Better Understanding the Costs and Benefits of Automatic Memory Management

Kunal Sareen
kunal.sareen@anu.edu.au
Australian National University
Canberra, Australia

Stephen M. Blackburn
steveblackburn@google.com
Google and Australian National University
Canberra, Australia

ABSTRACT

Automatic memory management relieves programmers of the burden of having to reason about object lifetimes in order to soundly reclaim allocated memory. However, this automation comes at a cost. The cost and benefits of garbage collection relative to manual memory management have been the subject of contention for a long time, and will likely remain so. However, until now, the question is surprisingly under-studied. We examine the costs and benefits of garbage collection through four studies, exploring: (i) the space overheads of garbage collection; (ii) the effects of garbage collection on the execution time of a mutator using a free-list allocator; (iii) how proximity to garbage collection events affects mutator performance; and (iv) the effects of the delay in memory reuse on manually managed workloads. We conduct this study in a contemporary setting using recent CPU microarchitectures, and novel methodologies including a mark-sweep collector built upon off-the-shelf free-list allocators, allowing us to shed new light on garbage collection overheads in a modern context.

We find that: (i) simple, fully-copying collectors such as semi-space have average space overheads of 65-80%, while immix has an overhead of 11-17% over a baseline approximating manual memory management; (ii) for the collection frequencies we evaluate, garbage collection has little impact on mutator time when using an optimized free-list allocator; (iii) the proximity of mutator work to the nearest collection has little impact on its performance, unless a free-list allocator is used; and (iv) postponing the reuse of memory generally has little effect on performance, but is allocator-specific, and is noticeable if the delay is significant relative to the workload's footprint. The costs and benefits of garbage collection are likely to remain subject to contentious discussion. However, the methodologies and evaluations we present here provide a deeper understanding of the differences in costs between manual memory management and garbage collection.

CCS CONCEPTS

• **Software and its engineering → Garbage collection**; **General programming languages**.

MPLR '22, September 14–15, 2022, Brussels, Belgium

ACM ISBN 978-1-4503-9696-7/22/09.
https://doi.org/10.1145/3546918.3546926

KEYWORDS

Garbage Collection, Memory Management, Manual Memory Management, Automatic Memory Management

ACM Reference Format:
Kunal Sareen and Stephen M. Blackburn. 2022. Better Understanding the Costs and Benefits of Automatic Memory Management. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR '22), September 14–15, 2022, Brussels, Belgium.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3546918.3546926

1 INTRODUCTION

Automatic memory management (garbage collection) provides an abstraction over memory that delivers memory safety, reducing opportunities for memory leaks and avoiding use-after-free bugs. By avoiding the need to reason about object lifetimes, garbage collection reduces cognitive load for programmers, and by allowing objects to be moved, it introduces spatial optimization opportunities for language implementers. Bugs such as use-after-frees, out-of-bounds reads and writes are commonplace in languages such as C and C++ and still dominate vulnerability databases [11].

The idea that garbage collection confers a productivity benefit is widely accepted, although hard to quantify [9]. On the other hand the costs associated with garbage collection are much less well understood. Zorn [29] compared conservative collection to manual memory management, finding that the principal overhead was due to the memory system, not the CPU load of performing collections. Huang et al. [18] showed that garbage collection can offer locality benefits to programs. Hertz and Berger [17] explored the question in depth, using a simulation framework to model an ideal (oracular) collector, suggesting that the overheads are substantial. Recently, Cai et al. [10] developed a methodology for creating a baseline that allowed a lower bound on the collector overhead to be established.

A fundamental challenge that any such comparison faces is that programs written for manually and automatically managed languages respectively tend to be applied to different problem domains and will have a different style, as a direct consequence of the availability / absence of automatic memory management. Retrofitting garbage collection into a manually managed language is problematic as unless that language is memory safe, the collector will not be able to move objects, which is one of the most important performance opportunities available to a garbage collector [18], and it will necessarily be conservative [8]. On the other hand, adding explicit `free()` operations to a garbage collected language is problematic both because of the effort associated with correctly doing so in any non-trivial code base, and because of the question of whether their placement reflects choices that would have been made natively by a programmer.

We introduce new methodologies with which we conduct four studies, three of which analyze different aspects of the impact of garbage collection on a garbage collected language and one which analyzes the effect of delaying object reuse in a manually managed language: (i) the space overheads of garbage collection relative to manually managed languages; (ii) the effects of garbage collection on the execution time of the mutator (application); (iii) the effects of garbage collection on the locality of the mutator; and (iv) the effects of inserting garbage collection-like behavior in manually managed applications. The new methodologies we introduce include using a mark-sweep collector that is built upon off-the-shelf free-list allocators [25], using large scale event-based workloads to explore the effect of proximity to collections, and using a quarantine [1] to model reclamation delays in unmanaged programs. These new methodologies allow us to deepen the understanding of the overheads and benefits of garbage collection in comparison to manual memory management, and we hope this will allow language developers and implementers to make an informed decision regarding their choice of whether to use automatic memory management. Our work also informs GC developers by identifying areas where GC introduces overheads. We conduct our study in a contemporary setting using modern hardware allowing us to reevaluate overheads and benefits of garbage collection in the face of rapid CPU and microarchitectural advancements.

2 BACKGROUND AND RELATED WORK

There is an extensive literature on manual and automatic memory management dating back to McCarthy's seminal 1960 paper [24]. Here we just cover a few concepts important to this paper.

2.1 Background

Lifetimes and Delayed Reclamation. Garbage collectors use reachability as a conservative approximation to liveness, so in general, an object will be reclaimed some time *after* its last use. This delay in reclamation is referred to as *heap drag* [26]. Tracing garbage collectors [24] and deferred reference counting collectors [12] reclaim garbage periodically, which means that there is generally an additional delay between when an object becomes unreachable and when the space it occupies becomes available for reuse. On the other hand, a non-deferred reference counting collector will reclaim an object immediately when the last reference to the object is (transitively) removed.[1] Because there are fixed costs associated with garbage collection, a longer period between collections will generally reduce collection costs, but comes at the cost of further delaying reclamation.

On the other hand, manual memory management relies on a programmer identifying the closest point in a program, statically, to where a object will no longer be used. This can be difficult to get right, particularly in the face of concurrency. If the object is freed too early, it may lead to a use-after-free error, which is a significant source of bugs and security vulnerabilities. On the other hand, if the object is freed too late, space will be wasted, and if it is not freed at all, the program will leak memory.

[1]Cyclic garbage can't be collected by direct reference counting, so in general it will suffer similar heap drag to an object in a system with tracing collection.

In Section 7, we directly examine the costs of delayed reclamation in the context of a manually managed system by placing objects in a 'quarantine' [1] for some time after they are freed before allowing their space to be reclaimed.

Sources of Space Overheads. Three sources of space overhead are relevant to our study: (i) space held in *reserve* by a collector into which objects may be copied; (ii) *fragmentation* intrinsic to the heap organization used by the memory manager; and (iii) space used for metadata that is essential to the memory manager, such as free lists, mark bits, and work queues.

At one end of the space efficiency spectrum, a simple semi-space copying collector [14] holds exactly half of the heap in reserve because it copies live objects into the reserved space and must account for the worst case where all objects survive. At the other end of the spectrum, a mark-compact collector moves objects in place [16, 27], and each time it collects it fully compacts the heap, so it suffers no fragmentation or copy reserve overheads. Free-list allocators, whether used for manual memory management or mark-sweep, suffer internal fragmentation, where an object does not fully occupy the available memory cell, and external fragmentation, where unused memory cells are not available for other purposes (such as other sized objects or for return to the operating system) because they share a memory region (e.g. a page) with some used cells.

Large Objects. Because the operating system and hardware already perform memory management at the page granularity, many garbage collection systems handle objects larger than a page differently to smaller objects. This detail is relevant to our study since this means that in practice most collectors are hybrids. For example, a semi-space collector uses the semi-space algorithm for all objects smaller than some threshold, and a simpler, non-moving strategy for larger objects.

2.2 Related Work

The question of how much garbage collection costs compared to manual memory management is longstanding and contentious. However, there are surprisingly few studies that directly address it.

Zorn [29] evaluates the costs of conservative garbage collection with respect to manual memory management. They compare the Boehm-Weiser conservative collector [8] against domain-specific and general purpose memory allocators for a variety of C benchmarks. They find that the execution time performance of conservative GC is often comparable and sometimes better than the performance of manual memory management. However, the space overheads in comparison to manual memory management range from 30% to 150% more space. They find that this high space overhead is due to internal fragmentation of the heap, increased heap size due to delayed reclamation, and due to the conservative nature of the collector they evaluated.

Huang et al. [18] describe an approach to exploit data locality by using copying garbage collection, the key insight being that a copying GC can reorder objects such that the most frequently accessed objects are located together. The authors introduce *Online Object Reordering* (OOR), a copying order that is based on the application's traversal patterns, into a generational copying collector

in JikesRVM [2]. The authors compare the execution time of OOR against an idealized manual memory management approximation. They model their approximation by using a mark-sweep collector with a free-list allocator and only measuring the mutator time (calculated as the difference of total execution time and time spent garbage collecting). This is an imperfect approximation as on one hand the mutator time does not include the cost of freeing objects, while on the other hand memory reclamation is delayed compared to manual memory management. They find that the OOR collector is generally comparable to, if not better, than the idealized manual memory management approximation even at small heap sizes.

Hertz and Berger [17] 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" which uses `malloc` and `free` to allocate and deallocate objects in Java programs. To avoid the costs of instrumentation affecting the measurement of the programs, their framework runs programs on the cycle-accurate simulator and injects calls to `free` from within the simulator whenever the trace indicates the death of an object. 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 are trace-based and require three preconditions to work: (i) deterministic program execution; (ii) fixed object allocation order; and (iii) a pre-computed 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 whether any objects are due 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 [19], and MMTk's explicit free-list allocator, *MSExplicit*, Hertz and Berger compare the space- and time-overheads for various garbage collectors in JikesRVM with MMTk [4, 5]. An often-cited result from this paper is that garbage collection is much slower than explicit memory management, requiring at least 5× more memory in order to provide the same execution time performance. However, automatic versus manual memory management is just one of three differences between the two systems compared in this headline result; the other two being the free list design (Lea versus MMTk's), and the method for accounting for memory usage. In Figure 6 they show an approximately 1.6× difference between Lea and MMTk's free lists. Ignoring the difference in space accounting, this suggests a 3.1× (5/1.6) space overhead to achieve the same performance when holding the free list design constant. For space overheads, they find (Table 4) that the best garbage collector in MMTk at that time, GenMS (a 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 (which, again, normalizing to MMTk's explicit memory manger, is a 1.25–1.56× space overhead). Our results in Section 4 suggest a space overhead of about 11-17% for a modern GC compared to manual memory management.

Cai et al. [10] recently presented a new methodology for empirically deriving a lower bound on the overhead imposed by garbage collection, relative to a hypothetical ideal garbage collector that bestows all the benefits of garbage collection and none of the costs. They reported a lower bound of 3–9% total time overheads for production OpenJDK collectors in heap sizes from 1.9–6.0× the minimum, and 124% for a heap 1.4× the minimum size. They did not explore the question of how garbage collection costs compare to manual memory management.

3 METHODOLOGY

Each of the four experiments uses a distinct methodology, with a number of elements in common, which we outline here.

Hardware and Operating System. All of the systems used in our experiments ran Ubuntu 18.04.6 LTS with Linux 5.4.0-105 kernels. All CPUs operate in 64-bit mode and use 64-bit kernels.

We used the following hardware platforms:

(i) 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 512 GB DDR4 RAM.
(ii) 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 DDR3 RAM.
(iii) Ryzen 9 5950X Zen 3 with a 3.4 GHz clock, a 16 x 32 KB, 64 B/line, 8-way L1 cache, a 16 x 512 KB, 64 B/line, 8-way L2 cache, and 64 GB DDR4 RAM.

The results for Section 4, which only measures space, and does not measure performance, were gathered on the Intel Xeon Gold 5118 machine. All other results were gathered on the Ryzen 9 5950X, including Section 6 which also used the Intel i7-6700K machine to explore microarchitectural sensitivity.

OpenJDK. When evaluating Java (Sections 4, 5, 6), we use OpenJDK 11[2] (the latest LTS release at the time). All experiments use the HotSpot C2 JIT compiler with pre-compilation enabled and explicit GCs disabled. Measurements were captured using the DaCapo benchmark harness. If we are measuring time-sensitive values, we perform four warmup iterations before starting our timing iteration in order to reduce any experimental noise due to JIT compilation or other VM operations.

Benchmarks. We use a snapshot of the Chopin evaluation version[3] of the DaCapo benchmarks [6]. 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 to their latest releases (i.e. from 2012 to 2022). The benchmark suite is available on the DaCapo Chopin GitHub repository. Section 7 describes the set of unmanaged benchmarks we use.

4 SPACE OVERHEADS OF GARBAGE COLLECTION

In our first study, we examine the claim that garbage collection is significantly less space efficient than manual memory management using a baseline that approximates a manually managed or naively-reference counted heap. This study *does not* examine the space-time trade-off inherent to garbage collection. For this study we introduce

[2]OpenJDK 11 version jdk-11.0.15+8
[3]Git hash `f480064`

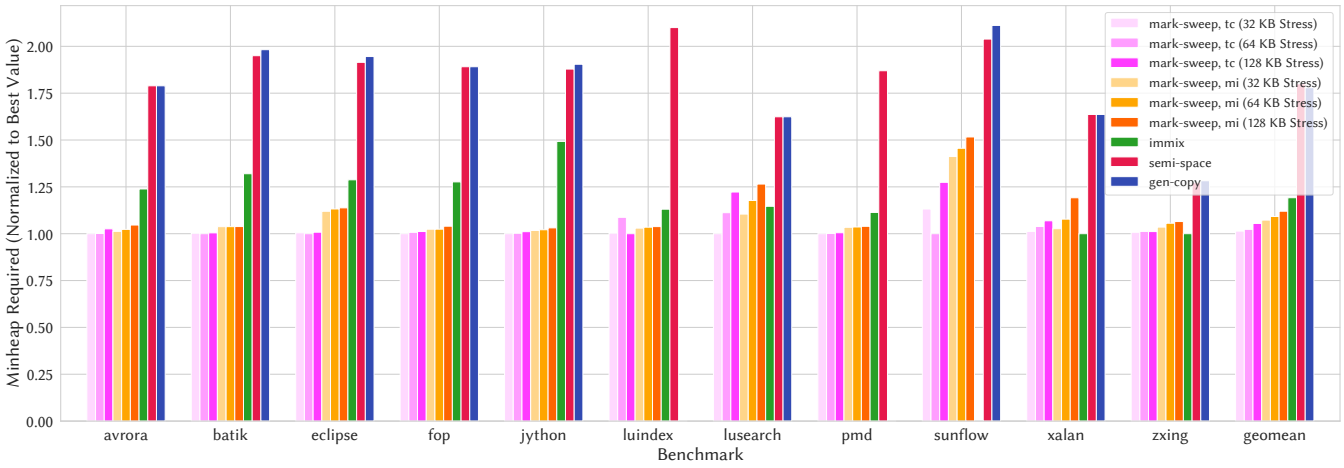


Figure 1: Minimum heap size required to complete a benchmark for the semi-space, gen-copy, and immix collectors compared to the 32 KB, 64 KB, and 128 KB mark-sweep baseline collectors on the Xeon Gold Skylake.

a methodology which uses a baseline that: (i) manages memory using TCMalloc and mimalloc, two modern high performance free-list allocators [15, 22]; and (ii) reclaims objects promptly, by identifying unreachable objects via very frequent heap traces.

4.1 Methodology

The basis for our evaluation is a mark-sweep collector that uses a conventional malloc/free library as its underlying free-list allocator [25]. This collector uses `malloc` to allocate each object. Like other collectors, the heap size is measured and specified in terms of used pages. Garbage collection is triggered automatically when the specified number of pages have been used, and may also be explicitly triggered by the user (e.g. `System.gc()`). The collector can be configured to use any standard malloc library. The collector only differs from an idealized manual memory management system on account of the delay in reclamation due to the periodicity of collection. We further configure the collector to perform collections at arbitrarily periodicity (as measured in bytes allocated), and in the limit can perform a collection at every allocation, thus eliminating heap drag [26] due to collection delay.

In practice, some of the DaCapo benchmarks perform so many allocations that collecting at every allocation would lead to impractical execution times, so we collect at coarser granularities. Collecting at a coarser granularity simply bounds the precision of our space analysis. We find that 64 KB is practical, so use this as our default. We also evaluate with 32 KB and 128 KB to establish the sensitivity of our methodology to this choice. In Section 7 we show in the context of an unmanaged language that delaying reclamation by these amounts leads to negligible space overheads, which gives us confidence that these provide good, fair baselines with respect to space overheads.

We used the TCMalloc [15] and mimalloc [22] allocators in our mark-sweep implementation. TCMalloc is widely used particularly for large-scale applications, notably within Google [15]. Mimalloc is a relatively new free-list allocator that is designed for use cases including highly parallel and concurrent applications as well as predictable performance overheads. Reeves [25] compares different free list implementations — such as jemalloc [13], Hoard [3], mimalloc, and glibc 2.27 — for the DaCapo Java benchmarks [6] and finds that mimalloc consistently outperforms the other allocators on those workloads in terms of the mutator performance.

We exclude cassandra, h2o, and tomcat due to internal errors or timeouts from the excessive GCs; biojava and h2 as they failed to complete in a reasonable timeout of 2 days; and graphchi and jme which refuse to initialize with small heaps.[4]

We measure the minimum heap size required to run the benchmarks for collectors such as semi-space, generational copying with a semi-space mature space ('gen-copy'), and immix [7] and compare them with the heap footprint as reported by the approximation of manual memory management as discussed above. The minimum heap size for each collector is measured by performing a bisection search over heap sizes.

4.2 Results and Analysis

Figure 1 shows the minimum heap required to run a benchmark for different GC algorithms in comparison to the 32 KB, 64 KB, and 128 KB mark-sweep baselines. The results are normalized to the best value for each benchmark. In a few cases, benchmarks did not complete, such as luindex and pmd for the gen-copy collector due to an internal VM crash, and sunflow for immix, which behaved inconsistently.

The two fully copying collectors, semi-space and gen-copy, must always hold half of the heap in reserve, so it is unsurprising that their space overhead approaches 100%. However, large objects do not require a copy reserve (Section 2.1) and have little fragmentation, and furthermore, the copying collectors have no fragmentation, which explains why the overhead is less than 100%. Taking the geometric mean over all benchmarks, immix has space overheads

[4]For example, jme only performs 5 GCs with the immix collector, suggesting that it could run on a smaller heap.

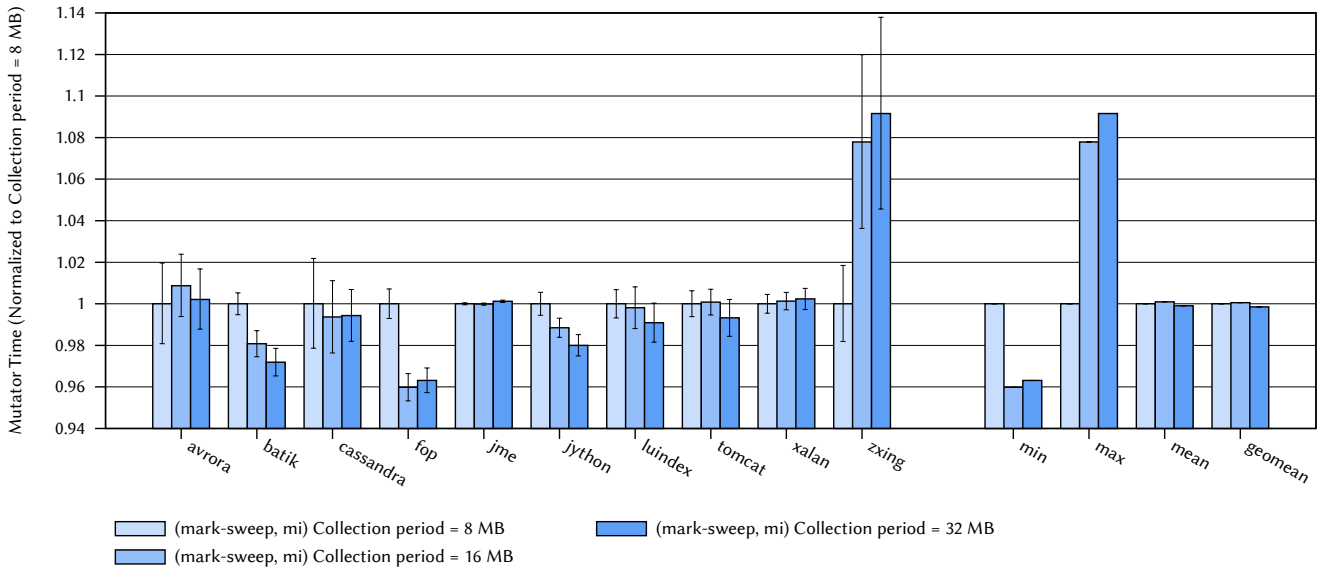


Figure 2: The impact of three collection periods on mutator execution time when using the mimalloc-based mark sweep collector. Results were gathered on the Zen 3 machine. Results are averaged over 10 invocations, and normalized to the 8 MB collection period.

of 17% and 11% relative to the TCMalloc and mimalloc baselines respectively. On xalan and zxing, immix is slightly more efficient than TCMalloc and mimalloc. To better understand this behavior, we ran those benchmarks with the mark-sweep collectors collecting every 16 KB. On zxing, TCMalloc is 0.6% more space efficient than immix, while mimalloc remains 3.5% less efficient. At 16 KB on xalan, TCMalloc and mimalloc are both 1.1% more space efficient than immix. For benchmarks such as batik, fop, jython, and pmd, the number of pages allocated for the three mark-sweep baseline collectors (per allocator) are within < 1% of each other, suggesting that these values are close to the space requirements for these benchmarks, had the memory been reclaimed with zero delay. On both luindex, and sunflow TCMalloc's space efficiency has a non-monotonic relationship to reclamation delay.

Summarizing, this study shows average space overheads for a modern whole-heap garbage collector [7] of 17% and 11% relative to TCMalloc [15] and mimalloc [22] respectively with near-immediate (32 KB) reclamation. This result is not directly comparable with Hertz and Berger because of the simulation-based methodology, slightly different collectors, and older workloads they used [17]. Nonetheless, our finding is much lower than the 2–2.5× space overhead often cited, but not so far from the 1.25–1.56× that their data appears to show.[5]

[5]It is sometimes reported that Hertz and Berger [17] show that garbage collection has a best case 2–2.5× space overhead compared to manual memory management. However that figure, taken from Table 4, is *not* Hertz and Berger's measure of the overhead of garbage collection (MS v MSExplicit), but a comparison between two different memory managers, using two different free-list allocators (MS v Lea). We refer to their MS v MSExplicit results, which hold the allocator constant, leaving GC versus manual as the only difference, as we describe in Section 2.2.

5 MUTATOR PERFORMANCE: GC FREQUENCY

In our second study we examine claim that garbage collection *indirectly* affects mutator performance by measuring the impact of collection frequency on mutator performance. For this study we develop a methodology that: (i) focuses on the costs imposed by garbage collection on the mutator; (ii) uses a non-moving collector to avoid conflating the established benefits of copying collection [18]; (iii) uses established high-performance explicit memory management libraries [15, 22] to allow costs to be viewed from the frame of explicit memory management; and (iv) allows us to issue non-reclaiming full heap traces, so we can separate the effects of delayed reclamation from the disturbance due to periodic stop-the-world traces of the heap by garbage collection.

5.1 Methodology

We use the mark-sweep collector developed by Reeves [25], as we did in the first evaluation (Section 4). The collector uses mimalloc [22] and TCMalloc [15] to allocate objects using `malloc`, and to reclaim dead objects, it uses `free`. By using this allocator, the mutator's memory management behavior is as close as possible to a high performance manual memory managed system, modulo the effects of garbage collection, which we will study. Specifically, we explore effects due to delayed reclamation, and the microarchitectural effects of periodic stop-the-world collection. We focus our discussion on mimalloc since Reeves showed it to perform better for these workloads.

By using a simple stop-the-world collector, there are no read or write barriers, so the only *direct* costs associated with garbage collection are stop-the-world pauses. We measure the GC pauses and

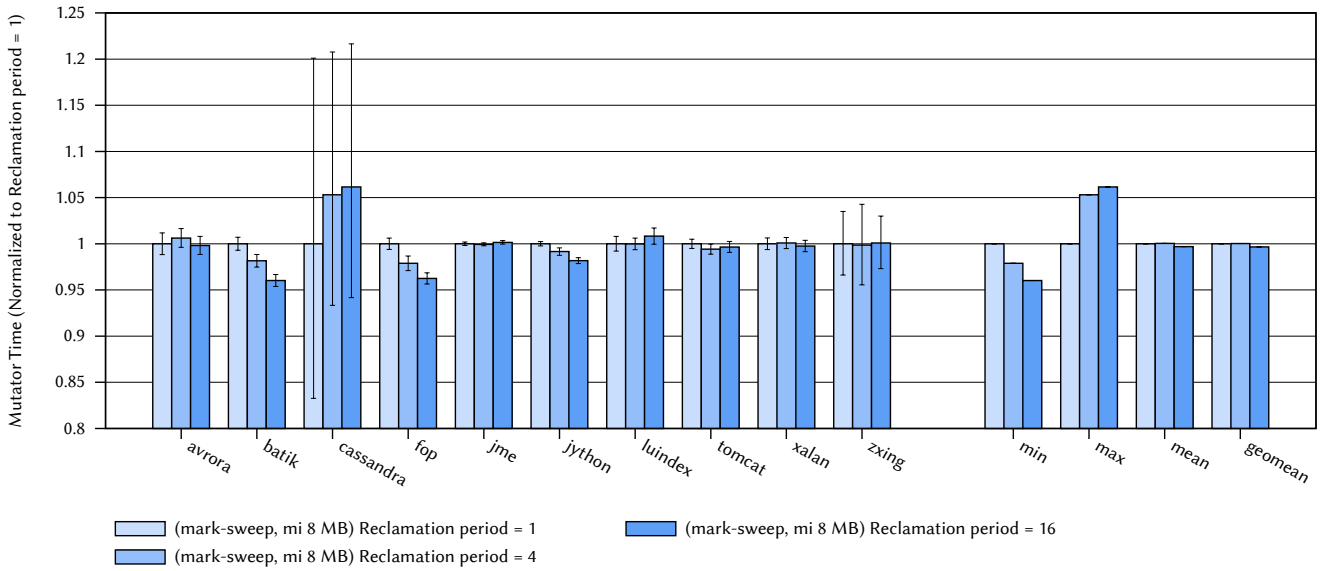


Figure 3: The impact on mutator execution time of three reclamation periods when performing collections at a fixed 8 MB period, using the mimalloc-based mark-sweep collector. Results were gathered on the Zen 3 machine. Results are averaged over 10 invocations, and normalized to the case where reclamations occur at every collection.

subtract from the total execution time to reveal the mutator time. The remainder of our methodology targets revealing the *indirect* costs of garbage collection imposed upon the mutator. We use two basic techniques to tease out those costs: (i) we carefully *control the periodicity* of garbage collections; and (ii) we selectively perform *full heap traces that do not reclaim garbage* so that we can separate out the impact on the allocator through delayed reclamation from the broader impact of microarchitectural perturbation due to periodic full heap traces.

Just as we did in Section 4, we force collections with a controlled frequency. In our first experiment we vary the collection frequency so that we can see the combined impact of delayed reclamation and periodic stop-the-world collections on the mutator. We then conduct a second experiment to tease apart reclamation from the rest of collection by only performing reclamation every Nth collection. We exclude eclipse, lusearch, pmd, and sunflow because they do not complete within a timeout of 1 day (for all invocations).

5.2 Results and Analysis

Figure 2 shows the average mutator execution time for the DaCapo benchmarks with collections every 8, 16, and 32 MB, with each result normalized to 8 MB. Three benchmarks, fop, batik, and jython show small performance improvements greater than 1% as the collection period increased from 8 MB, which point to frequent collections degrading performance. The other benchmarks saw little change, and zxing saw its best performance when collections were frequent.

There are two clear trends: (i) overall, mutator performance is largely insensitive to collection frequency; and (ii) with one exception (zxing), in the cases where there are changes, more frequent collections lead to small slowdowns.

In order to tease apart whether these effects were simply due to the microarchitectural displacement caused by a full heap collection, or due to the more timely reclamation of objects, we conducted two more experiments. In these experiments we modified the collector to separate *identification*, in which the collector traces the heap to identify live objects from *reclamation*, in which the collector sweeps the heap to reclaim unreachable objects, adding them to free lists. This methodology allows us to separate the microarchitectural displacement caused by the full heap trace performed during identification, from the effects such as improved locality due to more or less promptly reusing memory.

In the second experiment, we simply repeated the experiment illustrated in Figure 2, but only performed reclamation every sixteenth collection. The results (not shown) were very similar to those for the first experiment shown in Figure 2.

Figure 3 shows the results of the third experiment, where we performed collections every 8 MB but varied the frequency of reclamation. These results show that for fop and batik, and to a lesser extent jython, the speedups observed in Figure 2 occur here too, indicating that they were not due to collection frequency but to reclamation frequency. Interestingly, this shows that for these benchmarks, there is a small performance advantage in *delayed* reclamation, and for the other benchmarks there is no statistically significant change due to delayed reclamation.

Summarizing, our second study shows that for a mark-sweep collector based on the mimalloc free-list allocator, when evaluated with a collection of DaCapo benchmarks, collection frequency does not appear to have a significant impact on mutator performance in the range 8 MB to 32 MB, and for three benchmarks there appears to be a small mutator performance advantage in delaying reclamation. Our evaluation with TCMalloc was remarkably similar, revealing the same lack of sensitivity to GC frequency as we see here for mimalloc.

6 MUTATOR PERFORMANCE: GC PROXIMITY

In our third study, we again examine the claim that garbage collection events negatively and/or positively affect the performance of the mutator. We do this by applying an entirely new approach. This evaluation: (i) focuses on the *indirect costs* of garbage collection; (ii) uses five garbage collectors with distinct locality patterns and allocation behaviors; (iii) uses a single widely-used highly–tuned multithreaded workload comprised of a large number of small queries; and (iv) evaluates how *proximity to a garbage collection event impacts mutator performance.*

6.1 Methodology

Our insight is that we can observe the effect of proximity to garbage collection on the mutator by using a workload comprised of a large number of individually measurable events. We can see how a given event's proximity to a GC affects its performance. We use the DaCapo lusearch benchmark, which issues 512 K search queries in the Apache Lucene search framework. We control garbage collections so that they occur at regular intervals, and do so at three distinct resolutions (10, 100, and 1000 GCs per benchmark iteration). We then analyze the relationship between each query's execution time and its proximity to the most recent collection.

In this evaluation we are *not* examining the *direct* effects of garbage collection on an application (e.g. we explicitly ignore queries *interrupted* by a collection), but rather our focus is on the *indirect* effects of garbage collection. Intuitively, one might expect that code that executes soon after a garbage collection will be negatively affected due to the GC perturbing machine state such as caches and branch predictors, and perhaps gain some positive effects due to improved locality among older objects [18] and in the allocator.

We instrument lusearch to gather start and end timestamps for every query, and we record the start and end of every garbage collection. We perform 35 invocations of the benchmark, gathering the data from the fifth, warmed up iteration in each case. This produces approximately 18 M timing data points. We analyze the data offline, calculating the proximity of each query to the preceding collection, discarding each query that was interrupted by a collection. We then perform an analysis of the correlation between query performance and the query's proximity to the nearest preceding collection.

We repeat this experiment for five simple stop-the-world whole heap collectors and we configure heap sizes so as to trigger 10, 100 and 1000 collections per benchmark iteration. We use simple whole heap collectors because we do not want to conflate other concerns such as write barrier overheads, and we are explicitly not concerned in this experiment with the cost of the collection itself. By using canonical collectors rather than more complex ones, we may be better able to reason about the impact of particular collector types on mutator performance. These lessons can then be applied to a variety of concrete collector designs.

6.2 Results and Analysis

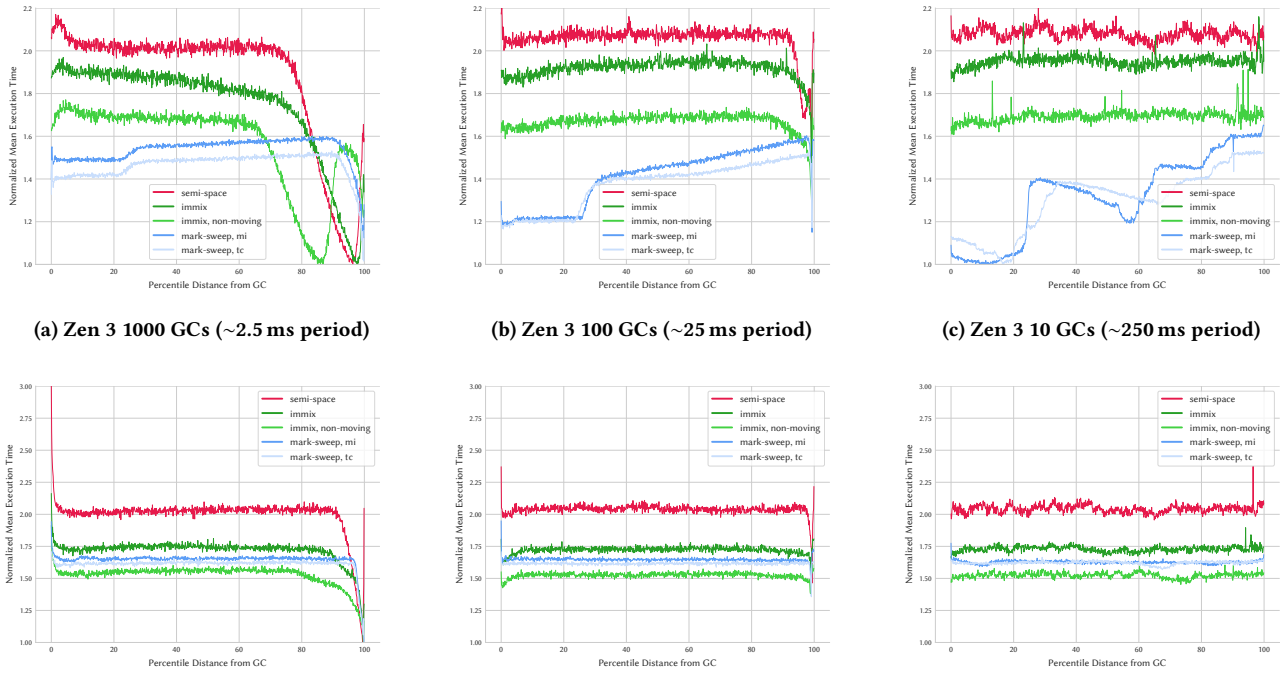
Figure 4 shows the results for three temporal resolutions (columns) and two architectures (rows). The left graphs show results for 1000 forced garbage collections which, given execution times of approximately 2.5 s and 10 s for the Zen 3 and Skylake respectively, means that garbage collections are approximately 2.5 ms and 10 ms apart respectively, longer for the mark-sweep collectors which are slower. The middle graphs have 100 forced collections (25 ms and 100 ms), and the right graphs have 10 forced collections (250 ms and 1 s).

For each line in each graph, we take 35×512 K search queries, and discard those that were interrupted by a collection, sorting the remainder by the distance from their start to the most recent garbage collection. We then divide the ordered data left to right into 1000 groups, each containing roughly 18 K queries, and find the mean execution time for that group. We then normalize these means to the fastest group from among all three time scales for that collector, and plot the results. The leftmost group is the 1000 queries closest to a GC, the rightmost is the 1000 queries furthest from the preceding GC. We repeat this for each garbage collector. Note that because each collector's results are normalized separately, the graphs in Figure 4 do not reveal the relative execution times between the different GC strategies (which is not our focus).

Figure 4(a) plots results for the Zen 3 with 1000 collections and, given a total execution time of about 2.5 seconds, the collections are approximately 2.5 ms apart, so we expect the furthest (rightmost) queries in this graph to start roughly 2.5 ms after the prior collection. Thus the scale of the x-axis in Figure 4(a) is approximately 0–2.5 ms. The average query runs for about 150 μs on the Zen 3, or roughly 1/16 the width of the x-axis in Figure 4(a). Survivor bias appears to be responsible for the downward spike on the far right. Search queries naturally have a distribution of execution times. Among the queries that are close to the next GC, long running ones will be interrupted by the GC, and are therefore discarded. Those that appear in the rightmost groups are thus naturally biased toward the shortest running queries. If this spike were not an observation artifact, we should expect to see it appear at the 10% and 1% marks of the graphs to the right which have respectively 10× and 100× longer observation periods. However, the spike does not appear at those points, but rather, a similar spike is visible at the far right of in Figure 4(b) on a much more compressed scale, and likewise for Figure 4(d) and Figure 4(e). These observations confirm that the downward spike is an observational artifact, which we can explain by survival bias.

Putting aside the right-most results, we see in Figure 4(a) a number of trends. First, for the three collectors that use a bump allocator (all aside from mark-sweep), the queries are slowest in the tenth percentile closest to the preceding collection (left). This suggests that the microarchitecturally disruptive effect of the collectors degrades mutator performance for about 200 μs or so after each collection by 5-10% on the Zen 3. By contrast, the two collectors with the free-list allocators get a brief boost from the collection. Between the 20th and 80th percentiles (roughly 500 μs to 2 ms) we see three different trends. Non-moving immix and semi-space are stable, while immix shows a small, steady improvement in performance of about 10%, while the mark-sweep collectors degrade sharply for a brief time and then continues to degrade at a modest but steady rate.

Figure 4(d) shows similar trends on the Skylake. The disruptive effect of the collection is far more pronounced than for the Zen 3, demonstrated by the very large spike at the lowest percentiles for all five collectors. On the Skylake each query is completing in approximately the same time as on the Zen 3, approximately 150 μs. Because the machine has one quarter as many hardware



(a) Zen 3 1000 GCs (~2.5 ms period)

(b) Zen 3 100 GCs (~25 ms period)

(c) Zen 3 10 GCs (~250 ms period)

(d) Skylake 1000 GCs (~10 ms period)

(e) Skylake 100 GCs (~100 ms period)

(f) Skylake 10 GCs (~1 s period)

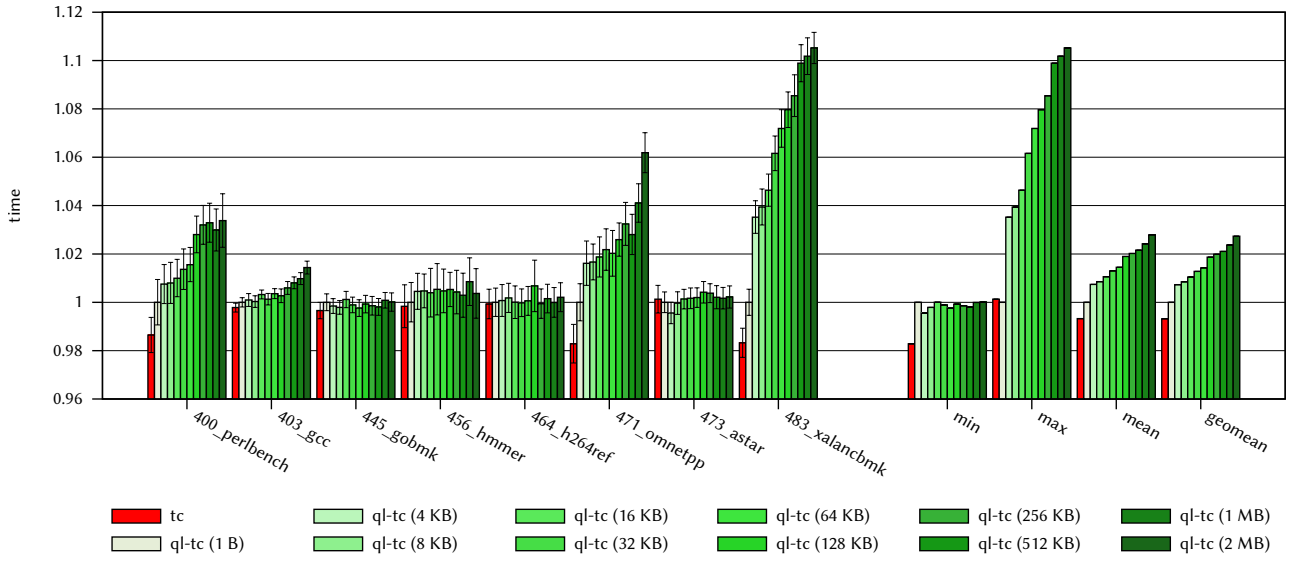
Figure 4: Performance of Lucene queries as a function of proximity to the most recent garbage collection. The three columns depict three orders of magnitude in GC frequencies, and thus three very different temporal resolutions. Each collector's results are normalized to the best performing percentile for that collector. Each experiment has 35×512 K queries. Queries interrupted by a GC are discarded. The remainder are sorted by the distance of their start time from the preceding GC. We plot the normalized mean execution time for each percentile (~18 K queries per percentile). Queries closest to the GC are on the left of each graph, those furthest are on the right. Survivor bias is visible in the rightmost percentiles (long-running queries are interrupted by the next GC).

threads, the benchmark runs for approximately four times as long. As a result each query is approximately 1/64 the width of the x-axis of Figure 4(d). The downward spike due to survivor bias seen in Figure 4(a) appears here, but is time-compressed by a factor of about four. Putting aside the results of the furthest and closest queries, all five collectors show a slow degradation in performance as they move further from the GC, however, the effect is small, about 5% in semi-space and barely observable in both mark-sweep collectors.

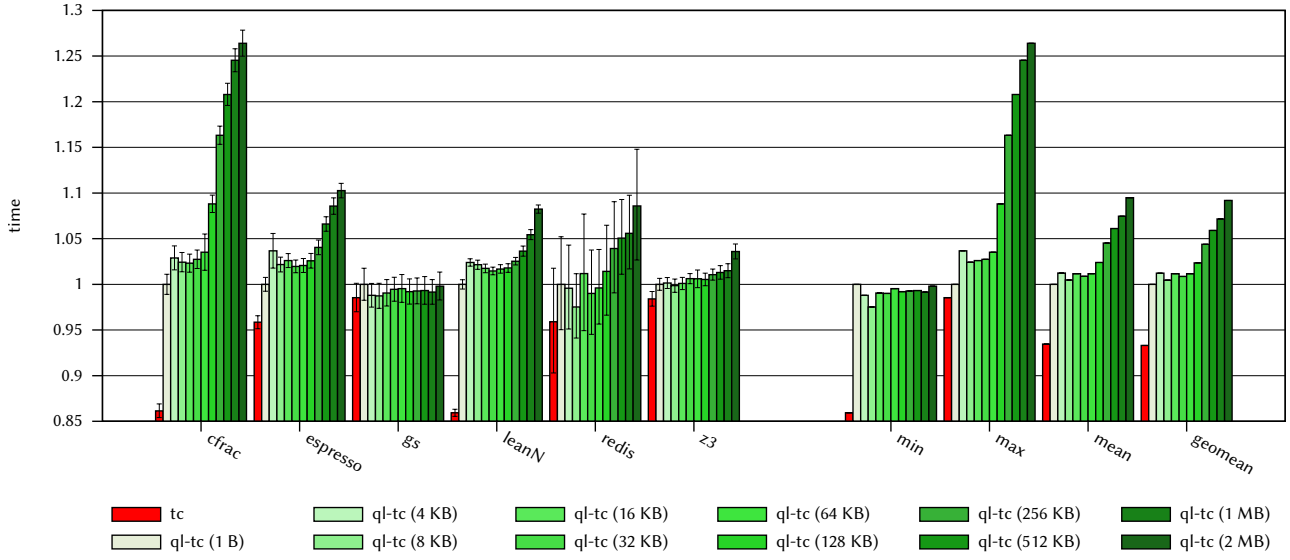
The flat lines in Figure 4(e) and Figure 4(f) show that at larger temporal granularities on Skylake proximity to the collection has very little effect.

Figure 4(b) and Figure 4(c) are similar to their Skylake counterparts with two notable differences. Since the timescale is compressed by a factor of four relative to Skylake, the features at the limits are more pronounced. Otherwise, the three bump pointer allocator collectors show a very similar trend, with almost uniform query performance as the distance from the GC changes. However, the collectors using TCMalloc and mimalloc free-list allocators respectively show much more interesting and hard-to-explain behavior. Figure 4(b) shows a steady degradation in query performance with a sharp step at about the 25th percentile. Figure 4(c) shows a sharp degradation below the 10th percentile, followed by a gradual improvement and then another sharp degradation around the 60th percentile. The mark-sweep results are difficult to interpret, but strikingly similar for both TCMalloc and mimalloc. We suggest two broad findings: (i) the free-list allocators are more sensitive to proximity to collections, due to the collections' direct impact on the free lists; (ii) the impact on free-list allocators is complex and is likely highly dependent on the particular free list design, the allocation pattern (w.r.t. size class) of the benchmark, and conspicuously, the microarchitecture.

Summarizing, this study suggests that: (i) the microarchitectural disruption due to garbage collection is observable but fleeting on modern machines, affecting the mutator only very briefly after the collection; (ii) for the three bump-pointer based collectors we studied, the impact of proximity to collections is barely observable; and (iii) for both malloc-based collectors we studied, proximity to the collection had little impact on the Skylake, but complex and significant impacts on mutator performance on the Zen 3, with variations of up to 40%. This study was designed around one workload. While Lucene is an important widely-used industrial-strength benchmark, it has distinctive allocation patterns which will color these results. Applying this methodology to other very different workloads is an interesting avenue for further work.



(a) SPEC CPU2006, time (TCMalloc).



(b) Mimalloc 'real' benchmarks, time (TCMalloc).

Figure 5: Time overheads due to delayed reclamation with the TCMalloc allocator, for delays between 4 KB and 2 MB, all normalized to the base case with quarantine but no delay (1 B). Results were gathered on the Zen 3 machine. Only six benchmarks show time overheads of more than 5%: omnetpp, xalancbmk, cfrac, espresso, leanN, and redis.

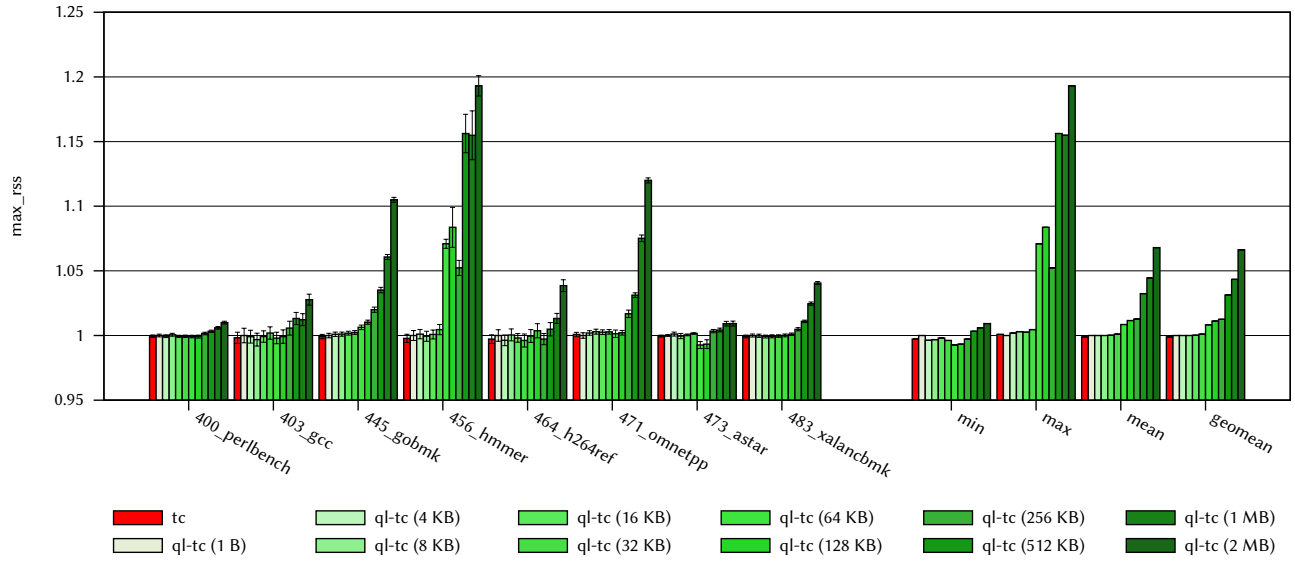
7 EFFECTS OF DELAYED RECLAMATION

In our final study we examine the claim that delaying reclamation reduces performance and increases space consumption. Garbage collection algorithms typically trade immediacy for efficiency, while manual memory management returns space as soon as the programmer's carefully-placed call to `free` is executed. This evaluation: (i) uses a quarantine to delay reclamation in C and C++ workloads [1]; (ii) uses three state of the art allocators [15, 22, 23]; and (iii) measures *how delayed reclamation affects time and space performance in manually managed languages.*

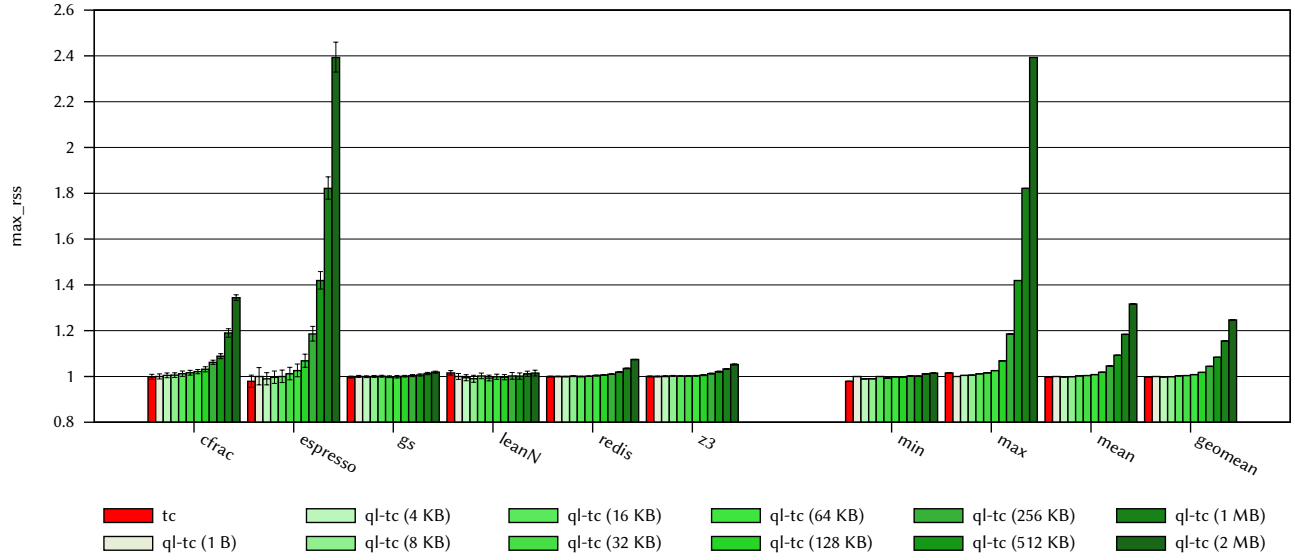
7.1 Methodology

Our methodological insight is that we can model delayed reclamation in an unmanaged setting via a *quarantine*, a mechanism used to identify and mitigate use-after-free bugs [1].

We use TCMalloc [15], mimalloc [22], and snmalloc [23], and add a thread-local quarantine buffer to their implementations of `free`. When the program calls `free`, instead of being freed immediately, the object is added to the quarantine buffer, and a global atomic variable that tracks the volume of quarantined objects is



(a) SPEC CPU2006, RSS (TCMalloc).



(b) Mimalloc 'real' benchmarks, RSS (TCMalloc).

Figure 6: Space overheads associated with delayed reclamation with the TCMalloc allocator, for delays between 4 KB and 2 MB, all normalized to the base case with quarantine but no delay (1 B). Results were gathered on the Zen 3 machine. Most benchmarks show little space sensitivity, with gobmk, hmmer, omnetpp, cfrac, and espresso as notable outliers.

incremented by the size of the freed object. We declare the quarantine to be *full* when a specified threshold of quarantined bytes is reached, and each thread returns their quarantined objects to their respective free lists. This mechanism emulates the effects of periodic reclamation. We measure the execution time and maximum RSS of standard benchmarks. Regardless of which threshold we use, we pre-allocate a fixed-size 2 MB quarantine buffer for each thread, sufficient to accommodate the volume thresholds we evaluate with the benchmarks we use. The baseline in each of our evaluations uses the quarantine with a threshold of 1 B which means objects are freed immediately. Hence, this baseline quantifies the overheads of incrementing the global atomic variable as well as allocating and accessing the thread-local quarantine buffer. The quarantine with no delay introduces an overhead of 0.7% and 6.7% compared to the unmodified TCMalloc allocator for the SPEC and mimalloc benchmarks respectively.

Benchmarks. We use the SPEC CPU 2006 benchmarks[6] and each of the 'real world' benchmarks used by mimalloc [21, 22].[7] We

[6]We do not have a license to SpecMark 2019 benchmarks used by Leijen et al. [21, 22].
[7]Git hash 5126dc8.

evaluated, but don't include the microbenchmarks used by mimalloc, since they don't contain meaningful computation on which to evaluate the effects of delayed reclamation.

Among these, leanN is multi-threaded while the remaining mimalloc benchmarks and the SPEC CPU benchmarks are single-threaded. We use the same inputs as those used by the authors [21], with the exception of z3. We changed the input of z3 from their small test example, test1 [20], to EntryCP from the z3 public test suite [28] because with the test1 input, z3 did not perform significant allocation and freeing. We exclude larson since it is a microbenchmark which only allocates and deallocates, while barnes and sed perform no deallocations, so we exclude them. We run each benchmark 10 times and report the average in order to reduce the effect of experimental noise in our measurements. For leanN, we fix the number of threads to be 8 as per the mimalloc authors [21].

Among the SPEC CPU benchmarks, we excluded bzip2, mcf, sjeng, and libquantum since they perform almost no dynamic memory management, all perform less than one free per second. By contrast, omnetpp performs 2 M frees per second.

7.2 Results and Analysis

We use TCMalloc [15] as the focus for our analysis here because it exhibited the most sensitivity and substantially larger overheads. Results for mimalloc [22] and snmalloc [23] appear in the appendix.

Figure 5 and Figure 6 respectively show the time and space impact of delayed freeing with TCMalloc for the SPEC CPU and mimalloc benchmarks, for quarantine sizes ranging from 1 byte to 2 MB.

Figure 5(a) and Figure 5(b) reveal an overall trend of time degradation as frees are delayed. For the SPEC benchmarks, there's a 2.7% slowdown on average when the delay is 2 MB, and for the mimalloc benchmarks it is 9.2%. These averages are driven up by omnetpp (6.2%), xalancbmk (10.5%), cfrac (26.4%), espresso (10.3%), leanN (8.2%), and redis (8.6%). The remaining benchmarks show very little change in execution time. Four of the benchmarks see worst case slowdowns of 1% or less due to delayed reclamation. Overall, these results indicate that for some benchmarks delayed reclamation introduces an observable slowdown, but in many cases there is none and in only six benchmarks is the worst case more than 5%.

Figure 7 and Figure 8 show that mimalloc and snmalloc are *substantially* less sensitive than TCMalloc with respect to time. They show averages for SPEC at 2 MB of 0.3% and 1.0% respectively (TCMalloc was 2.7%), and for the mimalloc benchmarks at 2 MB of 1.0% and 2.1% respectively (TCMalloc was 9.2%). The worst case results were similarly less sensitive for these allocators: 5.0% and 5.7% for redis on mimalloc and snmalloc respectively, both with substantial error bars, while for TCMalloc the worst was 5× greater, at 26.4% for cfrac.

Figure 6(a) shows that on SPEC, TCMalloc sees similar overheads for time and space, with an average worst case space overhead of 6.6% (when the delay is 2 MB) and with the highest overhead being 19.3% for hmmer. Four of the SPEC benchmarks have space overheads of 4.0% or less. On the mimalloc benchmarks, TCMalloc shows a more noticeable impact on space (Figure 6(b)). However, only two benchmarks, cfrac and espresso show substantial change, 1.34× and 2.4× respectively. These benchmarks have very small heap footprints in absolute terms: 3.5 MB and 4.4 MB respectively as compared to 492 MB for leanN. So the 2 MB delay in reclamation in this experiment is very substantial for these two benchmarks in relative terms, explaining their disproportional space overhead.

Figure 9 and Figure 10 show that mimalloc and snmalloc are also substantially less sensitive than TCMalloc with respect to space on the SPEC benchmarks, while showing similar sensitivity on the mimalloc benchmarks. They show averages for SPEC at 2 MB of 1.0% and 2.8% respectively (TCMalloc was 6.6%), and for the mimalloc benchmarks at 2 MB of 27.4% and 28.8% respectively (TCMalloc was 24.7%). Interestingly, on hmmer, mimalloc sees a striking *reduction* in space overhead of as much as 17.6% as reclamation is delayed. We have not established exactly why mimalloc behaves this way, but believe that it is due to hmmer inducing a fragmentation pathology that is relieved when reclamation is consolidated into less frequent episodes.

In summary, this study shows that the overheads induced by delaying reclamation are generally very small, but are also very dependant on the choice of allocator, with TCMalloc being substantially more sensitive than the other allocators we evaluate. Overall on the C and C++ benchmarks we evaluate, delayed reclamation has little impact on *time* and modest, allocator-specific impact on *space*, except in a few cases, notably cfrac (3.5 MB) and espresso (4.4 MB), where the reclamation delay (2 MB) is a substantial fraction of the total RSS footprint of the workload.

8 CONCLUSION

The relative cost of garbage collection compared to manual memory management is a longstanding and often contentious question. Although there is a large literature comparing amongst garbage collection algorithms and evaluating specific garbage collection mechanisms, there are few papers that directly explore how garbage collection compares to manual memory management.

In this paper we conduct four studies, each using a novel methodology to shed different light on the question. We find that: (i) relative to a baseline approximating manual memory management using a modern malloc/free library with frequent reclamation, the immix garbage collector has a space overhead of 11-17%, while fully copying collectors have an overhead of 65-80%; (ii) when using a mark-sweep collector based on a modern malloc library, the performance of the mutator is not very sensitive to delays in reclamation; (iii) when executing a large number of short queries, the proximity of those queries to the most recent garbage collection has little impact for systems that use a bump pointer allocator, except in the immediate aftermath of the collection, while having complex and significant microarchitecturally-sensitive impact on a free list based system; and (iv) delaying reclamation on C and C++ benchmarks generally results in negligible overheads in time and space but is allocator-dependent and can be substantial if the delay is a substantial fraction of the workload's footprint.

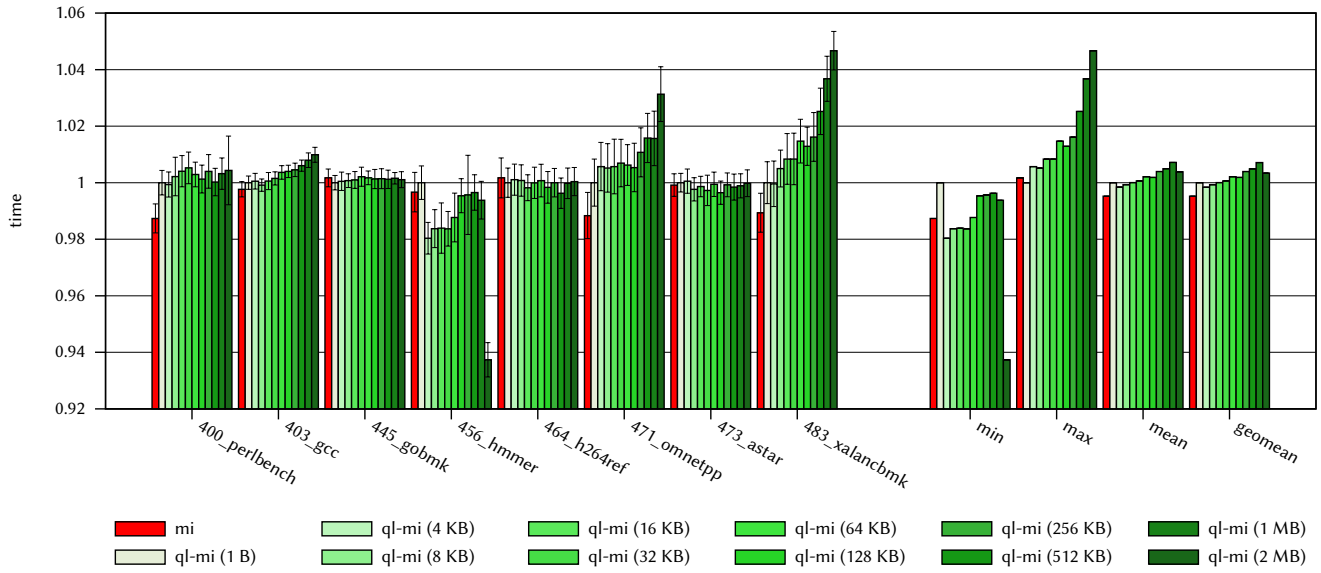
Our hope is that these analyses will help paint a richer picture of the relationship between manual memory management and automatic memory management performance, and that these new methodologies will provide others with new ways in which to examine the performance of automatic memory management.

ACKNOWLEDGMENTS

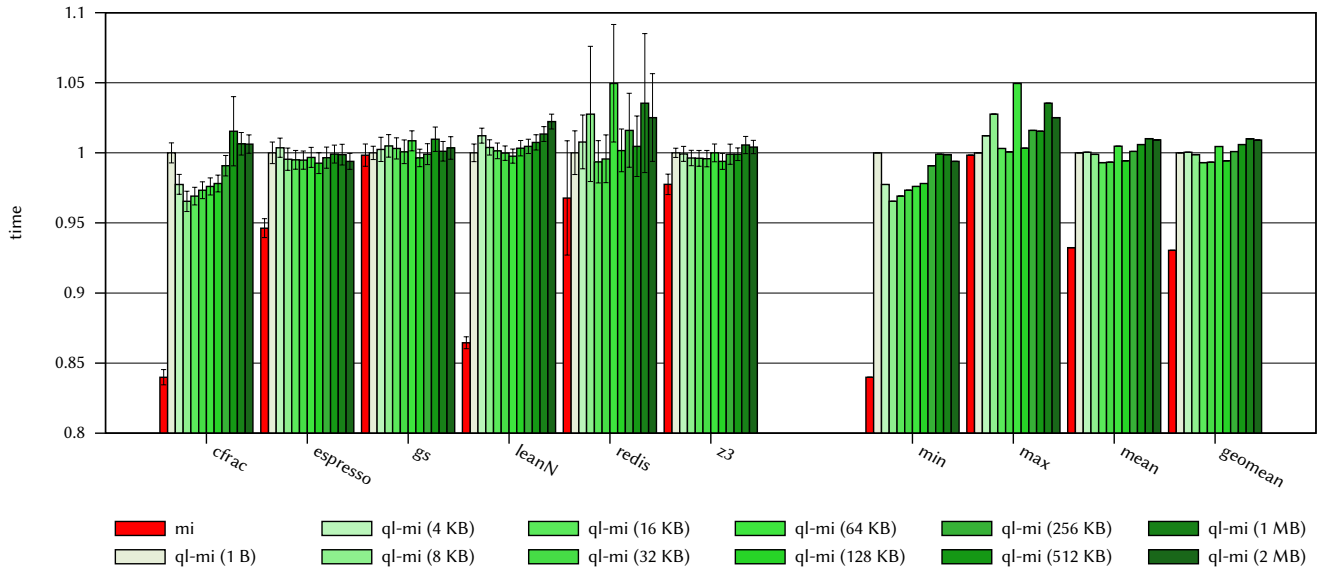
This material is based upon work supported by the Australian Research Council under award DP190103367. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Australian Research Council.

A ADDENDUM TO DELAYED RECLAMATION

This appendix extends Section 7 with results for mimalloc (Figure 7 and Figure 9) and snmalloc (Figure 8 and Figure 10).

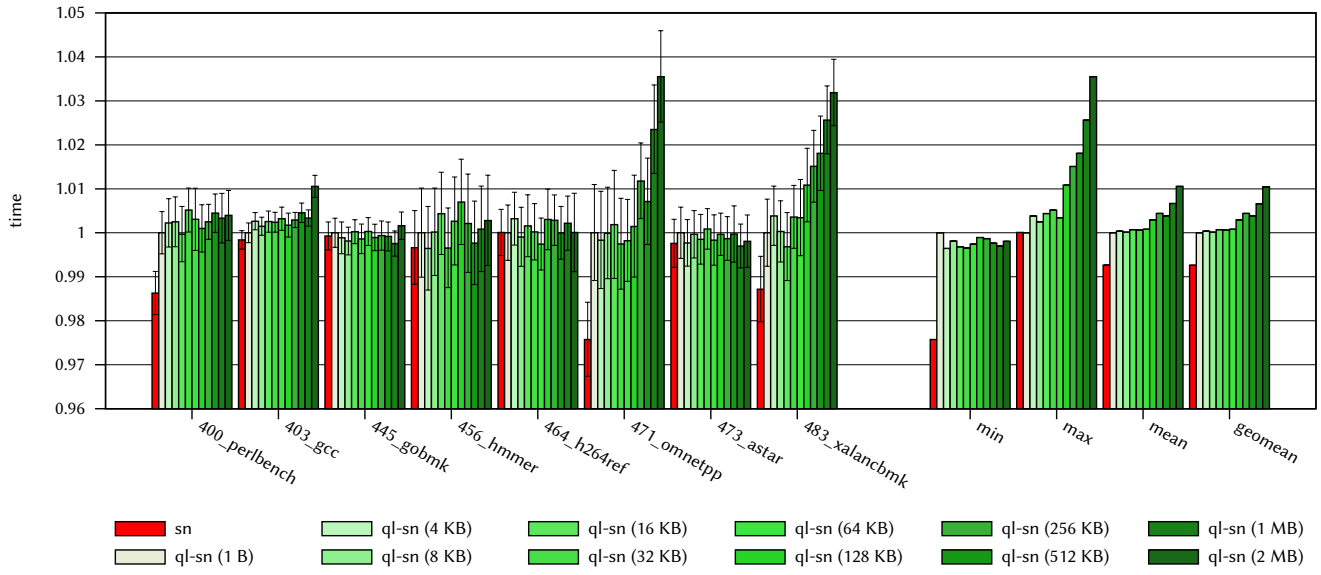


(a) SPEC CPU2006, time (mimalloc).

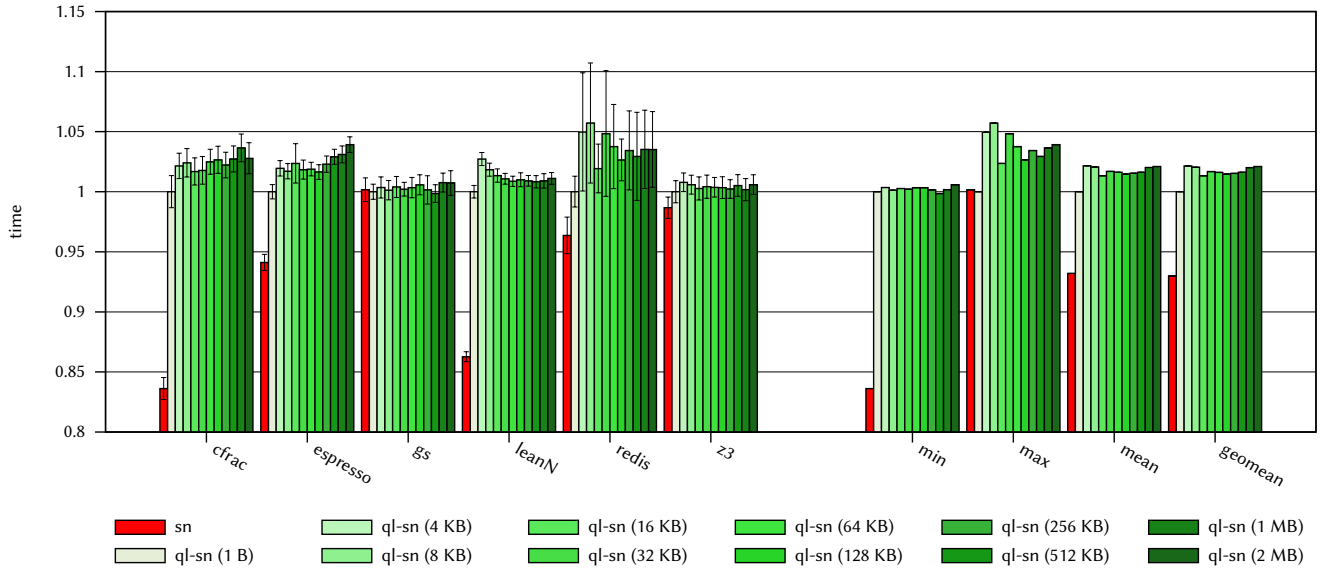


(b) Mimalloc 'real' benchmarks, time (mimalloc).

Figure 7: Time overheads due to delayed reclamation with the mimalloc allocator, for delays between 4 KB and 2 MB, all normalized to the base case with quarantine but no delay (1 B). Results were gathered on the Zen 3 machine.

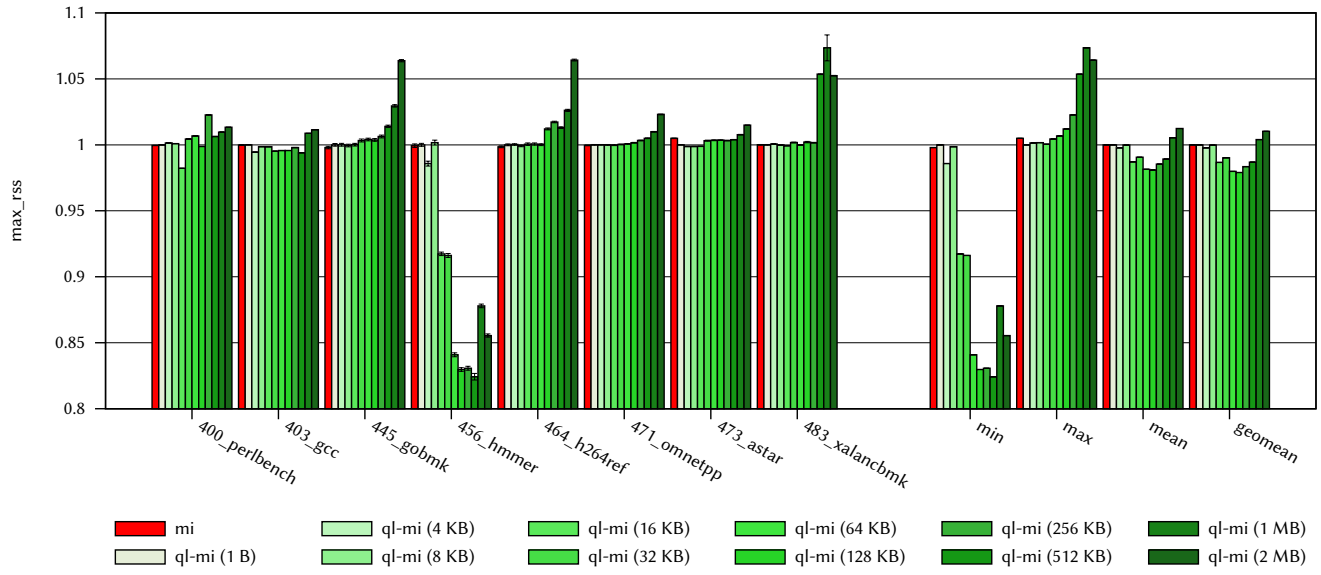


(a) SPEC CPU2006, time (snmalloc).



(b) Mimalloc 'real' benchmarks, time (snmalloc).

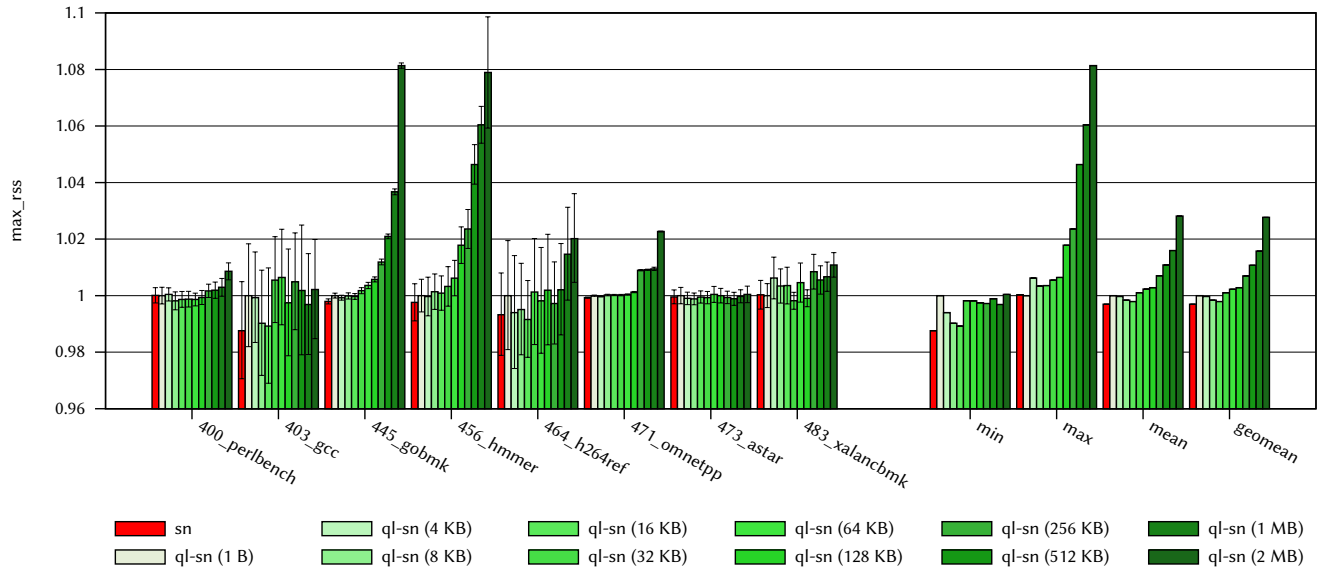
Figure 8: Time overheads due to delayed reclamation with the snmalloc allocator, for delays between 4 KB and 2 MB, all normalized to the base case with quarantine but no delay (1 B). Results were gathered on the Zen 3 machine.



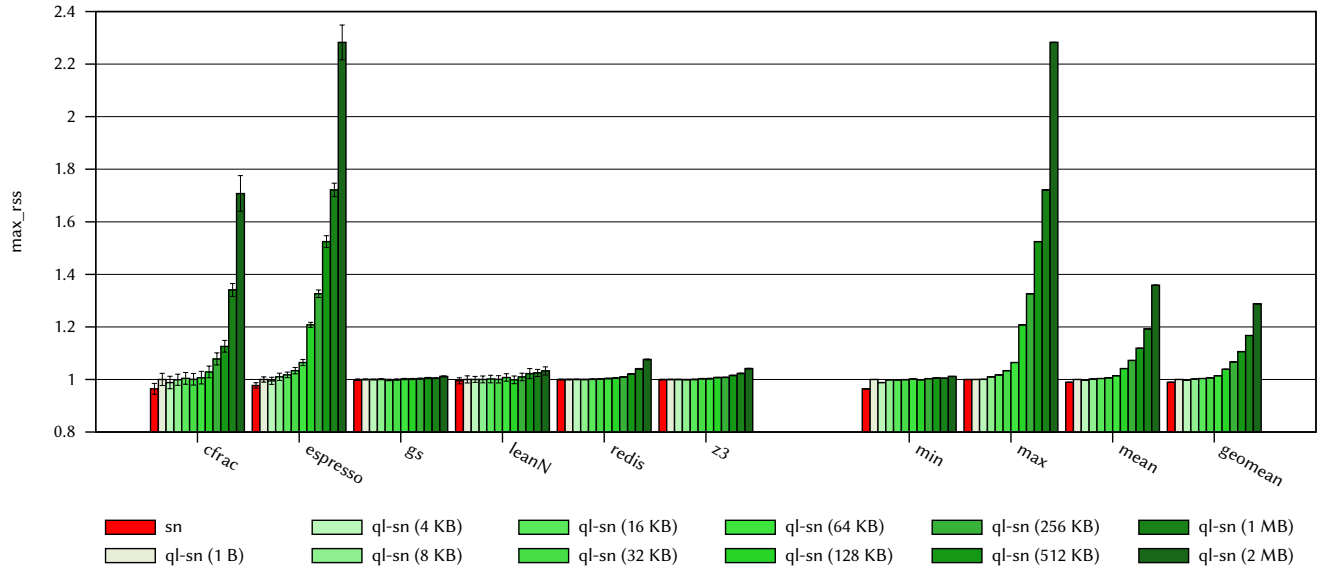
(a) SPEC CPU2006, RSS (mimalloc).

(b) Mimalloc 'real' benchmarks, RSS (mimalloc).

Figure 9: Space overheads associated with delayed reclamation with the mimalloc allocator, for delays between 4 KB and 2 MB, all normalized to the base case with quarantine but no delay (1 B). Results were gathered on the Zen 3 machine.



(a) SPEC CPU2006, RSS (snmalloc).



(b) Mimalloc 'real' benchmarks, RSS (snmalloc).

Figure 10: Space overheads associated with delayed reclamation with the snmalloc allocator, for delays between 4 KB and 2 MB, all normalized to the base case with quarantine but no delay (1 B). Results were gathered on the Zen 3 machine.

REFERENCES

[1] Sam Ainsworth and Timothy M. Jones. 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*. IEEE, 578–591. https://doi.org/10.1109/SP40000.2020.00058

[2] Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen E. Smith. 1999. Implementing Jalapeño in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1999, Denver, Colorado, USA, November 1-5, 1999*, Brent Hailpern, Linda M. Northrop, and A. Michael Berman (Eds.). ACM, 314–324. https://doi.org/10.1145/320384.320418

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 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*, Larry Rudolph and Anoop Gupta (Eds.). ACM Press, 117–128. https://doi.org/10.1145/378993.379232

[4] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. 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*, Edward G. Coffman Jr., Zhen Liu, and Arif Merchant (Eds.). ACM, 25–36. https://doi.org/10.1145/1005686.1005693

[5] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. 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*, Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum (Eds.). IEEE Computer Society, 137–146. https://doi.org/10.1109/ICSE.2004.1317436

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

[7] Stephen M. Blackburn and Kathryn S. McKinley. 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*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 22–32. https://doi.org/10.1145/1375581.1375586

[8] Hans-Juergen Boehm and Mark D. Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exp.* 18, 9 (1988), 807–820. https://doi.org/10.1002/spe.4380180902

[9] Albion M. Butters. 2007. *Total Cost of Ownership: A Comparison of C/C++ and Java.* Technical Report. Evans Data Corporation. http://docplayer.net/24861428-Total-cost-of-ownership-a-comparison-of-c-c-and-java.html

[10] Zixian Cai, Stephen M. Blackburn, Michael D. Bond, and Martin Maas. 2022. Distilling the Real Cost of Production Garbage Collectors. In *International IEEE Symposium on Performance Analysis of Systems and Software, ISPASS 2022, Singapore, May 22-24, 2022*. IEEE, 46–57. https://doi.org/10.1109/ISPASS55109.2022.00005

[11] Common Weakness Enumeration. 2021. 2021 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[12] L. Peter Deutsch and Daniel G. Bobrow. 1976. An Efficient, Incremental, Automatic Garbage Collector. *Commun. ACM* 19, 9 (1976), 522–526. https://doi.org/10.1145/360336.360345

[13] Jason Evans. 2006. Jemalloc. In *Proceedings of the 2006 BSDCan Conference, BSDCan'06, May 2006, Ottawa, CA.* http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

[14] Robert Fenichel and Jerome C. Yochelson. 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM* 12, 11 (1969), 611–612. https://doi.org/10.1145/363269.363280

[15] Google. 2014. tcmalloc. https://github.com/google/tcmalloc

[16] Bruce K. Haddon and William M. Waite. 1967. A Compaction Procedure for Variable-Length Storage Elements. *Comput. J.* 10, 2 (1967), 162–165. https://doi.org/10.1093/comjnl/10.2.162

[17] Matthew Hertz and Emery D. Berger. 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*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 313–326. https://doi.org/10.1145/1094811.1094836

[18] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 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*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 69–80. https://doi.org/10.1145/1028976.1028983

[19] Doug Lea. 1998. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html

[20] Daan Leijen. 2019. test1.smt2. https://github.com/daanx/mimalloc-bench/blob/8ee8911c02b7ee688d79a8a54effbf20ed52eede/bench/z3/test1.smt2

[21] Daan Leijen. 2021. mimalloc-bench: Suite for benchmarking malloc implementations. https://github.com/daanx/mimalloc-bench

[22] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 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 (Lecture Notes in Computer Science, Vol. 11893)*, Anthony Widjaja Lin (Ed.). Springer, 244–265. https://doi.org/10.1007/978-3-030-34175-6_13

[23] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. 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*, Jeremy Singer and Harry Xu (Eds.). ACM, 122–135. https://doi.org/10.1145/3315573.3329980

[24] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195. https://doi.org/10.1145/367177.367199

[25] Paige Reeves. 2021. MallocMS. https://www.mmtk.io/assets/videos/summer-2021-reeves.mp4

[26] Niklas Röjemo and Colin Runciman. 1996. Lag, Drag, Void and Use - Heap Profiling and Space-Efficient Compilation Revisited. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 34–41. https://doi.org/10.1145/232627.232633

[27] P. Styger. 1967. *LISP 2 Garbage Collector Specifications.* Technical Report TM-3417/500/00 1. System Development Cooperation, Santa Monica, CA.

[28] Z3 Theorem Prover. 2013. EntryCP.smt2. https://github.com/Z3Prover/z3test/blob/ad655e68bb118d7ba44d504d9b4cd7b5eb57ab70/regressions/verve/EntryCP.smt2

[29] Benjamin G. Zorn. 1993. The Measured Cost of Conservative Garbage Collection. *Softw. Pract. Exp.* 23, 7 (1993), 733–756. https://doi.org/10.1002/spe.4380230704