How Much Does Garbage Collection Cost? A Study on the Effects of Garbage Collection

Kunal Sareen

A thesis submitted for the degree of
Bachelor's of Advanced Computing (Honours)
(Research & Development)
The Australian National University

November 2021

© Kunal Sareen 2021

Except where otherwise indicated, this thesis is my own original work.

Kunal Sareen
18 November 2021

To my family and friends.

Acknowledgments

First and foremost, I would like to thank my supervisor Professor Steve Blackburn for guiding me through my journey. Thank you for being an inspiration and imparting your wisdom and insights to me. You have taught me how to be a good researcher for which I cannot thank you enough. Thank you for giving me this opportunity to work with you over the past year.

Over the course of completing my degree I have had the chance to interact and work with many great people. I've had the pleasure of working with Shoaib Akram, Ranald Clouston, Adrian Herrera, Peter Höfner, Tony Hosking, Charles Martin, Michael Norrish, Peter Strazdins, Ben Titzer, and Alwen Tiu. Thank you for providing valuable insight and opportunities to increase my understanding of computer science. I am truly fortunate to have been able to meet and work with you all. I would also like to thank the MMTk research group: Javad Amiri, Zixian Cai, Yi Lin, Kunshang Wang, and Wenyu Zhao, who made this thesis possible with their work on the MMTk codebase.

I would like to thank all my friends who added some zest to my life. I would like to especially thank Riley Baile, Niko Bakker, Zak Brighton-Knight, Zixian Cai, Aditya Chilukuri, Ben Gray, Fergus Rogers, Erik Still, Shiva Shah, James Taylor, V Vijendran, and Allie Zhou. Thank you for listening to my late-night rants and for your help and support throughout my university life.

Finally I would like to thank my family: my parents, Yogesh and Saroj Sareen, without whom I would not be able to achieve what I have today; my sister, Shivangi Sareen, who has been a pillar of support throughout my life; and our family dog, Snoopy, who was the best dog one could ask for.

Abstract

A key decision all modern programming language designers face is the choice of how to tackle dynamic memory management. Broadly, the two options are *manual memory management* and *automatic memory management* a.k.a. *garbage collection*. Both options provide their own sets of benefits and come with their own sets of costs. Manual memory management provides finer-grain control over how memory is allocated and freed, however it is error-prone with bugs like use-after-free dominating the Common Vulnerabilities and Exposures (CVE) database. On the other hand, while garbage collection greatly simplifies memory management, it is well-known that it has performance implications for a language. However there have only been a handful of comprehensive studies on its overheads and benefits in comparison to manual memory management.

What makes this comparison difficult is that it is quite hard to construct experiments to make a *fair* comparison. For example, it is not possible to insert state-of-the-art collectors into languages which were designed with manual memory management in mind since an application may encode raw pointers by XOR-ing them (a common example being an XOR linked list) and hence if we move objects around, we risk breaking the semantics of the application. On the other hand, applications written in a managed language can not easily be translated to manually memory managed applications as it is hard to determine where a certain object is no longer required and can be explicitly freed, or more importantly, where a programmer would insert a free. This is compounded by the fact that the programming idioms and paradigms of both styles of languages are quite different and so even direct translations between the two (if possible) may not be a fair comparison since the overall programming style may be biased to one form of memory management over the other. It is also easier to write code in a managed language since we don't have to reason about memory management. This is not an easy to quantify advantage which further muddies the water.

The key contributions of this thesis are the new methodologies and experiments we designed in order to understand and test certain aspects and effects of garbage collection. Most notably we investigate: (i) the space-overheads of garbage collection; (ii) the effects of garbage collection on the execution of the mutator; (iii) the effects of garbage collection on mutator locality; and finally (iv) the effects of inserting garbage collection-like behaviour to a manually memory managed application. We conduct this study in a modern setting using modern CPU microarchitectures allowing us to shed new light onto garbage collection overheads in a modern context given the rapid advancements in recent microarchitectures.

Our results show that common garbage collector algorithms such as SemiSpace and an Appel-style Generational Copying collector have space-overheads of around

1.75× an approximation of manual memory management, while Immix has a space-overhead of 1.15× on average. We measure locality effects of garbage collection using a transaction-based benchmark as a case-study and find a weak correlation between the execution time of a transaction and its proximity to the execution of a GC. Finally, we insert garbage collection-like behaviour to manually memory managed applications and find that the space- and time-overheads vary widely across different allocator architectures, having modest space-overheads but significant time-overheads (around 20% time-overheads on average, reaching $< 3\times$ overheads at a maximum) for the best allocator configurations.

This work deepens the understanding of the overheads and benefits of garbage collection in comparison to manual memory management allowing for language designers and implementers to make an informed decision regarding their choice of dynamic memory management. This work also enhances knowledge regarding the the locality benefits of garbage collectors. Finally, this work lays down a framework for systematically examining the effects of different garbage collectors on an application's behaviour.

Contents

Acknowledgments **vii**

Abstract **ix**

1 Introduction **1**
- 1.1 Thesis Statement . . . 3
- 1.2 Contributions . . . 3
- 1.3 Thesis Structure . . . 4

2 Background and Related Work **5**
- 2.1 Terminology and Taxonomy of Garbage Collection . . . 5
- 2.2 MMTk . . . 6
- 2.3 Related work . . . 7

3 Space Overheads of Garbage Collection **9**
- 3.1 Objectives . . . 9
- 3.2 Approximating Manual Memory Management . . . 9
- 3.3 Design and Implementation . . . 10
  - 3.3.1 Stress Garbage Collection . . . 10
  - 3.3.2 Configurable `malloc` Mark-Sweep . . . 11
- 3.4 Experimental Methodology . . . 12
  - 3.4.1 Benchmarks . . . 12
  - 3.4.2 Hardware and Operating System . . . 13
  - 3.4.3 MMTk and OpenJDK . . . 13
  - 3.4.4 Experimental Design . . . 14
- 3.5 Results and Evaluation . . . 14
- 3.6 Summary . . . 15

4 Time Overheads of Garbage Collection **17**
- 4.1 Objectives . . . 17
- 4.2 Garbage Collection “Signals” . . . 17
- 4.3 Design and Implementation . . . 19
- 4.4 Experimental Methodology . . . 20
  - 4.4.1 Benchmarks . . . 20
  - 4.4.2 Hardware and Operating System . . . 20
  - 4.4.3 MMTk and OpenJDK . . . 20
  - 4.4.4 Experimental Design . . . 20

4.5 Results and Evaluation 21
4.5.1 Discussion 24
4.6 Summary 24

5 Locality Effects of Garbage Collection 25
5.1 Objectives 25
5.2 Measuring Locality Effects 25
5.3 Design and Implementation 26
5.3.1 Instrumenting `lusearch` 26
5.3.2 Instrumenting MMTk 27
5.4 Experimental Methodology 27
5.4.1 Hardware and Operating System 27
5.4.2 MMTk and OpenJDK 27
5.4.3 Experimental Design 28
5.5 Results and Evaluation 29
5.5.1 Discussion 32
5.6 Summary 33

6 Garbage Collection Behaviour in an Unmanaged Context 35
6.1 Objectives 35
6.2 Approximating Garbage Collection Behaviour 35
6.3 Design and Implementation 36
6.4 Experimental Methodology 38
6.4.1 Benchmarks 38
6.4.2 Hardware and Operating System 39
6.4.3 Experimental Design 39
6.5 Results and Evaluation 39
6.5.1 mimalloc 40
6.5.2 jemalloc 44
6.5.3 hoard 46
6.5.4 ptmalloc2 47
6.5.5 Discussion 49
6.6 Summary 49

7 Conclusion 51
7.1 Future Work 51
7.1.1 Performance Evaluation on Different Microarchitectures 51
7.1.2 Time Overheads of Garbage Collection 52
7.1.3 Locality Effects of Garbage Collection 52
7.1.4 Garbage Collection Behaviour in an Unmanaged Context 52

A Figures 1

List of Figures

3.1 Minimum heap size required to complete a benchmark for the SemiSpace, Generational Copying, and Immix collectors in comparison to the 64 KB and 128 KB Mark-Sweep stress collectors. A lower value is better. 14

4.1 Noisy signal obtained due to the addition of the Gaussian noise to a clean signal. Note how the original signal can still be made out in the resultant additive signal. Image obtained from James Trichilo. 18

4.2 Schematic showcasing signal transmission over a noisy medium using differential signaling. Note the increased amplitude at the Receiver's end. Image obtained from Wikipedia [2021]. 19

4.3 Mutator execution time (normalized to best value) averaged over 30 runs for a GC limit of 1 using the mimalloc Mark-Sweep collector with three different stress factor values. A lower value is better. 21

4.4 Mutator execution time (normalized to best value) averaged over 30 runs for a GC limit of 16 using the mimalloc Mark-Sweep collector with three different stress factor values. A lower value is better. 22

4.5 Mutator execution time (normalized to best value) averaged over 30 runs for the mimalloc Mark-Sweep collector with a stress factor of 16MB and three different GC limit values. A lower value is better. . . . 23

5.1 Density of closeness to a GC for Query 0380 using the Immix collector on the Haswell system. Note that executions where the query was interrupted by a GC have been removed. The red line in 5.1a and dashed blue lines in 5.1b and 5.1c are the median for the entire dataset. The red lines in 5.1b and 5.1c are the median for the best and worst 5th percentile of executions with respect to execution time. 28

5.2 Density of closeness to a GC for Query 0380 using the SemiSpace collector on the Haswell system. Note that executions where the query was interrupted by a GC have been removed. The red line in 5.2a and dashed blue lines in 5.2b and 5.2c are the median for the entire dataset. The red lines in 5.2b and 5.2c are the median for the best and worst 5th percentile of executions with respect to execution time. 30

5.3 Density of closeness to a GC for Query 0380 using the Mark-Sweep collector on the Haswell system. Note that executions where the query was interrupted by a GC have been removed. The red line in 5.3a and dashed blue lines in 5.3b and 5.3c are the median for the entire dataset. The red lines in 5.3b and 5.3c are the median for the best and worst 5th percentile of executions with respect to execution time. 31

6.1 mimalloc average execution time over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `mi` configuration. A lower value is better. Note that the larson benchmark has a fixed execution time and hence we use the "relative execution time" as reported by the benchmark. 40

6.2 mimalloc average maximum RSS over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `mi` configuration. A lower value is better. Note the xmalloc-test results should be ignored since the benchmark allocates more the faster it runs. Hence, we have removed them from this (and all future) graph(s) in order to not affect other calculations. 41

6.3 mimalloc average execution time over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `mi` configuration. A lower value is better. 42

6.4 mimalloc average maximum RSS over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `mi` configuration. A lower value is better. 43

6.5 jemalloc average execution time over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `je` configuration. A lower value is better. 44

6.6 jemalloc average maximum RSS over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `je` configuration. A lower value is better. 45

6.7 hoard average execution time over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `hoard` configuration. A lower value is better. 46

6.8 hoard average maximum RSS over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `hoard` configuration. A lower value is better. 47

6.9 glibc average execution time over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `glibc` configuration. A lower value is better. 48

6.10 glibc average maximum RSS over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `glibc` configuration. A lower value is better. 49

A.1 Mutator execution time (normalized to best value) averaged over 30 runs for a GC limit of 4 using the mimalloc Mark-Sweep collector with three different stress factor values. A lower value is better. 1
A.2 Mutator execution time (normalized to best value) averaged over 30 runs for the mimalloc Mark-Sweep collector with a stress factor of 32MB and three different GC limit values. A lower value is better. . . . 2
A.3 Mutator execution time (normalized to best value) averaged over 30 runs for the mimalloc Mark-Sweep collector with a stress factor of 64MB and three different GC limit values. A lower value is better. . . . 2
A.4 Density of closeness to a GC for Query 0667 using the Immix collector on the Haswell system. 3
A.5 Density of closeness to a GC for Query 0667 using the SemiSpace collector on the Haswell system. 4
A.6 Density of closeness to a GC for Query 0667 using the Mark-Sweep collector on the Haswell system. 5
A.7 Density of closeness to a GC for Query 1009 using the Immix collector on the Haswell system. 6
A.8 Density of closeness to a GC for Query 1009 using the SemiSpace collector on the Haswell system. 7
A.9 Density of closeness to a GC for Query 1009 using the Mark-Sweep collector on the Haswell system. 8
A.10 Density of closeness to a GC for Query 0380 using the Immix collector on the Xeon system. 9
A.11 Density of closeness to a GC for Query 0380 using the SemiSpace collector on the Xeon system. 10
A.12 Density of closeness to a GC for Query 0380 using the Mark-Sweep collector on the Xeon system. 11
A.13 Density of closeness to a GC for Query 0667 using the Immix collector on the Xeon system. 12
A.14 Density of closeness to a GC for Query 0667 using the SemiSpace collector on the Xeon system. 13
A.15 Density of closeness to a GC for Query 0667 using the Mark-Sweep collector on the Xeon system. 14
A.16 Density of closeness to a GC for Query 1009 using the Immix collector on the Xeon system. 15
A.17 Density of closeness to a GC for Query 1009 using the SemiSpace collector on the Xeon system. 16
A.18 Density of closeness to a GC for Query 1009 using the Mark-Sweep collector on the Xeon system. 17
A.19 jemalloc average execution time over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `je` configuration. A lower value is better. 18

A.20 jemalloc average maximum RSS over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `je` configuration. A lower value is better. 19
A.21 hoard average execution time over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `hoard` configuration. A lower value is better. 19
A.22 hoard average maximum RSS over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `hoard` configuration. A lower value is better. 20
A.23 glibc average execution time over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `glibc` configuration. A lower value is better. 20
A.24 glibc average maximum RSS over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `glibc` configuration. A lower value is better. 21

List of Tables

1.1 The 2021 Common Weakness Enumeration Top 5 Most Dangerous Software Weaknesses. Data obtained from CWE [2021]. 2

3.1 Minimum heap size (in MB) measured for different collectors. Note that certain benchmarks did not complete for some collectors. Here, "mi-MS (64KB)" and "mi-MS (128KB)" refer to the mimalloc Mark-Sweep 64KB and 128KB stress collectors respectively. 13

5.1 Geometric mean of normalized medians for the best and worst 5th percentile of executions (per query) over all 2048 queries per collector. Note how most of the best queries for Immix are farther away from a GC. 29

5.2 Geometric mean of normalized medians for the best and worst 5th percentile of executions (per query) over all 2048 queries per collector. Note how most of the best queries for Immix and SemiSpace are farther away from a GC, whereas for Mark-Sweep they are much closer to the execution of a GC. 32

Chapter 1

Introduction

Programming languages have been a foundational building block of computer science since the advent of the field. Modern research focusses on improving both the usability of programming languages (generally in terms of high-level language features such as object-orientation, abstract data types, etc.) as well as their performance. However, it is often the case that improving the usability of a programming language negatively affects the performance. For example, a key decision that a language implementer has to make while creating a new language is how to handle dynamic memory allocations. The two broad approaches are *manual memory management* (like with C, C++, etc.) and *automatic memory management* (like with Java, Python, JavaScript etc.) a.k.a. *garbage collection*. It is well-known that the addition of garbage collection to a language affects the execution time of an application since the application will spend part of its time collecting garbage. Fundamentally, garbage collection is a space-time trade-off, in other words, the larger the heap we provide to an application, the fewer times it has to collect garbage.

Garbage collection allows a language to provide an abstraction over memory instead of exposing raw pointers to application programmers. An abstraction over memory is quite useful, both cognitively (i.e. a programmer does not have to reason about pointers and pointer arithmetic) as well as functionally since all objects are only accessible by a reference which can allow the garbage collector to move objects around in memory to reduce the heap footprint. Garbage collection also provides a key benefit to programmers, namely memory safety. Manual memory management is notoriously error prone. Bugs such as use-after-frees, out-of-bounds reads and writes are commonplace in languages such as C and C++. Table 1.1 lists the top 5 most dangerous software weaknesses of 2021 as measured by the CWE Team [2021]. Note how both out-of-bounds reads and writes are within the top 3. Such bugs are generally impossible in a garbage collected language since all memory is managed by the language.

There has long been a debate in the world of programming languages regarding the overheads of garbage collection. To some, the costs and overheads imposed by garbage collection are unacceptable, while others believe the benefits of garbage collection far outweigh the costs. However, we note that literature comparing the costs and benefits of modern garbage collection against manual memory management is scarce. This has lead to the spread of several misconceptions and misunderstandings

Table 1.1: The 2021 Common Weakness Enumeration Top 5 Most Dangerous Software Weaknesses. Data obtained from CWE [2021].

Rank	ID	Name	Count
1	CWE-787	Out-of-bounds Write	3033
2	CWE-79	Cross-site Scripting	3564
3	CWE-125	Out-of-bounds Read	1448
4	CWE-20	Improper Input Validation	1120
5	CWE-78	OS Command Injection	833

surrounding the true costs and benefits of garbage collection. The effects of garbage collection on the locality of an application are also poorly understood. Huang et al. [2004] suggest that garbage collection can in fact improve an application's locality, resulting in better performance, whereas in direct contrast, Hertz and Berger [2005] suggest that garbage collection adversely affects an application's locality, degrading its performance.

What makes the comparison difficult between the two styles of memory management is that it is quite hard to construct experiments to make a *fair* comparison. More concretely, it is impossible to insert a state-of-the-art garbage collector into a language designed with manual memory management in mind as an application may encode raw pointers, for example, by XOR-ing them (a common use-case being an XOR linked list). This means that we cannot move objects around in memory since we may break the semantics of the application. In addition, any garbage collector in a language without type-accurate information (as is the case with most manually memory managed languages) has to be *conservative* when it scans for heap references [Boehm and Weiser, 1988]. What this means is that the collector assumes any sequence of bytes that *looks like* a pointer to be a pointer. This can include a (large) number that just happens to be a valid address. Hence this leads to an overestimation of the number of objects that are actually alive.

On the other hand, in applications written in a managed language, it is difficult to ascertain where a certain object is not required anymore and should be freed, or more precisely where a programmer will insert a call to `free`. In addition to the above, each style of language has its own set of programming paradigms and idioms that are quite different from each other. Hence direct translations between the two, if possible, may not be a fair comparison since the code may be biased to one form of memory management over the other.

These poorly understood overheads and benefits can lead to language implementers committing to one form of memory management (or in extreme cases, one algorithm) without the ability to switch to another. Famously, Swift cites the Hertz and Berger [2005] paper in a talk [Lattner, 2016] as a motivation for using naïve reference counting [Collins, 1960] – an automatic memory management algorithm which has long been dismissed by the research community [Jones et al., 2011, Chapter 5; Jibaja et al., 2011, Shahriyar et al., 2012; Wang, 2017]. Another example is PHP which

embeds reference counting semantics into its specification [PHP Community Foundation, 2019, Section 4] [1]. This has lead to the PHP community largely suffering due to the poor performance of the reference counting algorithm [Jibaja et al., 2011] as well as being unable to move to a different garbage collection algorithm.

1.1 Thesis Statement

Given the poor understanding of the effects of garbage collection, we naturally pose questions regarding the true overheads and benefits of garbage collection in comparison to manual memory management: Does a garbage collector actually provide locality benefits to an application? What space- and time-overheads do common garbage collection algorithms have? and so-on. This thesis thus tries to demystify and deepen our understanding of the overheads and benefits of garbage collection in comparison to manual memory management and clear up misconceptions surrounding the costs of garbage collection.

1.2 Contributions

We focus on using the Memory Management Toolkit [Blackburn et al., 2004a; MMTk Research Group, 2021] with Hotspot, the Java virtual machine by OpenJDK [OpenJDK Community, 2021]. Our key contributions and insights from this work are:

1. We introduce a new methodology for measuring different aspects of garbage collectors, namely:
   (a) the space-overheads of garbage collection;
   (b) the effects of garbage collection on the execution of an application;
   (c) the effects of garbage collection on the locality of an application;
   (d) the effects of inserting garbage collection-like behaviour in manual memory managed applications;
2. We deepen the understanding of the overheads and benefits of garbage collection in comparison to manual memory management allowing for language developers and implementers to make an informed decision regarding their choice of memory management;
3. We hopefully clear up some misconceptions in the literature regarding the locality benefits of different garbage collectors;
4. We conduct our study in a contemporary setting using modern hardware (and more importantly modern CPU microarchitectures) allowing us to reevaluate

[1] Note that the specification states that a reference counting algorithm is not necessary, however a lot of applications depend on precise reference counting and deterministic destruction [Yamauchi, 2012; Wang, 2017] which makes it hard to move away from a precise reference counting algorithm.

overheads and benefits of garbage collection in the face of rapid CPU and microarchitecural advancements.

1.3 Thesis Structure

Over the course of our research, we devised multiple experiments that tested different aspects of an application's performance. Since each of these experiments are mostly self-contained, we discuss the design, methodologies, results, and evaluations in their respective chapters. Chapter 2 provides relevant background information and discusses prior studies on the overheads and benefits of garbage collection. Chapter 3 details our first experiment which dealt with understanding the space-overheads of garbage collection. Chapter 4 discusses experiments conducted to understand the effects of garbage collection on an application's execution. Chapter 5 explores the effects of garbage collection on an application's locality. Chapter 6 tackles the thesis statement from a different direction by approximating garbage collection behaviour in manual memory management. Finally, Chapter 7 summarizes our results and contributions as well as discusses future avenues of research.

Chapter 2

Background and Related Work

Automatic memory management or *garbage collection* is a key aspect of modern managed languages. First introduced as a method to simplify memory management in LISP by McCarthy [1960], it has been widely adopted by other languages since then. In this chapter we introduce background knowledge key to understanding our thesis. We describe common terminology in garbage collection literature and provide a brief breakdown of the taxonomy of different garbage collection algorithms in Section 2.1. In Section 2.2, we briefly introduce the framework most of our work is implemented in. Finally, in Section 2.3, we discuss prior work that is directly relevant to our thesis.

2.1 Terminology and Taxonomy of Garbage Collection

In garbage collection terminology, an application in a managed language can be divided into two parts: (i) the *mutator*; and (ii) the *collector*. The mutator executes the application code which allocates new objects and *mutates* the object graph[2] by (re)assigning references. The collector (or *garbage collector*) executes garbage collection code which deallocates objects which are no longer used or needed.

An interesting observation in the behaviour of garbage collected applications is that it is periodic. That is to say, over the course of the execution of an application, there are periods of bulk allocations followed by periods of bulk deallocations. We term such behaviour as the *inhale-exhale behaviour* of garbage collection.

Garbage collection has two broad strategies to identify objects to reclaim: (i) *tracing garbage collection*; and (ii) *reference counting*. A tracing garbage collector performs a transitive closure of the object graph, keeping track of any reachable objects. Tracing garbage collectors work under the assumption that any object that is reachable can be used in the future by the programmer and should be kept alive. More precisely, they approximate the *liveness* of an object with its *reachability*. We note this implicitly marks unreachable objects as dead, i.e. ready for reclamation. A reference counting garbage collector, on the other hand, does not perform a transitive closure of the object graph, instead operating directly on the object references. In such a scheme, each object has an associated *reference count* that specifies how many objects currently

[2] If we consider objects allocated in the heap, we find that they form what is termed an *object graph*, where each node is an object and each edge is a reference.

own a reference to it. An object is considered dead if and only if its reference count is zero. Hence, tracing garbage collectors work on live objects and reference counting collectors work on dead objects.

Tracing garbage collection can generally be divided further into different strategies. *Mark-Sweep* is one of the simplest tracing garbage collectors. It involves two phases: (i) the *mark*-phase, where reachable objects are marked with a mark bit denoting that an object should not be reclaimed; and (ii) the *sweep*-phase, wherein the collector traverses the heap and finds and frees objects that do not have the mark bit set. Such a collector is non-moving, that is to say, it suffers from heap fragmentation. Multiple strategies have been invented to combat heap fragmentation. *Copying* collectors such as the *Semi-Space* collector [Cheney, 1970] employ one such strategy. The Semi-Space collector partitions the heap into two regions (termed *semispaces*) and switches between the two every collection cycle. After tracing, live objects are *evacuated* from one space to the other, leaving all dead objects behind. There is no need to explicitly free dead objects as the collector will just allocate over them. Note that the collector has to rewrite references to objects after moving them in order to preserve program correctness. Another common strategy to deal with fragmentation is *Compacting*. Such collectors generally compact the heap in-place instead of moving objects to a new space as with copying collectors. A common compacting collector is the *Mark-Compact* collector. Here, reachable objects are marked live as with the Mark-Sweep collector, but instead of the sweep-phase, it compacts the heap by relocating live objects and rewriting references.

The Mark-Sweep collector works on the smallest region-granularity possible, i.e. object-granularity, while the Semi-Space collector works on a rather large region-granularity (i.e. the semispaces). Midway between the two are what are termed *Mark-Region* collectors. Mark-Region collectors partition the heap into regions, similar to Semi-Space, however they keep the size of regions small. This allows for the collector to support contiguous allocation inside these small regions, while only collecting objects at a fixed granularity. Immix [Blackburn and McKinley, 2008] is common example of a Mark-Region collector.

A key concept in garbage collection literature is the *(weak) generational hypothesis* [Lieberman and Hewitt, 1983; Ungar, 1984]. The hypothesis states that most objects die young, i.e. in other words, most objects are short-lived. This key insight has lead to the development of *generational* garbage collectors which partition the heap into a young *nursery* space and an old *mature* space. Most high-performance garbage collectors use such a scheme nowadays.

2.2 MMTk

Often garbage collector implementations are deeply entrenched in their respective runtimes (famous examples being the Ruby [2021] and C# garbage collectors [2021] which are massive monolithic files that implement their respective GCs). Hence, breakthroughs and novel GC implementations in one language are usually very hard

to translate into other languages.

The Memory Management Toolkit (MMTk) [2021] is a language-agnostic library that provides language implementers with a high-performance framework for memory management. It is a re-write of the original MMTk [Blackburn et al., 2004a,b] which was written in Java as a part of the JikesRVM Java virtual machine. This new version is written in Rust after a successful pilot study using Rust for a high-performance GC implementation [Lin et al., 2016]. MMTk currently maintains support for the OpenJDK (Java), V8 (JavaScript), and JikesRVM (Java) runtimes. A core objective of MMTk is to allow for researchers and language implementers to quickly design and test garbage collector (GC) implementations. MMTk facilitates this due to its novel approach of breaking GCs down to different key components which allows for a GC algorithm to be defined by composing different components together. This *composability* is a key driving factor which allows MMTk to be an excellent tool for garbage collection researchers.

As of writing this thesis, all garbage collectors in the MMTk re-write are *stop-the-world* (STW) collectors, that is to say, they pause the mutator in order to let the garbage collector collect garbage. This obviously induces larger pause times on the mutator. The alternative is *concurrent* garbage collection wherein the collector and mutator runs at the same time. We note that there is work being done to support concurrent garbage collectors into MMTk.

2.3 Related work

Huang et al. [2004] describe an approach to exploit data locality using copying garbage collection. Standard copying collectors generally only use a single static order (often simple breadth-first or depth-first ordering). Such an order is not necessarily optimal for all applications. Huang et al. introduce *online object reordering* (OOR) which uses copying orders that are based on application traversal patterns. They achieve this by piggybacking off of the runtime method sampling in the adaptive JIT compiler in JikesRVM, a research Java virtual machine. The JIT compiler uses (timer-driven) sampling to identify hot methods in order to recompile them at higher optimization tier. OOR analysis collects information regarding field accesses during compilation. When the JIT compiler recompiles hot methods, OOR marks hot field accesses using information from before. Later, during garbage collection, the hot fields are traversed (and copied) first. The authors conduct an experiment comparing the execution time of this OOR collector against an idealized manual memory management approximation. They use a Mark-Sweep collector with a free list allocator and only measure the mutator time (which is calculated as the difference of total execution time and GC time). This is an imperfect approximation, as the mutator time does not include time to free objects, as well as the recycling of allocation slots is not as prompt as in manual memory management. They find that the OOR collector is generally comparable to if not better than the idealized manual memory management even at small heap sizes. We note however, such an implementation

is non-trivial as well as requires support from the target VM so it is not necessarily applicable to all languages and collectors. We also note that modern freelist allocator designs have improved significantly, in terms of both space- and time-efficiency. Hence the Mark-Sweep mutator results may not be indicative of the current state of manual memory management.

Perhaps the most relevant prior work to our thesis is by Hertz and Berger [2005] who develop a framework comparing the costs of garbage collection to manual memory management. Using Dynamic SimpleScalar (an architectural simulator) and JikesRVM, they create an "oracular memory manager" for Java which uses `malloc` and `free` to allocate and deallocate objects. The framework directly inserts calls to `free` as soon as objects are considered dead. They describe a *reachability-based* oracle, which frees objects just before they become unreachable; and a *liveness-based* oracle, which uses object lifetimes to free objects just after their last use. These oracles require two preconditions to work: (i) fixed object allocation order; and (ii) a list ordered by allocation time that indicates which objects should be freed at that time. Hence, when an application allocates an object in this framework, the simulator first checks if any objects need to be freed. If some objects need to be freed then it calls `free` on them explicitly before it passes control onto `malloc` to actually service the allocation request.

Using the lea allocator [Doug Lea, 1998] and a modified version of the freelist allocator[3] with support for explicit freeing (named *MSExplicit*), they compare the space- and time-overheads for various garbage collectors in the JikesRVM MMTk. They find that garbage collection in MMTk requires at least 5× more memory (than the explicit memory management scheme using the lea allocator) in order to provide the same execution time performance. For space-overheads, they find the best garbage collector in MMTk at that time GenMS (an Appel-style generational collector with a Mark-Sweep mature space) requires at least 2–2.5× the heap size of the explicit memory manager using the lea allocator. They also find that the MSExplicit allocator generally has a 60% space-overhead in comparison to the lea allocator.

We believe there are certain aspects the authors overlook or forget to account for. The authors use heap footprints in terms of total pages allocated as a measure of space-overheads instead of heap usage numbers reported by MMTk. We note that MMTk did not have support to unmap allocated pages at that time.[4] Hence, the space-overhead results reported for all collectors (and even the MSExplicit allocator) are not indicative of actual space-overheads since MMTk does not unmap allocated pages. On the other hand, as mentioned previously, there have been significant improvements to freelist allocator architectures and hence the above results may not be indicative of modern manual memory management.

[3]The freelist allocator implemented in MMTk at that time was based directly on the lea allocator.
[4]Confirmed by MMTk authors.

Chapter 3

Space Overheads of Garbage Collection

It is well known that garbage collection is fundamentally a space-time tradeoff. However this space-time tradeoff is generally only measured between different garbage collectors and not between garbage collection and manual memory management. In this chapter we describe our methodology, design and implementation, and results for an experiment that aims to understand the *space-overheads* of classic garbage collection algorithms against manual memory management.

3.1 Objectives

As discussed in Chapter 2, Hertz and Berger [2005] find that certain garbage collection algorithms require a heap size 2–2.5× larger than if the heap was manually managed. We explore the space-overheads of garbage collection by creating an experiment to measure the space-overheads of classic garbage collection algorithms against an approximation of manual memory management.

3.2 Approximating Manual Memory Management

The core insight we use to calculate space-overheads is simple: A manually memory managed application will free objects whenever they are considered dead by the programmer. Hence, in order to emulate an ideal manually memory managed application in a managed language such as Java, we can simply perform a GC every allocation. Note how we've inserted the manual memory management-like behaviour here, i.e. we free an object as soon as possible. This gives us the *perfect baseline*. That is to say, we can never do better (in terms of reducing the heap size) than performing a GC every allocation. Note that programmers cannot achieve the same in manual memory management since calls to `free` are generally placed as they see fit.

However, some benchmarks allocate in the order of magnitude of *gigabytes*. It is infeasible to run experiments where we perform a full heap GC at every allocation

as they may take *months* to complete[5]. Hence, we approximate this perfect baseline by decreasing the periodicity of GCs, or more precisely, by increasing the allowed number of bytes allocated between subsequent GCs.

We choose to model our perfect baseline on the Mark-Sweep algorithm (described in Chapter 2) for this experiment. Mark-Sweep best approximates the manual memory management behaviour of `malloc-free` with its freelist allocator. Since our experiment performs a GC every allocation (or every X bytes of allocation), we can also exploit the fact that we can immediately reuse freelist slots (if the new object allocation fits) further improving the accuracy of our approximation of manual memory management. A Mark-Sweep garbage collector faces similar issues regarding fragmentation as `malloc-free` due to its non-moving characteristic.

We pick the mimalloc [Leijen et al., 2019] allocator as the freelist allocator of choice for our Mark-Sweep implementation. Mimalloc is a relatively new freelist allocator that is designed for use-cases including highly parallel and concurrent applications as well as predictable performance overheads[6]. Paige Reeves [2021] compares different freelist implementations – such as jemalloc [Evans, 2006], hoard [Berger et al., 2000], mimalloc, and glibc 2.27 – for the DaCapo Java benchmarks [Blackburn et al., 2006a,b] and finds that mimalloc consistently outperforms the other allocators in terms of the mutator performance.

We can then calculate the minimum heap required to run the benchmarks for classic GC algorithms such as Semispace, Generational copying, Immix etc. and compare against our perfect baseline to give us a better idea regarding the space-overheads of these algorithms.

3.3 Design and Implementation

There are two key aspects of the design: (i) a mechanism to allow for changing the periodicity of a GC (based on number of allocated bytes) a.k.a. *Stress GC*; and (ii) internally using a `malloc` library as a freelist allocator. We break down our design and implementation for the two in this section.

3.3.1 Stress Garbage Collection

Generally when an allocation request comes to MMTk, it can either take the *fast path* or the *slow path*. The fast path is the usual case and is heavily optimized in order to reduce the time taken to allocate an object. In a bump-pointer allocator, for example, the fast path simply checks if there is enough (local) space to allocate the object. If there is, then it will allocate it, otherwise it will go to the slow path. The slow path is the exceptional case and we are less concerned about its performance since it is executed infrequently. It is responsible for various things such as allocating more local space for an allocator, or checking if we have exhausted the heap and a GC is

[5]And indeed, it did take more than a month for one benchmark to finish with a GC every allocation.
[6]Mimalloc is intended to be used as a backend for reference counting.

required, etc. Hence, if we trick the allocation to always go through the slow path, then we can forcefully perform a GC every X bytes of allocation.

This is simple to accomplish in MMTk. We can maintain a global atomic variable denoting the number of bytes allocated since the last GC. This variable is updated with the number of bytes allocated every allocation. We can then check this variable every allocation and force a GC if we have allocated more than X bytes, resetting the variable to 0 after a GC. A key aspect of this design is that it can be achieved with a *run time* flag allowing us to control the periodicity of a GC at run time instead of recompiling every time we want to change the periodicity.

3.3.2 Configurable `malloc` Mark-Sweep

Paige Reeves [2021] implemented a Mark-Sweep implementation with a configurable `malloc` freelist allocator in MMTk. We first briefly describe her work and then describe extensions we made to this implementation for the purposes of our work.

The `malloc` freelist allocator in MMTk (as implemented by Reeves) allows one to choose between the glibc, Hoard [Berger et al., 2000], jemalloc [Evans, 2006], and mimalloc [Leijen et al., 2019] `malloc` implementations. This allocator is selected at compile time using compiler flags. Since MMTk can't control where the `malloc` library allocates, in order to keep track of allocated objects, a side-metadata bitmap is used where each bit represents whether an object starting at that address has been allocated or not. This is termed as the *alloc-bit*. This alloc-bit is set when an object is initialized by MMTk. The alloc-bits are used in the sweep-phase to perform a linear scan of all the allocated objects. As with a normal Mark-Sweep implementation, any unmarked (i.e. dead) objects are deallocated in the sweep-phase. However, `free` is used to deallocate objects here.

In the above implementation, in order to determine the current heap footprint of an application, MMTk simply keeps track of the sum of the sizes of all allocated objects, rounding it to the nearest OS page-aligned size. This is an approximation, however, as the actual heap footprint could be larger owing to fragmentation. We extend the original implementation by adding a side-metadata *bytemap* where each byte represents if an OS page starting at that address is *active* or not, with active defined as containing a live object. This is termed as the *active page-byte*. We chose to use a byte over a bit due to synchronization overheads in the case where multiple threads are trying to set different bits in the same metadata byte. Having each page be represented as a byte allows us to not worry about access races. The active page-bytes are set when an object is allocated (and in the case where the object size is larger than a page, we set the bytes for all the pages the object occupies). We then unset them in the sweep-phase if an object has been deallocated. The sum of these bytes then gives us the number pages that are active which we can then use to determine the current heap footprint.

Note that in the current `malloc` Mark-Sweep implementation all allocations directly go through the slow path since there is no internal local freelist we can allocate from in a fast path. Hence, no changes were required to the above implementation

in order to support Stress GC. Also note that all allocations regardless of size go through the `malloc` allocator (as opposed to having another space and allocator for large objects). This allows us to more accurately measure the heap footprint.

3.4 Experimental Methodology

We use the following experimental methodology.

3.4.1 Benchmarks

We use the DaCapo benchmarks [Blackburn et al., 2006a,b] (Chopin evaluation version[7]) for our benchmarks. The DaCapo benchmark suite is a suite of real-world open source Java applications each with different levels of parallelism, data access patterns, and memory requirements etc. The Chopin update to the DaCapo benchmark suite adds new benchmarks and updates previous ones with more intensive workloads. We give a brief outline of each of the benchmarks used:

- **avrora** A cycle-accurate simulator for embedded sensing programs.
- **batik** A toolkit for dealing with SVG images.
- **biojava** A set of tools to process biological data.
- **cassandra**† A NoSQL database management system.
- **eclipse** An integrated development environment (IDE).
- **fop** An output-independent print formatter.
- **graphchi** A large-scale graph computing engine.
- **jme** A 3D game engine in Java.
- **jython** A python interpreter written in Java.
- **luindex** A text indexing tool.
- **lusearch** A text search tool.
- **pmd** A source code analyzer for Java.
- **sunflow** A rendering system for photo-realistic image synthesis.
- **tomcat** An HTTP web server environment.
- **xalan** An XSLT processor for transforming XML documents.
- **zxing** A barcode image processing library.

The benchmark suite is packaged as a jar file. The jar file contains just the benchmarks. The testing data is located in a separate folder. The source code and the benchmark harness are available on the DaCapo Chopin GitHub repository [Steve Blackburn, 2021]. Note that benchmarks labelled with † crashed or did not terminate in our experiments.

[7]Obtained with permission from authors. Git hash used: `69a704e` [2021].

Table 3.1: Minimum heap size (in MB) measured for different collectors. Note that certain benchmarks did not complete for some collectors. Here, "mi-MS (64KB)" and "mi-MS (128KB)" refer to the mimalloc Mark-Sweep 64KB and 128KB stress collectors respectively.

benchmark	mi-MS (64KB)	mi-MS (128KB)	SemiSpace	GenCopy	Immix
avrora	7	7	13	14	9
batik	280	280	487	495	331
biojava	-	188	377	382	191
eclipse	224	224	374	380	254
fop	21	22	41	41	29
graphchi	176	177	256	256	256
jython	57	57	105	106	84
luindex	27	27	53	-	28
lusearch	25	26	36	37	24
pmd	173	178	313	-	173
sunflow	19	20	29	29	20
tomcat	-	59	81	98	65
xalan	11	13	18	19	11
zxing	100	102	125	124	99

3.4.2 Hardware and Operating System

We ran our experiments on the following hardware platforms:

(i) Intel i7-4770 Haswell with a 3.4 GHz clock, a 4 x 32 KB, 64 B/line, 8-way L1 cache, a 4 x 256 KB, 64 B/line, 8-way L2 cache, and 16 GB DDR3 RAM;

(ii) AMD FX-8320 Piledriver with a 3.5 GHz clock, a 8 x 16 KB, 64 B/line, 4-way L1 cache, a 4 x 2 MB, 64B/line, 16-way L2 cache, and 8GB DDR3 RAM;

(iii) 4 (sockets) x Intel Xeon Gold 5118 Skylake with a 2.3 GHz clock, a 12 x 32 KB, 64 B/line, 8-way L1 cache, a 12 x 1 MB, 64 B/line, 16-way L2 cache, and a 512 GB DDR4 RAM.

All systems ran Ubuntu 18.04.6 LTS with Linux 5.4.0-87 kernels. All CPUs operate in 64-bit mode and use 64-bit kernels.

3.4.3 MMTk and OpenJDK

We use the Rust rewrite of MMTk (see Chapter 2) with `mmtk-core` revision `42328e24`, mmtk-openjdk revision `55cc6f7`, and a modified version of OpenJDK 11 (revision `425d41085f1`) which has support for MMTk. All experiments use the HotSpot C2 JIT compiler with pre-compilation enabled and explicit GCs disabled. Measurements were captured using the DaCapo benchmark harness and probes. Since our measurements are not time-sensitive, we do not perform any warmup iterations.

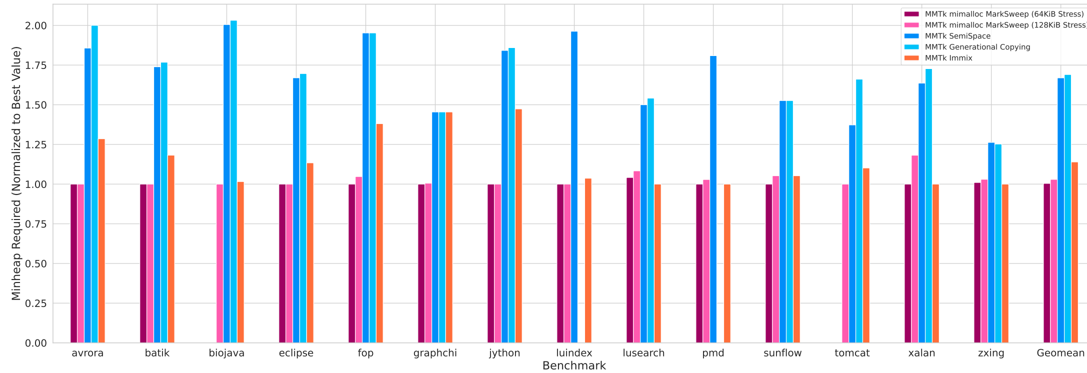


Figure 3.1: Minimum heap size required to complete a benchmark for the SemiSpace, Generational Copying, and Immix collectors in comparison to the 64 KB and 128 KB Mark-Sweep stress collectors. A lower value is better.

3.4.4 Experimental Design

We measure the minimum heap size required to run the benchmarks for collectors such as SemiSpace, Appel-style Generational Copying, and Immix and compare them with the minimum heap footprint as reported by the approximation of manual memory management as discussed above. The Immix collector was configured with opportunistic defragmentation enabled in order to compact the heap when possible. We chose relatively low stress factors (64KB and 128KB) which allow us to get a reasonable confidence in their accuracy. We conduct our experiment in a standalone setting where we run without any other significant resource-consuming processes in order to limit the experimental noise in our measurements. Since our experiment is not measuring time-sensitive components, we simply report the minimum heap size required by all our GCs across the different hardware. For the stress collectors, we use an arbitrarily large heap size since our highly periodic GCs will reduce the total heap footprint anyway.

3.5 Results and Evaluation

Figure 3.1 and Table 3.1 show the minimum heap required to run a benchmark for different GC algorithms in comparison to the 64KB and 128KB Mark-Sweep stress GCs. The results are normalized to the best value for that benchmark. Note that some benchmarks did not complete with certain GCs such as `luindex` for the Generational Copying collector. We simply leave the respective column blank in such cases.

Almost immediately we can see that the different GC algorithms are in the 1–2× range in terms of heap size required to complete most of the benchmarks in comparison to the Mark-Sweep stress collectors. Surprisingly, Immix seems to be competitive to the Mark-Sweep stress collectors in the minimum heap size required. SemiSpace and Generational Copying both require around 2× the Mark-Sweep stress collectors which is in-line with expectations due to their large space requirements. Note that in

certain benchmarks such as `lusearch`, we have the Immix collector beating out the Mark-Sweep stress collectors. We believe this suggests that we can reduce the heap size for `lusearch` further by increasing the periodicity of the Mark-Sweep stress collector. This could also partly be explained by heap fragmentation since `lusearch` is a highly parallel benchmark. Since we use the opportunistic defragmentation mode for Immix, we could theoretically make a highly fragmented heap smaller.

The geometric mean of the space-overheads per collector give us an idea of the average (space) cost for that collector. Notably, we see that both the SemiSpace and Generational Copying collectors have around a $1.75\times$ space-overhead in comparison to the Mark-Sweep stress collectors. Immix fairs much better, only having around a $1.15\times$ space-overhead. We believe this is due to Immix's mark-region design which helps in minimizing space in comparison to other copying collectors. The opportunistic defragmentation mechanism in Immix also contributes to its lower space-overheads.

We compare and contrast our results against the work by Hertz and Berger [2005] (see Section 2.3). They report at least a 2–2.5$\times$ space-overhead for the GenMS (an Appel-style generational collector with a Mark-Sweep mature space) in comparison to their explicit memory manager using the lea allocator [Doug Lea, 1998]. While we did not test the GenMS algorithm, we find that the SemiSpace and Generational Copying algorithms, which are arguably worse than GenMS in terms of space requirements, have a $1.75\times$ space-overhead on average. This apparent contradiction confirms our belief that they overestimate the space-overheads for garbage collectors in their work due to MMTk not unmapping allocated pages.

3.6 Summary

We outline a novel methodology for calculating the space-overheads of GC algorithms in comparison to manual memory management by modelling an approximation of manual memory management-like behaviour by increasing the periodicity of garbage collection cycles. We find and demonstrate through an experiment that common GC algorithms generally use around 1–2$\times$ the heap size in comparison to (an approximation of) manual memory management. We find that the Immix allocator is competitive with our manual memory management approximation, with SemiSpace and an Appel-style Generational Copying collector having a space-overhead of roughly $1.75\times$, which is in-line with expectations due to their large space requirements.

Chapter 4

Time Overheads of Garbage Collection

Due to the nature of garbage collection, there is an appreciable effect of the execution of the garbage collector on the execution of the mutator. This could be a positive effect such as collecting garbage so that the mutator can allocate more objects, or it could be negative such as the overheads imposed by garbage collection by stopping the mutator's execution or through the execution of barriers. In this chapter we describe our methodology, design and implementation, and results for an experiment that aims to tease away the *benefits of garbage collection from its cost* in order to understand the *effects of garbage collection on the mutator's execution*.

4.1 Objectives

Garbage collection imposes some form of overhead on the execution of a mutator. However, it can also potentially improve the execution time of a mutator by reorganizing objects or allowing for faster allocation, for example. Hence it is difficult to actually understand the true cost of garbage collection (on the mutator's execution) since the benefits might hide or even overshadow the true costs. We construct an experiment to tease away the costs and benefits of garbage collection and measure how the mutator's execution changes in hopes of understanding how the garbage collection "signal" affects it.

4.2 Garbage Collection "Signals"

Signal processing theory has a concept of *interference* wherein a signal we want to measure has some noise added to it resulting in a noisy signal. In order to measure or extract the signal we're interested in, we need to be able to remove or dampen the noise from the noisy signal somehow. We build upon this signal processing theory concept. We believe that the execution of an application can be considered a "signal" with occasional noise added to it due to the execution of the garbage collector. Hence, here, the signal we want to measure is the steady-state of the mutator and the added noise is the costs and benefits of garbage collection.

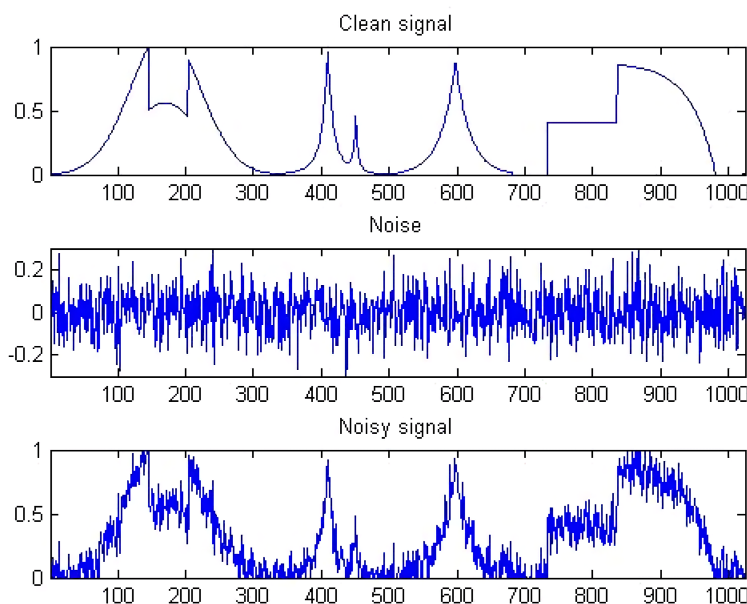


Figure 4.1: Noisy signal obtained due to the addition of the Gaussian noise to a clean signal. Note how the original signal can still be made out in the resultant additive signal. Image obtained from James Trichilo.

Figure 4.1 gives an example of a noisy signal in signal processing theory. Here, a clean signal (top) has some Gaussian noise (middle) added to it in order to get a noisy signal (bottom). While in signal processing, noise is usually removed through techniques such as using filters on Fourier Transforms, we do not have the liberty for such an approach here as there is no literal "signal" that we can measure. Hence, we need to construct a different experiment to be able to extract or dampen the "noise" due to garbage collection. We propose that we can extract this noise by separating the costs and benefits of garbage collection. By separating these costs and benefits we can create multiple "views" of the application's (noisy) signal which could help us understand the mutator's underlying steady-state "signal".

While not exactly what we are describing above, an analogous concept in signal processing is *differential signaling*. Figure 4.2 depicts a schematic showing how differential signaling can be used to transmit signals over a noisy medium. Note how the Sender sends two copies of the signal, with one being the negation of the true signal we want to send. Given the noisy medium, both signals have noise added to them resulting in the signals received on the Receiver's end. The Receiver can then subtract the two signals to remove the noise and get the original signal back. Note how the amplitude of the resultant signal is much larger at the Receiver. This is due to the subtraction of the two signals since one is the negation of the other. Due to its simplicity and effectiveness, differential signaling is widely used today in making twisted-pair cables such as Ethernet as well as on modern printed circuit boards.

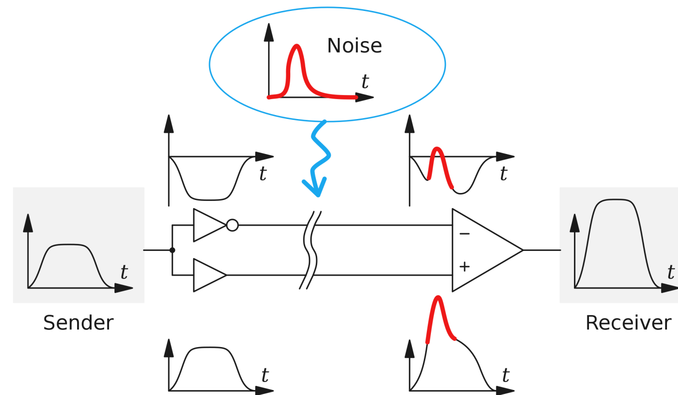


Figure 4.2: Schematic showcasing signal transmission over a noisy medium using differential signaling. Note the increased amplitude at the Receiver's end. Image obtained from Wikipedia [2021].

One method we can use to separate the costs and benefits of garbage collection is by *deferring* the freeing of objects (also termed *quarantining*)[8]. Hence, instead of freeing objects as soon as they are dead, we place them in a quarantine buffer and only free them after N GCs have occurred. Simultaneously, we increase the periodicity of a GC to once every X bytes (described in Section 3.2) in order to make the effects of the GC's "noise" on the mutator's "signal" more prominent. The combination of the two techniques above means that applications do not see the benefits of a GC immediately while still incurring the heavy cost of a GC (such as polluting the cache, tracing the entire object graph, etc.), effectively divorcing the costs from the benefits of a GC. Further, we can change the periodicity as well as the number of GCs between actual collections to get different views or pictures of how a GC (or the lack thereof) affects the mutator's execution time.

4.3 Design and Implementation

In principle, the technique described above is not limited to any GC algorithm, however we choose to focus on Mark-Sweep as it makes the implementation of deferred freeing easier than other collectors owing to its non-moving behaviour.

In order to change the periodicity of garbage collection, we use the same implementation as described in Section 3.3. On top of this, we allow a user to configure the periodicity of an actual GC as well. We implement support for quarantining using a simple global buffer where pointers to dead objects are placed. During the sweep-

[8]Deferred freeing in manually memory managed applications is explored in much more detail in Chapter 6.

phase of the Mark-Sweep collector, we identify dead objects as usual, however we place them inside this global buffer instead of freeing them immediately. Given we allow a user to configure the periodicity of an actual GC, at the end of the sweep, we check if we have crossed this threshold and if we have, then we free all objects in the quarantine buffer. While the simple global buffer require synchronization overheads between different GC threads, we also implemented quarantining using thread-local buffers, but did not observe any appreciable differences in the mutator performance in our (simple) tests. Hence, we stick with the global buffer implementation due to its simplicity.

4.4 Experimental Methodology

We use the following experimental methodology.

4.4.1 Benchmarks

We use the DaCapo benchmarks [Blackburn et al., 2006a,b] (Chopin evaluation version[9]) for our benchmarks. The list of benchmarks and a brief description of them can be found in Section 3.4.1.

4.4.2 Hardware and Operating System

We ran our experiments on the following hardware platform: Intel i7-6700K Skylake with a 4 GHz clock, a 4 x 32 KB, 64 B/line, 8-way L1 cache, a 4 x 256 KB, 64 B/line, 4-way L2 cache, and 16 GB DDR4 RAM. The system ran Ubuntu 18.04.6 LTS with a Linux 5.4.0-87 kernel. The CPU operates in 64-bit mode and uses 64-bit kernels.

4.4.3 MMTk and OpenJDK

We use the Rust rewrite of MMTk (see Chapter 2) with `mmtk-core` revision `37a229c9`, mmtk-openjdk revision `55cc6f7`, and a modified version of OpenJDK 11 (revision `425d41085f1`) which has support for MMTk. All experiments use the HotSpot C2 JIT compiler with pre-compilation enabled and explicit GCs disabled. Measurements were captured using the DaCapo benchmark harness and probes. Since we are measuring time-sensitive values, we perform two warmup iterations before starting our timing iteration in order to reduce any noise due to compilation or other VM operations.

4.4.4 Experimental Design

The key idea of this experiment is we want to measure the mutator's execution time to see if it is affected by varying the periodicity of "fake" garbage collections (i.e.

[9] Obtained with permission from authors. Git hash used: `69a704e` [2021].

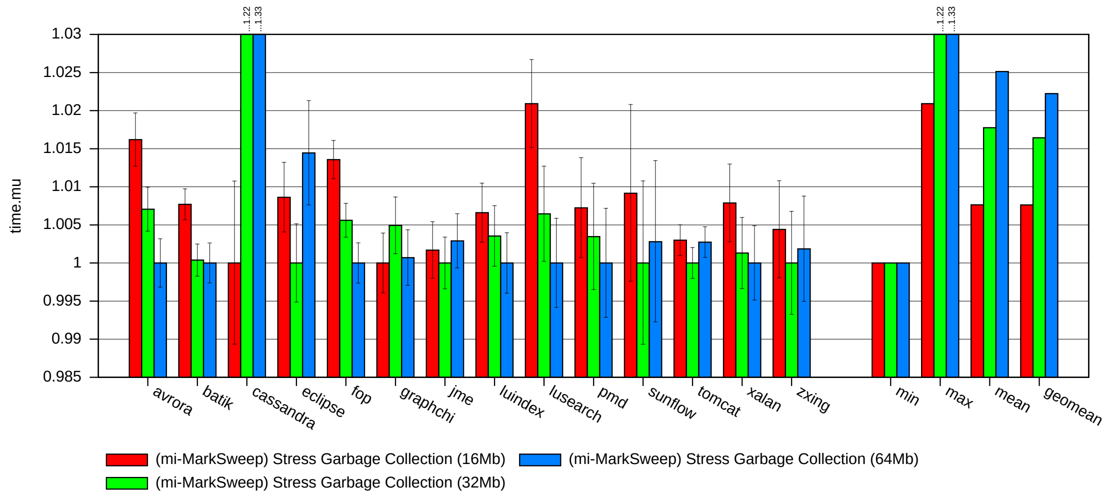


Figure 4.3: Mutator execution time (normalized to best value) averaged over 30 runs for a GC limit of 1 using the mimalloc Mark-Sweep collector with three different stress factor values. A lower value is better.

stress factor) as well as "real" garbage collections (i.e. *GC limit*). This can give us insight into how the underlying mutator's steady-state is affected by garbage collection operations. We use the above Mark-Sweep collector with the mimalloc freelist allocator as described in Section 3.3. We vary the stress factor in coarsely, with the values 16MB, 32MB, and 64MB. We generally note that these result in a $> 6\times$ increase in the number of GCs if ran with the minimum heap size as well as do not take too long to finish execution. On the other hand, we vary the GC limit as 1, 4, and 16 which means that we perform a "real" GC every 1, 4, and 16 "fake" GCs. We run each benchmark 30 times, averaging the mutator execution time over them in order to reduce the effect of noise in our measurements. We conduct our experiment in a standalone setting where we run without any other significant resource-consuming processes in order to limit the experimental noise in our measurements. We use an arbitrarily large heap size for the benchmarks since we are artificially controlling the heap size with our periodic GCs.

4.5 Results and Evaluation

We present our results through two views: per GC limit and per stress factor. We discuss the per GC limit results first. For the sake of conciseness and completeness, we place some figures in Appendix A.

Figure 4.3 depicts the average mutator execution time for the DaCapo benchmarks with a GC limit of 1. The idea of this experiment is to estimate the costs associated with our framework. We can immediately see that even though we have a lot of runs, there is still a lot of noise present in the experimental results. Note that most of the trends for the mutator execution times are statistically insignificant given they are within $< 1\%$ of the best configuration as well as are extremely noisy.

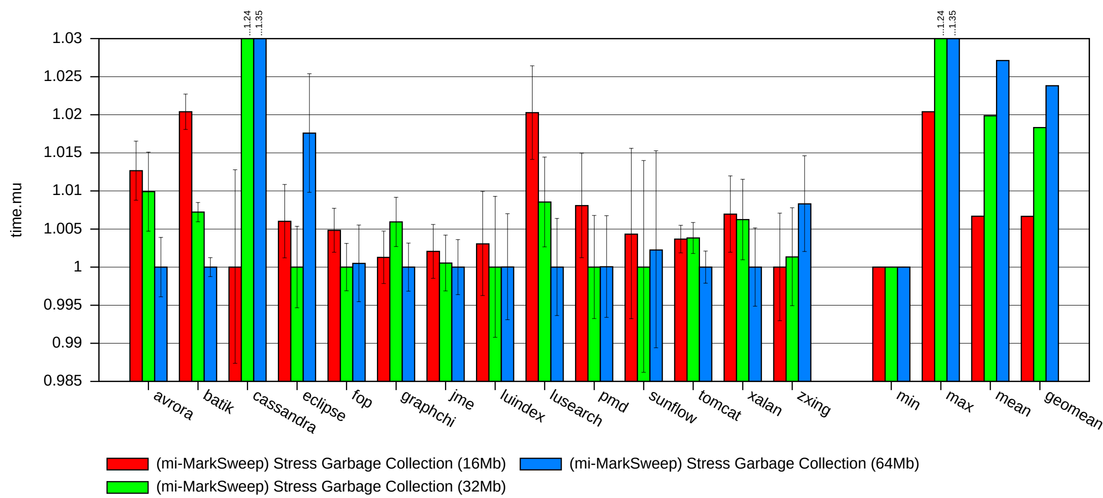


Figure 4.4: Mutator execution time (normalized to best value) averaged over 30 runs for a GC limit of 16 using the mimalloc Mark-Sweep collector with three different stress factor values. A lower value is better.

Unfortunately, we believe this is due to our choice of using the `malloc` Mark-Sweep as the basis of our implementation. We believe the significant mutator overhead for allocating objects (recall that a library call to a `malloc` library of choice is performed) hides or overshadows the costs of our periodic "fake" and "real" GCs. Note that most of the benchmarks which have the statistically insignificant results are benchmarks which have a high allocation rate, lending credence to our theory that the latency for allocating objects far outweighs any other costs.

We note that we see certain benchmarks showcase interesting behaviours regardless of the above flaws in our implementation. The results for `cassandra` are most intriguing. While generally we would expect more periodic GCs to degrade mutator performance, it seems like the mutator time for `cassandra` actually significantly benefits from having more frequent GCs. We believe this is to do with the database workload that `cassandra` simulates. The benchmark uses the YCSB core workload [Cooper et al., 2010] which performs multiple operations such as inserting, updating, reading, etc. against a database. These operations tend to create many short-lived objects. We believe frequent GCs can help with locality in such a case as these allocation slots are recycled more quickly.[10] This also suggests that the costs of a GC such as trashing the cache etc. are less significant for the benchmark. Indeed if we run `cassandra` while further decreasing the stress factor (i.e. increasing the periodicity), we find that the smallest stress factor has the best mutator time. Note that the total time for the benchmark increases, however, given that the GC time increases dramatically.

The `avrora`, `fop`, and `lusearch` benchmarks behave more as expected, with the smallest stress factor having the largest mutator overhead. These are still within $< 2\%$ of the best configuration, however, likely due to the high latency for allocations

[10] Hence, we believe a generational collector could be very beneficial for a benchmark like `cassandra`.

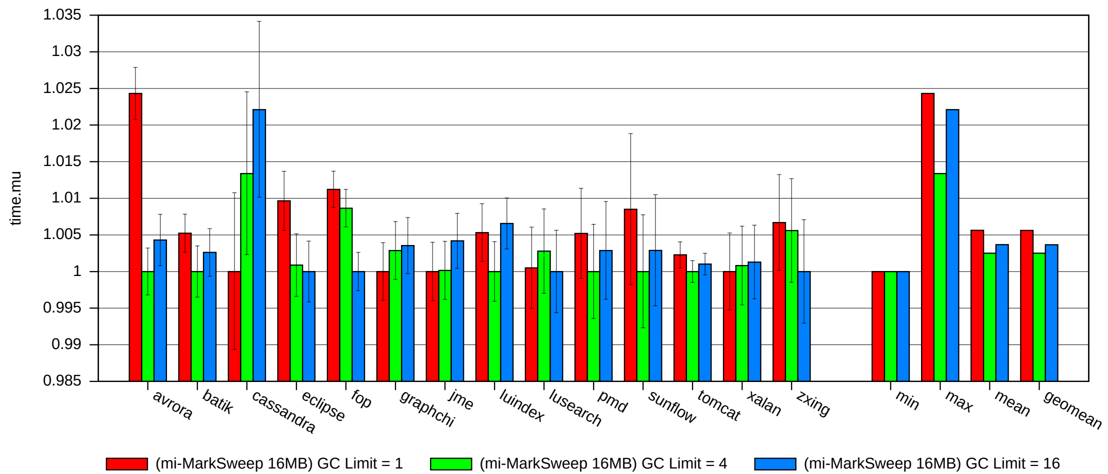


Figure 4.5: Mutator execution time (normalized to best value) averaged over 30 runs for the mimalloc Mark-Sweep collector with a stress factor of 16MB and three different GC limit values. A lower value is better.

as mentioned previously. These costs are likely due to the excessive GCs that can trash the cache and other machine state.

Figure A.1 depicts the average mutator execution time for the DaCapo benchmarks with a GC limit of 4. Hence, every 4th "fake" GC we actually collect the quarantine buffer. The idea for this experiment was to see how separating the costs from the benefits of a GC affects mutator performance. We note that the trends for the GC limit = 1 case are still present. The only differences are with `batik` wherein the GC limit = 1 case had $< 0.75\%$ overhead for the stress factor = 16MB configuration, while there is no perceivable overhead for the GC limit = 4 case. Note that these overheads are still statistically insignificant.

Figure 4.4 depicts the average mutator execution time for the DaCapo benchmarks with a GC limit of 16. The idea for this experiment was to understand the effects of further increasing the time between proper collections. We note that the overheads for most of the benchmarks except `batik` are unchanged. Interestingly, `batik` seems to have significantly larger overheads in comparison to the experiment with GC limit = 4 (see Figure A.1). We believe this suggests that the increased periodicity of the deferred freeing of objects adversely affects its execution.

We now discuss the results per stress factor. Figure 4.5 depicts the average mutator execution time for the DaCapo benchmarks with a stress factor of 16MB. Unfortunately, as before, most the the results are statistically insignificant with overheads being within $< 1\%$ of the best configuration as well as are extremely noisy. The only benchmark we can reasonably comment on is `avrora`. Here, the GC limit = 1 case has significantly more overhead than the other two. Given that the only differences between the configurations is the GC limit, we believe this suggests that the benchmark either suffers significantly from the cost of freeing continuously (with GC limit = 1) or benefits from deferred freeing. We don't see any reason why freeing immediately will perform worse (if anything it should allow for allocation slots to be recycled

quicker), hence we believe that the benchmark benefits from deferred freeing instead. This could be due to the due to the fact that the cost of freeing continuously (in the `malloc` Mark-Sweep) can be expensive and deferred freeing can amortize the cost.

Figures A.2 and A.3 depict the average mutator execution time for stress factors 32MB and 64MB respectively. We note no appreciable differences in comparison to the 16MB case except for `batik` and `cassandra`. We note that the overheads for the GC limit = 16 case for `batik` are minimum, with the other two configurations having a $< 2\%$ overhead. It is likely the increased periodicity of "fake" GCs causes this trend to not show up for the stress factor = 16MB case (see Figure 4.5). While we do not have any conclusive evidence to back this up, we believe that increasing the GC limit for `batik` results in amortizing the costs of freeing objects.

4.5.1 Discussion

We believe that using the `malloc` Mark-Sweep collector in MMTk for our implementation was poor a choice in hindsight. The significant mutator overhead due to the slow allocation may actually hide the costs associated with increasing the periodicity of "fake" and "real" GCs. We suspect this may also be the cause why we didn't see any improvements (in terms of mutator time) for the thread-local quarantining implementation. Unfortunately, due to these oversights, we do not have any statistically significant results. However, we believe, that our broad proposal and experiments are novel and have merit to provide interesting insight into the costs of garbage collection. In the future we hope to implement a similar decoupling of the costs and benefits for a GC like SemiSpace or Immix or revisit Mark-Sweep with a native freelist allocator.

4.6 Summary

In this chapter we describe a novel methodology to understand the costs imposed by garbage collection in managed applications. We appeal to signal processing theory, wherein there is a concept of noise interfering with a signal we want to measure. We further draw the analogy that garbage collection interferes with the steady-state of an application by adding "noise". Hence, our core idea is to decouple the costs from the benefits of garbage collection. One proposed method could be to increase the periodicity of GCs while only actually collecting dead objects after a certain number of GCs. This way we have compounded the costs of garbage collection – through periodic "fake" GCs that perform all GC operations except collection – without any associated benefits as we only collect objects occasionally. Unfortunately, we find that using the `malloc` Mark-Sweep collector in MMTk as a basis for our implementation was a poor choice due to the significant overheads it incurs for allocating objects. This overhead hides or overshadows the costs of increasing the periodicity of both "fake" and "real" GCs. In the future we would like to revisit this methodology for a different GC algorithm such as SemiSpace, Immix or Mark-Sweep with a native freelist allocator.

Chapter 5

Locality Effects of Garbage Collection

How an application manages and uses its data locality can drastically affect its performance. Good cache locality and data placement/alignment can dramatically improve the runtime of an application. However, the effects of garbage collection on a mutator's locality are not well understood. In this chapter we describe our methodology, design and implementation, and results for an experiment that aims to understand the *effects of garbage collection on the locality of the mutator*.

5.1 Objectives

We believe garbage collection can have both positive and negative effects on an application's data locality, depending on the garbage collection algorithm used. A copying or compacting GC could rearrange objects in memory to make objects that are frequently accessed sit on the same cache line. On the other hand, garbage collectors such as the classical Mark-Sweep are unlikely to improve application locality, and in fact could hurt it due to internal fragmentation. We construct an experiment that tries to understand the locality effects of classical GC algorithms such as Immix, SemiSpace, and Mark-Sweep.

5.2 Measuring Locality Effects

We believe that certain garbage collection algorithms can benefit an application's locality. However, these locality benefits may not be immediate, that is to say, an application will see improved locality some period of time after a GC disrupts the application's execution. More precisely, while a GC may negatively the affect the locality of an application at the moment it occurs (due to trashing the cache and the working set of the application), the application could benefit from the rearranged and compacted objects in the future.

In order to measure these delayed benefits, we want to use a benchmark with many small transactions that we can individually measure. We can coalesce various measurements for these transactions such as start and end times, total execution

time, cache misses, etc. Each transaction is executed multiple times in order to get different measurements in time. We then calculate the proximity of execution of these transactions to the execution of the "closest" GC. Here, we define the *closest GC* as the GC just prior to the query executing. Given we want to measure the "delayed effects" of a GC, we believe this is a sound definition of "closest". Using this we can see if there is a correlation with the (best) execution time(s) of a transaction to how close it was to a GC.

We pick the `lusearch` benchmark from the DaCapo suite of Java benchmarks [Blackburn et al., 2006a,b] as our benchmark of choice. `lusearch` is a simple text search application using the Apache Lucene Java search library [Apache, 2021]. Here, each transaction is a single text query. We can then instrument the benchmark to report various measurements of each query. Further, running the benchmark using OpenJDK with MMTk, we can generate a trace of the start and ends of all GCs. Now we can place each query in comparison to its distance from a *previous* GC on a timeline.

5.3 Design and Implementation

Our implementation is simple: we want to instrument both the benchmark (i.e. `lusearch`) and MMTk. We discuss both in this section.

5.3.1 Instrumenting `lusearch`

Given `lusearch` is written in Java where (almost) everything is an object, we want to minimize our memory footprint (as well as allocation rate) in order to not trigger excessive GCs. To this end, we bulk allocate three arrays when the benchmark starts, namely: two `long` arrays to store the *start* and *execution* times (in nanoseconds) of queries, and a `String` array to store a unique query identifier. We felt the use of a `String` array is reasonable since we do not actually allocate more objects as each query internally stores a unique `String` identifier anyway. Just before a query is primed to execute, we get the current time in nanoseconds. Immediately after the end of the query, we calculate the execution time (in ns) and update the three arrays.

Given `lusearch` is a highly parallel benchmark, we need to synchronize access to each of the arrays. Instead of using locks or other synchronization primitives, we opted to use a simple shared atomic integer that is atomically incremented to get an index value before a query starts execution. This allowed us to have a fairly low profile in terms of both memory and execution time.

Finally, at the end of the benchmark, we create and dump a CSV file of all the queries that were executed. The CSV file is named with a timestamp in order to easily distinguish between runs. Our changes to the benchmark can be found publicly on GitHub at `k-sareen/dacapobench`.

Unfortunately, due to time constraints, we were not able to implement support for measuring different performance counters (such as cache misses, branch mispredictions, etc.) per query without directly using simple Java Native Interface (JNI)

calls. Simple JNI calls[11] are generally expensive operations, and hence are unsuitable for the scale of measurements we want. We note, however, that there is prior work by Yang et al. [2016] that could be used to support measuring performance counters on a per query-granularity. We leave investigation into this area as potential future work.

5.3.2 Instrumenting MMTk

On the MMTk side, we want to report the start and end of a GC in order to place the execution of a query to its closest GC. This is easy to implement in MMTk. Every GC algorithm implemented in MMTk schedules a collection by registering work (through the form of *work packets*, i.e. small units of GC work) with the GC scheduler. We can then get the current time (in ns) just before all the work is scheduled, and calculate the total execution time (in ns) when all work has finished. We then create a CSV file and store the GC's unique ID (assigned via a simple atomic integer), start time, and execution time. The CSV file is named with a timestamp in order to easily associate it with the CSV files generated by `lusearch`.

5.4 Experimental Methodology

We use the following experimental methodology.

5.4.1 Hardware and Operating System

We ran our experiments on the following hardware platforms:

(i) Intel i7-4770 Haswell with a 3.4 GHz clock, a 4 x 32 KB, 64 B/line, 8-way L1 cache, a 4 x 256 KB, 64 B/line, 8-way L2 cache, and 16 GB DDR3 RAM;

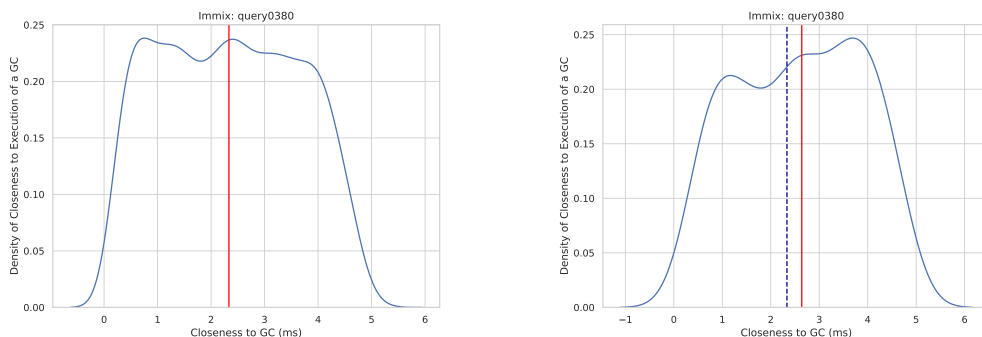
(ii) Intel Xeon Gold 5118 Skylake with a 2.3 GHz clock, a 12 x 32 KB, 64 B/line, 8-way L1 cache, a 12 x 1 MB, 64 B/line, 16-way L2 cache, and a 512 GB DDR4 RAM.

All systems ran Ubuntu 18.04.6 LTS with Linux 5.4.0-87 kernels. All CPUs operate in 64-bit mode and use 64-bit kernels. In order to avoid NUMA effects on the Intel Xeon, we limit our experiment to just a single socket, that is to say, only 12 cores (24 threads).

5.4.2 MMTk and OpenJDK

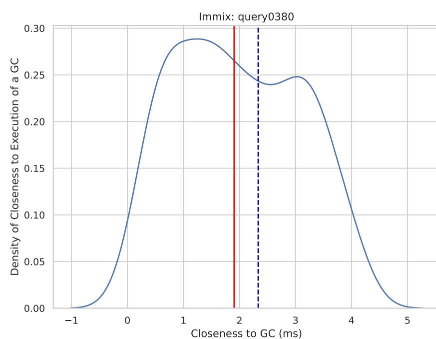
We use the Rust rewrite of MMTk (see Chapter 2) with `mmtk-core` revision `0ccf09cc`, mmtk-openjdk revision `c1bc366`, and a modified version of OpenJDK 11 (revision `6dc618e2811`) which has support for MMTk. All experiments use the HotSpot C2

[11] We specify *simple* JNI calls here as the JVM does include backdoors for certain performance-critical JNI calls such as timers and array copying which generally avoid the overheads associated with JNI.



(a) Density of closeness to a GC for all executions

(b) Density of closeness to a GC for the best 5th percentile of executions



(c) Density of closeness to a GC for the worst 5th percentile of executions

Figure 5.1: Density of closeness to a GC for Query 0380 using the Immix collector on the Haswell system. Note that executions where the query was interrupted by a GC have been removed. The red line in 5.1a and dashed blue lines in 5.1b and 5.1c are the median for the entire dataset. The red lines in 5.1b and 5.1c are the median for the best and worst 5th percentile of executions with respect to execution time.

JIT compiler with pre-compilation enabled and explicit GCs disabled. Measurements were captured using the DaCapo benchmark harness and probes. Since we are measuring time-sensitive values, we perform two warmup iterations before starting our timing iteration in order to reduce any noise due to compilation or other VM operations.

5.4.3 Experimental Design

As mentioned previously in Section 5.2, we measure the start and execution times of text search queries in the `lusearch` DaCapo benchmark. The workload for the `lusearch` benchmark in DaCapo consists of 2048 text search queries. Each query is executed 256 times per run of the benchmark. We run the benchmark 20 times in order to get a large dataset to work with. Hence for each query, there are 5120 unique executions.

Table 5.1: Geometric mean of normalized medians for the best and worst 5th percentile of executions (per query) over all 2048 queries per collector. Note how most of the best queries for Immix are farther away from a GC.

	Immix	SemiSpace	Mark-Sweep
Best 5th Percentile	1.30	1.08	1.06
Worst 5th Percentile	0.76	0.86	0.90

We gather results for three classic garbage collector algorithms: SemiSpace, Immix, and Mark-Sweep. We use an Immix configuration that opportunistically defragments the heap. Note the the Mark-Sweep implementation still internally uses a `malloc` library as a freelist (see Chapter 3). Hence we stress that the results for our implementation of Mark-Sweep collector do not necessarily reflect the results of a native freelist Mark-Sweep collector. As of writing this thesis, MMTk does not currently have a native freelist implementation, however there is work being done to port the mimalloc freelist design into MMTk. We chose a heap size large enough for each of the different collectors to complete the benchmark in a reasonable time, while also small enough such that we have a significant number of GCs (around 3000 GCs per run). This is to make sure that we have enough queries that are actually affected (both directly and indirectly) by the execution of a GC.

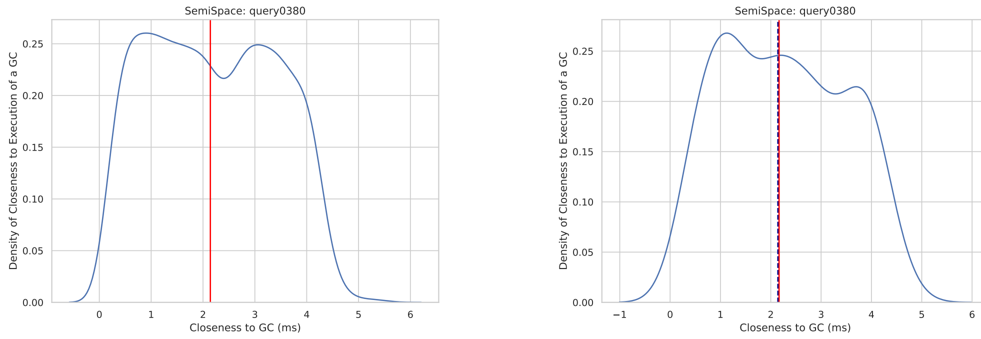
We conduct our experiment in a standalone setting where we run the benchmark without any other significant resource-consuming processes on the system in order to limit the experimental noise in our measurements. After the benchmark finishes, we collect the CSV files generated by both MMTk and `lusearch` in order to analyse their data on a separate device.

5.5 Results and Evaluation

We select three representative queries from the 2048 in order to study them in more detail. For the sake of conciseness and completeness, we place some figures in Appendix A. We discuss the Intel i7-4770 Haswell results first.

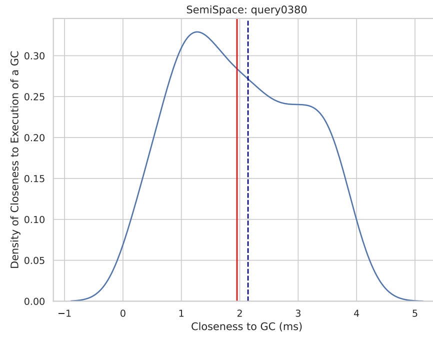
Figures 5.1–5.3 plot the density of closeness of execution to a GC for Query 0380, where *closeness* of execution to a GC is defined as before. Note that we remove all executions where the query was interrupted by a GC. We note that only around 150 executions (each per collector) are actually interrupted by a GC, still giving us a sizable dataset to work with.

From Figure 5.1a, we see that most queries actually execute close to a GC. Hence, it is not a surprise that we note that the queries in the best 5th percentile in terms of execution time (i.e. the shortest execution times) and the worst 5th percentile (i.e. the longest execution times) are also generally close to the execution of a GC. Interestingly, we note that it is generally the case that the best queries are skewed towards being farther away from a GC (with respect to the global median) while the worst queries are skewed towards being closer to a GC (with respect to the global median). We note that this trend is apparent for the rest of the collectors as well.



(a) Density of closeness to a GC for all executions

(b) Density of closeness to a GC for the best 5th percentile of executions



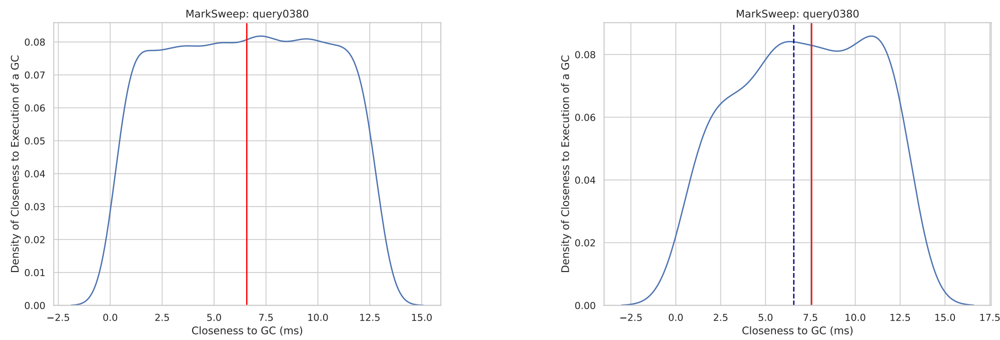
(c) Density of closeness to a GC for the worst 5th percentile of executions

Figure 5.2: Density of closeness to a GC for Query 0380 using the SemiSpace collector on the Haswell system. Note that executions where the query was interrupted by a GC have been removed. The red line in 5.2a and dashed blue lines in 5.2b and 5.2c are the median for the entire dataset. The red lines in 5.2b and 5.2c are the median for the best and worst 5th percentile of executions with respect to execution time.

For the Mark-Sweep collector, we can see that the distribution of the proximity to a GC is 3× larger than for the rest of the collectors. We attribute this to the slowdown imposed by using the `malloc` Mark-Sweep implementation.

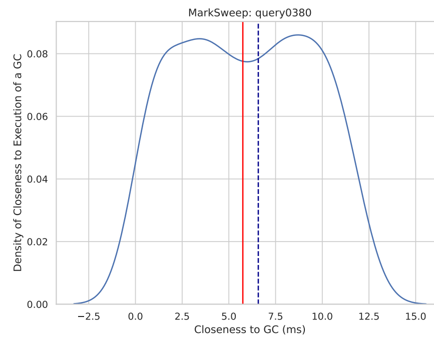
Further, we note that the above trends are generally true for the other selected queries as well (see Figures A.4–A.9). We believe this trend suggests that the best queries are generally farther away from a GC. We note that this trend is more apparent for the Immix collector than the SemiSpace and Mark-Sweep collectors. For the latter two collectors, the median of the best queries nearly coincides with the median of all the executions (for the queries we have chosen).

We investigate our claim that the best executions of a query are generally farther away from a GC by using a crude metric to estimate the variance of these medians: we simply calculate the median for the best and worst 5th percentile of executions for each query and normalize them with respect to the median of all the executions for that particular query. Table 5.1 reports the geometric mean for each of the two



(a) Density of closeness to a GC for all executions

(b) Density of closeness to a GC for the best 5th percentile of executions



(c) Density of closeness to a GC for the worst 5th percentile of executions

Figure 5.3: Density of closeness to a GC for Query 0380 using the Mark-Sweep collector on the Haswell system. Note that executions where the query was interrupted by a GC have been removed. The red line in 5.3a and dashed blue lines in 5.3b and 5.3c are the median for the entire dataset. The red lines in 5.3b and 5.3c are the median for the best and worst 5th percentile of executions with respect to execution time.

sequences generated. Note that the raw numbers do not necessarily mean anything and we only mention them in order to understand if there is a significant variance in the medians of the best and worst executions per query or not.

We can immediately see that most of the best queries for Immix are farther away from a GC, while the worst performing queries are closer to the execution of a GC. We note that for SemiSpace and Mark-Sweep, while the median for the best queries do not significantly differ from the global median, the worst performing queries are closer to the execution of a GC. We believe the above trend is due to the GC disrupting an application's steady state and then the application converging back to its steady state after a while. Hence, queries which are sufficiently farther away from a GC generally have the shortest execution times.

Table 5.2: Geometric mean of normalized medians for the best and worst 5th percentile of executions (per query) over all 2048 queries per collector. Note how most of the best queries for Immix and SemiSpace are farther away from a GC, whereas for Mark-Sweep they are much closer to the execution of a GC.

	Immix	SemiSpace	Mark-Sweep
Best 5th Percentile	1.30	1.17	0.34
Worst 5th Percentile	0.69	0.72	0.98

We now discuss the results from the Intel Xeon Gold 5118 Skylake processor. From Figures A.10–A.18 and Table 5.2, we note that the general trends we find in the Haswell results are still apparent (if not more pronounced) with the exception of the Mark-Sweep collector. For the Mark-Sweep collector on the Skylake system, the queries that are much closer to the execution of a GC perform the best. This goes against the general trend set for other collectors as well as the Haswell system. We do not have any conclusive hypotheses or explanations for this surprising behaviour. One potential cause could be due to the freelist allocator recycling blocks for allocation right after a GC, however we would have expected to see similar results for the Haswell system then. An interesting observation we note for the Mark-Sweep collector is how the execution of a query is much more spread out in terms of proximity to a GC (even reaching to 150 ms away from a GC for some executions) than for the other collectors. We believe this is due to the very slow allocation of the freelist allocator, as the allocator will perform a library call to mimalloc internally. We see that in comparison to the Haswell system, the distribution for proximity to a GC is larger by a factor of 10. We believe this is a side effect of the much slower clock speed of the Xeon, which results in dilation of the execution time.

We note that the number of executions (per query) interrupted by a GC also far exceeds that in the Haswell system. Here, generally > 600 executions per query are interrupted by the execution of a GC. We believe that this is a side effect of the larger core count of the system. Given we have more queries executing parallelly (in fact we have $3\times$ more queries running in parallel than the Haswell system), it is more likely that multiple queries are interrupted by the same GC. Indeed we find that the number of queries interrupted in the Xeon system are roughly $3\times$ that of the Haswell system.

5.5.1 Discussion

While we did not find any conclusive evidence, we believe there is a (weak) correlation between the execution time of a query to its proximity to the execution of a GC, with queries farther away from a GC executing significantly faster than queries closer to a GC. The results for the SemiSpace collector generally suggests that the execution time of a query is indifferent to its proximity to a GC. We note that since most of the executions per query were also distributed being close to the execution of a GC, the experiment results could also suggest there is no substantive delayed benefit of a

GC to the locality of an application. Further measurements (such as measuring cache and branch prediction misses per query) need to be done in order gain more insight regarding the locality benefits of garbage collection.

5.6 Summary

In this chapter we outlined a simple experiment to measure the effects of a GC on an application's locality using the case-study of a transaction-based benchmark. We instrument `lusearch` – a text query search benchmark – in order to gather data regarding the execution time of a query and its proximity to a GC. Benchmarking with a variety of different GC algorithms, we find a weak correlation of the execution time of a query to its proximity to a GC: it is generally the case (more apparent in certain GC algorithms) that the best queries execute farther away from a GC. We believe this is due to the GC disrupting the application's steady state and the application converging back to its steady state after a while. However, these benefits are not too pronounced as we find that most queries execute close to a GC anyway. We note that the lack of significant benefits *could* suggest that there is no substantive delayed benefit of a GC to the locality of an application. Future work could deal with measuring more relevant quantities such as cache and branch prediction misses per query which could further help elucidate the locality effects of garbage collection.

Chapter 6

Garbage Collection Behaviour in an Unmanaged Context

It is impossible to insert a fully functional state-of-the-art garbage collector in a manually memory managed language. However, we can easily *emulate* garbage collection behaviour, i.e. insert the inhale-exhale behaviour of garbage collection to a manually memory managed application. In this chapter we describe our methodology, design and implementation, and results for an experiment that aims to understand the *effects of inserting garbage collection-like behaviour in applications that are manually memory managed*.

6.1 Objectives

Most work surrounding the costs of garbage collection (including ours) seem to approach understanding it from the managed-to-manual angle. We believe there is insight to be gained if we approach the question from the other direction as well – i.e. from manual-to-managed. We construct an experiment that adds GC-like behaviour (i.e. bulk allocation and deallocation) to manually managed C and C++ programs and measure their maximum heap size as well as their execution time.

6.2 Approximating Garbage Collection Behaviour

We implement the idea of *deferred freeing* or *quarantining*, which is often associated with reference counting [Deutsch and Bobrow, 1976; Blackburn and McKinley, 2003] or preventing security vulnerabilities such as use-after-free bugs [Ainsworth and Jones, 2020], and use it as an approximation to the *inhale-exhale* behaviour of garbage collectors as discussed in Chapter 2. We then measure the overheads imposed by this deferred freeing in terms of both the maximum resident set size (i.e. the heap) as well as the execution time. This gives us an idea of how garbage collection could affect a manually memory managed application.

Deferred freeing or quarantining does not immediately free a dead object, instead opting to free it at a later time, and often freeing dead objects in bulk. This can

be achieved by delaying the updating of reference counts (in a reference counting-based approach) [Deutsch and Bobrow, 1976; Blackburn and McKinley, 2003], or via placing the data in a quarantine buffer to free later (in a quarantine-based approach) [Ainsworth and Jones, 2020]. We note that many manually memory languages such as C++ and Rust have standardized reference counting through their standard library, however, these are not easily portable to an application written in C, which remains widely used to this day. Hence, we focus on the quarantining-based approach in our work since it is universal.

In a quarantining-based approach to adding inhale-exhale behaviour, when a programmer calls `free` on an object, instead of directly freeing the associated memory, we place the object in a (local) buffer termed *quarantine list*. When either the quarantine list is full or if the total volume of objects in the quarantine list[12] exceeds a certain threshold, the quarantine list is walked, freeing all the objects. Note that this is different from using a conservative garbage collector as we do not perform any transitive closure or walk the heap. Instead, we rely on the programmer to insert (hopefully) correct calls to `free` when objects are no longer required. Of interest is that unlike automatic memory management, here it is the call to `free` that will stall and result in the bulk freeing, instead of an allocation call. Our quarantining-based approach was not developed to prevent use-after-frees or other memory bugs, though we note that we can significantly reduce the chances of a use-after-free exploit just by deferring the deallocation of an object to a later time.

We note that by deferring frees to a later point in time, we add the inefficiencies of garbage collection to a manually memory managed application. Most notably, since dead objects are not freed immediately, there is a noticeable space-overhead with applications potentially having larger heap footprints. Given that there is extra work being done (in terms of placing dead objects in a quarantine list and walking the quarantine list to free them), there is an aspect of time-overhead as well since the application cannot continue execution until it returns from the `free` call. Most importantly however, like a Mark-Sweep collector, we do not recycle free slots immediately as dead objects are still kept alive in the quarantine list. This could potentially result in poorer mutator locality as well as contribute to the time-overhead.

Given the above machinery, we can also vary the `malloc` implementation used in order to understand how the inhale-exhale behaviour affects the different `malloc` implementations. We focus on the jemalloc [Evans, 2006], hoard [Berger et al., 2000], mimalloc [Leijen et al., 2019], and glibc 2.27 allocators in our work.

6.3 Design and Implementation

We implement our quarantining-based approach as a shared library in C. We name this shared library `libql`. `libql` aims to be a drop-in replacement for `malloc-free` (as well as `new-delete` for C++). We can dynamically use this library by preloading it in the dynamic linker before executing an application. Such a design is non-intrusive

[12] *Volume* is defined as the sum of sizes of all objects in the quarantine list.

as it means we don't have to recompile programs to link to `libql`. This design also allows `libql` to hide the `malloc` and `free` implementations of other allocators such as mimalloc, jemalloc, etc. We refer to the underlying allocator whose functions we override as the *backing allocator* or *backing implementation*.

Internally, `libql` allocates a thread-local buffer of fixed size (the default being 40960 bytes, i.e. 5120 pointers on a 64-bit system) for each application thread that calls into the library to allocate memory. This thread-local buffer acts as the quarantine list[13]. We use a thread-local buffer since we want to avoid synchronization costs of having serialize access to a global buffer in the case of a multi-threaded application. This quarantine list is allocated at the first call to `free` for that application thread. We also maintain two thread-local pieces of metadata associated with the quarantine list: (i) the current volume of objects in the quarantine list (in bytes); and (ii) the index of the first free element in the quarantine list. These variables help us check if we need to perform a bulk free operation.

All allocation functions (i.e. `malloc`, `calloc`, `realloc`, etc.) remain unchanged and we directly call into the backing allocator to service the allocation request. However, whenever a programmer frees an object, we place the pointer to this object into the quarantine list. The current volume is incremented by the size of the freed object. If either the quarantine list is full or the current volume of the objects in the quarantine list has crossed a user-defined threshold, then we iterate through the quarantine list freeing all the dead objects. Note that this should only stall one application thread owing to our use of thread-local data structures.[14] We term this bulk freeing operation as a *collection*. If the collection is initiated due to the thread-local buffer being exhausted, we term it a *capacity-collection*; and if the collection is initiated due to the volume threshold being crossed, we term it a *volume-collection*. In a collection, we always iterate through the quarantine list backwards, i.e. we free in a last-freed, first-freed (LIFO) order. In the case of a volume-collection, freeing LIFO means that we avoid iterating through the entire quarantine list since we know the index of the object last inserted into the list. Hence, theoretically, if we set the threshold to 1 B, then we don't have to do too much extra work in comparison to not using a quarantine list: there should only be a few extra comparison operations and then the actual free. However, as we see in Section 6.5, this may not necessarily be true.

As mentioned previously, the threshold (in terms of volume) for when to collect objects is configurable at runtime. We define an environment variable (`QL_SIZE`) to set this threshold. The user can then specify (in bytes) the threshold to collect objects at runtime by setting this environment variable. The default threshold is 40960 bytes. Note that we do not allow a `QL_SIZE` of 0 as it has no meaning. In the case a user specifies a `QL_SIZE` of 0, we use the default threshold.

We attach constructors and destructors to the library as well as a destructor for when POSIX threads (pthreads) exit. At library initialization, we get a function pointer to the backing `free` implementation by calling into the runtime dynamic

[13]Even though we may call it a quarantine *list*, it is not a linked list and is in fact a fixed size array.

[14]Unless, of course, another application thread is waiting on a response from the stalled one, which is out of our hands.

linker. We also check if the user has overridden the `QL_SIZE` environment variable, setting the global threshold appropriately. We expect multi-threaded applications to use the pthreads library (in some form) and hence, we associate a destructor with the exit of a pthread. This destructor will collect all objects currently in the quarantine list and unmap the quarantine list to prevent memory leaks. On program exit, the library destructor attempts to collect the quarantine list to prevent memory leaks for single-threaded applications or applications which heavily rely on `fork` calls.

Finally, we alias all deallocation functions in both C and C++ to our `free` implementation in order to override the backing implementation. Note that it does not matter if we override the allocation functions or not since they remain unchanged. The source code for the library can be viewed on GitHub: `k-sareen/deferred-free`.

6.4 Experimental Methodology

We use the following experimental methodology.

6.4.1 Benchmarks

We use standard C and C++ allocation benchmarks to understand the overheads imposed by the inhale-exhale behaviour we have inserted. We briefly describe the benchmarks used. We used the following single-threaded benchmarks:

- **barnes** A hierarchical n-body particle solver [Barnes and Hut, 1986].
- **cfrac** A continued fraction factorization program.
- **espresso** A programmable logic array analyzer [Grunwald et al., 1993].

We used the following multi-threaded benchmarks:

- **cache-scratch** A test for passive false sharing of cache lines [Berger et al., 2000].
- **larson** A server workload simulation [Larson and Krishnan, 1998].
- **sh6bench** A test where some objects are deallocated in LIFO order, and others in reverse (i.e. FIFO) order.
- **sh8bench** A test where some objects are deallocated in reverse (i.e. FIFO) order, and others by other threads.
- **threadtest** A test which continuously allocates and deallocates objects.
- **xmalloc-test** A produce-consumer benchmark with X allocating threads and X deallocating threads.

We base the inputs of each benchmark from the benchmark suite curated by the mimalloc authors [Daan Leijen, 2021].

6.4.2 Hardware and Operating System

We ran our experiments on the following hardware platform: Intel i7-6700K Skylake with a 4 GHz clock, a 4 x 32 KB, 64 B/line, 8-way L1 cache, a 4 x 256 KB, 64 B/line, 4-way L2 cache, and 16 GB DDR4 RAM. The system ran Ubuntu 18.04.6 LTS with a Linux 5.4.0-87 kernel. The CPU operates in 64-bit mode and uses 64-bit kernels.

6.4.3 Experimental Design

The key idea of this experiment is to measure the space- and time-overheads imposed by inserting our quarantine-based inhale-exhale behaviour to a manually memory managed application. We measure the execution time as well as the maximum resident set size (RSS) (i.e. heap size) per run. While our implementation is explicitly designed around keeping space-overheads low, we believe it is still interesting to measure and compare them specially alongside the time-overheads. We use four different backing allocators: jemalloc, mimalloc, hoard, and the system glibc 2.27 allocator. We pair each allocator with the `libql` library in order to insert the inhale-exhale behaviour to them. We further vary the `QL_SIZE` environment variable (with values of 1 B, 4 KB, 40 KB, 512 KB, 1 MB) to get an idea of how the periodicity of collections affects the performance and max RSS. Hence, each backing allocator has six different configurations. We also vary the size of the thread-local buffer for the quarantine list (with values of 40 KB and 128 KB). We conduct our experiment in a standalone setting where we run without any other significant resource-consuming processes in order to limit the experimental noise in our measurements. We run each benchmark 20 times and report the average in order to reduce the effect of experimental noise in our measurements. Note that we fix the number of threads for multi-threaded benchmarks with configurable threads to 8 (maximum for our system).

6.5 Results and Evaluation

We breakdown our results per allocator per thread-local buffer size since we are trying to understand how the inhale-exhale behaviour affects an allocator implementation. Inter-allocator comparisons are not easy to make given the dramatically different implementations. Each allocator has six different configurations:

(i) `{alloc}`: A release build of the *{alloc}* allocator, where *alloc* can be {`mi`, `je`, `hoard`, `glibc`};

(ii) `ql-{alloc} (1 B)`: A release build of the *{alloc}* allocator with quarantining and `QL_SIZE=1`;

(iii) `ql-{alloc} (4 KB)`: A release build of the *{alloc}* allocator with quarantining and `QL_SIZE=4096`;

(iv) `ql-{alloc} (40 KB)`: A release build of the *{alloc}* allocator with quarantining and `QL_SIZE=40960`;

(v) `ql-{alloc} (512 KB)`: A release build of the *{alloc}* allocator with quarantining and `QL_SIZE=524288`; and

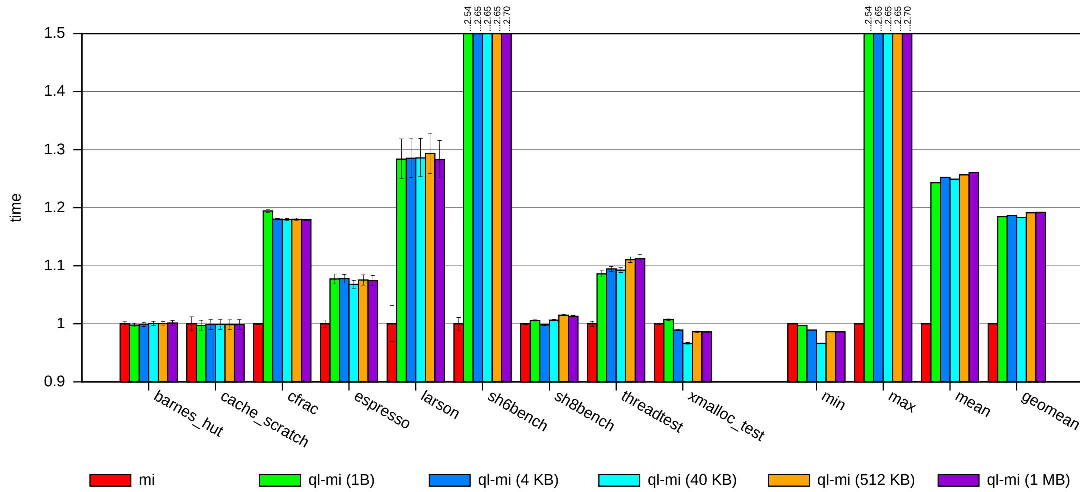


Figure 6.1: mimalloc average execution time over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `mi` configuration. A lower value is better. Note that the larson benchmark has a fixed execution time and hence we use the "relative execution time" as reported by the benchmark.

(vi) `ql-{alloc} (1 MB)`: A release build of the *{alloc}* allocator with quarantining and `QL_SIZE=1048576`.

For the sake of conciseness and completeness, we place some figures in Appendix A.

6.5.1 mimalloc

Figures 6.1 and 6.2 show us the average execution time and maximum RSS (respectively) reported over 20 runs with a quarantine thread-local buffer size of 40 KB for the mimalloc allocator with its six different configurations. For brevity, we shall refer to the configurations by their number as listed above. For example, the `mi` configuration will be referred to as *configuration (i)*.

We can immediately see that for benchmarks such as barnes and cache-scratch, the added inhale-exhale behaviour does not significantly affect the execution time or the heap size. These benchmarks generally do not allocate much which could explain their indifference to the inhale-exhale behaviour.

There is almost no space-overhead for the cfrac benchmark between the different configurations (at max 1.1× *configuration (i)*). However, there seems to be a constant execution time-overhead of 1.2× *configuration (i)* regardless of the periodicity of collections. The cfrac benchmark allocates many (small) short-lived objects during its execution. We believe this relatively constant time-overhead comes from frequent collections due to the quarantine list filling up. Note how *configuration (ii)* is marginally slower than the others. We believe this is due to its more frequent collections.

The space-overhead for the espresso benchmark shows a step-like pattern with each increasing value of `QL_SIZE` increasing the heap size. At the most extreme,

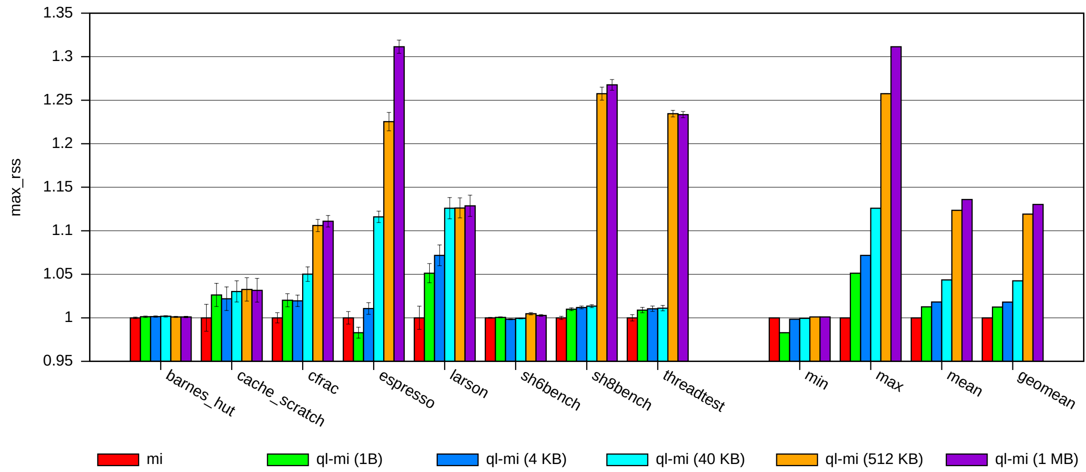


Figure 6.2: mimalloc average maximum RSS over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `mi` configuration. A lower value is better. Note the xmalloc-test results should be ignored since the benchmark allocates more the faster it runs. Hence, we have removed them from this (and all future) graph(s) in order to not affect other calculations.

espresso requires a 30% larger heap size in comparison to *configuration (i)*. This intuitively makes sense given increasing the value of `QL_SIZE` decreases the periodicity of collections. What is interesting, however, is that the time-overhead is relatively constant, with execution time of the quarantine list configurations being $< 1.1\times$ *configuration (i)*.

Larson follows a similar trend as above with the space-overhead follows a step-like pattern (at max $1.1\times$ *configuration (i)*). As mentioned previously, one of the input parameters for larson is the execution time (in seconds). Hence, we use the "relative execution time" as reported by the larson benchmark. The relative execution time is calculated as $1/throughput$, where *throughput* is the total number of allocations per second. We note that around 7 million objects are allocated over the course of the benchmark's execution. Hence, for *configuration (ii)*, we have 7 million collection operations, and on the other end we have around 400,000 collection operations for *configuration (vi)*.

We believe that even though the collections for *configuration (ii)* are very frequent, the cost of iterating through the quarantine list is negligible in comparison to allocating, accessing, and deallocating the thread-local buffer. In order to confirm this, we made two different builds of `libql`. The first build directly calls the `free` of the backing allocator without allocating any thread-local buffers. We refer to this build as *no-ql*. The second build allocates the thread-local buffer, accesses it to set the pointer value, updates the current volume, and then immediately frees the object. The thread-local buffer is unmapped in the destructor. We refer to this build as *no-collect-ql*. Note that we never iterate or collect the quarantine list in this build. The execution time for *no-ql* was comparable to *configuration (i)*, whereas the execution time for *no-collect-ql* was comparable to the rest. Hence, we believe that the exe-

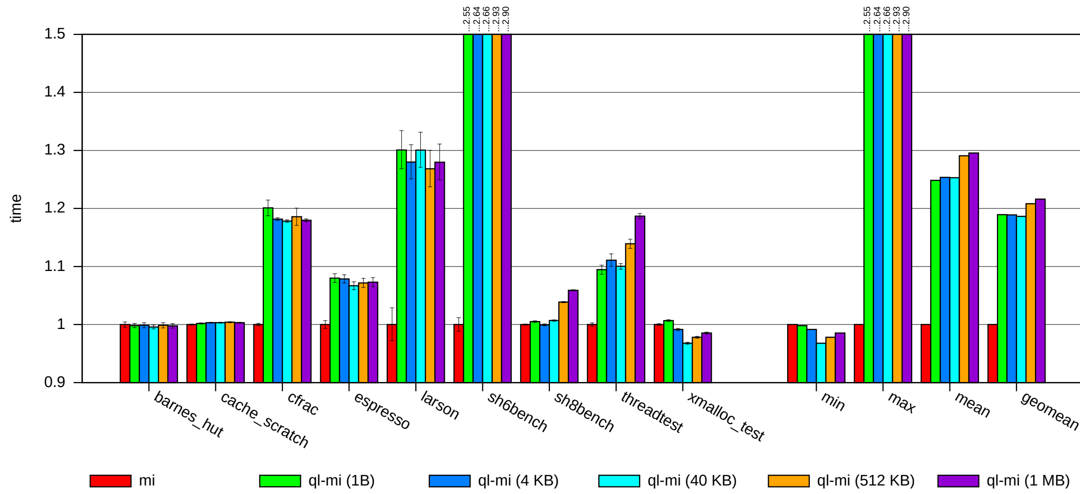


Figure 6.3: mimalloc average execution time over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `mi` configuration. A lower value is better.

cution time-overhead for the quarantine configurations evens out regardless of the periodicity of collections due to the overheads imposed by allocating, deallocating, and accessing the thread-local buffer[15]. The larson benchmark continuously spawns (and kills) threads throughout its execution which results in the large execution time-overhead we see.

This trend is also apparent in sh6bench. While there is no significant space-overhead for the benchmark, the execution time-overhead is $> 2.5\times$ *configuration (i)*. We find that the *no-ql* build from above behaves exactly like *configuration (i)*, whereas the *no-collect-ql* build behaves exactly like the rest of the quarantine configurations. While sh6bench does not continuously spawn threads, we believe the costs of accessing the thread-local buffer are significant. Annotating the execution of the *no-collect-ql* build running sh6bench using `perf`, we find that nearly 9% of the measured samples were accessing the thread-local buffer and nearly 35% of the measured samples were in our `free` implementation.

Interestingly, sh8bench has no significant time-overheads in comparison to *configuration (i)*. The space-overhead for the larger values of `QL_SIZE`, however, is significant, requiring a 30% larger heap size than *configuration (i)*. We find that most collections for *configuration (v)* and *configuration (vi)* are capacity-collections and not volume-collections, while most collections for the rest of the quarantine configurations are volume-collections. Hence, *configurations (v)* and *(vi)* have similar space-overheads, whereas the others retain a low profile due to more frequent collections.

Threadtest seems to have an execution time-overhead of about 10% in comparison to *configuration (i)*. Using the *no-ql* and *no-collect-ql* builds we find that the time-overhead likely comes from allocating, accessing, and deallocating the thread-local

[15]Note that we are also considering any extra comparison operations that are inserted into the code due to the thread-local buffer (such as a check to first allocate it) as part of the overheads.

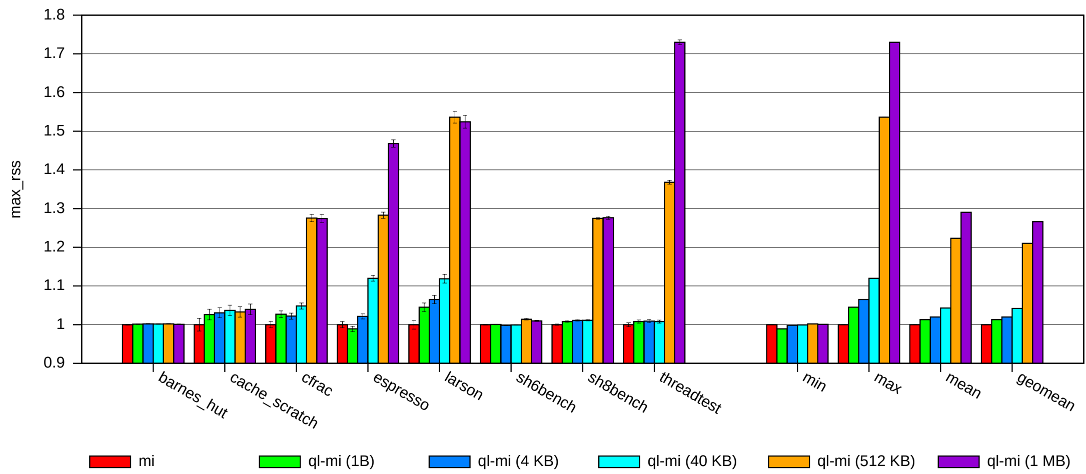


Figure 6.4: mimalloc average maximum RSS over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `mi` configuration. A lower value is better.

buffer. We annotate the execution of the *no-collect-ql* build running the threadtest benchmark using `perf`. We find that around 5% of the measured samples were in our `free` implementation. We note that the space-overheads for threadtest follow a similar pattern to sh8bench, wherein larger values of `QL_SIZE` lead to larger heap sizes. As with sh8bench, most of the collections for *configurations (v)* and *(vi)* are capacity-collections.

Finally, as mentioned previously, the heap size values for xmalloc-test should be ignored since the benchmark allocates more the faster it runs. This is the only benchmark where quarantining actually improves the execution time in comparison to *configuration (i)*, with *configuration (iv)* being 0.97× *configuration (i)*. We believe this slight performance win is related to the asymmetrical producer-consumer workload that xmalloc-test simulates, however, we do not have any conclusive hypotheses or explanations to back up this claim. Note that sh8bench simulates a similar workload (i.e. wherein objects allocated in one thread are freed by another thread)[16], with performance of the quarantine configurations comparable to *configuration (i)* lending credence to our theory. We also note that running xmalloc-test under `perf` shows that nearly 30% of measured samples were in a mimalloc function (`_mi_page_free_collect`) for all configurations, suggesting the bottleneck in this benchmark could be due to mimalloc's design.

Figures 6.3 and 6.4 show us the average execution time and maximum RSS (respectively) reported over 20 runs with a quarantine thread-local buffer size of 128 KB for the mimalloc allocator with the previously mentioned six configurations.

Broadly, the trends from Figures 6.1 and 6.2 are still visible when we increase the buffer size to 128 KB. Notably, we see that benchmarks such as cfrac, larson,

[16]We note that larson also simulates a similar workload, however it suffers from a significant execution time-overhead due to its continuous spawning of threads.

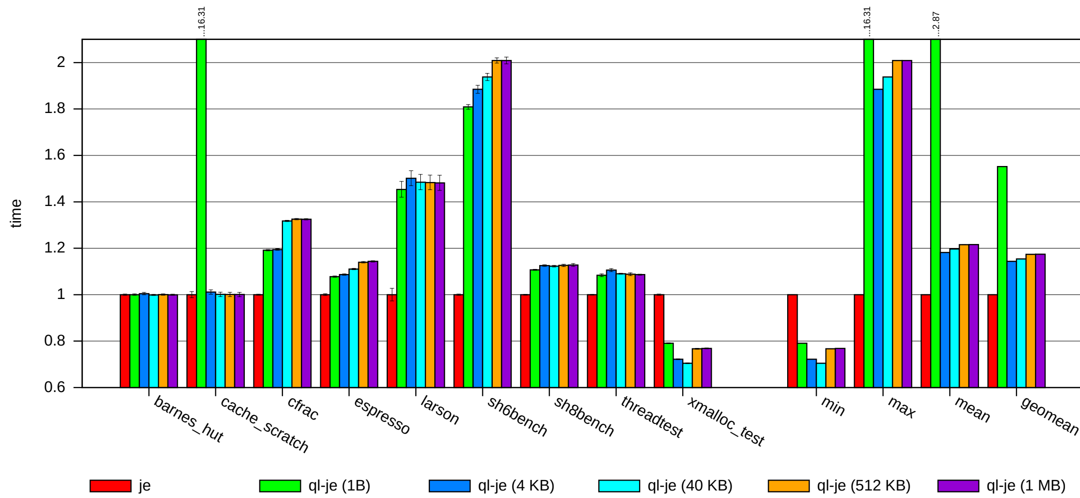


Figure 6.5: jemalloc average execution time over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `je` configuration. A lower value is better.

and sh8bench have much larger space-overheads for larger values of `QL_SIZE`. We can attribute this to the fact that, for *configurations (v)* and *(vi)*, most collections in these benchmarks are capacity-collections. We would have expected to see a more gradual step-like pattern due to the buffer size increasing (as we see in threadtest), however it seems like these benchmarks allocate a lot of objects which can quickly fill up the quarantine list. We note that the space-overheads for *configurations (v)* and *(vi)* for threadtest have significantly increased in comparison to the 40 KB case. We can attribute this to less frequent collections because of the larger buffer size.

Interestingly, we note that the execution time-overheads for *configurations (v)* and *(vi)* for sh8bench and threadtest are slighly larger than for other quarantine configurations. We believe this can be explained by the fact that each collection takes longer and is generally more expensive, given the larger buffer size.

6.5.2 jemalloc

Figures 6.5 and 6.6 show us the average execution time and maximum RSS (respectively) reported over 20 runs with a quarantine thread-local buffer size of 40 KB for the jemalloc allocator with the previously mentioned six configurations.

For the execution time-overheads, generally, the jemalloc results agree with the trends in the mimalloc results. We note, however, that the overheads are generally more pronounced for jemalloc than for mimalloc (with the exception of sh6bench). Taking cfrac as an example, Figure 6.1 reports a maximum time-overhead of 20% for mimalloc, whereas it is around 30% for jemalloc. We attribute these more pronounced effects of quarantining to different allocator architectures and designs. Interestingly, we see that for sh6bench, the overheads are less than for mimalloc, with the maximum execution time being 2× *configuration (i)*. It seems like the xmalloc-

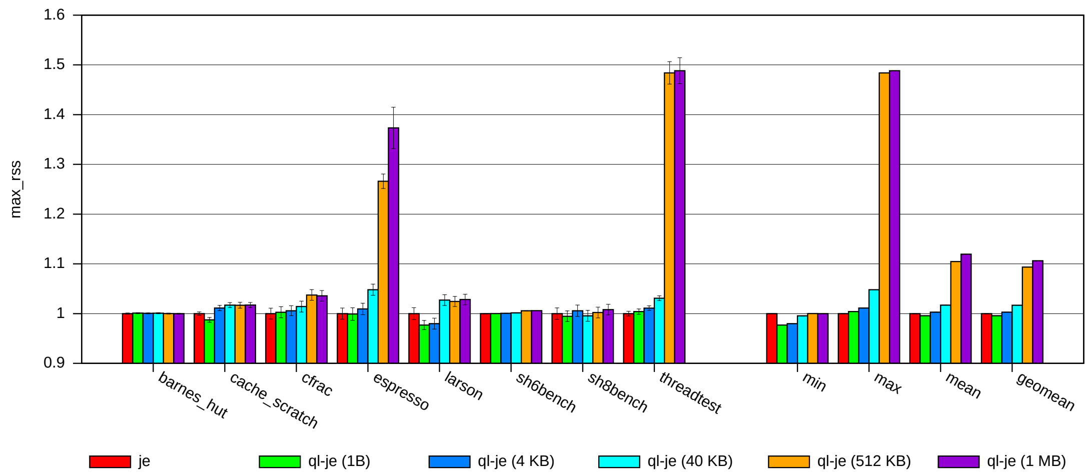


Figure 6.6: jemalloc average maximum RSS over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `je` configuration. A lower value is better.

test benchmark *strongly* favours quarantining since the quarantine configurations are significantly faster than *configuration (i)*.

Of note is that for cache-scratch, *configuration (ii)* has a execution time of 16× *configuration (i)*! Checking with `perf`, we find that *configuration (ii)* has 2× the cache misses, and nearly 7× the branch mispredictions of *configuration (i)*. While these could be symptomatic of the longer runtime (that is to say, we measure more events since we execute for longer), we believe that these branch mispredictions and cache misses do contribute to the overhead. Leijen et al. [2019] find that jemalloc suffers from false cache line sharing when running this benchmark with multiple threads. We believe this is further compounded by the frequent collection operations for *configuration (ii)*.

We generally note that there are no space-overheads between different configurations for jemalloc. The only notable exceptions are espresso and threadtest wherein *configurations (v)* and *(vi)* require much larger heaps. We believe this is due to the fact that most collections for these benchmarks are capacity-collections (as discussed in Section 6.5.1), leading to fewer total collections.

Figures A.19 and A.20 show us the average execution time and maximum RSS (respectively) reported over 20 runs with a quarantine thread-local buffer size of 128 KB for the jemalloc allocator with the previously mentioned six configurations.

We see that the trends for the execution time-overhead are near identical to the 40 KB case. The only notable differences are the values for *configurations (v)* and *(vi)* for sh6bench are slightly larger. We attribute this to the fact that (capacity-)collections are more expensive due to the larger buffer size.

The trends in the heap sizes for the different configurations continue from the 40 KB case. Notably, for *configuration (v)*, espresso and threadtest space-overheads have stabilized, that is to say, most of the collections are now volume-collections instead of capacity-collections. We note that for *configuration (vi)*, most collections for threadtest

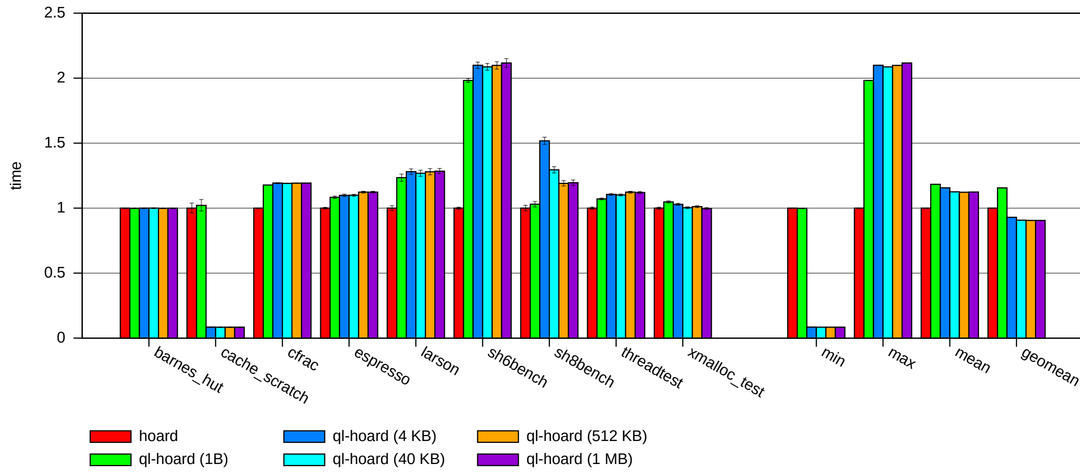


Figure 6.7: hoard average execution time over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `hoard` configuration. A lower value is better.

are still capacity-collections, whereas around two-thirds of the collections are now volume-collections for espresso. Interestingly, we see that the space-overheads for benchmarks such as cfrac and larson are more prominent for *configuration (v)* and *(vi)*. We find that most collections are still capacity-collections for these benchmarks explaining their larger heap sizes.

6.5.3 hoard

Figures 6.7 and 6.8 show us the average execution time and maximum RSS (respectively) reported over 20 runs with a quarantine thread-local buffer size of 40 KB for the hoard allocator with the previously mentioned six configurations.

We note that the execution time-overheads for hoard, while generally following the trend of the previous two allocators, differs significantly for the cache-scratch and sh8bench benchmarks. The execution time for the cache-scratch benchmark for *configurations (iii)-(vi)* are nearly 90% faster than for *configurations (i)-(ii)*. We believe this is due to the nature of the benchmark as well as the deferred freeing. The cache-scratch benchmark allocates multiple small objects and passes them onto each thread. The threads immediately free the object and then allocates another object, accessing the new object multiple times. Given we don't free objects immediately, and hence can't immediately reuse its free slot to allocate a new object, we find that this greatly helps in avoiding this passive false sharing case.

sh8bench, on the other hand, displays an interesting trend. Here, *configuration (iii)* has a significantly large execution time-overhead, more so than the other quarantine configurations. We do not have a conclusive explanation for this behaviour, but we believe it may have to do with the periodicity of collections. Note that even though *configuration (ii)* has more collections, it performs much less work each collection. Hence, this behaviour may also depend on the amount of work per collection

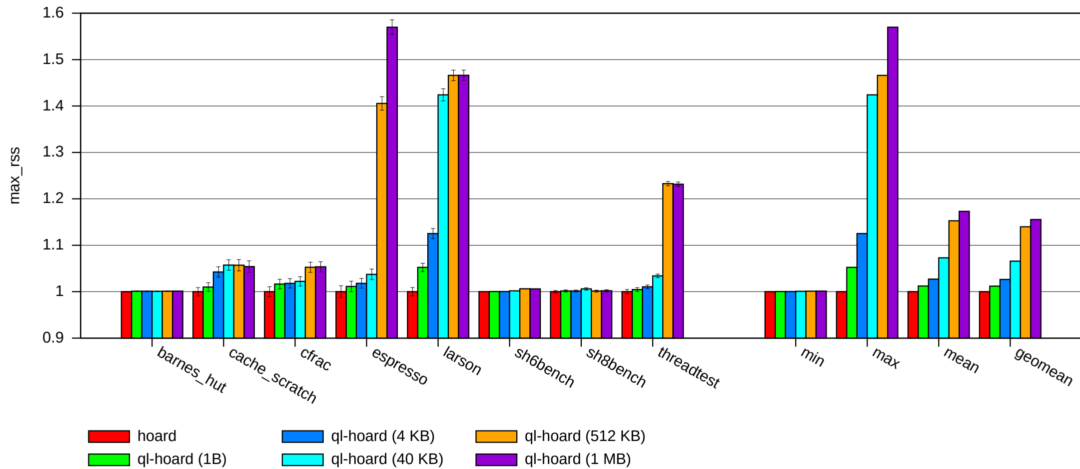


Figure 6.8: hoard average maximum RSS over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `hoard` configuration. A lower value is better.

as well. Interestingly, we see that the execution time for xmalloc-test for the quarantine configurations are comparable to *configuration (i)*, going against trends set by both mimalloc and jemalloc. Leijen et al. [2019] note that most industrial allocators perform very poorly for the xmalloc-test benchmark. We believe these architectural and design issues sneak into the quarantine configurations as well resulting in sub-optimal performance.

The space-overheads for hoard follow the trends of the previous two allocators. The only benchmark where they notably vary is larson. While the quarantine configurations for mimalloc and jemalloc did not have any significant space-overheads for larson, we see that *configurations (iv)-(vi)* for hoard have $>$ 40% space-overheads in comparison to *configuration (i)*.

Figures A.21 and A.22 show us the average execution time and maximum RSS (respectively) reported over 20 runs with a quarantine thread-local buffer size of 128 KB for the hoard allocator with the previously mentioned six configurations. We note no appreciable differences between the 128 KB and 40 KB case for the hoard allocator, for both the time- and space-overheads. We note that the 128 KB case has larger space-overheads for *configurations (v)* and *(vi)* for a few benchmarks such as espresso, larson, and threadtest. The larger space-overheads can be explained by the fact that these benchmarks primarily perform capacity-collections in the 40 KB case. Notably, larson still primarily performs capacity-collections for the 128 KB case.

6.5.4 ptmalloc2

Figures 6.9 and 6.10 show us the average execution time and maximum RSS (respectively) reported over 20 runs with a quarantine thread-local buffer size of 40 KB for the glibc allocator with the previously mentioned six configurations.

The quarantine configurations follow a more prominent step-like pattern for the

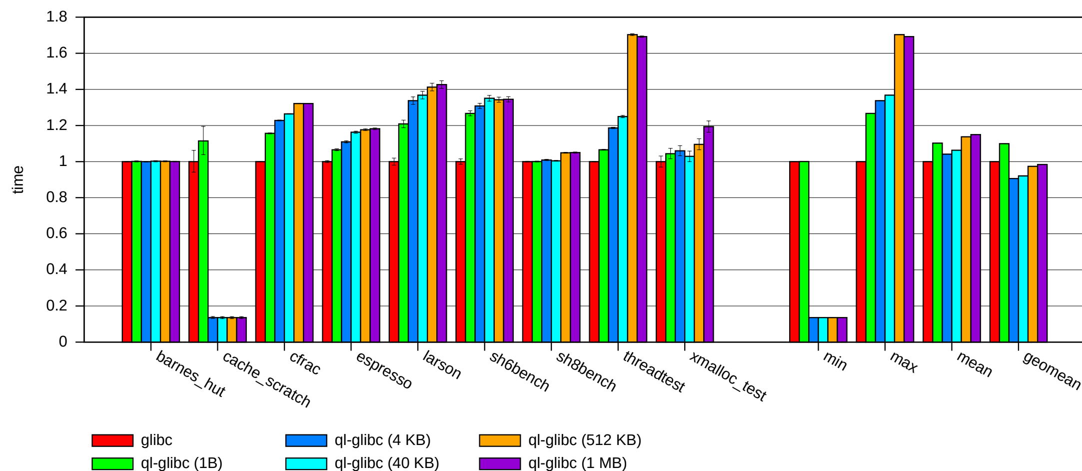


Figure 6.9: glibc average execution time over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `glibc` configuration. A lower value is better.

time-overheads for the glibc allocator in comparison to the previous three allocators. We can immediately see that the glibc configurations follow the hoard configurations' trend for the cache-scratch benchmark. We believe the explanation of this behaviour for the hoard configurations can also be applied here (see Section 6.5.3). Interestingly, we find that the time-overheads for threadtest are the same for *configurations (v)* and *(vi)*. We believe this is due to both configurations primarily performing capacity-collections which results in an equal number of collections (and hence similar overheads) for both configurations.

The space-overheads for the glibc configurations generally conform to what we have previously seen with the three other allocators. Of note, however, is that benchmarks such as cache-scratch, cfrac, and larson have much larger space-overheads than for other allocators.

Figures A.23 and A.24 show us the average execution time and maximum RSS (respectively) reported over 20 runs with a quarantine thread-local buffer size of 128 KB for the glibc allocator with the previously mentioned six configurations. We note no significant differences between the 128 KB and 40 KB case for the glibc allocator, for both the time- and space-overheads. We note that the 128 KB case has larger time-overheads for *configuration (vi)* and *(vi)* for a few benchmarks such as sh6bench and threadtest. This could be explained by capacity-collections being more expensive than the 40 KB case. We further note that the 128 KB case has space-overheads for *configurations (v)* and *(vi)* for benchmarks such as larson and threadtest. These benchmarks primarily perform capacity-collections in the 40 KB case which could explain why there is a larger space-overhead for the 128 KB case. Notably, larson still primarily performs capacity-collections for the 128 KB case for both the configurations.

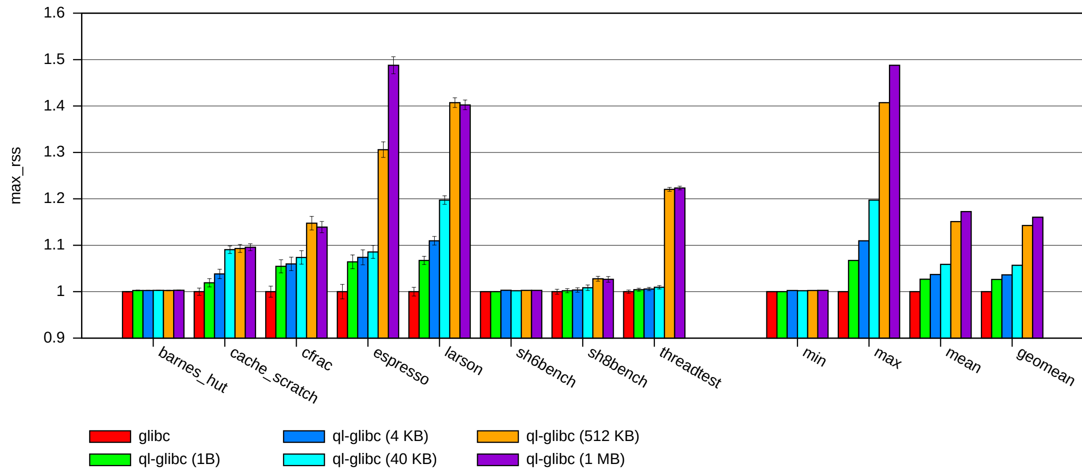


Figure 6.10: glibc average maximum RSS over 20 runs with a thread-local buffer size of 40 KB and with six different configurations. The values are normalized to the `glibc` configuration. A lower value is better.

6.5.5 Discussion

Coalescing results across the different allocators and configurations, we find that, for the configurations we chose, quarantining generally adds modest space-overheads ($\leq$ 5% on average) but considerable time-overheads ($\leq$ 20% on average). We find that these overheads also strongly depend on the backing allocator, with mimalloc generally being the best-pick allocator for quarantining in terms of both (raw) space- and time-overheads. We believe a `QL_SIZE` of 4 KB or 40 KB and the default thread-local buffer size of 40 KB are a good compromise to minimize the space- and time-overheads for quarantining.

6.6 Summary

In this chapter we demonstrated a simple, yet effective, technique to insert the inhale-exhale behaviour of garbage collection into manually memory managed applications using *deferred freeing* or *quarantining*. We implement an allocator-agnostic library in C that inserts inhale-exhale behaviour to applications at runtime. We test our implementation using a variety of different production allocators and find that our approach generally adds a modest space-overhead of $\leq$ 5% and a significant time-overhead of $\leq$ 20% on average. We find the worst time-overheads widely vary for different allocator architectures. We find that most of this time-overhead comes from allocating, accessing, and deallocating internal data structures in our implementation. We find that quarantining can sometimes help avoid passive false sharing since we do not immediately free dead objects to reuse their slots.

Chapter 7

Conclusion

The costs and benefits of garbage collection in comparison to manually managed applications is a hard problem. In this thesis we introduced various novel methodologies for systematically examining and understanding the effects of garbage collection on an application's behaviour. We outlined four experiments which investigate: (i) the space-overheads of garbage collection; (ii) the effects of garbage collection on the execution of an application; (iii) the effects of garbage collection on an application's locality; and finally (iv) the effects of inserting garbage collection-like behaviour in manually memory managed applications.

Our thesis deepens the understanding of garbage collection behaviour in managed languages as well as in comparison to manual memory management. We find that classic GC algorithms such as SemiSpace and Appel-style Generational Copying collectors in general have a 1.75× space-overhead in comparison to an approximation of manual memory management, while Immix is competitive to this approximation with a 1.15× space-overhead on average. We draw parallels between garbage collection behaviour and interference in signal processing and propose a novel methodology that decouples the costs from the benefits of garbage collection in order to understand how garbage collection affects an application's execution. We measure locality effects of garbage collection using a transaction-based benchmark as a case-study and find that there is a weak correlation between the execution time of a transaction and its proximity to the execution of a GC. Finally, we insert the "inhale-exhale" behaviour of garbage collection to manually memory managed applications and find that the behaviour has modest space-overheads but can have significant time-overheads for certain applications. We also find that the worst time-overheads widely vary for different allocator architectures.

7.1 Future Work

We point out four avenues of future work based on our experiments.

7.1.1 Performance Evaluation on Different Microarchitectures

We would like to run our experiments with more hardware configurations in order to understand and contrast garbage collection behaviour on different microarchitec-

tures. This could reveal interesting performance tradeoffs and microarchitectural idiosyncrasies.

7.1.2 Time Overheads of Garbage Collection

With the benefit of hindsight, we find that using the `malloc` Mark-Sweep in MMTk for estimating the time-overheads of garbage collection in Chapter 4 was a poor choice due to its considerable overheads for allocating objects. We believe this methodology has a lot of potential to unlock deeper understandings of garbage collection behaviour. In the future we would like to revisit this methodology by implementing a way to decouple the costs and benefits of garbage collection in algorithms such as SemiSpace, Immix, or Mark-Sweep with a native freelist allocator.

7.1.3 Locality Effects of Garbage Collection

For our experiment which measured locality effects of garbage collection in Chapter 5, we only measure the execution time of queries. We would like to measure other statistics as well such as using hardware performance counters to measure cache and branch prediction misses per query. This could give us more insight regarding how the machine state and locality are affected due to the presence of garbage collection.

7.1.4 Garbage Collection Behaviour in an Unmanaged Context

In Chapter 6, we find that most time-overhead for quarantining comes from allocating, accessing, and deallocating the thread-local buffer. We could investigate ways to reduce these overheads by potentially being more branch and cache friendly. We believe if we can reduce the time-overhead associated with quarantining, we could see performance benefits for manually memory managed applications with only a modest space-overhead. We also would like to run our quarantining experiment with more allocators (such as tcmalloc [Google, 2014], snmalloc [Liétar et al., 2019], etc.) and more benchmarks in order to see how quarantining affects the space- and time-overheads of these allocators.

Bibliography

AINSWORTH, S. AND JONES, T. M., 2020. Markus: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, 578–591. IEEE. doi:10.1109/SP40000.2020.00058. https://doi.org/10.1109/SP40000.2020.00058. (cited on pages 35 and 36)

APACHE, 2021. Apache Lucene. https://lucene.apache.org/. (cited on page 26)

BARNES, J. H. AND HUT, P., 1986. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324 (1986), 446–449. (cited on page 38)

BERGER, E. D.; MCKINLEY, K. S.; BLUMOFE, R. D.; AND WILSON, P. R., 2000. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000*, 117–128. ACM Press. doi:10.1145/378993.379232. https://doi.org/10.1145/378993.379232. (cited on pages 10, 11, 36, and 38)

BLACKBURN, S. AND MCKINLEY, K. S., 2003. Ulterior reference counting: fast garbage collection without a long wait. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, 344–358. ACM. doi:10.1145/949305.949336. https://doi.org/10.1145/949305.949336. (cited on pages 35 and 36)

BLACKBURN, S. M.; CHENG, P.; AND MCKINLEY, K. S., 2004a. Myths and realities: the performance impact of garbage collection. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, 25–36. ACM. doi:10.1145/1005686.1005693. https://doi.org/10.1145/1005686.1005693. (cited on pages 3 and 7)

BLACKBURN, S. M.; CHENG, P.; AND MCKINLEY, K. S., 2004b. Oil and water? high performance garbage collection in java with mmtk. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, 137–146. IEEE Computer Society. doi:10.1109/ICSE.2004.1317436. https://doi.org/10.1109/ICSE.2004.1317436. (cited on page 7)

BLACKBURN, S. M.; GARNER, R.; HOFFMAN, C.; KHAN, A. M.; MCKINLEY, K. S.; BENTZUR, R.; DIWAN, A.; FEINBERG, D.; FRAMPTON, D.; GUYER, S. Z.; HIRZEL, M.; HOSKING, A.; JUMP, M.; LEE, H.; MOSS, J. E. B.; PHANSALKAR, A.; STEFANOVIĆ,

D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006a. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications* (Portland, OR, USA, Oct. 2006), 169–190. ACM Press, New York, NY, USA. doi:http://doi.acm.org/10.1145/1167473.1167488. (cited on pages 10, 12, 20, and 26)

Blackburn, S. M.; Garner, R.; Hoffman, C.; Khan, A. M.; McKinley, K. S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006b. The DaCapo Benchmarks: Java benchmarking development and analysis (extended version). Technical Report TR-CS-06-01, ANU. Http://www.dacapobench.org. (cited on pages 10, 12, 20, and 26)

Blackburn, S. M. and McKinley, K. S., 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, 22–32. ACM. doi:10.1145/1375581.1375586. https://doi.org/10.1145/1375581.1375586. (cited on page 6)

Boehm, H. and Weiser, M. D., 1988. Garbage collection in an uncooperative environment. *Softw. Pract. Exp.*, 18, 9 (1988), 807–820. doi:10.1002/spe.4380180902. https://doi.org/10.1002/spe.4380180902. (cited on page 2)

Cheney, C. J., 1970. A nonrecursive list compacting algorithm. *Commun. ACM*, 13, 11 (1970), 677–678. doi:10.1145/362790.362798. https://doi.org/10.1145/362790.362798. (cited on page 6)

Collins, G. E., 1960. A method for overlapping and erasure of lists. *Commun. ACM*, 3, 12 (1960), 655–657. doi:10.1145/367487.367501. https://doi.org/10.1145/367487.367501. (cited on page 2)

Cooper, B. F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; and Sears, R., 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, 143–154. ACM. doi:10.1145/1807128.1807152. https://doi.org/10.1145/1807128.1807152. (cited on page 22)

CWE, 2021. 2021 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html. (cited on pages xvii, 1, and 2)

Daan Leijen, 2021. mimalloc-bench: Suite for benchmarking malloc implementations. https://github.com/daanx/mimalloc-bench. (cited on page 38)

DEUTSCH, L. P. AND BOBROW, D. G., 1976. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19, 9 (1976), 522–526. doi:10.1145/360336.360345. https://doi.org/10.1145/360336.360345. (cited on pages 35 and 36)

DOUG LEA, 1998. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html. (cited on pages 8 and 15)

EVANS, J., 2006. Jemalloc. In *Proceedings of the 2006 BSDCan Conference, BSDCan'06, May 2006, Ottawa, CA*. http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf. (cited on pages 10, 11, and 36)

GOOGLE, 2014. tcmalloc. https://github.com/gperftools/gperftools. (cited on page 52)

GRUNWALD, D.; ZORN, B. G.; AND HENDERSON, R., 1993. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, 177–186. ACM. doi:10.1145/155090.155107. https://doi.org/10.1145/155090.155107. (cited on page 38)

HERTZ, M. AND BERGER, E. D., 2005. 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, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, 313–326. ACM. doi:10.1145/1094811.1094836. https://doi.org/10.1145/1094811.1094836. (cited on pages 2, 8, 9, and 15)

HUANG, X.; BLACKBURN, S. M.; MCKINLEY, K. S.; MOSS, J. E. B.; WANG, Z.; AND CHENG, P., 2004. The garbage collection advantage: improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, 69–80. ACM. doi:10.1145/1028976.1028983. https://doi.org/10.1145/1028976.1028983. (cited on pages 2 and 7)

JAMES TRICHILO. Simple Denoising Methods. Part I: Signal and Image Noise Models. http://mason.gmu.edu/~jtrichil/d1. (cited on pages xiii and 18)

JIBAJA, I.; BLACKBURN, S. M.; HAGHIGHAT, M. R.; AND MCKINLEY, K. S., 2011. Deferred gratification: engineering for high performance garbage collection from the get go. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness: held in conjunction with PLDI '11, San Jose, CA, USA, June 5, 2011*, 58–65. ACM. doi:10.1145/1988915.1988930. https://doi.org/10.1145/1988915.1988930. (cited on pages 2 and 3)

JONES, R. E.; HOSKING, A. L.; AND MOSS, J. E. B., 2011. *The Garbage Collection Handbook: The art of automatic memory management*. Chapman and Hall / CRC Applied Algorithms and Data Structures Series. CRC Press. ISBN 978-1-4200-8279-1. http://gchandbook.org/. (cited on page 2)

Larson, P. and Krishnan, M., 1998. Memory allocation for long-running server applications. In *International Symposium on Memory Management, ISMM '98, Vancouver, British Columbia, Canada, 17-19 October, 1998, Conference Proceedings*, 176–185. ACM. doi:10.1145/286860.286880. https://doi.org/10.1145/286860.286880. (cited on page 38)

Lattner, C., 2016. Swift: Challenges and Opportunity for Language and Compiler Research. https://researcher.watson.ibm.com/researcher/files/us-lmandel/lattner.pdf. (cited on page 2)

Leijen, D.; Zorn, B.; and de Moura, L., 2019. Mimalloc: Free list sharding in action. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*, vol. 11893 of *Lecture Notes in Computer Science*, 244–265. Springer. doi:10.1007/978-3-030-34175-6_13. https://doi.org/10.1007/978-3-030-34175-6_13. (cited on pages 10, 11, 36, 45, and 47)

Lieberman, H. and Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26, 6 (1983), 419–429. doi:10.1145/358141.358147. https://doi.org/10.1145/358141.358147. (cited on page 6)

Liétar, P.; Butler, T.; Clebsch, S.; Drossopoulou, S.; Franco, J.; Parkinson, M. J.; Shamis, A.; Wintersteiger, C. M.; and Chisnall, D., 2019. snmalloc: a message passing allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management, ISMM 2019, Phoenix, AZ, USA, June 23-23, 2019*, 122–135. ACM. doi:10.1145/3315573.3329980. https://doi.org/10.1145/3315573.3329980. (cited on page 52)

Lin, Y.; Blackburn, S. M.; Hosking, A. L.; and Norrish, M., 2016. Rust as a language for high performance GC implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management, Santa Barbara, CA, USA, June 14 - 14, 2016*, 89–98. ACM. doi:10.1145/2926697.2926707. https://doi.org/10.1145/2926697.2926707. (cited on page 7)

McCarthy, J., 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3, 4 (1960), 184–195. doi:10.1145/367177.367199. https://doi.org/10.1145/367177.367199. (cited on page 5)

MMTk Research Group, 2021. The Memory Management Toolkit. https://www.mmtk.io/. (cited on pages 3 and 7)

.NET Platform, 2021. runtime/gc.cpp. https://github.com/dotnet/runtime/blob/main/src/coreclr/gc/gc.cpp. (cited on page 6)

OpenJDK Community, 2021. HotSpot Group. https://openjdk.java.net/groups/hotspot/. (cited on page 3)

Paige Reeves, 2021. MallocMS. https://www.mmtk.io/assets/videos/summer-2021-reeves.mp4. (cited on pages 10 and 11)

PHP Community Foundation, 2019. *PHP Language Specification*. https://phplang.org. (cited on page 3)

Ruby Programming Language, 2021. ruby/gc.c. https://github.com/ruby/ruby/blob/master/gc.c. (cited on page 6)

Shahriyar, R.; Blackburn, S. M.; and Frampton, D., 2012. Down for the count? getting reference counting back in the ring. In *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, 73–84. ACM. doi:10.1145/2258996.2259008. https://doi.org/10.1145/2258996.2259008. (cited on page 2)

Steve Blackburn, 2021. The DaCapo Benchmark Suite git hash 69a704e. https://github.com/dacapobench/dacapobench/commit/69a704ef4436196295e8e107f0a5f5f5212c125a. (cited on pages 12 and 20)

Ungar, D. M., 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*, 157–167. ACM. doi:10.1145/800020.808261. https://doi.org/10.1145/800020.808261. (cited on page 6)

Wang, M., 2017. The Future of HHVM. https://hhvm.com/blog/2017/09/18/the-future-of-hhvm.html. (cited on pages 2 and 3)

Wikipedia, 2021. Differential Signaling. https://commons.wikimedia.org/wiki/File:Differential_signal_transmission_over_balanced_line.svg. (cited on pages xiii and 19)

Yamauchi, O., 2012. On Garbage Collection. https://hhvm.com/blog/431/on-garbage-collection. (cited on page 3)

Yang, X.; Blackburn, S. M.; and McKinley, K. S., 2016. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, 309–322. USENIX Association. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yang. (cited on page 27)

Figures

Time Overheads of Garbage Collection

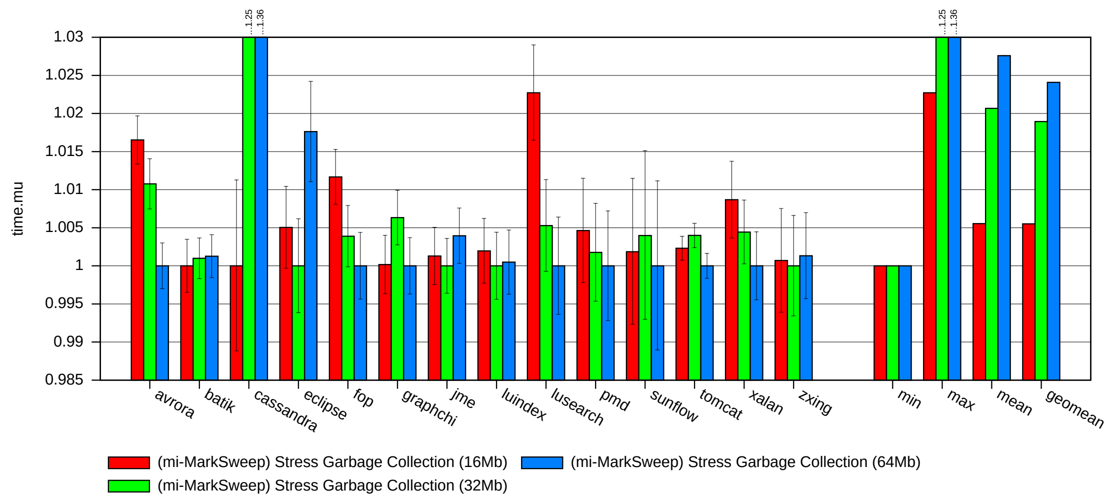


Figure A.1: Mutator execution time (normalized to best value) averaged over 30 runs for a GC limit of 4 using the mimalloc Mark-Sweep collector with three different stress factor values. A lower value is better.

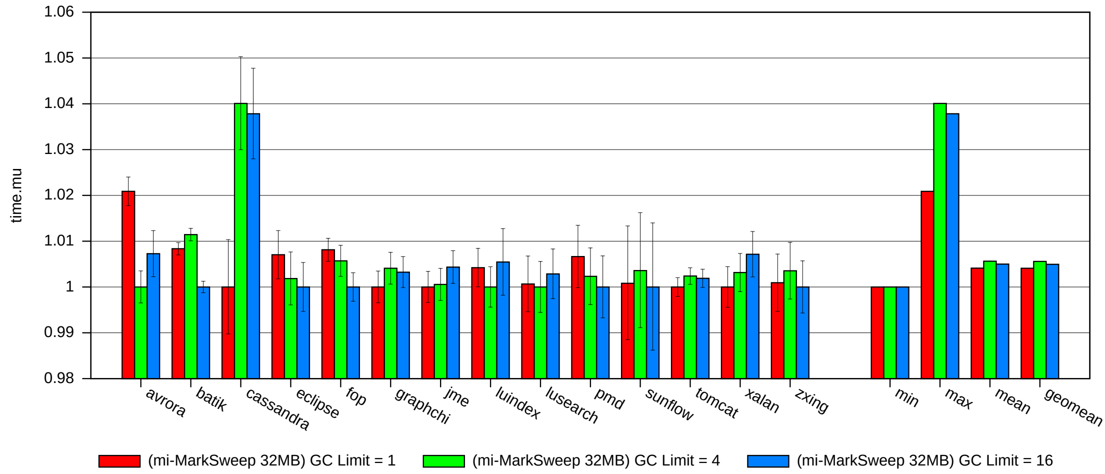


Figure A.2: Mutator execution time (normalized to best value) averaged over 30 runs for the mimalloc Mark-Sweep collector with a stress factor of 32MB and three different GC limit values. A lower value is better.

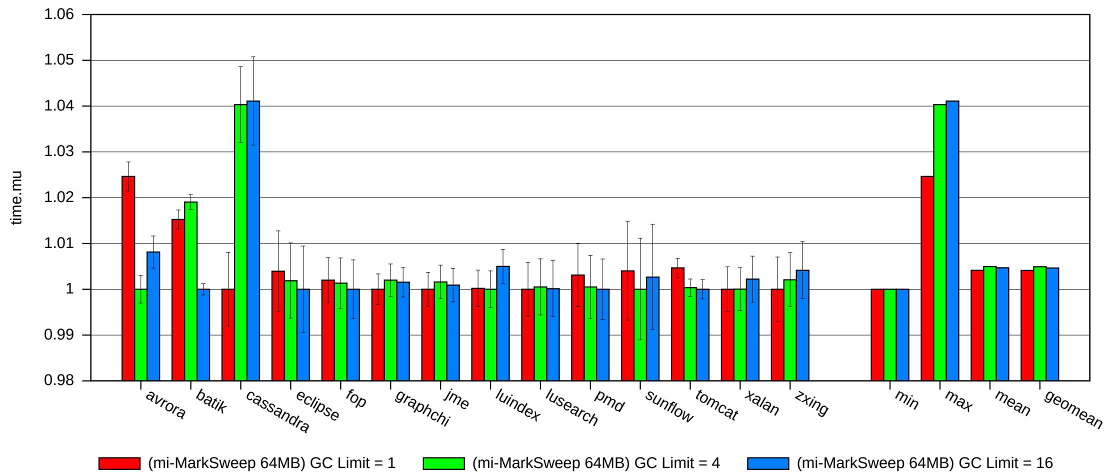
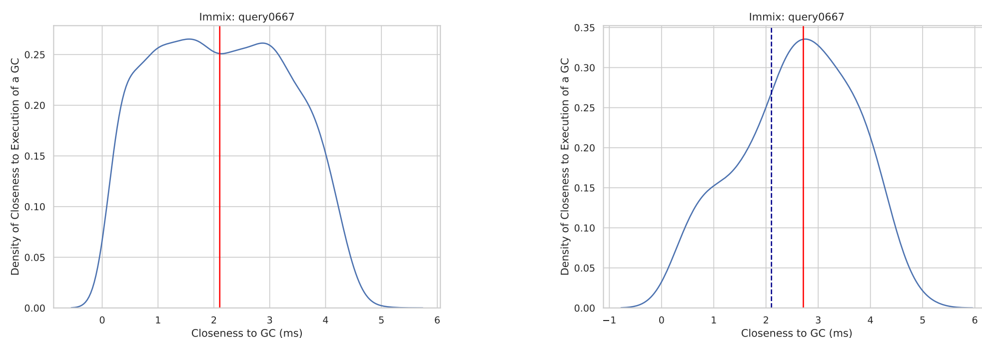


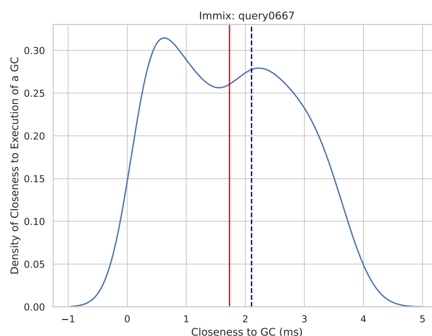
Figure A.3: Mutator execution time (normalized to best value) averaged over 30 runs for the mimalloc Mark-Sweep collector with a stress factor of 64MB and three different GC limit values. A lower value is better.

Locality Effects of Garbage Collection



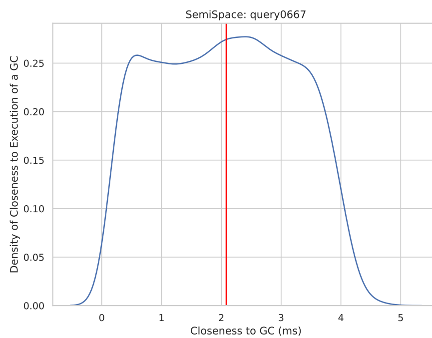
(a) Density of closeness to a GC for all executions

(b) Density of closeness to a GC for the best 5th percentile of executions

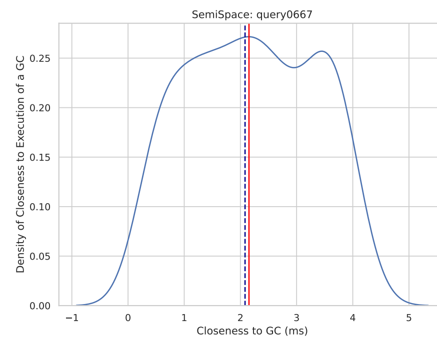


(c) Density of closeness to a GC for the worst 5th percentile of executions

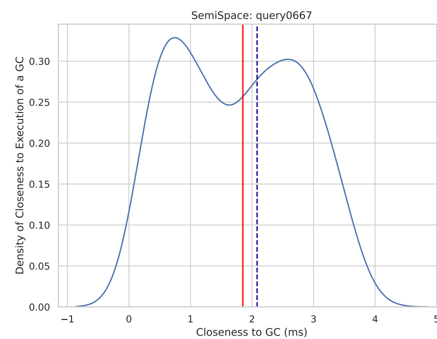
Figure A.4: Density of closeness to a GC for Query 0667 using the Immix collector on the Haswell system.



(a) Density of closeness to a GC for all executions

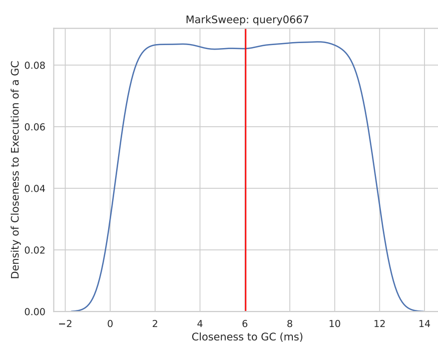


(b) Density of closeness to a GC for the best 5th percentile of executions

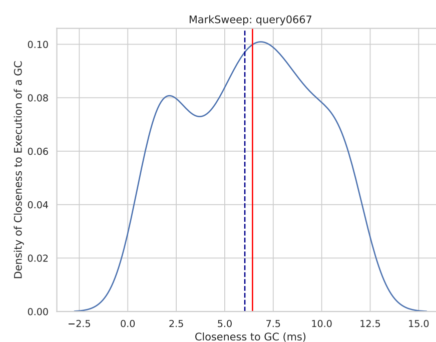


(c) Density of closeness to a GC for the worst 5th percentile of executions

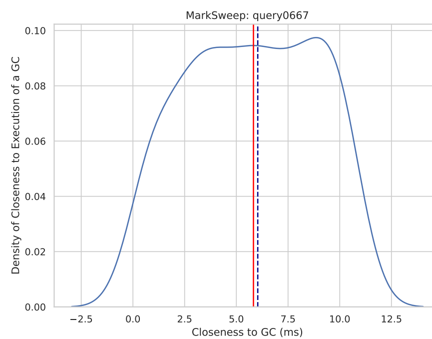
Figure A.5: Density of closeness to a GC for Query 0667 using the SemiSpace collector on the Haswell system.



(a) Density of closeness to a GC for all executions

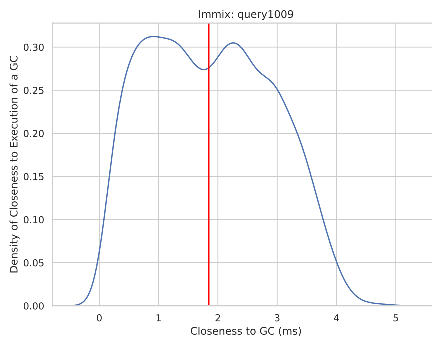


(b) Density of closeness to a GC for the best 5th percentile of executions

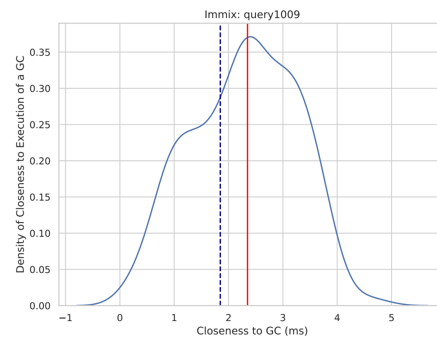


(c) Density of closeness to a GC for the worst 5th percentile of executions

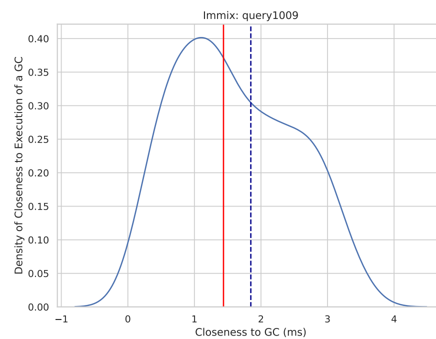
Figure A.6: Density of closeness to a GC for Query 0667 using the Mark-Sweep collector on the Haswell system.



(a) Density of closeness to a GC for all executions

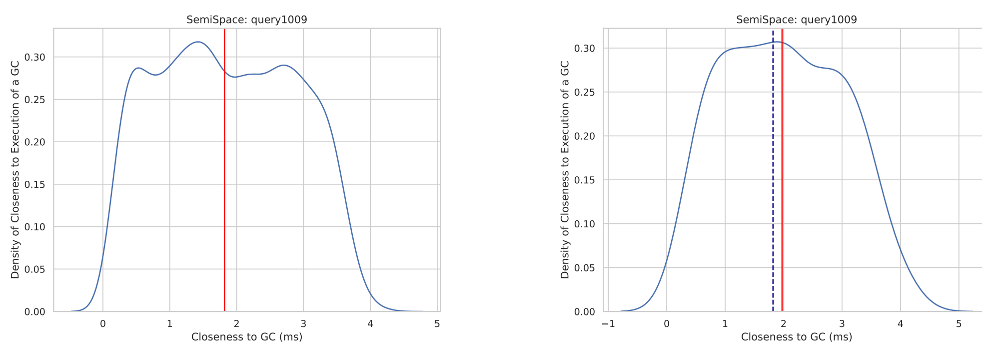


(b) Density of closeness to a GC for the best 5th percentile of executions



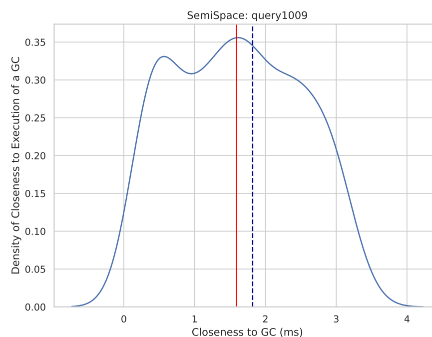
(c) Density of closeness to a GC for the worst 5th percentile of executions

Figure A.7: Density of closeness to a GC for Query 1009 using the Immix collector on the Haswell system.



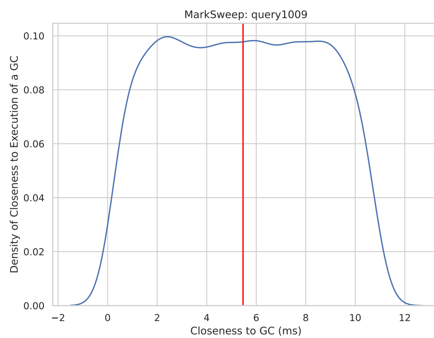
(a) Density of closeness to a GC for all executions

(b) Density of closeness to a GC for the best 5th percentile of executions

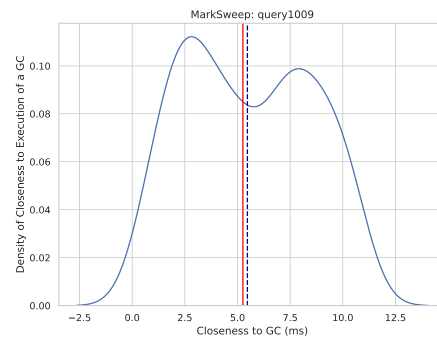


(c) Density of closeness to a GC for the worst 5th percentile of executions

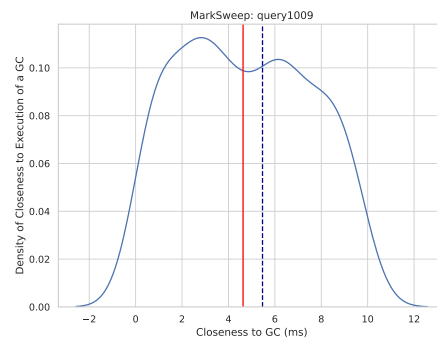
Figure A.8: Density of closeness to a GC for Query 1009 using the SemiSpace collector on the Haswell system.



(a) Density of closeness to a GC for all executions

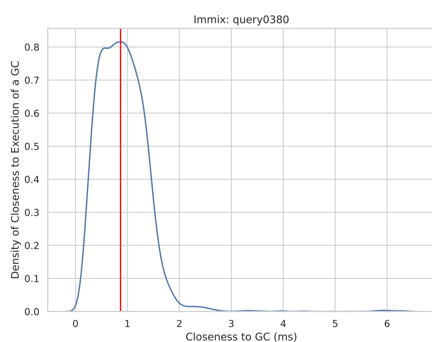


(b) Density of closeness to a GC for the best 5th percentile of executions

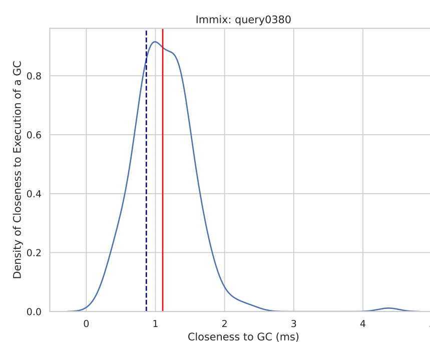


(c) Density of closeness to a GC for the worst 5th percentile of executions

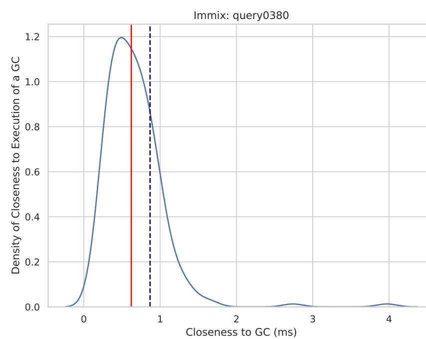
Figure A.9: Density of closeness to a GC for Query 1009 using the Mark-Sweep collector on the Haswell system.



(a) Density of closeness to a GC for all executions

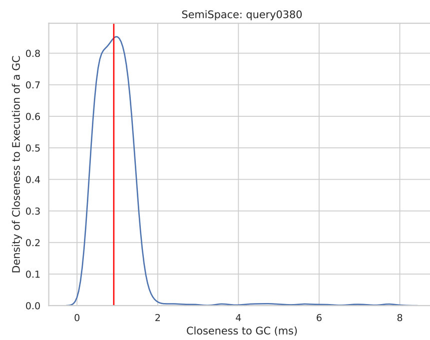


(b) Density of closeness to a GC for the best 5th percentile of executions

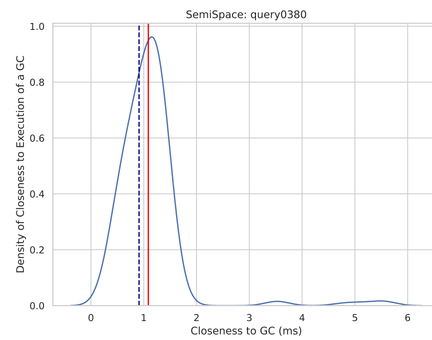


(c) Density of closeness to a GC for the worst 5th percentile of executions

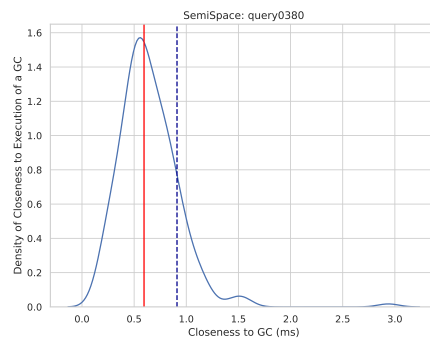
Figure A.10: Density of closeness to a GC for Query 0380 using the Immix collector on the Xeon system.



(a) Density of closeness to a GC for all executions

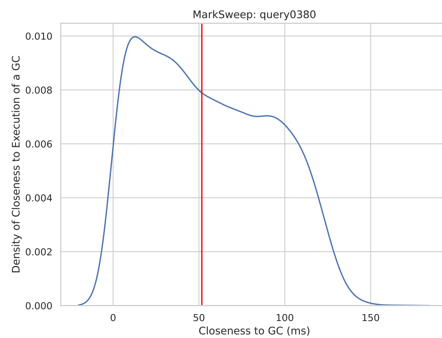


(b) Density of closeness to a GC for the best 5th percentile of executions

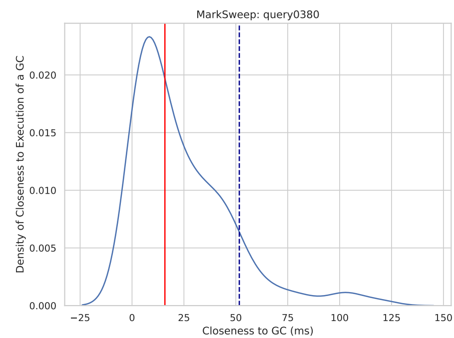


(c) Density of closeness to a GC for the worst 5th percentile of executions

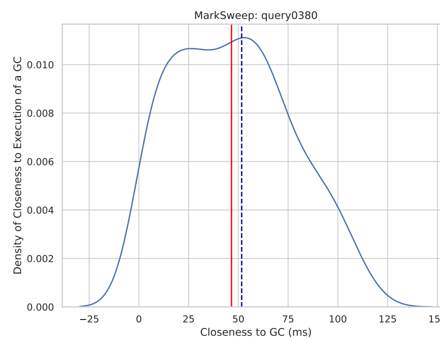
Figure A.11: Density of closeness to a GC for Query 0380 using the SemiSpace collector on the Xeon system.



(a) Density of closeness to a GC for all executions

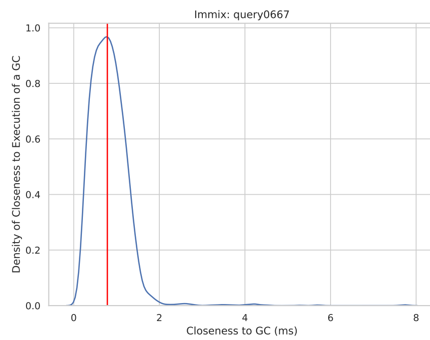


(b) Density of closeness to a GC for the best 5th percentile of executions

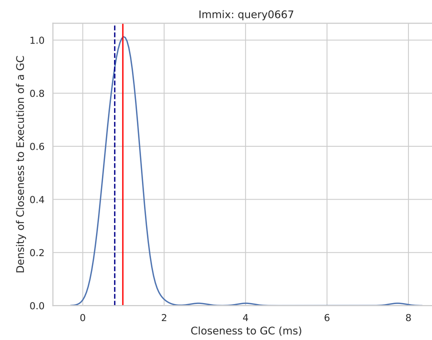


(c) Density of closeness to a GC for the worst 5th percentile of executions

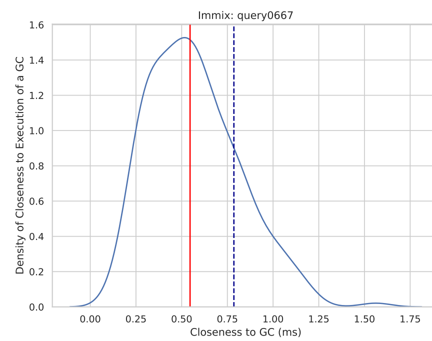
Figure A.12: Density of closeness to a GC for Query 0380 using the Mark-Sweep collector on the Xeon system.



(a) Density of closeness to a GC for all executions

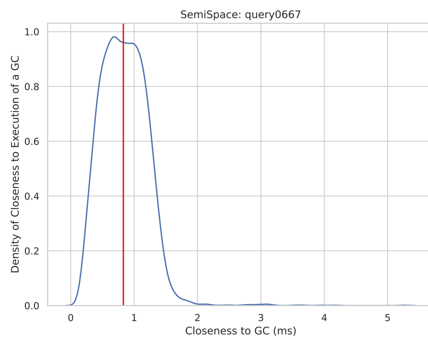


(b) Density of closeness to a GC for the best 5th percentile of executions

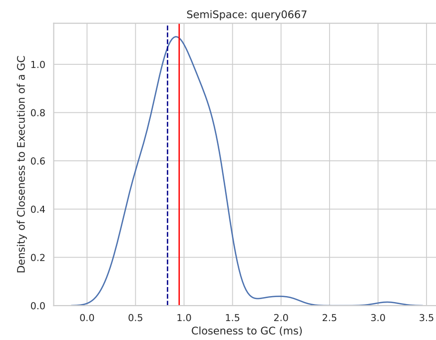


(c) Density of closeness to a GC for the worst 5th percentile of executions

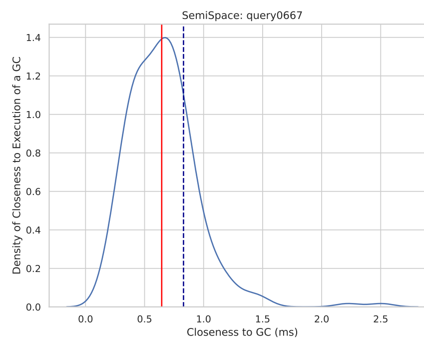
Figure A.13: Density of closeness to a GC for Query 0667 using the Immix collector on the Xeon system.



(a) Density of closeness to a GC for all executions

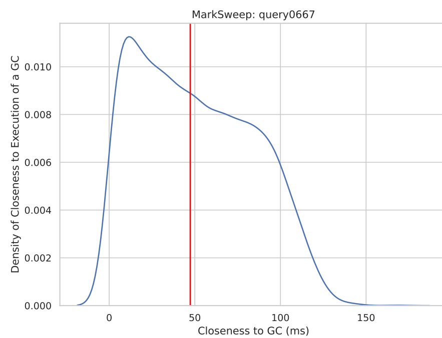


(b) Density of closeness to a GC for the best 5th percentile of executions

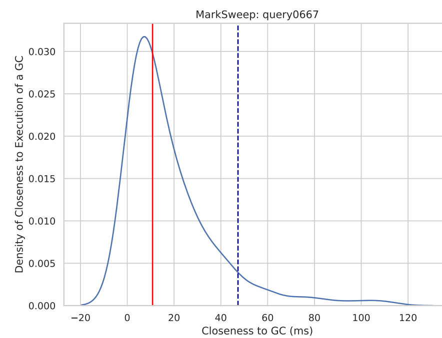


(c) Density of closeness to a GC for the worst 5th percentile of executions

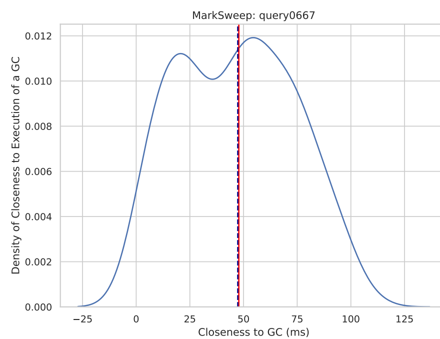
Figure A.14: Density of closeness to a GC for Query 0667 using the SemiSpace collector on the Xeon system.



(a) Density of closeness to a GC for all executions

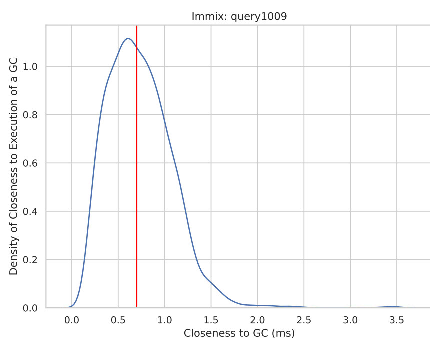


(b) Density of closeness to a GC for the best 5th percentile of executions

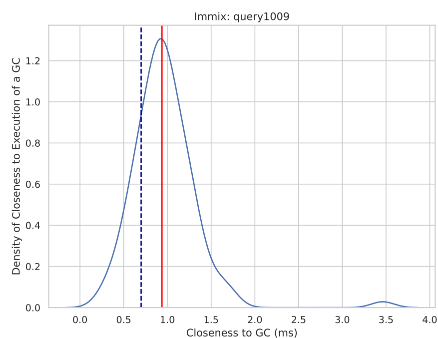


(c) Density of closeness to a GC for the worst 5th percentile of executions

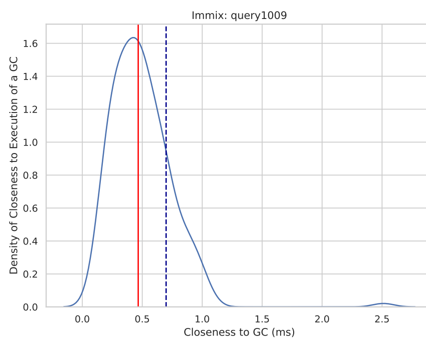
Figure A.15: Density of closeness to a GC for Query 0667 using the Mark-Sweep collector on the Xeon system.



(a) Density of closeness to a GC for all executions

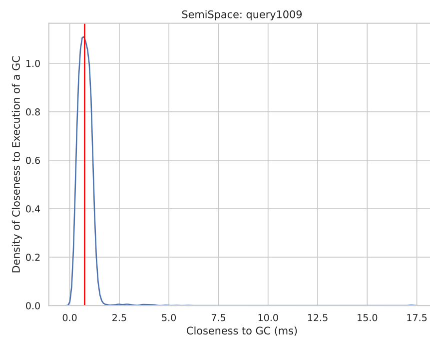


(b) Density of closeness to a GC for the best 5th percentile of executions

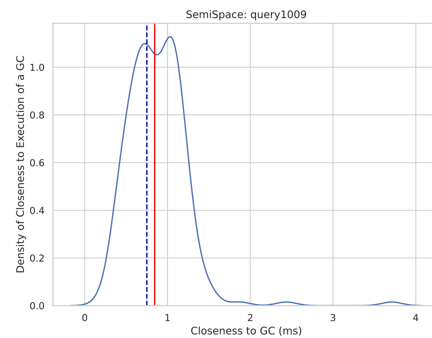


(c) Density of closeness to a GC for the worst 5th percentile of executions

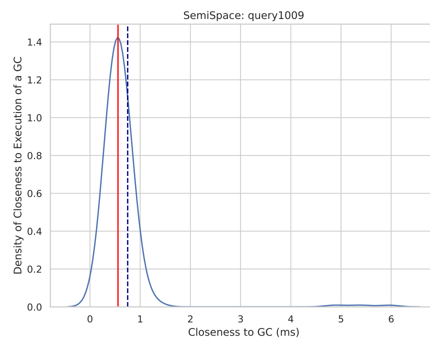
Figure A.16: Density of closeness to a GC for Query 1009 using the Immix collector on the Xeon system.



(a) Density of closeness to a GC for all executions

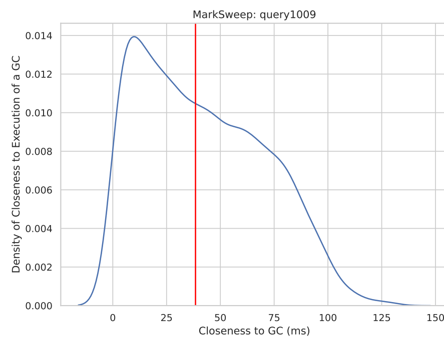


(b) Density of closeness to a GC for the best 5th percentile of executions

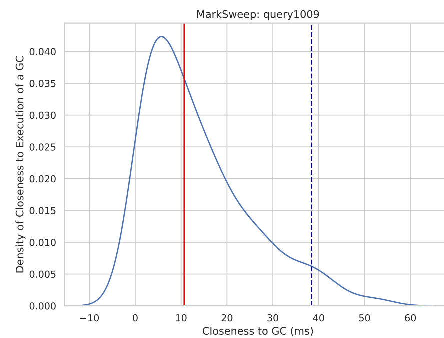


(c) Density of closeness to a GC for the worst 5th percentile of executions

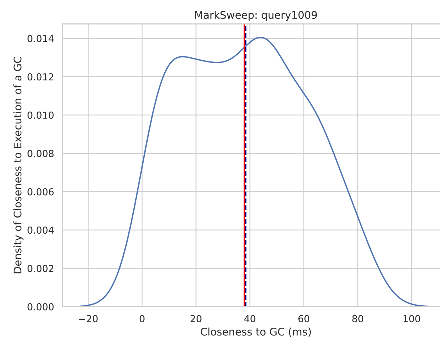
Figure A.17: Density of closeness to a GC for Query 1009 using the SemiSpace collector on the Xeon system.



(a) Density of closeness to a GC for all executions



(b) Density of closeness to a GC for the best 5th percentile of executions



(c) Density of closeness to a GC for the worst 5th percentile of executions

Figure A.18: Density of closeness to a GC for Query 1009 using the Mark-Sweep collector on the Xeon system.

Garbage Collection Behaviour in an Unmanaged Language

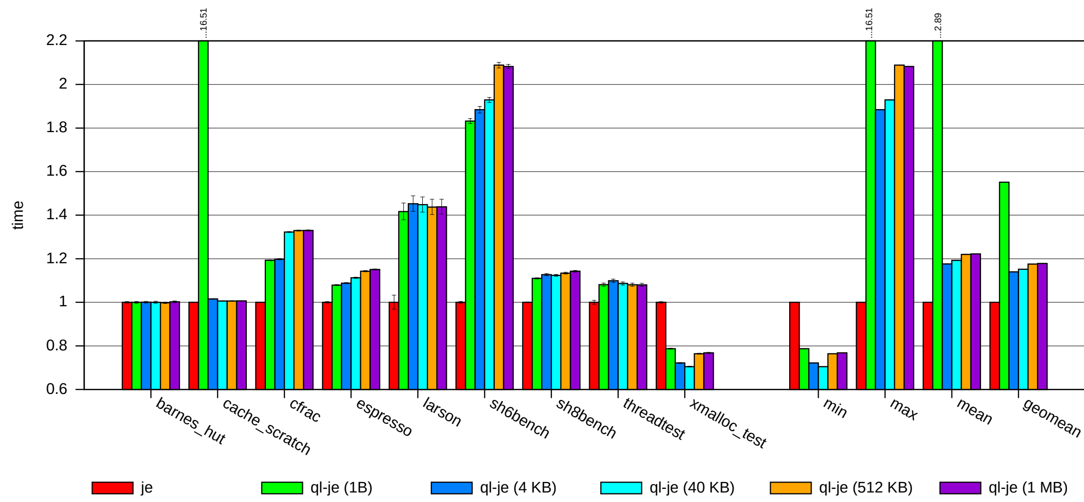


Figure A.19: jemalloc average execution time over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `je` configuration. A lower value is better.

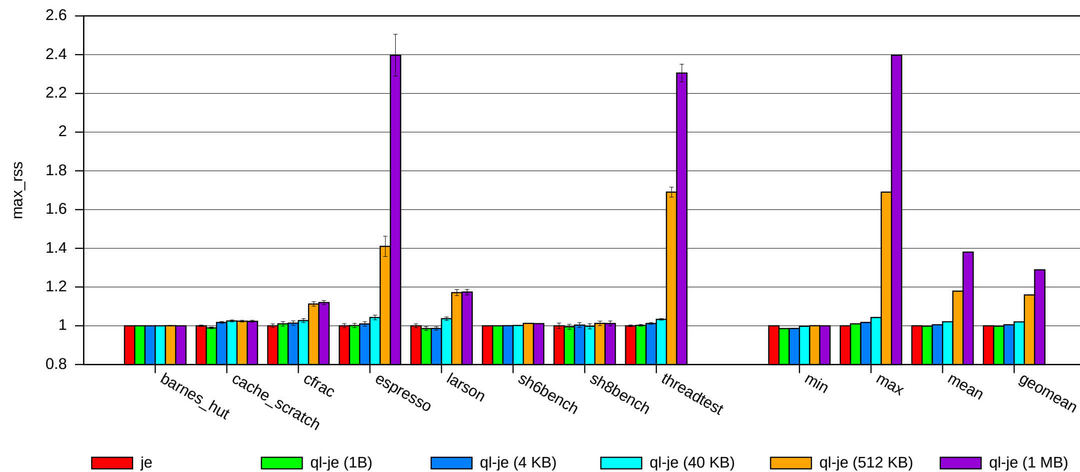


Figure A.20: jemalloc average maximum RSS over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `je` configuration. A lower value is better.

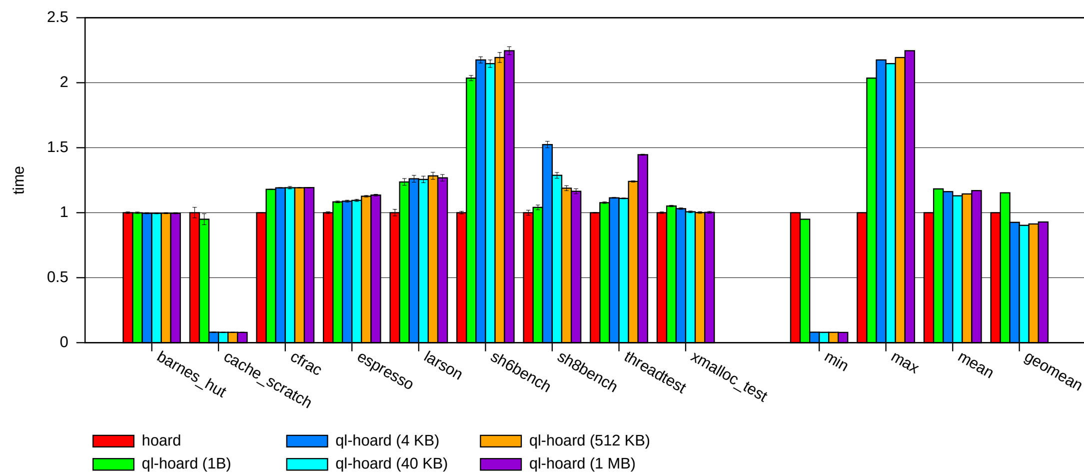


Figure A.21: hoard average execution time over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `hoard` configuration. A lower value is better.

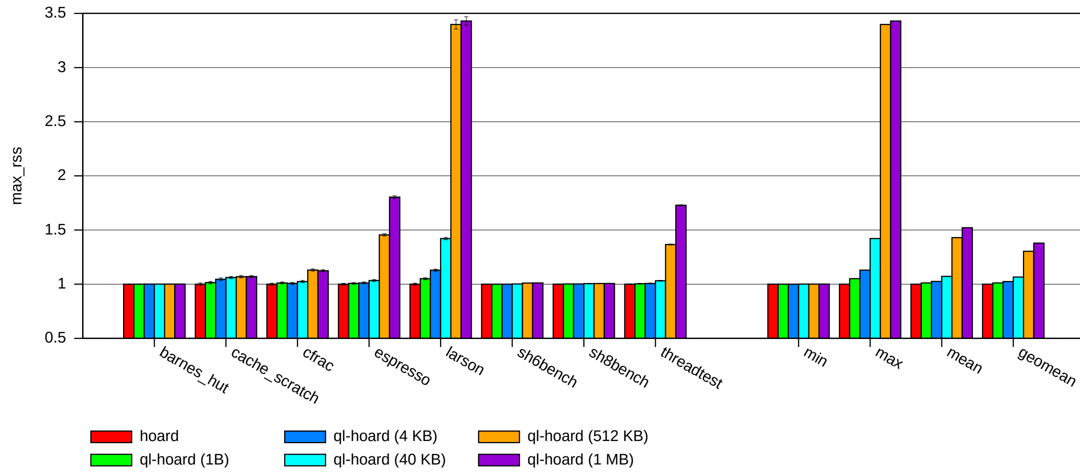


Figure A.22: hoard average maximum RSS over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `hoard` configuration. A lower value is better.

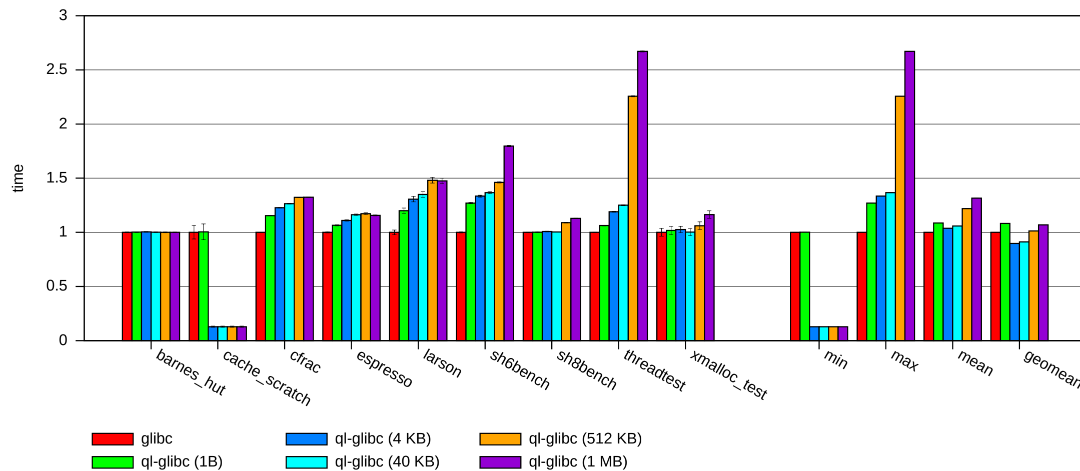


Figure A.23: glibc average execution time over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `glibc` configuration. A lower value is better.

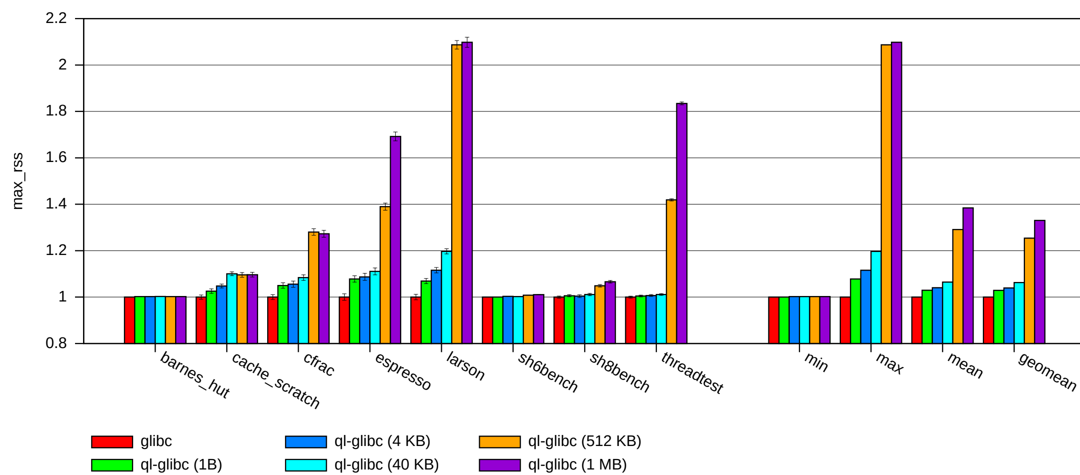


Figure A.24: glibc average maximum RSS over 20 runs with a thread-local buffer size of 128 KB and with six different configurations. The values are normalized to the `glibc` configuration. A lower value is better.