. [25] present a strategy for managing the code cache in memory-constrained Java environments. The technique involves selective compilation and intelligent replacement policies to manage the code cache effectively, prioritizing methods based on their frequency and recency of use. This approach minimizes the need for re-compilation and reduces energy consumption by making better use of limited memory resources. The paper demonstrates that their strategies can significantly reduce energy consumption compared to both pure interpretation and traditional Least Recently Used (LRU) compilation strategies without specifying exact percentages of energy reduction.

Jimborean et al. [75] introduce compiler techniques to automatically identify extended data-race-free (xDRF) regions in parallel programs. These regions stretch across synchronization points, such as function calls and loop back-edges, while preserving data-race-free semantics. xDRF regions can enhance parallel applications' performance and energy efficiency by reducing unnecessary synchronization and cache coherence traffic. By evaluating these techniques within a dual-mode cache coherence protocol, they demonstrate performance improvements of 6.8% and energy efficiency gains of 11.7% over standard directory-based coherence protocols.

Operating Systems

Operating systems (OS) not only occupy a significant portion of all machine cycles but can also consume a dominant part of the total energy, especially in OS-intensive workloads. For instance, Li and John [83] reveal that high-performance and general-purpose microprocessors require sophisticated methodologies for real-time power estimation to offer dynamic and precise power management solutions. This is crucial as modern applications like database servers exercise the operating system significantly, leading to substantial power dissipation if not managed properly. The paper proposes enhanced power modeling techniques that integrate detailed instruction-level and routine-level power analysis, providing a more granular and accurate energy profile for operating systems.

Another way to achieve energy efficiency is to make the OS aware of special types of memory or hardware on which it is running. For example, Nakagawa and Oikawa [92] proposes an operating system architecture that leverages NVM(see section 2.2.1 "Embedded Systems for details) and AMPs (see section 2.2.1 "Hardware" for details) to reduce power consumption. The approach involves running the OS kernel in Java on a simpler core for efficiency while user programs run on more complex cores. This setup aims to utilize the low-power benefits of NVM for main memory and the flexible, energy-efficient processing capabilities of AMPs. The architecture anticipates energy savings by avoiding the constant refreshing required by DRAM and enabling more effective hibernation processes. The paper discusses a preliminary implementation and experimentation with this architecture but does not quantify the specific amount of energy reduction achieved.

Programming Languages

There are multiple ways to go about the energy consumption of programming languages:

- Choosing a more energy-efficient language [55, 105, 45, 18].
- Choosing more energy-efficient data structures [63, 103, 89, 94, 104, 106] or programming patterns [44].

The following sections go into depth for each of the mentioned methods.

More Energy Efficient Programming Languages Different languages have different energy profiles, as shown, for instance, by Georgiou et al. [55]. Using data from Rosetta Code and focusing on popular languages, the study finds that compiled languages like C, C++, and Go are more energy-efficient than interpreted languages like JavaScript, PHP, and Ruby for the tasks tested. This is not surprising, given the nature of these languages.

Peters et al. [105] assess the effects of migrating Android apps from Java to Kotlin, focusing on runtime efficiency metrics such as CPU and memory usage, garbage collection, and energy consumption. The study analyzed 10 Android apps that fully transitioned to Kotlin and compared their Java and

Kotlin versions across several performance metrics. The findings reveal that while migrating to Kotlin statistically affects CPU usage, memory usage, and frame render times, the effect size is negligible. Essentially, the migration does not significantly impact the number of garbage collection calls, delayed frames, app size, or energy consumption. This suggests that developers can expect comparable runtime efficiency when migrating their Android apps to Kotlin, with minor variations in performance metrics that do not significantly affect the overall app behavior or user experience.

Ertel et al. [45] present a new programming model and framework, Ohua, that leverages dataflow programming concepts to design scalable concurrent applications. Ohua combines object-oriented and functional programming using Java for stateful function implementation and Clojure for implicitly expressing algorithms and deriving dataflow graphs. The evaluation showcases Ohua's scalability by comparing a simple web server developed with Ohua to the highly optimized Jetty web server, demonstrating improved energy efficiency.

Canino and Liu [18] introduce ENT, a novel programming language designed to facilitate energy-efficient programming. ENT combines proactive and adaptive energy management at the application level by characterizing program fragments with modes that reflect their energy behavior. These modes can be statically assigned or determined dynamically at runtime, depending on the program's state, configuration settings, and external factors like battery levels or CPU temperatures. This approach unifies proactiveness and adaptiveness under a mixed type system, blending static and dynamic typing to regulate interactions between program components and potentially expose energy bugs through compile-time or run-time errors. ENT has been implemented as an extension to Java. The authors also present FJP (Featherweight Java-like Predication) [19], a minimal calculus incorporating energy prediction into programming language abstractions. The main technique explored is dynamic and partial energy consumption prediction, which shifts away from attempting to fully and statically predict a program's energy use. FJP allows programmers to annotate parts of the code to specify where energy consumption should be recorded or predicted. Then, the framework calculates the share of energy consumption by each thread based on its activity, measured in OS time units called "jiffies."

Energy Analysis of Data Structures Different collection classes, data structures, and constructs have different energy profiles within the same language. Hasan et al. [63] discuss the energy consumption associated with operations on Java List, Map, and Set abstractions. They reveal significant differences in energy usage among different implementations of these data structures, depending on the operations performed. For example, array lists are more energy-efficient for insertions at the middle or end, while linked lists consume less energy for insertions at the start. Based on these findings, modi-

fying the collection classes used in six applications and libraries demonstrated that inappropriate choices could lead to up to 3× higher energy consumption than the most efficient option. Similar research was done by Pereira et al. [103]. However, they found that by selecting the most energy-efficient implementations based on their profile, an average energy savings of 6.2% could be achieved.

To facilitate finding the most energy-efficient collection class, multiple tools emerged, like SEEDS, as introduced by Manotas et al. [89], Stanley, developed by Pereira et al. [104], along with the tool from Pereira et al. [103]. There is also jBrainy developed by Couderc et al. [38], although it focuses more on performance and less directly on energy. To put these studies in perspective, Pereira et al. [104], *e.g.,* showed energy savings between 2% and 17% and a reduction in execution time between 2% and 13%.

Pinto et al. [106] explore how different thread management constructs in Java affect energy consumption on multicore platforms. They focus on three main constructs: explicit threading, fixed-size thread pooling, and work stealing, and explore how variations in thread number, task division strategies, and data characteristics influence energy usage. The study found that for I/O-bound tasks, the explicit threading model tends to be more energy-efficient, while for highly parallel tasks, the work-stealing model often shows better energy efficiency.

Anti-patterns At a higher level of abstraction, El-Dahshan et al. [44] focus on software engineering. They look at identifying and correcting software design flaws, known as anti-patterns, in mobile applications using reverse engineering and UML modeling. The study compares nine UML tools to find the best for reverse and forward engineering capable of detecting anti-patterns. They propose and apply a method to 29 mobile apps, identifying and correcting ten specific anti-patterns 749 times across the apps. Even though the paper does not provide direct energy measurements, empirical studies have assessed that the anti-patterns harm energy efficiency [91].

Runtimes

Runtimes, or execution environments, play a pivotal role in shaping the energy efficiency of software systems. These dynamic platforms execute applications, manage system resources, and provide essential services to ensure smooth program execution. Understanding the energy dynamics within runtimes is crucial for optimizing overall system energy consumption.

Energy of Runtimes Ournani et al. [99] examine the energy efficiency of various JVM distributions and configurations. They found significant variations in energy consumption among different JVMs, noting that some configurations can lead to up to 100% more energy usage. The study introduces J-Referral, an open-source tool that recommends the most energy-efficient JVM distribution and configuration for Java applications.

Badea et al. [9] improved the efficiency of the JVM on embedded systems by using "superoperators." Superoperators bundle frequently executed bytecode sequences into optimized, single bytecode instructions. This approach minimizes the overhead of bytecode fetching, decoding, and executing individual instructions, significantly reducing memory accesses and computational steps. Consequently, it leads to considerable energy savings, especially beneficial for resource-constrained embedded systems. The study demonstrates energy savings between 40% to 60% for processor and DRAM combined.

Liu et al. [86] have another approach to improving the energy efficiency of JVM. They propose method-grained DVFS, where the JVM is augmented to profile the energy consumption of hot, aka frequently executed, methods at different CPU frequencies and then optimize their execution by selecting the most energy-efficient frequency for each method. This technique allows for a 14.9% reduction in energy usage compared to the built-in power management features of Linux.

Runtime Energy Support and Tools Multiple tools and extensions allow for optimizing the energy consumption of runtime environments like the JVM. For example, Babakol et al. [7] introduce Chappie, a runtime system for Java applications that produces an energy footprint, detailing the relative energy consumption of application logical units, such as methods, classes, and packages within an application. It operates by periodically sampling both the JVM's call stacks and the system's raw energy readings. These samples are then used to distribute energy consumption data among all active methods and provide an energy footprint of the application. This design allows Chappie to run concurrently with the application, requiring no modifications to the application code, JDK, JVM, OS, or hardware. Zhu et al. [147] introduce the Eco Java extension — a programming model designed for energy-aware and temperature-aware applications. It means that Eco allows programs to adjust their behavior based on energy or temperature budgets to avoid deficits (battery drain or CPU overheating) and surpluses (underutilized energy resources). If a program in Eco does not fit into the specified budget, it switches to a different implementation, which is defined by developers to provide alternative application behaviors that achieve the same goals but with different energy or thermal footprints. However, deploying energy-aware applications in shared environments, such as cloud computing, presents the challenge of accurately attributing energy consumption to individual applications at a high resolution. Babakol et al. [8] addresses this issue with Eflect, the first application software framework designed for energy reflection in shared settings. Eflect employs energy virtualization, utilizing OS time (measured in "jiffies") to gauge the activity level of each thread. The principle is straightforward: the longer a thread runs, the more energy it consumes. Within an "energy domain"—a group of CPU cores with identical power settings—Eflect assumes uniformity in voltage and frequency across all cores due to DVFS. This as-

sumption allows for the consistent application of power metrics across these cores. Eflect calculates the share of energy each thread uses in an energy domain based on its proportion of activity within that domain.

Rua et al. [111] introduce the GreenSource infrastructure and the AnaDroid tool. The work focuses on energy efficiency in Android app development by analyzing and executing open-source Android apps to gather energy consumption data. The goal is to provide developers and researchers with a dataset for studying energy-efficient software development. The infrastructure combines a collection of Android applications, a benchmarking framework, and a repository of metrics obtained from application execution, aiming to characterize energy consumption in the Android ecosystem.

Development Frameworks

Even a development framework makes a difference in energy through the abstraction layers it introduces. These frameworks might employ different programming languages, runtime environments, and compilation strategies that can impact how efficiently code is executed on hardware. The framework's choice influences how code is compiled into the binary, potentially introducing overhead regarding memory usage, CPU cycles, and energy consumption due to differences in how efficiently they manage resources, execute parallel tasks, and interact with the device's native APIs. Oliveira et al. [95] assess the energy, CPU, and memory usage of mobile apps developed using three popular frameworks: Flutter, React Native, and Ionic, compared to native Java implementations. They found that while cross-platform and hybrid frameworks can offer competitive performance for CPU-intensive tasks, with some reducing energy consumption by up to 81% and execution time by up to 83%, the overhead varies significantly across different types of applications. Flutter often showed the least overhead, while React Native generally had the highest, particularly in benchmarks. However, React Native was notably efficient in an app scenario involving continuous image animation, using the least CPU and energy by leveraging native libraries.

As we delve deeper into the realm of development tools and frameworks, it becomes increasingly evident that they play a pivotal role in shaping the energy footprint of software systems. Without these tools, achieving optimal energy efficiency would be a formidable challenge. For example, to improve embedded system development cycles, Silva et al. [117] outline the SIMPLE framework. It emphasizes the development and testing of applications and hardware components interconnected by a network-on-chip. SIMPLE supports multithreaded real-time Java applications adhering to the Real-Time Specification for Java and facilitates performance, power, and energy evaluation.

Another example of a helper tool for development is XTREM [34], a power-simulation tool tailored for the Intel XScale microarchitecture. XTREM shows

an average performance error of only 6.5% and a power error of 4%. The paper uses XTREM to analyze the energy consumption of Java CDC and CLDC applications, discovering that up to 35% of the total energy consumption is dedicated to VM support functions.

2.2.3 Concluding Remarks

In conclusion, optimizing for energy efficiency in computing necessitates a holistic and interdisciplinary approach. The diverse range of techniques explored in the previous section, spanning from hardware advancements to high-level programming languages and development frameworks, highlights the multifaceted nature of the energy efficiency challenge, which must be addressed across all layers of the system stack. Researchers and practitioners can uncover synergies and forge more impactful solutions by tackling energy consumption from various perspectives.

This thesis is an example of such an interdisciplinary approach, bridging the realms of programming languages with a focus on memory management and runtime environments, operating systems with their scheduling policies, and energy-efficient hardware. By integrating insights from these diverse disciplines, we have been able to deepen our understanding of what optimization may or may not work in the realm of high-level dynamic languages.

One important aspect to consider is the anticipated level of energy reduction achievable through our efforts. At a broad level, it's evident that hardware-based optimizations often yield substantial energy savings, frequently exceeding a 10x improvement. While these numbers are undoubtedly appealing, integrating such solutions into mainstream hardware poses significant challenges. One obstacle is that many hardware optimizations target specific applications, limiting their profitability on a broader scale. Consequently, these solutions often remain confined to the ad-hoc realm, where companies develop custom accelerators tailored to their needs.

On the other hand, software-based optimizations typically result in more modest energy reductions, typically around 10% better. However, they offer advantages in terms of implementation feasibility, as they don't necessitate the establishment of new hardware manufacturing processes. Furthermore, the benefits can propagate across all software instances utilizing the optimized environment depending on the approach, such as compiler or runtime optimizations. As highlighted in chapter 1, even a 1% reduction in energy consumption within the JVM runtime environment holds significant weight, given Java's widespread usage as one of the most popular programming languages globally, with billions of instances in operation. So basically, existing Java users can immediately benefit from the optimizations without additional effort or workflow changes.

In the subsequent section, we will present the methodologies employed in conducting our interdisciplinary study, describing approaches and techniques that have enabled us to study energy efficiency in runtime environments.

3. Methods

In this chapter, we lay the groundwork for our experiments by introducing the benchmarks we used, outlining our methodology for measuring energy consumption and detailing the techniques employed to modify GC configurations. This chapter will aid in reasoning about experiments conducted in the subsequent chapter, where we discuss our research findings and their implications.

3.1 Software and Runtime Environment

In this thesis, OpenJDK [36] serves as an experiment platform. OpenJDK is an implementation of the Java Platform, Standard Edition (Java SE). The use of OpenJDK is motivated by its status as a freely available open-source project and its extensive deployment in real-world applications. OpenJDK is distributed under the GNU General Public License [12] (GPL-2.0-only, GNU GPL, or simply GPL), a key license in the free software movement that grants users the freedom to run, study, share, and modify the software. This licensing model facilitates research opportunities, allowing one to communicate findings easily with the public.

According to the official Oracle website, Java SE is the most popular modern development platform globally, with millions of developers operating over 60 billion JVMs worldwide [37]. Since OpenJDK is an open-source implementation of the Java SE specification, it's reasonable to assume that many of the JVMs mentioned could be running on OpenJDK. Consequently, even marginal enhancements in energy efficiency within this platform can translate into substantial global environmental benefits. Our use of OpenJDK, therefore, aligns with both a pragmatic approach to software development and a commitment to broad-scale impact, leveraging the platform's ubiquity to maximize the potential reach and significance of our research outcomes.

3.2 Java Benchmarks

To the best of our knowledge, there is currently no established standard benchmark within the community for measuring the energy consumption of Java server workloads. Consequently, we have conducted our measurements across multiple sets of benchmarks (BMs) to ensure a comprehensive

assessment covering a wide spectrum of real-world applications and providing representative energy measurements.

Specifically, we have employed the following benchmark suites in our study: DaCapo [14], Renaissance [108], Hazelcast [54], and Specjbb15 [127]. These benchmark suites have been categorized into two distinct groups: throughput and latency.

3.2.1 Throughput BMs

Throughput benchmarks are those whose primary performance metric is execution time without factoring in jitter or stalls. This category includes some programs within DaCapo and encompasses all programs in the Renaissance benchmark suite.

Renaissance and DaCapo are well-established benchmark suites. DaCapo comprises nineteen distinct applications (nine of which are latency benchmarks), each offering configurable parameters, such as the number of mutators, the size of input data, and the number of iterations. Depending on the purpose of each work, we excluded some of the benchmarks due to known bugs, unsupported Java versions, or low GC activity. Each paper explains and motivates benchmark selection in more detail.

Renaissance is a benchmark suite focusing on concurrent and object-oriented workloads designed to exercise different concurrency primitives within JVM. This extensive suite comprises twenty-five individual programs; however, we have excluded *db-shootout* and *neo4j-analytics* due to unsupported Java versions and *rx-scarbel* due to the absence of significant GC activity.

3.2.2 Latency Benchmarks

A benchmark is a latency benchmark if it explicitly gauges the response time of discrete events in an application. Examples of such benchmarks include Hazelcast and Specjbb15, as well as select benchmarks in the DaCapo suite.

The Specjbb15 benchmark defined a special metric, critical-jOPS, that denotes the rate at which incoming requests accumulate when an application's latency exceeds predefined intervals (10ms, 25ms, 50ms, 75ms, and 100ms). An increase in latency leads to a decrease in critical-jOPS. The Hazelcast benchmark uses another latency metric that measures the time interval from request initiation to successful completion; in this context, lower is better.

In November 2023, DaCapo was extended by two different latency metrics: "simple" and "metered." For our evaluation, we have used the metered latency metric, which encompasses components such as request processing time, queuing delays, and GC [146]. The metered latency metric adjusts for the fact that both the producer of a request and the consumer of the same request run inside the same virtual machine. A pause, such as a GC pause, will,

therefore, affect not only the processing of requests but also the production of requests. Metered latency, therefore, adjusts the start times of tasks to fit a metered rate, even though they may have arrived late because a pause in the VM held them up.

SPECjbb2015 simulates the operations of an online store. Unlike throughput benchmarks, SPECjbb2015 does not employ a fixed workload; instead, it gradually increases the injection rate (IR) and the speed at which requests come into the store, thereby increasing the workload until it fully saturates the system. It is possible to run the benchmark with a fixed load (a fixed IR) and duration, although this configuration does not comply with the SPECjbb2015 guidelines and, therefore, does not yield any critical-jOPS and max-jOPS scores. However, external measuring tools are necessary since SPECjbb2015 does not produce energy measurements. When selecting a fixed IR, setting it below the maximum machine capacity is crucial to mirror real-world deployment conditions. To determine the appropriate level for a fixed workload, we relied on the max-jOPS parameter from the SPECjbb2015 compliant report, specifically without fixed IR, setting the fixed IR to 60%–80% of max-jOPS. The reason for choosing that range is because latency-sensitive workloads usually run on middle-range CPU loads, as we explain and demonstrate in Paper IV. Running on lower CPU loads wastes resources, while running on high CPU loads impacts latency negatively.

Hazelcast is a real-time stream processing system and functions as a pipeline for handling events that perform sliding window aggregation. Such aggregation is essential for tasks like noise removal (smoothing) in high-frequency data. We configured Hazelcast using parameters recommended in the benchmark guide [126]. Hazelcast operates with a fixed workload determined by its keyset size. The keyset size affects the allocation rate: the higher the keyset size, the higher the allocation rate. Thus, we can control the allocation rate of the benchmark through an input parameter.

We conducted experiments with varying IRs or keyset sizes whenever possible to capture a broader spectrum of behaviors.

3.3 Heap Size

As emphasized by Blackburn et al. [15], managed languages introduce an additional layer of flexibility in experimental evaluation, manifesting as a trade-off between space and time due to GC activity. In OpenJDK, this trade-off is exposed by a heap size parameter. Therefore, a comprehensive methodology would be incomplete without addressing the strategies for configuring heap sizes. Various methods can be employed for this purpose, and one common approach is to select a default heap size for an application *e.g.,* approximately three times larger than the minimum heap size required to allow the application to execute to completion using Serial GC [82] or G1 [17]. While this is

standard practice, it remains uncertain whether such an approach accurately reflects the heap sizes developers choose for production systems. For instance, developers often opt for heap sizes that are powers of two [46], as observed in numerous GitHub projects.

The ideal heap size is contingent upon numerous factors, including allocation rate and the size of live objects, both of which change during execution. Consequently, it becomes important to experiment with multiple heap sizes to reproduce more execution scenarios. After grappling with the challenge of identifying a suitable heap size, we have developed a dynamic approach for automatically determining a heap size, which we explain in Paper II. In the dynamic approach, we refrain from setting a fixed heap size for the entire execution. Instead, we allow it to fluctuate by controlling GC frequency by setting the amount of CPU resources we want to spend on GC work. As the allocation rate and size of the live set naturally vary during runtime, the frequency of GC activity will also change. This is because the CPU resource allocation remains constant while the workload fluctuates. Therefore, the frequency of GC activity adjusts based on whether GC can effectively manage the application's allocation rate. Rather than controlling GC frequency by statically setting a heap size, we choose to regulate it through CPU utilization while adjusting the heap size dynamically. In this manner, the application takes responsibility for selecting an appropriate heap size, while the user must determine the suitable amount of CPU resources for GC tasks. Although it still necessitates experimentation with various parameter values, the range is narrower — ranging from 0% to 100% instead of from 0MB to the maximum memory capacity of the machine.

3.4 Measuring Steady-State Performance

We use multiple techniques to ensure the stability and reliability of our results. To comprehend them, a reader must grasp two fundamental concepts: *iterations* and *runs*. Each run initiates a new instance of the JVM. Conducting multiple runs is crucial to mitigate inherent fluctuations in execution across different JVM instances. An iteration takes place within the same run. For short-running benchmarks, repeating iterations multiple times is essential to minimize the jitter induced by the Just-In-Time (JIT) compiler.

Our first technique involves determining the number of iterations per run for each benchmark to reach a steady state as suggested by Blackburn et al. [15]. These iterations are divided into two phases: a warm-up phase, comprising several initial iterations, and a measurement phase, consisting of the last five iterations, during which we collect data for analysis. The reason for considering the five last iterations is to account for variance in execution time within one JVM. We deem a state to be steady when the coefficient of variance in execution time remains below 0.05. For long-running benchmarks

like Hazelcast and SPECjbb2015, we assume that any initial jitter or variability is smooth throughout the benchmark run. Consequently, our analysis focuses on overall performance trends and measurements gathered throughout the run.

A cache flush operation is performed between each iteration to reset the states of the L1, L2, and L3 caches. This procedure ensures that each iteration starts with a pristine cache state, eliminating any residual data from previous iterations that might otherwise affect performance. This approach allows each iteration to commence with a clean slate, making it closer to an independent execution. It aims to eliminate noise attributed to the JVM while maintaining the integrity of each iteration.

To further enhance the reliability of our measurements, we run each benchmark a minimum of 10 times in separate JVM instances. This strategy mitigates the variance that can occur across different JVM instances, which could significantly impact performance.

Some additional JVM flags contribute to the stability of measurements. For instance, the DaCapo benchmarks trigger an explicit GC between each iteration. We disable these triggers, meaning the first GC of every iteration after the first may see some data from the previous iteration. In theory, explicit GCs achieve the same goal as cache flushes but also negatively affect internal behavior in the GCs themselves. ZGC incorporates built-in heuristics that enable it to discern patterns within long-running server applications, which is the typical deployment scenario for ZGC. By refraining from manually triggering GC, we allow ZGC heuristics to operate undisturbed.

For experiments where we set a maximum heap size, we set both initial (flag `-Xms`) and maximum heap size (flag `-Xmx`) to the same value. It minimizes fluctuations in the memory behavior of benchmarks. If set to different values, the benchmark starts with the set initial heap size value and then grows to the maximum heap size value if needed. However, memory expansion (committing memory) adds jitter to the execution, so we opt against it.

3.5 Measuring Energy Consumption

As energy consumption is the core of this thesis, we will briefly describe the energy measurement approach. We measured energy consumption using RAPL (Running Average Power Limit) [72], a feature supported by recent Intel systems. RAPL provides an interface to access machine-specific registers (MSRs) that estimate energy consumption across various system domains.

RAPL classifies energy consumption into four distinct categories:

- *PKG*: This category encompasses the entire package, including cores, shared caches, memory controllers, and optional uncore devices.
- *PP0*: Refers specifically to the cores themselves.

- *PP1*: Corresponds to the uncore devices, typically encompassing components such as GPUs.
- *DRAM*: Represents the memory subsystem.

Different architectures support different combinations of listed MSRs. Khan et al. [78] conducted a study on RAPL accuracy and concluded that RAPL readings are highly correlated with plug power, promisingly accurate enough, and have negligible performance overhead. Desrochers et al. [40] studied the accuracy of DRAM energy accuracy and concluded that the RAPL results match overall energy and power trends. Thus, we opted to use it in our experiments.

The impact of some energy optimizations can be difficult/impossible to measure. In Paper II, *e.g.,* we introduced a new approach to setting up a Java runtime parameter. This approach does not require extensive profiling of an application, which reduces the energy cost of pre-deployment. This reduction depends on the energy cost of a single application run multiplied by the number of required runs, which can vary depending on the method used to set up the parameter. Although numerical estimations are not provided in this case, it is a positive impact that is also important to acknowledge.

3.6 Statistical Analysis

Statistical analysis empowers us to compare outcomes across different scenarios and determine if one is "better" in some aspects, such as energy efficiency. Given the inherent variability in data, directly comparing two datasets, each with its unique variance and distribution, is not straightforward. Such variability can obscure true differences or similarities between the groups under comparison. We need to use statistical tests to address this challenge and make informed conclusions about the comparative performance or characteristics of two populations. These tests are designed to evaluate a null hypothesis.

The null hypothesis is a fundamental concept in statistics and scientific research, serving as a default position suggesting no effect or difference between two or more groups [48]. It is a statement used as a starting point for statistical testing, aiming to either be rejected or not rejected based on the evidence from the data. The null hypothesis posits that any observed differences or relationships in the data occur by chance or are insignificant in the study context. For example, suppose you were studying the energy efficiency effect of rewriting a Java program from C++. In that case, the null hypothesis might state that there is no difference in energy efficiency between a program written in Java and the same program written in C++.

The alternative hypothesis, on the other hand, suggests that there is a significant effect or difference. The researcher aims to support this through data analysis. Continuing with the previous example, the alternative hypoth-

esis would propose that there is a significant difference in energy efficiency between a program written in Java and the same program written in C++.

Statistical tests are used to evaluate the null hypothesis by determining the likelihood that the observed data would occur if the null hypothesis were true. If this likelihood (typically expressed through a p-value) is below a predetermined threshold (often 0.05 or 5% [32]), the null hypothesis is rejected in favor of the alternative hypothesis. Rejecting the null hypothesis suggests that the observed effect or difference is statistically significant and not likely due to chance.

It is important to note that failing to reject the null hypothesis does not prove it is true; it merely indicates that there is insufficient evidence to support the alternative hypothesis. The null hypothesis serves as a critical tool for ensuring that findings are not attributed to chance, thereby enhancing the reliability of scientific research.

As mentioned above, a data set has two attributes: variance and distribution. Variance measures the dispersion [42] or spread of data points around their mean (average) value. It quantifies how much the numbers in the dataset differ from each other and the mean. In other words, variance tells you the degree to which the data points are spread out or clustered around the mean. In general, we would like to have a smaller variance, which we achieve by improving the stability of experiments and consecutively precision of measurements as described in section 3.4. However, how can we test that two data sets have the same variance, which is required by the Student t-test [90]?

A distribution describes how dataset values are spread across different values. It shows the frequency of each value or range of values. There are multiple distribution types: normal, binomial, Poisson, uniform, and exponential.

When it comes to benchmarking for performance values, a common data distribution is a normal distribution, a.k.a. the Gaussian distribution. Its bell-shaped curve characterizes it. Its mean and standard deviation define it.

When we refer to statistical analysis in the papers, we mean that we used Welch's t-test [139], Grubb's outlier test [61], and Yuen's t-test [145]. Welch's and Yuen's t-tests were used because we do not assume equal variances but can assume a normal data distribution. Grubb's test identifies outliers, and Yuen's test is used if present.

4. Research summary

This chapter explains how each paper contributes to the goal of improving Java's energy efficiency.

4.1 Energy Consumption of Fully Concurrent GCs

As discussed in section 2.1, Java has automatic memory management, meaning it uses GC for memory allocation and deallocations. There exist multiple GC algorithms. However, as stated in section 2.2.1, fully concurrent GCs were overlooked from the energy perspective. Thus, the first goal of this thesis was to measure the energy consumption of fully concurrent GCs and compare it to other GC algorithms.

In Paper I, we measured the energy of programs using multiple GC algorithms in OpenJDK (see section 3.1). We studied all the algorithms available in OpenJDK[1]: Serial, Parallel, CMS, G1, ZGC, and Shenandoah. These GCs have different properties, which makes them useful in different scenarios. For example, Parallel and G1 are commonly used to achieve high throughput, while ZGC and Shenandoah, both fully concurrent GCs, are used for low-latency scenarios.

GCs also have many parameters that can be configured manually. These parameters differ between collectors. For example, one can specify the number of threads used in Parallel GC, while one can specify the size factor between young and old generations in G1. The common parameter across all GCs is heap size — a crucial parameter that affects performance parameters, like throughput or latency [140] (see section 2.1). Too much or too little memory negatively impacts data locality or frequency of GC interruptions, which affects how much GC competes for CPU resources with mutators. Hence, selecting an optimal heap size is important, yet their reliability is not absolute despite available methods. So one of the gaudiness [15] is experimenting with multiple heap sizes to identify trends.

By running benchmarks with different GCs and their parameters, we wanted to know:

[1]Including algorithms outside OpenJDK would be possible in principle, but given the large differences in energy efficiency between programming languages and runtimes as discussed in section 2.2.2, it would be very difficult if not impossible to attribute changes to specific algorithms if we allowed language or runtime to vary.

1. Is there an energy benefit to changing a default GC?
2. What is the energy profile of fully concurrent GCs compared to other GCs?
3. What is the energy impact of different parameters?

To answer these questions, we experimented with four different sets of benchmarks (see section 3.2) following the best benchmarking practices (see section 3.4).

By using a "naive" brute-force approach to try all possible GCs and their configurations, including multiple Java versions, and then picking the best in terms of energy efficiency, we found that if there are no performance constraints, we could use up to 47% less energy than running OpenJDK with the default GC (G1) and the heap 3× more than memory required to run with Serial GC without OOM. If a 5% performance reduction is allowed, many GC configurations are excluded as they do not meet that bar. Still, switching from G1 to other collectors can reduce energy consumption by 40% compared to 47%. While comparing specific combinations with specific heap sizes under made-up performance thresholds is somewhat speculative, these results clearly demonstrate that the choice of GC can significantly impact the application's energy consumption and that relying on the default GC in OpenJDK is not always the best solution for sustainability.

The fully concurrent GCs ZGC and Shenandoah were the least energy-efficient GCs in OpenJDK in our measurements. We identified two reasons for that. First, we used single-generation versions of both collectors as the generational versions were unavailable then. This means that each GC cycle processed young and old generations, which is less efficient than separating the young and old heap and managing their memory with different frequencies. We took this into account, and for Paper II, III and IV, we used generational ZGC.

The second reason for the energy inefficiency of ZGC and Shenandoah is that concurrent GCs must carry out substantial work to synchronize worker threads with mutators. This is unnecessary in less concurrent GCs because mutators can be stopped when GC runs. For example, remapping all pointers in a GC cycle in ZGC requires atomic operations. This additional unavoidable work might suggest that fully concurrent GCs inherently can not be more energy-efficient than other GCs. However, we claim that full concurrency can also be exploited to benefit energy efficiency, which we will demonstrate in Paper III and IV.

To conclude, Paper I shows that fully concurrent GCs are the least energy-efficient GCs on OpenJDK. In Paper III and IV, we will explore techniques to improve the energy efficiency of those GCs, building proof-of-concepts on top of ZGC.

4.2 Improving Heap Sizing

Before we get into optimizing ZGC for energy efficiency, we will take a "detour" to address another issue that was apparent in Paper I, namely the problem of manually picking a maximum heap size or its impact on performance. There are multiple methods to set a heap size discussed in section 3.3, but neither of the approaches can be done without extensive profiling. There are various issues with this:

1. If a program needs to be run multiple times to set a parameter, then the energy cost of pre-deployment increases, especially if a program needs to run on multiple architectures, where a heap size needs to be set for each one.
2. Setting the heap size requires benchmarking with a representative load on a representative machine. As this is a complicated task, the risk of making mistakes increases, meaning not achieving desired performance goals. Running a program in a suboptimal way has an implicit energy penalty, like paying the extra runtime cost if the program runs slower than it could have.
3. At the same time, the connection between heap size and energy is not straightforward and needs to be made more transparent to programmers. Smaller heap sizes will reduce the DRAM energy required, increasing CPU energy as more GC needs to be performed. The trade-offs between the two are not obvious because, as discussed above, changing heap size also affects performance, affecting CPU energy.
4. Last but not least, the current way of setting a heap size for the whole program may not be optimal for all program phases if different phases exhibit different behavior. The heap size is currently set to accommodate the phase with the highest allocation rate. Otherwise, a program will crash with an OOM error or perform poorly. Dynamically adjusting heap size depending on the current requirements of a program seems sensible. As we discussed before, too much memory can negatively affect performance.

Because of these reasons, a parameter as critical as heap size should be intuitive to set without profiling; the values should have a transparent connection with performance and energy outcomes and dynamically adjust for the program's current memory requirements.

There are multiple approaches outside of OpenJDK to adjust heap size dynamically, for example, used in Immix and .NET. Immix [13] is a stop-the-world GC that resizes the heap by allocating additional memory blocks if necessary during GC pauses. Otherwise, Immix triggers a GC cycle to reclaim unused memory when free memory drops below a threshold. .NET [138] allows objects to be pinned, meaning that these objects can not change their location. While pinning is useful for interoperability with unmanaged code, it challenges the GC, especially when adjusting the heap size. To mitigate

the challenges posed by object pinning and memory fragmentation, .NET offers a ConserveMemory API for interacting with the GC programmatically. This API allows applications to opt for a mode where memory conservation is prioritized, albeit at potential costs. Activating this setting can lead to more frequent GC cycles and possibly longer pause times during these collections. Frequent GCs can affect application performance but are used here as a strategy to manage memory more efficiently in constrained environments. ZGC does not have object pinning, therefore adjusting heap size can be done without extra complexity connected to it.

Paper II describes a new dynamic approach to setting a heap size at runtime that we propose. The main insight is that memory and GC CPU utilization (CPU resources required to perform GC) have an inverse correlation. To understand it, let us first consider the correlation between memory and the frequency of GC. GC can run less frequently if more memory is available. And conversely, less memory leads to more frequent GC. In the original approach, the heap size parameter controls the frequency of GC by setting the memory. Memory is static, but the GC frequency is dynamic depending on the dynamic metrics of an application, like allocation rate.

GC frequency has a direct correlation with CPU resources required to execute GC. More frequent GCs require more CPU. Thus, memory and CPU utilization are connected through GC frequency. If previously, the GC frequency was controlled by memory, we propose to use CPU utilization as a proxy for the allocation rate, which controls the GC frequency. This shift in perspective allows for multiple benefits. First, GC CPU utilization has a fixed range of values from 0% (no CPU resources to GC) to 100% (GC gets as much CPU time as a program). This range is smaller than an application's possible memory range, making it easier to choose a value that is not too much off and to pick a sensible default value. The GC CPU utilization parameter is a proxy for allocation rate across architectures, ensuring that the GC will allocate the same requested portion, for instance, 10% of available cores, regardless of the underlying architecture. It is not the same with memory, where 25% of RAM (default ZGC value) might not be enough on one machine and too big on another.

All of the above, although very valuable, does not address the problem of transparency between parameter values and performance and energy goals. To address it, we conducted experiments on a range of throughput and latency-oriented benchmarks to see if there is a connection between certain GC CPU utilization values and common optimization goals(execution time, memory usage, and energy). We experimented with 5%, 10%, 15%, and 20% GC CPU utilization rates. The rates higher than 20% seem counterproductive for achieving low latency because we want to avoid GC competing with mutators for CPU resources, but nothing prevents a user from trying a wider range.

We observed that higher GC CPU utilization rates tend to lead to less memory usage as GC executes frequently to free up memory (as one would expect).

Execution time and energy have similar trends; they were the lowest in a range between 10 and–15%. The technique does not seem to have a negative impact on latency, which is crucial for concurrent GCs, and setting a parameter wrong will not result in an OOM error if an application fits in 90% of the RAM of the entire machine. Compared to manually setting heap size for the entire application, we could observe improvements in performance and energy, probably due to the dynamically changing memory, as discussed at the beginning of this section.

Overall, Paper II shows that energy reduction goals can be aligned with other important aspects of the deployment process, like making it easier for a programmer to find a "good enough" value of runtime parameters. This approach has implicit impacts on energy, such as reduced energy cost of predeployment and reduced cost of running an application in a suboptimal mode due to wrongly set runtime parameters.

4.3 GC Scheduling for Energy Efficiency

At the intersection of runtime environments and operating systems, GC scheduling exemplifies how interdisciplinary research can boost energy efficiency. Here, we explore two techniques to improve energy efficiency and hardware utilization in server workloads: scheduling GC work on energy-efficient cores and scheduling GC on idle cores. In the following sections, we will explore how these approaches contribute to the goal of this thesis: improving the environmental sustainability of ICT by enhancing the energy efficiency of Java garbage collection.

4.3.1 Energy Impact of Scheduling GC on Energy-Efficient Cores

As discussed in Paper I, fully concurrent GCs pay extra costs for synchronization, which is hard to avoid. Therefore, fully concurrent GCs seem inherently less energy-efficient than stop-the-world GCs. However, we can utilize full-concurrency to reduce energy consumption. The main insight here is that fully concurrent GCs are not on a critical path of the application regarding latency because they rarely and only briefly stop mutators from running. This is not the case for stop-the-world collectors, as GC needs to run as fast as possible to restart the application threads during every pause. So, the insight can be reformulated as slowing down fully concurrent GCs should not affect application performance metrics. Of course, GC has an intricate connection to performance, so the trade-off between doing less GC work and performance loss needs to be explored.

Why do we want to slow down GC? To run at a reduced frequency, *e.g.,* is a common technique to get energy reduction[28]. In Paper III, we explored an alternative scheduling technique, namely running a GC on simple energy-efficient cores instead of more complex and more power-hungry cores in heterogeneous systems, to investigate if slowing down a fully concurrent collector can lead to energy reduction. Heterogeneous systems consist of several types of cores. For example, CPUs and GPUs or CPUs and FPGAs. In our case, we looked into the so-called big. LITTLE architecture. Some CPUs — called p-cores or performance cores — are complex, out-of-order cores with big caches, and some — called e-cores or energy efficiency cores — are small, out-of-order but with a shorter re-order buffer than on p-cores, with smaller cache sizes. The difference in features leads to energy consumption differences even if they run at the same frequency (in GHz). Simple energy-efficient cores are a good fit for GC because GC is memory-bound and has poor temporal and spatial locality, so it does not benefit from additional complexity for out-of-order execution and big caches.

Two main research questions in this context are: (1) can we reduce the energy consumption of the whole application by scheduling GC on e-cores instead of p-cores, and (2) can it be done without sacrificing performance metrics such as execution time and latency?

Our experiments showed that scheduling GC work on e-cores instead of p-cores overall can lead to $\approx 3\%$ energy reduction without performance and mean latency degradation on the system we used in our benchmarks. The overall energy reduction can increase to $5.3\% \pm 0.0225$ by tuning the number of e-cores. Importantly, a programmer does not need extra effort to get the energy benefit we demonstrated. It is achieved by running a program on a server with e-cores and using the automatic scheduling policy provided by (our modified) JVM.

Another interesting aspect of this study is that it is a semi-replication study. The basic principles were explored more than a decade ago. However, technology does not stand still, and some assumptions that were true in the past do not hold true any longer. For example, Cao et al. [20] deliberately experimented with small heterogeneous systems because the assumption was that adding more big cores would not be feasible in the future because of power or energy constraints. Modern heterogeneous designs are bigger than previously anticipated. In addition, fully concurrent GCs were not available a decade ago, while ZGC is a production GC today. Testing old hypotheses in a modern context holds value for the scientific community. Our findings differed from those reported by Cao et al. [20], who observed an 11% energy efficiency improvement. We achieved a 3% improvement. Although differences can be explained by various factors, like testing a new GC algorithm on contemporary hardware (more details in Paper III), the core principle stays the same - running GC on energy-efficiency cores has the potential for energy reduction.

4.3.2 Scheduling Concurrent GC on Idle Cores to Improve Hardware Utilization.

Finally, Paper IV explores opportunistic scheduling of GC by only running GC on idle cores or during the application's idle periods. This approach is called slack-based scheduling, widely explored in the context of real-time systems to reduce latency [6, 64]. We explored this technique in the context of server workloads to delay horizontal scaling and improve hardware utilization.

Horizontal scaling, a prevalent technique in managing server workloads with latency constraints, is frequently employed in scenarios where workloads operate within stringent limits on average CPU utilization [135, 141]. Should an application surpass this threshold, the typical recourse adopted by vendors is to scale up (horizontal scaling) by relocating the application to another server to sustain acceptable latency levels [135].

In OpenJDK, GC operates alongside application threads sharing system resources, especially during concurrent GC operations. The underlying operating system manages the threads' scheduling, with workers and mutators assigned identical thread priorities. Consequently, during concurrent GC cycles, GC workers may preempt mutator threads in instances of CPU resource contention within the system. This preemptive action disrupts mutators engaged in critical tasks, such as request processing, resulting in heightened latency. As CPU load escalates, this contention intensifies, often necessitating premature scaling to fulfill Service Level Agreements (SLAs). However, workloads characterized by sporadic spikes in activity frequently experience periods of reduced activity, marked by decreased incoming request rates. If concurrent GC operations could be deferred to these periods, it might be feasible to sustain higher average CPU loads without adverse effects on latency. Such an approach could elevate the threshold for horizontal scaling, thereby enhancing hardware utilization.

We anticipated three behaviors based on CPU load: low, middle, and high. When the CPU load is low, mutators and workers can co-exist without affecting each other since they have no contention for CPU resources. Thus, scheduling GC activity on idle cores should not make any difference from the default scheduling technique. On middle-range CPU loads, contention increases, and delaying GC to idle periods should improve latency. On high CPU loads, GC might experience starvation, meaning that GC would never get to execute due to a lack of idle CPUs. In this case, an application will sooner or later run out of memory and perform poorly or crash. To counter this, we measured application CPU utilization and allowed GC to preempt mutators (fallback) when the CPU load exceeds a threshold.

We tested our hypothesis using a workload focused on request processing. This workload gradually ramps up the influx of incoming requests until the application response time is undesirable. The CPU load scales proportionally with the request volume, enabling us to examine performance across all

three CPU load ranges. As anticipated, we found no discernible difference in response times under low CPU loads. However, in the middle range, we observed improvements in latency. We witnessed a significant deterioration in latency at higher CPU loads without timely reversion to a default scheduling policy. Nonetheless, a fallback approach nearly restored latency levels to those of the default configuration. Additionally, our solution demonstrated the ability to handle up to 15% more requests within a 25ms latency threshold compared to the default setup.

Overall, this solution could delay the horizontal scaling of applications working on middle-range CPU loads and, as a consequence, improve hardware utilization. In particular, the implementation technique is simple enough to incorporate into OpenJDK easily.

5. Future Work

This chapter briefly discusses future work.

The exploration of using machine-learning techniques for GC tuning, as elucidated in Paper I, paves the way for further research endeavors. While our initial inquiry primarily focused on tuning for energy efficiency, numerous other aspects can be optimized. Presently, we are looking into fine-tuning memory-related JVM parameters within the G1 default collector. An example of such crucial parameters is the relative size between young and old generations, which impacts both performance and latency. For example, allocating more memory to the young generation can reduce the frequency of young generation collections, which tend to be faster but more frequent. On the other hand, allocating more memory to the old generation can reduce the frequency of full garbage collections, which are slower but less frequent. The frequency of GC affects total pause time (related to throughput) and maximum pause time (related to latency) as well as data layout. Flags such as `-XX:G1MaxNewSizePercent`, which sets the maximum young generation size as a percentage of the heap, and `-XX:G1NewSizePercent`, which sets the minimum young generation size, play pivotal roles in this optimization process. However, configuring these memory-related flags, as discussed in Paper II, poses considerable challenges, often necessitating multiple runs for profiling, the results of which can be specific to one specific machine. GC logs generated by the JVM offer invaluable insights into application behavior, encompassing GC events, pause times, object allocation rates, and live-set information. Leveraging this wealth of data through machine learning models presents a compelling opportunity to dynamically optimize JVM performance and memory management.

The continuation of Paper II is its integration into OpenJDK, marking a milestone in the journey towards enhancing ICT sustainability (JDK Enhancement Proposal building on this work is now available online [150]). This integration represents a crucial step forward, amplifying the impact of this research. The second future direction is investigating dynamic heap sizing for multi-tenant environments, *i.e.,* where multiple applications coexist on the same machine concurrently. Also, we are interested in comparing this approach with the methodology devised by Kirisame et al. [81]. They derived a square-root formula to dynamically set heap sizes based on various runtime parameters, including GC duration and application allocation rates. Remarkably, this formula operates autonomously without prior knowledge regarding the behavior of co-applications. This comparison could shed light

on the efficacy and adaptability of dynamic heap size adjustment within the dynamic landscape of multi-tenant environments as well as present possible ways of combining the two approaches.

Expanding the investigation into memory tradeoffs highlighted in Paper III, a promising direction for future research involves mitigating the memory overhead associated with scheduling GC work on e-cores. Our preliminary analysis suggests that memory overhead is because the JVM is not aware of the available underlying hardware resources. Addressing this gap represents an essential initial step towards optimizing memory usage. Additionally, given the similarity between e-cores and p-cores, further exploration into frequency reduction strategies could amplify the observed energy reduction benefits. By exploring these areas, we aim to refine the efficiency and performance of ZGC in heterogeneous computing environments.

6. Conclusions

This research contributes to the understanding and enhancement of the energy efficiency of ZGC.

The central theme of this study is enhancing the energy efficiency of Java applications by optimizing GC scheduling and heap sizing techniques. Using systematic benchmarking, algorithm design, and empirical evaluation, this research narrows the gap between energy efficiency objectives and performance goals in Java applications.

Our results show that the selection of the GC algorithm plays a crucial role in the energy consumption of a Java application. Extensive benchmarking and analysis reveals that fully concurrent GCs, such as ZGC and Shenandoah, exhibit higher energy consumption compared to other GC algorithms due to the additional overhead associated with synchronizing highly concurrent threads.

To address the challenges posed by manual heap size configuration, we propose a dynamic heap sizing strategy based on GC CPU utilization. By simplifying the heap size parameter-setting process for programmers without the need for extensive profiling, we give programmers direct control of the energy consumption of GC as well as reduce the energy required for pre-deployment.

Moreover, we propose two scheduling-based techniques for reducing energy consumption in fully concurrent GCs. These techniques involve scheduling GC work on energy-efficient cores on heterogeneous systems or idle cores. Both approaches demonstrate promising energy savings without compromising performance or latency metrics and, importantly, without requiring any changes to the programs deployed on the JVM.

In conclusion, the findings of this research lay a foundation for future endeavors aimed at enhancing energy efficiency in Java applications. By tackling the challenges associated with GC selection, heap sizing, and GC scheduling, this research contributes to the advancement of sustainable computing practices.

7. Popular summary in English

Running software requires energy. The work presented in this thesis aims to reduce the amount of energy needed to run programs that use automatic memory management. Memory is a finite resource in a machine, and automatic memory management means that the program manages its own memory automatically instead of forcing the programmer to keep track of certain parts of memory that are no longer needed and explicitly delete them. The technique for automatic memory management used in this thesis is called garbage collection. There are multiple algorithms for implementing garbage collection, from throughput collectors that need to stop the program to perform a collection to concurrent collectors that permit the program to keep running to various degrees while its garbage is being collected. A garbage collector that never stops the program to do work is called a fully concurrent garbage collector. Usually, such collectors are needed to keep the response time of a program within some tight bounds. A typical example is a web server that serves incoming requests. If the server could be stopped at any moment because memory was "full," this could translate into dropped requests or users losing interest and lost business opportunities. Since garbage collection rarely stands in the way of program work in such collectors, the server can keep processing incoming requests without interruptions. However, collecting garbage concurrently with a program typically requires more work because actions in the program and actions in the garbage collector must be coordinated, or else the system might crash. In terms of energy, concurrent collectors are less energy-efficient than throughput collectors. This thesis explores ways to reduce the energy consumption of concurrent collectors and how that can impact performance. We experiment with ZGC, one of the garbage collectors bundled in Java, one of the world's most-used programming languages, and, therefore, most likely one of the most used concurrent collectors worldwide.

The work in this thesis is divided into four papers, each exploring one aspect of this space.

In Paper I, we demonstrated that the choice of garbage collector significantly impacts a program's energy consumption, highlighting the importance of optimizing the energy efficiency of garbage collectors to improve the sustainability of computing. In particular, we investigated the energy consumption of various garbage collector algorithms to understand how much energy fully concurrent collectors consume compared to throughput collectors. We found that fully concurrent collectors exhibit higher energy consumption due to the inherent overhead of coordinate work done by the garbage collection and the program work, which is necessary when both run concurrently.

In Paper II, we addressed the problem of manually selecting heap size in Java. The heap size controls how much memory a program can use and, therefore, the frequency of garbage collection. Thus, this parameter directly impacts energy consumption and performance. Traditional methods for setting heap size often require extensive profiling and produce a machine-specific value that may change as the program evolves. We proposed a dynamic, automatic approach that simplifies parameter selection to simplify deployment and lower the risk of improper deployment, which may cause energy inefficiencies, performance regressions, or both. In this approach, we set the amount of CPU resources we want to spend on garbage collection, and then heap size dynamically adjusts at runtime based on what we set. A benefit of this is a reduced range of values that the set parameter can change from 0–available memory to [0,1] for CPU overhead. We reduced the range even further by conducting experiments with multiple values and identifying that a range from 0.1–0.2 is optimal for high performance and low energy.

Papers III and IV explore harnessing the concurrency in garbage collection to reduce energy consumption. Unlike throughput collectors, the (almost-fully) concurrent collectors seldom halt application threads, meaning garbage collection work is not critical for a program's performance. This implies that as long as the rate at which garbage collection frees up memory can match the rate at which the program allocates memory, garbage collection does not impact response time negatively. If the garbage collector cannot keep up, performance may suffer, or a program may crash due to insufficient memory.

Our work demonstrates that strategically slowing down garbage collection operations can yield significant energy savings without compromising a program's response or execution times. Slowing down can be achieved by adjusting the scheduling of garbage collection work on available hardware resources. Modern processors are equipped with multiple cores and require efficient thread-core assignment, which is the essence of scheduling.

Two primary scheduling techniques were explored: scheduling garbage collection work on energy-efficient cores in heterogeneous systems and scheduling garbage collection only when there are more processors in the system than there is program work. Energy-efficient cores are specialized to consume less power than standard cores, contributing to overall energy savings. Heterogeneous systems, such as those found in smartphones and computer processors like the Apple M-family of processors and the Intel Alder Lake family, combine ordinary and energy-efficient cores for optimized performance and energy efficiency.

Opportunistic scheduling means that garbage collection only runs when the system has available processors. These are currently not doing any program work, so they can be used without competing with the program for resources. A key challenge that must be overcome is dealing with situations when the program's performance becomes dependent on garbage collection or when garbage collection is not scheduled frequently enough to keep up

with the program's allocation rate. This allows latency guarantees to be met using a smaller number of machines, which saves energy.

In summary, the work described in this thesis contributes to a more sustainable information and communication technology sector by improving the energy consumption of concurrent garbage collection.

8. Popular summary in Swedish

Att köra ett program kräver energi. Målet med arbetet som presenteras i denna avhandling är att minska mängden energi som krävs för att köra program som använder automatisk minneshantering. Minne är en begränsad resurs i en dator och automatisk minneshantering innebär att programmet hanterar sitt eget minne automatiskt istället för att programmeraren skall behöva hålla reda på när delar av minnet inte längre behövs och explicit ta bort dem. I denna avhandling fokuserar vi på en teknik för automatisk minneshantering som på svenska kallas skräpsamling. Det finns flera algoritmer för att implementera skräpsamling, från sådana som fokuserar på genomströmning och som stoppar programmet för att utföra arbete till skräpsamlare som låter programmen fortsätta köra medan skräp samlas in. På engelska kalls en sådan skräpsamlare "concurrent". En skräpsamlare som aldrig stoppar programmet för att utföra arbete kallas för "fully concurrent". Vanligtvis behövs sådana skräpsamlare för att hålla svarstiden för ett program inom hårda gränser. Ett typiskt exempel är en webbserver som hanterar inkommande förfrågningar. Om servern kunde stoppas när som helst på grund av att minnet var "fullt" skulle detta kunna leda till försvunna förfrågningar eller att användare tappar intresset på grund av fördröjningen, vilket leder till förlorade inkomster. Eftersom skräpsamling sällan hindrar programarbete i sådana ("concurrent") skräpsamlare kan servern fortsätta bearbeta inkommande förfrågningar utan avbrott. Dock kräver skräpsamling utan att avbryta programmets exekvering vanligtvis mer arbete eftersom händelser i programmet ständigt måste koordineras med händelser i skräpsamlaren, annars kan systemet krascha. Med avseende på energieffektivitet är skräpsamlare som är "concurrent" mindre energieffektiva än sådana som fokuserar på genomströmning. Denna avhandling utforskar sätt att minska energiförbrukningen hos de föregeående och hur det kan påverka prestandan. Vi experimenterar med ZGC, som är en av skräpsamlarna som ingår i Java, ett av världens mest använda programmeringsspråk och därmed förmodligen en av de mest använda "concurrent" skräpsamlarna i världen.

Avhandlingen är uppdelad i fyra olika artiklar som var och en utforskar en aspekt inom området.

I artikel I visar vi att valet av skräpsamlare påverkar ett programs energiförbrukning avsevärt, vilket betonar vikten av att optimera energieffektiviteten hos skräpsamlare för att förbättra hållbarheten i ICT-sektorn. Specifikt undersöker vi energiförbrukningen hos olika skräpsamlaralgoritmer för att förstå hur mycket energi skräpsamlare som är "concurrent" förbrukar jämfört med sådana som fokuserar på genomströmning. Vi visar att den föregående kategorin skräpsamlare har en högre energiförbrukning på grund av den tidigare

nämnda koordinationen av arbete mellan skräpsamlare och program, som blir nödvändigt när båda körs samtidigt.

I artikel II behandlar vi problemet med manuellt val av heapstorlek i Java. Heapstorleken kontrollerar hur mycket minne ett program kan använda och därmed skräpsamlingsfrekvensen. Denna parameter påverkar sålunda energiförbrukningen och prestandan. Traditionella metoder för att ställa in heapstorlek kräver ofta omfattande profilering och producerar ett maskinspecifikt värde som kan ändras när programmet utvecklas. För att förenkla distribution och minska risken för felaktig driftsättning som kan orsaka energiineffektivitet, prestandaregression eller båda delarna, föreslog vi en dynamisk, automatisk metod som förenklar valet av heapstorlek. I vår föreslagna metod väljer programmeraren istället hur mycket beräkningsresurser som skall läggas på skräpsamling, som en faktor av programmets beräkningsansvändning, och sedan justeras heapstorleken dynamiskt vid körning baserat på detta. En fördel med detta är en minskad sökrymd av möjliga värden för heapstorleksparametern, från 0–tillgängligt minne när vi manuellt måste bestämma en heapstorlek, till [0,1] för beräkningsoverhead för GC. Vi minskar intervallet ytterligare genom att genomföra experiment med flera värden och identifierar att en overhead på 0,1–0,2 ger en bra balans mellan hög prestanda och låg energiförbrukning.

Artiklarna III och IV utforskar möjligheten att utnyttja att skräpsamling och program exekverar samtidigt för att minska energiförbrukningen. Till skillnad från skräpsamlare som fokuserar på genomströmning pausar "concurrent" skräpsamlare sällan programmet, vilket innebär att skräpsamling inte är kritisk för ett programs prestanda. Detta innebär att så länge som takten i vilken skräpsamlaren kan frigöra minne kan matcha takten som programmet allokerar minne, påverkar inte skräpsamlingen svarstiden negativt. Om skräpsamlaren inte kan hålla jämna steg kan prestandan påverkas eller ett program kan krascha på grund av otillräckligt minne.

Våra resultat visar att betydande energibesparingar är möjliga, utan att kompromissa med programs svarstid eller exekveringstid, genom att sakta ned skräpsamlingen. Detta kan uppnås genom att justera schemaläggningen av skräpsamlingsarbete på tillgängliga maskinresurser. Moderna processorer är utrustade med flera parallella beräkningselement och kräver att beräkningar läggs ut på beräkningselementen på ett effektivt sätt. Detta är kärnan i schemaläggning.

Två primära schemaläggningstekniker utforskas: schemaläggning av skräpsamlingsarbete på energieffektiva beräkningselement i heterogena system och schemaläggning av skräpsamling endast när det finns lediga beräkningselement i systemet. Energieffektiva beräkningselement är specialiserade för att dra mindre ström än vanliga beräkningselement, vilket bidrar till energibesparingar. Heterogena system, som de som återfinns i smartphones och processorer som Apples M-processorer och Intels Alder Lake-familj, kom-

binerar både vanliga och energieffektiva beräkningselement för optimerad prestanda och energieffektivitet.

Opportunistisk schemaläggning innebär att skräpsamling bara utförs när det finns lediga beräkningselement i systemet. Eftersom dessa för närvarande inte utför något programarbete kan de användas utan att konkurrera med programmet om maskinens resurser. En nyckelutmaning som måste övervinnas är att hantera situationer när programmets prestanda blir beroende av skräpsamling eller när skräpsamlingen inte schemaläggs tillräckligt ofta för att hålla jämna steg med hur programmets konsumerar minne. Detta gör att latensgarantier kan uppfyllas med färre antal maskiner, vilket sparar energi.

Sammanfattningsvis bidrar det arbete som beskrivs i denna avhandling till en mer hållbar informations- och kommunikationstekniksektor genom att förbättra energiförbrukningen hos “concurrent” skräpsamling.

9. Acknowledgments

I extend my deepest gratitude to my supervisor, Tobias Wrigstad, for his unwavering support during the most challenging moments of my journey. Tobias, your guidance, and mentorship have been invaluable, and I am profoundly grateful for your insights into research and your genuine concern for the well-being of those under your guidance. Working alongside you has been an absolute pleasure, and I am truly fortunate to have you as my supervisor. I also want to express my gratitude to Stefanos Kaxiras for his co-supervision.

My heartfelt thanks go to my Oracle team—Erik, Roberto, Albert, Jesper, and all the incredible individuals I had the pleasure of working with. Your camaraderie and dedication brought meaning to my research by presenting real-world problems to solve and sharing constructive feedback.

I am indebted to my "seniors"—Sofia, Matteo, and Davide—for their wise advice and inspiration. Your exemplary leadership and wisdom have been a constant source of motivation.

I am immensely grateful to all my co-authors for their collaboration and contribution to our shared research.

A special acknowledgment goes to my dear friend and collaborator, Mihail, whose friendship has been a source of inspiration and encouragement. Your influence on my life cannot be overstated.

I am incredibly fortunate to have had remarkable colleagues in both the UART and PL groups. Johan and Chris, your meaningful contributions have enriched my life in countless ways. To my office mates—Amanda, David, and Anders—thank you for the engaging conversations. I will cherish our time together.

To Ellias, Eva, Matilda, Jonas, Ellen, and all the wonderful colleagues from UART and PL groups, thank you for your insightful discussions and constructive feedback. I will always remember the party house.

My heartfelt appreciation goes out to my GGs—Amanda, Clara, Vilda, Anna, Louie, Belinda, Claudia, and Cecilia. Meeting each of you was a highlight of my PhD journey, and I am grateful for the bond we share.

I am deeply touched by the friendship and support of my dear friends Tharyk and German. Your presence in my life is a source of immense joy and strength.

I want to give a special mention to my friend Olga, whose unwavering support sustained me through challenging times. Your determination and resilience are truly admirable.

Campus1477 holds a special place in my heart, thanks to incredible people like Starla, Ludo, Kunal, Ulrika, Merima, Arianne, Elin, and others. Your

friendship and camaraderie have made my experience unforgettable. Keep getting those sport-induced endorphins.

I am grateful for the adventures shared with my "gangsters"—Georgii, Vijay, Maria, and Arezoo. Together, we embraced new challenges and created lasting memories by doing one new thing each month. Here's to many more!

A heartfelt thanks to my fellow book club members for reigniting my love for fiction and enriching our discussions.

To my Russian community in Uppsala — Georgii, Tatiana, and Dimitry — thank you for the memorable gatherings and insightful conversations.

I am immensely grateful to Kate for her friendship, despite everything that is going on in the world.

Thanks to Kedar, Jacob, Francesco, Maria, Alba, and others, my involvement in BFC has been a rewarding experience. You are my inspiration!

To Elke, Myrthe, Kristel, Marta, and Giselle — your friendship has brought joy and laughter into my life.

I am deeply thankful for my best friend Polina's and my niece Masha's unwavering support. Your love and encouragement mean the world to me.

Last but not least, I want to express my heartfelt thanks to my family for their enduring love and support. Your faith in me sustains me through life's challenges.

References

[1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An efficient parallel heap compaction algorithm. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, page 224–236, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138318. doi: 10.1145/1028976.1028995. URL https://doi.org/10.1145/1028976.1028995.

[2] Shoaib Akram, Jennifer B. Sartor, Kenzo Van Craeynest, Wim Heirman, and Lieven Eeckhout. Boosting the priority of garbage: Scheduling collection on heterogeneous multicore processors. *ACM Trans. Archit. Code Optim.*, 13(1), mar 2016. ISSN 1544-3566. doi: 10.1145/2875424. URL https://doi.org/10.1145/2875424.

[3] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 62–77, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356985. doi: 10.1145/3192366.3192392. URL https://doi.org/10.1145/3192366.3192392.

[4] D.H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–259, 1999. doi: 10.1109/MICRO.1999.809463.

[5] Anders S. G. Andrae and Tomas Edler. On global electricity usage of communication technology: Trends to 2030. *Challenges*, 6(1):117–157, 2015. ISSN 2078-1547. doi: 10.3390/challe6010117. URL https://www.mdpi.com/2078-1547/6/1/117.

[6] Joshua Auerbach, David F. Bacon, Perry Cheng, David Grove, Ben Biron, Charlie Gracie, Bill McCloskey, Aleksandar Micic, and Ryan Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, page 245–254, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605584683. doi: 10.1145/1450058.1450092. URL https://doi.org/10.1145/1450058.1450092.

[7] Timur Babakol, Anthony Canino, Khaled Mahmoud, Rachit Saxena, and Yu David Liu. Calm energy accounting for multithreaded Java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 976–988, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409703. URL https://doi.org/10.1145/3368089.3409703.

[8] Timur Babakol, Anthony Canino, and Yu David Liu. Eflect: porting energy-aware applications to shared environments. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 823–834, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510145. URL https://doi.org/10.1145/3510003.3510145.

[9] Carmen Badea, Alexandru Nicolau, and Alexander V. Veidenbaum. Impact of jvm superoperators on energy consumption in resource-constrained embedded systems. *SIGPLAN Not.*, 43(7):23–30, jun 2008. ISSN 0362-1340. doi: 10.1145/1379023.1375661. URL https://doi.org/10.1145/1379023.1375661.

[10] Antonio Carlos S. Beck, Mateus B. Rutzig, and Luigi Carro. Cache performance impacts for stack machines in embedded systems. In *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design*, SBCCI '06, page 155–160, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934790. doi: 10.1145/1150343.1150385. URL https://doi.org/10.1145/1150343.1150385.

[11] Lotfi Belkhir and Ahmed Elmeligi. Assessing ICT global emissions footprint: Trends to 2040 & recommendations. *Journal of Cleaner Production*, 177: 448–463, 2018. ISSN 0959-6526. doi: https://doi.org/10.1016/j.jclepro.2017.12.239. URL https://www.sciencedirect.com/science/article/pii/S095965261733233X.

[12] Inc Black Duck Software. *Top Open Source Licenses*, 2016. URL https://web.archive.org/web/20160719043600/https://www.blackducksoftware.com/top-open-source-licenses.

[13] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. *SIGPLAN Not.*, 43(6):22–32, jun 2008. ISSN 0362-1340. doi: 10.1145/1379022.1375586. URL https://doi.org/10.1145/1379022.1375586.

[14] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 169–190, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933484. doi: 10.1145/1167473.1167488. URL https://doi.org/10.1145/1167473.1167488.

[15] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM*, 51

(8):83–89, aug 2008. ISSN 0001-0782. doi: 10.1145/1378704.1378723. URL https://doi.org/10.1145/1378704.1378723.

[16] Randal E Bryant and David Richard O'Hallaron. *Computer systems: a programmer's perspective*. Prentice Hall, 2011.

[17] Zixian Cai, Stephen M Blackburn, Michael D Bond, and Martin Maas. Distilling the real cost of production garbage collectors. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 46–57. IEEE, 2022.

[18] Anthony Canino and Yu David Liu. Proactive and adaptive energy-aware programming with mixed typechecking. *SIGPLAN Not.*, 52(6):217–232, jun 2017. ISSN 0362-1340. doi: 10.1145/3140587.3062356. URL https://doi.org/10.1145/3140587.3062356.

[19] Anthony Canino and Yu David Liu. Toward a language design for energy prediction. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, Programming '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362573. doi: 10.1145/3328433.3328445. URL https://doi.org/10.1145/3328433.3328445.

[20] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, page 225–236, USA, 2012. IEEE Computer Society. ISBN 9781450316422.

[21] Maria Carpen-Amarie, Patrick Marlier, Pascal Felber, and Gaël Thomas. A performance study of java garbage collectors on multicore architectures. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, page 20–29, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334044. doi: 10.1145/2712386.2712404. URL https://doi.org/10.1145/2712386.2712404.

[22] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Wolf. Energy savings through compression in embedded java environments. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, page 163–168, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135424. doi: 10.1145/774789.774823. URL https://doi.org/10.1145/774789.774823.

[23] G. Chen, Mahmut T. Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Mario Wolczko. Adaptive garbage collection for battery-operated environments. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, page 1–12, USA, 2002. USENIX Association. ISBN 1931971013.

[24] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection for reducing memory system energy in an embedded Java environment. *ACM Trans. Embed. Comput. Syst.*, 1(1): 27–55, nov 2002. ISSN 1539-9087. doi: 10.1145/581888.581892. URL https://doi.org/10.1145/581888.581892.

[25] G. Chen, G. Chen, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-aware code cache management for memory-constrained java devices. In *IEEE International [Systems-on-Chip] SOC Conference, 2003. Proceedings.*, pages 179–182, 2003. doi: 10.1109/SOC.2003.1241488.

[26] G. Chen, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, and M. Wolczko. Tracking object life cycle for leakage energy optimization. In *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*, pages 213–218, 2003. doi: 10.1109/CODESS.2003.1275286.

[27] Lei Chen, Jiacheng Zhao, Chenxi Wang, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, Guoqing Harry Xu, and Huimin Cui. Unified holistic memory management supporting multiple big data processing frameworks over hybrid memories. *ACM Trans. Comput. Syst.*, 39(1–4), jul 2022. ISSN 0734-2071. doi: 10.1145/3511211. URL https://doi.org/10.1145/3511211.

[28] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DianNao family: energy-efficient hardware accelerators for machine learning. *Commun. ACM*, 59(11):105–112, oct 2016. ISSN 0001-0782. doi: 10.1145/2996864. URL https://doi.org/10.1145/2996864.

[29] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13 (11):677–678, nov 1970. ISSN 0001-0782. doi: 10.1145/362790.362798. URL https://doi.org/10.1145/362790.362798.

[30] K. F. Chong, C. Y. Ho, and Anthony S. Fong. Pretenuring in java by object lifetime and reference density using scratch-pad memory. In *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*, pages 205–212, 2007. doi: 10.1109/PDP.2007.67.

[31] Cliff Click, Gil Tene, and Michael Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, page 46–56, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930477. doi: 10.1145/1064979.1064988. URL https://doi.org/10.1145/1064979.1064988.

[32] Jacob Cohen. The earth is round (p < .05). *American Psychologist*, 49:997–1003, 1994. URL https://api.semanticscholar.org/CorpusID:380942.

[33] Gilberto Contreras and Margaret Martonosi. Techniques for real-system characterization of java virtual machine energy and power behavior. In *2006 IEEE International Symposium on Workload Characterization*, pages 29–38, 2006. doi: 10.1109/IISWC.2006.302727.

[34] Gilberto Contreras, Margaret Martonosi, Jinzhang Peng, Guei-Yuan Lueh, and Roy Ju. The xtrem power and performance simulator for the intel xscale core: Design and experiences. *ACM Trans. Embed. Comput. Syst.*, 6(1):4–es, feb 2007. ISSN 1539-9087. doi: 10.1145/1210268.1210272. URL https://doi.org/10.1145/1210268.1210272.

[35] Intel Corporation. *Intel DBPXA255 Development Platform for the Intel Personal Internet Client Architecture,*, 2003.

[36] Oracle Corporation. *OpenJDK*, 2015. URL https://openjdk.org/.

[37] Oracle Corporation. *Java*, 2023. URL https://www.oracle.com/java/.

[38] Noric Couderc, Emma Söderberg, and Christoph Reichenbach. JBrainy: Micro-benchmarking Java collections with interference. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 42–45, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371094. doi: 10.1145/3375555.3383760. URL https://doi.org/10.1145/3375555.3383760.

[39] Common Weakness Enumeration (CWE™). *Common Weakness Enumeration: CWE Top 25 Most Dangerous Software Weaknesses*, 2021. URL https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[40] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, page 455–470, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343053. doi: 10.1145/2989081.2989088. URL https://doi.org/10.1145/2989081.2989088.

[41] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526, sep 1976. ISSN 0001-0782. doi: 10.1145/360336.360345. URL https://doi.org/10.1145/360336.360345.

[42] J. L. Devore and Kenneth N. Berk. Modern mathematical statistics with applications. *Springer Texts in Statistics*, 2021. URL https://api.semanticscholar.org/CorpusID:118767523.

[43] Kerstin Eder, John P. Gallagher, Pedro López-García, Henk Muller, Zorana Banković, Kyriakos Georgiou, Rémy Haemmerlé, Manuel V. Hermenegildo, Bishoksan Kafle, Steve Kerrison, Maja Kirkeby, Maximiliano Klemen, Xueliang Li, Umer Liqat, Jeremy Morse, Morten Rhiger, and Mads Rosendahl. Entra: Whole-systems energy transparency. *Microprocessors and Microsystems*, 47: 278–286, 2016. ISSN 0141-9331. doi: https://doi.org/10.1016/j.micpro.2016.07.003. URL https://www.sciencedirect.com/science/article/pii/S0141933116300862.

[44] Kamal A. El-Dahshan, Eman K. Elsayed, and Naglaa E. Ghannam. Comparative study for detecting mobile application's anti-patterns. In *Proceedings of the 8th International Conference on Software and Information Engineering*, ICSIE '19, page 1–8, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450361057. doi: 10.1145/3328833.3328834. URL https://doi.org/10.1145/3328833.3328834.

[45] Sebastian Ertel, Christof Fetzer, and Pascal Felber. Ohua: Implicit dataflow programming for concurrent systems. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, page 51–64, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450337120. doi: 10.1145/2807426.2807431. URL https://doi.org/10.1145/2807426.2807431.

[46] Ben Evans. What tens of millions of VMs reveal about the state of java, March 2020. URL https://thenewstack.io/what-tens-of-millions-of-vms-reveal-about-the-state-of-java/.

[47] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, nov 1969. ISSN 0001-0782. doi: 10.1145/363269.363280. URL https://doi.org/10.1145/363269.363280.

[48] R. A. Fisher. *Statistical Methods for Research Workers*. Springer New York, New York, NY, 1992. ISBN 978-1-4612-4380-9. doi: 10.1007/978-1-4612-4380-9_6. URL https://doi.org/10.1007/978-1-4612-4380-9_6.

[49] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341356. doi: 10.1145/2972206.2972210. URL https://doi.org/10.1145/2972206.2972210.

[50] Daniele Folegnani and Antonio González. Energy-effective issue logic. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 230–239, 2001.

[51] Charlotte Freitag, Mike Berners-Lee, Kelly Widdicks, Bran Knowles, Gordon S. Blair, and Adrian Friday. The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations. *Patterns*, 2(9):100340, 2021. ISSN 2666-3899. doi: https://doi.org/10.1016/j.patter.2021.100340. URL https://www.sciencedirect.com/science/article/pii/S2666389921001884.

[52] Qi Gao, Wenbin Zhang, Yan Tang, and Feng Qin. First-aid: surviving and preventing memory management bugs during production runs. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 159–172, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584829. doi: 10.1145/1519065.1519083. URL https://doi.org/10.1145/1519065.1519083.

[53] David Gay, Rob Ennals, and Eric Brewer. Safe manual memory management. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, page 2–14, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938930. doi: 10.1145/1296907.1296911. URL https://doi.org/10.1145/1296907.1296911.

[54] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B. Kahveci, Ali Gürbüz, Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, Mehmet Doğan, Mohamed Mandouh, Marios Fragkoulis, and Asterios Katsifodimos. Hazelcast jet: Low-latency stream processing at the 99.99th percentile. *Proc. VLDB Endow.*, 14(12):3110–3121, jul 2021. ISSN 2150-8097. doi: 10.14778/3476311.3476387. URL https://doi.org/10.14778/3476311.3476387.

[55] Stefanos Georgiou, Maria Kechagia, and Diomidis Spinellis. Analyzing programming languages' energy consumption: An empirical study. In *Proceedings of the 21st Pan-Hellenic Conference on Informatics*, PCI '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450353557. doi: 10.1145/3139367.3139418. URL https://doi.org/10.1145/3139367.3139418.

[56] Stefanos Georgiou, Maria Kechagia, Panos Louridas, and Diomidis Spinellis. What are your programming language's energy-delay implications? In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 303–313, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357166. doi: 10.1145/3196398.3196414. URL https://doi.org/10.1145/3196398.3196414.

[57] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 229–240, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318709. doi: 10.1145/2451116.2451142. URL https://doi.org/10.1145/2451116.2451142.

[58] T. Gilmont, J.-D. Legat, and J.-J. Quisquater. Enhancing security in the memory management unit. In *Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium*, volume 1, pages 449–456 vol.1, 1999. doi: 10.1109/EURMIC.1999.794507.

[59] Paul Griffin, Witawas Srisa-an, and J. Morris Chang. An energy efficient garbage collector for Java embedded devices. *SIGPLAN Not.*, 40(7):230–238, jun 2005. ISSN 0362-1340. doi: 10.1145/1070891.1065943. URL https://doi.org/10.1145/1070891.1065943.

[60] Paul Griffin, Witawas Srisa-An, and J. Morris Chang. On designing a low-power garbage collector for java embedded devices: A case study. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, page 868–873, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139640. doi: 10.1145/1066677.1066875. URL https://doi.org/10.1145/1066677.1066875.

[61] Frank E Grubbs. Procedures for detecting outlying observations in samples. *Technometrics*, 11(1):1–21, 1969.

[62] Yao Guo, Pritish Narayanan, Mahmoud Abdullah Bennaser, Saurabh Chheda, and Csaba Andras Moritz. Energy-efficient hardware data prefetching. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(2):250–263, 2011. doi: 10.1109/TVLSI.2009.2032916.

[63] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of Java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 225–236, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884869. URL https://doi.org/10.1145/2884781.2884869.

[64] R. Henriksson. Scheduling garbage collection in embedded systems. In *PhD thesis*. Lund Institute of Technology, 1998.

[65] Matthew Hertz and Emery D Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 313–326, 2005.

[66] Shiwen Hu and Lizy K. John. Impact of virtual execution environments on processor energy consumption and hardware adaptation. In *Proceedings of the*

2nd International Conference on Virtual Execution Environments, VEE '06, page 100–110, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933328. doi: 10.1145/1134760.1134775. URL https://doi.org/10.1145/1134760.1134775.

[67] Claire Huang, Stephen Blackburn, and Zixian Cai. Improving garbage collection observability with performance tracing. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2023, page 85–99, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703805. doi: 10.1145/3617651.3622986. URL https://doi.org/10.1145/3617651.3622986.

[68] Xianglong Huang, Stephen M Blackburn, Kathryn S McKinley, J Eliot B Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. *ACM SIGPLAN Notices*, 39(10):69–80, 2004.

[69] Lorenz Huelsbergen and Phil Winterbottom. Very concurrent mark-and-sweep garbage collection without fine-grain synchronization. *SIGPLAN Not.*, 34(3):166–175, oct 1998. ISSN 0362-1340. doi: 10.1145/301589.286878. URL https://doi.org/10.1145/301589.286878.

[70] Ahmed Hussein, Antony L. Hosking, Mathias Payer, and Christopher A. Vick. Don't race the memory bus: Taming the gc leadfoot. *SIGPLAN Not.*, 50(11): 15–27, jun 2015. ISSN 0362-1340. doi: 10.1145/2887746.2754182. URL https://doi.org/10.1145/2887746.2754182.

[71] Ahmed Hussein, Mathias Payer, Antony Hosking, and Christopher A. Vick. Impact of gc design on power and performance for android. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336079. doi: 10.1145/2757667.2757674. URL https://doi.org/10.1145/2757667.2757674.

[72] Intel. *Intel Architecture Software Developer's manual*, volume Volume 3: System Programming Guide. Intel®, 2009.

[73] Jaeyoung Jang, Jun Heo, Yejin Lee, Jaeyeon Won, Seonghak Kim, Sung Jun Jung, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. Charon: Specialized near-memory processing architecture for clearing dead objects in memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 726–739, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358297. URL https://doi.org/10.1145/3352460.3358297.

[74] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. A specialized architecture for object serialization with applications to big data analytics. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 322–334, 2020. doi: 10.1109/ISCA45697.2020.00036.

[75] Alexandra Jimborean, Jonatan Waern, Per Ekemark, Stefanos Kaxiras, and Alberto Ros. Automatic detection of extended data-race-free regions. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 14–26, 2017. doi: 10.1109/CGO.2017.7863725.

[76] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2023.

[77] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE, 2017.

[78] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. RAPL in action: Experiences in using RAPL for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), mar 2018. ISSN 2376-3639. doi: 10.1145/3177754. URL https://doi.org/10.1145/3177754.

[79] Channoh Kim, Jaehyeok Kim, Sungmin Kim, Dooyoung Kim, Namho Kim, Gitae Na, Young H. Oh, Hyeon Gyu Cho, and Jae W. Lee. Typed architectures: Architectural support for lightweight scripting. *SIGARCH Comput. Archit. News*, 45(1):77–90, apr 2017. ISSN 0163-5964. doi: 10.1145/3093337.3037726. URL https://doi.org/10.1145/3093337.3037726.

[80] Sangmi Kim, Sanjiv Tomar, Vijaykrishnan Narayanan, M. Kandemir, and M.J. Irwin. Energy-efficient Java execution using local memory and object co-location. *Computers and Digital Techniques, IEE Proceedings* -, 151:33 – 42, 02 2004. doi: 10.1049/ip-cdt:20040186.

[81] Marisa Kirisame, Pranav Shenoy, and Pavel Panchekha. Optimal heap limits for reducing browser memory use. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022. doi: 10.1145/3563323. URL https://doi.org/10.1145/3563323.

[82] Haoyu Li, Mingyu Wu, Binyu Zang, and Haibo Chen. Scissorgc: Scalable and efficient compaction for Java full garbage collection. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, page 108–121, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450360203. doi: 10.1145/3313808.3313820. URL https://doi.org/10.1145/3313808.3313820.

[83] Tao Li and Lizy Kurian John. Run-time modeling and estimation of operating system power consumption. *SIGMETRICS Perform. Eval. Rev.*, 31(1):160–171, jun 2003. ISSN 0163-5999. doi: 10.1145/885651.781048. URL https://doi.org/10.1145/885651.781048.

[84] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, jun 1983. ISSN 0001-0782. doi: 10.1145/358141.358147. URL https://doi.org/10.1145/358141.358147.

[85] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[86] Kenan Liu, Khaled Mahmoud, Joonhwan Yoo, and Yu David Liu. Vincent: Green hot methods in the JVM. *Science of Computer Programming*, 230:102962, 2023. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2023.102962. URL https://www.sciencedirect.com/science/article/pii/S0167642323000448.

[87] Xutong Ma, Jiwei Yan, Wei Wang, Jun Yan, Jian Zhang, and Zongyan Qiu. Detecting memory-related bugs by tracking heap memory management of

C++ smart pointers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 880–891, 2021. doi: 10.1109/ASE51524.2021.9678836.

[88] Martin Maas, Krste Asanović, and John Kubiatowicz. A hardware accelerator for tracing garbage collection. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 138–151, 2018. doi: 10.1109/ISCA.2018.00022.

[89] Irene Manotas, Lori Pollock, and James Clause. Seeds: a software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 503–514, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568297. URL https://doi.org/10.1145/2568225.2568297.

[90] Douglas C Montgomery. *Design and analysis of experiments*. John wiley & sons, 2017.

[91] Rodrigo Morales, Ruben Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. Anti-patterns and the energy efficiency of Android applications. *arXiv preprint arXiv:1610.05711*, 2016.

[92] Gaku Nakagawa and Shuichi Oikawa. An architecture of operating system utilizing non-volatile main memory and heterogeneous multi-core. In *2013 IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS)*, pages 559–663, 2013. doi: 10.1109/ICIS.2013.6607900.

[93] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Scratch-pad memory allocation without compiler support for Java applications. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '07, page 85–94, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938268. doi: 10.1145/1289881.1289899. URL https://doi.org/10.1145/1289881.1289899.

[94] Wellington Oliveira, Renato Oliveira, Fernando Castor, Benito Fernandes, and Gustavo Pinto. Recommending energy-efficient java collections. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 160–170, 2019. doi: 10.1109/MSR.2019.00033.

[95] Wellington Oliveira, Bernardo Moraes, Fernando Castor, and João Paulo Fernandes. Analyzing the resource usage overhead of mobile app development frameworks. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, page 152–161, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700446. doi: 10.1145/3593434.3593487. URL https://doi.org/10.1145/3593434.3593487.

[96] Matthew Benjamin Olson, Joseph T. Teague, Divyani Rao, Michael R. JANTZ, Kshitij A. Doshi, and Prasad A. Kulkarni. Cross-layer memory management to improve DRAM energy efficiency. *ACM Trans. Archit. Code Optim.*, 15(2), may 2018. ISSN 1544-3566. doi: 10.1145/3196886. URL https://doi.org/10.1145/3196886.

[97] Oracle. *Timeline of key Java milestones*, 2020. URL https://www.oracle.com/java/moved-by-java/timeline/.

[98] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent gc for servers. *SIGPLAN Not.*, 37(5):129–140, may 2002. ISSN 0362-1340. doi: 10.1145/543552.512546. URL https://doi.org/10.1145/543552.512546.

[99] Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, and Joël Penhoat. Evaluating the impact of java virtual machines on energy consumption. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450386654. doi: 10.1145/3475716.3475774. URL https://doi.org/10.1145/3475716.3475774.

[100] Chen Pan, Mimi Xie, Chengmo Yang, Zili Shao, and Jingtong Hu. Nonvolatile main memory aware garbage collection in high-level language virtual machine. In *2015 International Conference on Embedded Software (EMSOFT)*, pages 197–206, 2015. doi: 10.1109/EMSOFT.2015.7318275.

[101] David A Patterson and John L Hennessy. *Computer organization and Design*. Morgan Kaufmann„ 1994.

[102] Somnath Paul, Robert Karam, Swarup Bhunia, and Ruchir Puri. Energy-efficient hardware acceleration through computing in the memory. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2014. doi: 10.7873/DATE.2014.279.

[103] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. The influence of the java collection framework on overall energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software*, GREENS '16, page 15–21, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341615. doi: 10.1145/2896967.2896968. URL https://doi.org/10.1145/2896967.2896968.

[104] Rui Pereira, Pedro Simão, Jácome Cunha, and João Saraiva. jstanley: placing a green thumb on Java collections. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 856–859, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. doi: 10.1145/3238147.3240473. URL https://doi.org/10.1145/3238147.3240473.

[105] Michael Peters, Gian Luca Scoccia, and Ivano Malavolta. How does migrating to Kotlin impact the run-time efficiency of Android apps? In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 36–46, 2021. doi: 10.1109/SCAM52516.2021.00014.

[106] Gustavo Pinto, Fernando Castor, and Yu David Liu. Understanding energy behaviors of thread management constructs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, page 345–360, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325851. doi: 10.1145/2660193.2660235. URL https://doi.org/10.1145/2660193.2660235.

[107] Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of

multiple datapath resources. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 90–101. IEEE, 2001.

[108] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, et al. Renaissance: Benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–47, 2019.

[109] Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, and Mikel Luján. Type information elimination from objects on architectures with tagged pointers support. *IEEE Transactions on Computers*, 67(1):130–143, 2018. doi: 10.1109/TC.2017.2709739.

[110] Alberto Ros and Stefanos Kaxiras. Racer: Tso consistency via race detection. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016. doi: 10.1109/MICRO.2016.7783736.

[111] Rui Rua, Marco Couto, and João Saraiva. Greensource: A large-scale collection of Android code, tests and energy metrics. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 176–180, 2019. doi: 10.1109/MSR.2019.00035.

[112] Narendran Sachindran and J Eliot B Moss. Mark-copy: Fast copying gc with less space overhead. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 326–343, 2003.

[113] Narendran Sachindran and J Eliot B Moss. Mark-copy: Fast copying gc with less space overhead. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 326–343, 2003.

[114] Patrick M Sansom. Combining single-space and two-space compacting garbage collectors. In *Functional Programming, Glasgow 1991: Proceedings of the 1991 Glasgow Workshop on Functional Programming, Portree, Isle of Skye, 12–14 August 1991*, pages 312–323. Springer, 1992.

[115] Kunal Sareen and Stephen Michael Blackburn. Better understanding the costs and benefits of automatic memory management. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, MPLR '22, page 29–44, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450396967. doi: 10.1145/3546918.3546926. URL `https://doi.org/10.1145/3546918.3546926`.

[116] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. Cooperative cache scrubbing. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 15–26, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328098. doi: 10.1145/2628071.2628083. URL `https://doi.org/10.1145/2628071.2628083`.

[117] Elias T. Silva, Daniel Barcelos, Flávio R. Wagner, and Carlos E. Pereira. A virtual platform for multiprocessor real-time embedded systems. In *Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '08, page 31–37, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605583372. doi:

10.1145/1434790.1434796. URL https://doi.org/10.1145/1434790.1434796.

[118] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. Cache coherence for GPU architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 578–590, 2013. doi: 10.1109/HPCA.2013.6522351.

[119] Darko Stefanović, Kathryn S McKinley, and J Eliot B Moss. Age-based garbage collection. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 370–381, 1999.

[120] Yu Sun and Wei Zhang. Efficient code caching to improve performance and energy consumption for Java applications. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, page 119–126, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605584690. doi: 10.1145/1450095.1450115. URL https://doi.org/10.1145/1450095.1450115.

[121] Sun Microsystems. J2ME Building Blocks for Mobile Devices: White Paper on KVM and the Connected, Limited Device Configuration (CLDC). PDF, 2000. Retrieved 2024-04-30.

[122] Jie Tang and Chen Liu. An energy and memory trade-off study on resource constrained embedded jvm. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 448–452, 2014. doi: 10.1109/ICPPW.2014.65.

[123] Jie Tang, Shaoshan Liu, Zhimin Gu, Xiao-Feng Li, and Jean-Luc Gaudiot. Hardware-assisted middleware: Acceleration of garbage collection operations. In *ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 281–284, 2010. doi: 10.1109/ASAP.2010.5541011.

[124] Jie Tang, Shaoshan Liu, Zhimin Gu, Xiao-Feng Li, and Jean-Luc Gaudiot. Achieving middleware execution efficiency: Hardware-assisted garbage collection operations. *J. Supercomput.*, 59(3):1101–1119, mar 2012. ISSN 0920-8542. doi: 10.1007/s11227-010-0493-0. URL https://doi.org/10.1007/s11227-010-0493-0.

[125] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: the continuously concurrent compacting collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, page 79–88, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450302630. doi: 10.1145/1993478.1993491. URL https://doi.org/10.1145/1993478.1993491.

[126] Marko Topolnik. *Performance of Modern Java on Data-Heavy Workloads: The Low-Latency Rematch*, June 2020. URL https://jet-start.sh/blog/2020/06/23/jdk-gc-benchmarks-rematch.

[127] Marko Topolnik. *Standard Performance Evaluation Corporation*, January 2021. URL https://www.spec.org/jbb2015/.

[128] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM Sigplan notices*, 19(5):157–167, 1984.

[129] Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet, Russell Tessier, Deepak Unnikrishnan, and Kris Gaj. Memory security management for reconfigurable embedded systems. In *2008 International Conference on Field-Programmable Technology*, pages 153–160, 2008. doi: 10.1109/FPT.2008.4762378.

[130] Jose M. Velasco, David Atienza, Katzalin Olcoz, Francky Catthoor, Francisco Tirado, and Mendias. Energy characterization of garbage collectors for dynamic applications on embedded systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 3728 LNCS, pages 69–78, 2005. ISBN 9783540290131.

[131] Jose Manuel Velasco, David Atienza, and Katzalin Olcoz. Exploration of memory hierarchy configurations for efficient garbage collection on high-performance embedded systems. In *Proceedings of the 19th ACM Great Lakes Symposium on VLSI*, GLSVLSI '09, page 3–8, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585222. doi: 10.1145/1531542.1531549. URL https://doi.org/10.1145/1531542.1531549.

[132] Jose Manuel Velasco, David Atienza, and Katzalin Olcoz. Memory power optimization of Java-based embedded systems exploiting garbage collection information. *Journal of Systems Architecture*, 58(2):61–72, 2012. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2011.11.002. URL https://www.sciencedirect.com/science/article/pii/S1383762111001251.

[133] David Vengerov. Modeling, analysis and throughput optimization of a generational garbage collector. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, page 1–9, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583471. doi: 10.1145/1542431.1542433. URL https://doi.org/10.1145/1542431.1542433.

[134] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam, and M. J. Irwin. Energy behavior of java applications from the memory perspective. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, page 23, USA, 2001. USENIX Association.

[135] VMware, Inc. Best practices for enterprise java applications running on vmware. Technical white paper, VMware, Inc., 2020. Available at: https://www.vmware.com [Accessed 11 April 2024].

[136] Sophie Vos, Patricia Lago, Roberto Verdecchia, and Ilja Heitlager. Architectural tactics to optimize software for energy efficiency in the public cloud. In *2022 International Conference on ICT for Sustainability (ICT4S)*, pages 77–87, 2022. doi: 10.1109/ICT4S55073.2022.00019.

[137] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 347–362, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314650. URL

https://doi.org/10.1145/3314221.3314650.

[138] Genevieve Warren, Maoni Stephens, Sébastien Ros, GitHubPang, Andrew Au, and Peter Sollich. *Runtime configuration options for garbage collection*, feb 2023. URL https://learn.microsoft.com/en-us/dotnet/core/runtime-config/garbage-collector#conserve-memory.

[139] Bernard L Welch. The significance of the difference between two means when the population variances are unequal. *Biometrika*, 29(3/4):350–362, 1938.

[140] David R. White, Jeremy Singer, Jonathan M. Aitken, and Richard E. Jones. Control theory for principled heap sizing. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, page 27–38, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450321006. doi: 10.1145/2464157.2466481. URL https://doi.org/10.1145/2464157.2466481.

[141] Mahendra Yadav, Gaurav Raj, Harishchandra Akarte, and Dharmendra Yadav. Horizontal scaling for containerized application using hybrid approach. *Ingénierie des systèmes d information*, 25:709–718, 12 2020. doi: 10.18280/isi.250601.

[142] Albert Mingkun Yang and Tobias Wrigstad. Deep dive into ZGC: A modern garbage collector in OpenJDK. *ACM Trans. Program. Lang. Syst.*, 44(4), sep 2022. ISSN 0164-0925. doi: 10.1145/3538532. URL https://doi.org/10.1145/3538532.

[143] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. Improving program locality in the GC using hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 301–313, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385977. URL https://doi.org/10.1145/3385412.3385977.

[144] Sun Yu and Wei Zhang. Adaptive drowsy cache control for java applications. In *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, volume 1, pages 185–191, 2008. doi: 10.1109/EUC.2008.88.

[145] Karen K Yuen. The two-sample trimmed t for unequal population variances. *Biometrika*, 61(1):165–170, 1974.

[146] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. Low-latency, high-throughput garbage collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 76–91, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523440. URL https://doi.org/10.1145/3519939.3523440.

[147] Haitao Steve Zhu, Chaoren Lin, and Yu David Liu. A programming model for sustainable software. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 767–777, 2015. doi: 10.1109/ICSE.2015.89.

[148] Benjamin Zorn. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 87–98, New York, NY, USA, 1990. Association for Computing Machinery. ISBN 089791368X. doi: 10.1145/91556.91597. URL https://doi.org/10.1145/91556.91597.

[149] Benjamin Zorn. The measured cost of conservative garbage collection. *Software: Practice and Experience*, 23(7):733–756, 1993.

[150] Erik Österlund. ZGC: Automatic heap sizing. `https://bugs.openjdk.org/browse/JDK-8329758`, 2024. Accessed: April 2024.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 2412

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-527713

ACTA UNIVERSITATIS
UPSALIENSIS
2024