An Empirical Lower Bound on the Overheads of Production Garbage Collectors

Zixian Cai
Australian National University
zixian.cai@anu.edu.au

Stephen M. Blackburn
Australian National University
steve.blackburn@anu.edu.au

Michael D. Bond
Ohio State University
mikebond@cse.ohio-state.edu

Martin Maas
Google
mmaas@google.com

Abstract—**Despite the long history of garbage collection (GC) and its prevalence in modern programming languages, there is surprisingly little clarity about its true overheads. Even when evaluated using modern, well-established methodologies, crucial tradeoffs made by GCs can go unnoticed, which leads to misinterpretation of evaluation results. In this paper, we 1) develop a methodology that allows us to place a lower bound on the absolute overhead (LBO) of GCs, and 2) expose key performance tradeoffs of five production GCs in OpenJDK 17, a high-performance Java runtime. We find that with a modest heap size and across a diverse suite of modern benchmarks, production GCs incur substantial overheads, spending 7–82 % more wall-clock time and 6–92 % more CPU cycles relative to a zero-cost GC scheme. We show that these overheads can be masked by concurrency and generous provision of memory/compute. In addition, we find that newer low-pause GCs are significantly more expensive than older GCs, and sometimes even deliver *worse* application latency than stop-the-world GCs. Our findings reaffirm that GC is by no means a solved problem and that a low-cost, low-latency GC remains elusive. We recommend adopting the LBO methodology and using a wider range of cost metrics for future GC evaluations. This will not only help the community more comprehensively understand the performance characteristics of different GCs, but also reveal opportunities for future GC optimizations.**

I. Introduction

Garbage collection (GC) is ubiquitous in software systems. Managed languages, such as C#, Java, and JavaScript, continue to grow in popularity due to their productivity and safety benefits, which are in part provided by GC. On servers, many widely used web services, such as Twitter, GitHub, Shopify, and Alibaba, make extensive use of such languages. On clients, JavaScript engines are embedded in every web browser, and Java runtimes are embedded in every Android phone.

Despite the ubiquity of GC, there is a surprising lack of clarity regarding its costs. In this paper, we address two key problems: a) understanding the overheads of GC algorithms relative to a notional zero-cost baseline, and b) mitigating the misinterpretation of GC evaluations, which is common even when modern, well-established methodologies are used.

A lack of clarity regarding the overheads of a technology such as garbage collection can have profound impacts. For potential GC users, the decision of which GC algorithm to use—or even to use GC at all—hinges on a clear understanding of the true cost of GC algorithms. Without such information, a user might not realize how much a GC algorithm costs; such cost might not actually be acceptable to the user if they had known it. Furthermore, the decision of whether to use a language with GC cannot be easily reversed, as switching to an alternative, such as C, C++, or Rust, is typically a huge undertaking. For GC developers, the importance of a clear understanding of GC overheads is twofold. First, a costly GC technique presented as cheap can mislead the community into overusing it, discouraging future optimizations, and dismissing cheaper alternatives. Second, identifying costly techniques can reveal opportunities for hardware acceleration and optimization across the stack [1], especially in datacenter settings.

We now offer more detail regarding these two problems and outline our contributions.

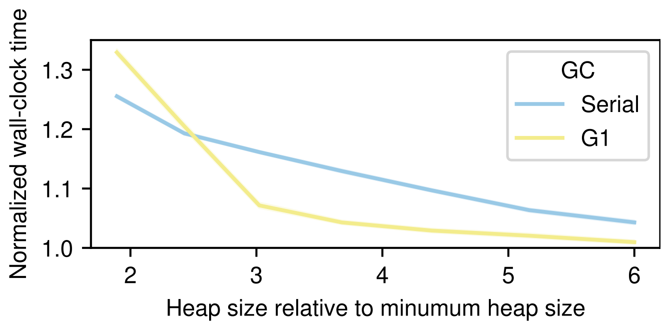
a) Unclear absolute overheads: When evaluating GCs (*i.e.*, GC algorithms), it is rare to report the overheads of GCs relative to a zero-cost baseline, presumably because of the lack of a good methodology for establishing such a baseline. Hertz and Berger [2] attempted to measure these overheads. However, their approach relies on simulation and it is unclear whether the conclusions can be extrapolated to GCs running on real hardware.

To address this problem, we introduce a *lower-bound overhead (LBO)* methodology to empirically estimate the absolute overheads of GC by approximating a zero-cost GC scheme. In contrast with Hertz and Berger [2], the LBO methodology can measure GCs running on real hardware, and does not require invasive runtime instrumentation. Due to its simplicity, the LBO methodology makes minimum assumptions about the GC implementation, and should therefore be easy to apply to different languages, runtimes, and GC algorithms.

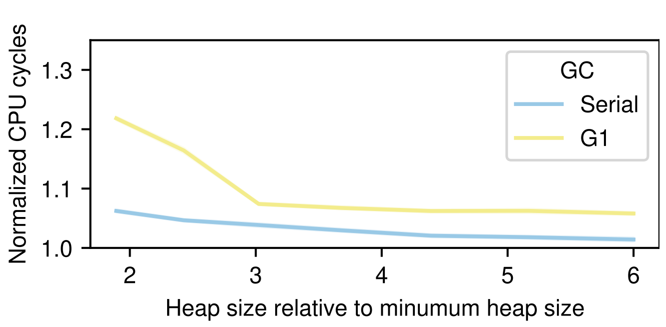
We systematically evaluate five production GCs in OpenJDK 17, the latest release of an industrial-strength, high-performance JVM. The LBO methodology offers clarity with respect to the true overheads of production GCs. We find that these GCs impose substantial overheads to the program execution across a diverse set of workloads. Using a modest heap size, our LBO methodology estimates that by using GC, applications spend 7–82 % more wall-clock time and 6–92 % more CPU cycles relative to a zero-cost GC scheme.

b) Misinterpretation of evaluation results: Not seeing the full picture of different collectors' performance can lead to misinterpretation. Specifically, knowing the limitations of a GC can change our verdict on its suitability in certain use cases.

In Fig. 1 and Fig. 2, we investigate three production OpenJDK collectors—Serial, G1, and Shenandoah—running the lusearch benchmark. We use four commonly reported

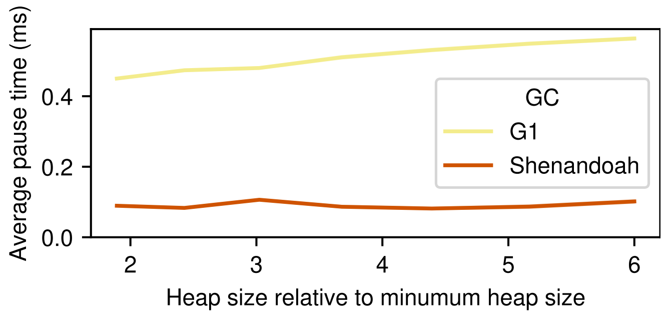


(a) Total wall-clock time, normalized to the best value. Lower is better. G1 outperforms Serial for all but the two smallest heap sizes.

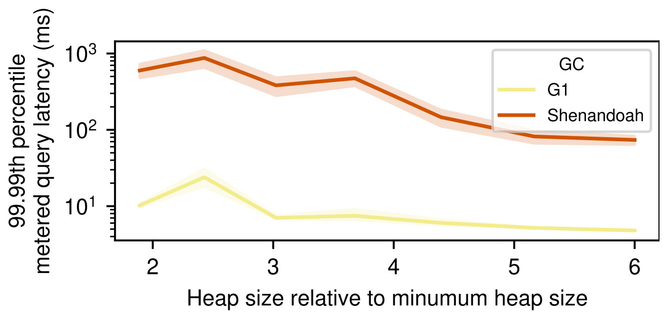


(b) Total CPU cycles, normalized to the best value. Lower is better. Serial outperforms G1 for all heap sizes.

Fig. 1: Two cost metrics for the total execution cost of Serial and G1 running the lusearch benchmark. The heap size is relative to the minimum required by G1. Each line and its shade represent the mean and 95 % confidence interval over 20 invocations.



(a) Average time (ms) per pause. Lower is better. Shenandoah consistently outperforms G1 for all heap sizes.



(b) 99.99th percentile metered query latency (ms). Lower is better. G1 consistently outperforms Shenandoah for all heap sizes.

Fig. 2: Two metrics for measuring the suitability of GCs on the latency-sensitive lusearch benchmark. The heap size is relative to the minimum required by G1. Each line and its shade represent the mean and 95 % confidence interval based on 20 invocations.

performance metrics. When looking at (parts of) each subfigure in isolation, we can arrive at different, sometimes diametrical, verdicts on the suitability of these collectors.

1) It is well known that GC is fundamentally a time–space tradeoff [3], [4]. If we chose to focus on a single heap size in Fig. 1a, we could arrive at two opposite conclusions. At 1.9× heap, G1 is about 7 % slower than Serial, while at 3.0× heap, G1 is about 9 % faster than Serial.
2) G1 appears to be better if we optimize for the application throughput (Fig. 1a). However, its true cost is masked by parallelism, which we can only see when we measure the CPU cycles consumed. On a machine with less free hardware to spare, the performance of G1 might degrade due to the overall higher CPU consumption (Fig. 1b). Similarly, in a multi-tenant environment, extra cycles used by the GC lead to fewer cycles available for other applications on the same host (opportunity cost).
3) When optimizing for latency-sensitive workloads, such as lusearch, Shenandoah seems better suited because of shorter pause times[1] (Fig. 2a). However, Shenandoah ends up delivering worse application latency than G1 (Fig. 2b) due to stalling the application when it cannot keep up with the allocation rate (discussed in detail in Section IV-C).

[1]The distribution of the pause times is more appropriate for latency-sensitive workload. The average pause time is used here for the simplicity of illustration.

The above examples highlight the risks of evaluating GC with limited metrics. In this paper, we measure five production GCs using multiple metrics including wall-clock time, CPU cycles, and application latency. Our results suggest that the tradeoffs made by different GCs can go unnoticed when employing naïve evaluation regimes such as those shown above. Our surprising findings include situations where low-pause GCs (Shenandoah and ZGC) achieve worse application latency than simple stop-the-world GCs while also having much worse throughput.

The LBO methodology is a simple way to estimate the absolute overheads of GCs, revealing the substantial costs incurred by widely used production GCs. We recommend using richer sets of metrics when evaluating GCs. This is important for revealing the tradeoffs and limitations of different collectors, which assists GC users in choosing the most appropriate GC for a plethora of existing and emerging workloads. Our findings about the overheads of production GCs invite future research.

TABLE I: Collectors we studied.

Collector	Year	Generational	Parallel GC	Concurrent GC	Barriers
Epsilon	2018	No (no GC)	No (no GC)	No (no GC)	No (no GC)
Serial	1998	Yes	No	No	Write (card-marking)
Parallel	2005	Yes	Yes	No	Write (card-marking)
G1 [5]	2009	Yes	Yes	Yes (tracing)	Write (card-marking and SATB)
Shenandoah [6]	2019	No[2]	Yes	Yes (tracing and copying)	Write (SATB) and read (LVB[3])
ZGC [7], [8]	2018	No	Yes	Yes (tracing and copying)	Read (LVB)

II. Background and Related Work

In this section, we review sources of GC overhead and describe the current, established evaluation methodologies. We highlight three sources of GC cost, which on their own are insufficient to adequately characterize GC overheads. There are also a few attempts to measure the overall cost of GC, and we will point out their limitations. Finally, we will give an overview of the collectors that we study in this work, and how they contain these hard-to-measure sources of GC cost.

A. Costs Tightly Coupled with Application Execution

A GC can be considered in terms of three principal activities: *allocation*, *identification*, and *reclamation* [9]. Allocation is performed by the application using space provided by the GC. Identification establishes which part of the heap is live and which may be reclaimed. Reclamation makes unused space available for re-use. For performance reasons, some GC mechanisms such as allocators and read and write barriers are tightly coupled with application (mutator) execution, implemented using the fast-and-slow-path paradigm [10]. Fast paths are frequently executed, often inlined into the mutator code by the JIT compiler. Because they are so tightly integrated into the mutator, their costs are hard to measure.

Blackburn *et al.* [4] placed an upper bound on the cost of a bump-pointer allocator by compiling the allocation fast-path out-of-line. Then, that upper bound was used to derive the cost of a free-list allocator using their relative performance.

Barriers are code snippets that mediate mutators' heap operations on behalf of the GC. Their cost has been investigated by prior work [11]–[14]. The methodologies used in prior work remove the requirement of barriers for correctness, *e.g.*, through a full-heap trace for a generational GC, and then measure the execution with and without barriers. These approaches require deep understanding of the GC implementation and modification of the collector and/or runtime, and consequently are not trivially applied to arbitrary language runtimes.

Reclamation is typically performed directly by the GC but can also be performed by the mutator. For example, in the case of naïve reference counting (RC) [15], the site that decrements the reference count of an object to zero can immediately reclaim the memory. Blackburn *et al.* [4] measured the cost of a deferred reference counting collector in terms of the mutator time.

B. Indirect Costs/Benefits

Apart from the direct costs of GC, it can also bring indirect costs, and sometimes benefits. A common source is mutator locality. GC impacts locality through sharing caches with the mutator. In the case of concurrent GC, GC threads contend with mutator threads when they share caches. In the case of stop-the-world pauses, GC code displaces the cache, leaving mutator threads resuming with a cold cache. However, GC can also improve the mutator locality through rearranging objects. A compacting or evacuating GC can improve the spatial locality by moving objects that are frequently accessed together to be spatially closer to each other [16].

Some of the locality impact might be observable through hardware performance counters, such as cache miss events (*e.g.*, LLC and TLB). However, in general, it is hard to tease out the locality impact of GC running on real hardware, because that would require achieving the same GC effects (like compacting the heap) without affecting cache state. It may be possible to achieve such an effect under simulation, but it would be a major undertaking, likely requiring invasive changes to the runtime and significant compute resources to evaluate.

C. Absolute Overheads of Garbage Collection

Hertz and Berger [2] attempted to quantify the overhead of garbage collection over an explicit memory management regime. By tracing the program execution, they constructed an oracle which guided the insertion of `malloc` and `free` calls. However, they relied on simulation, the fidelity of which is unknown. In addition, the algorithms and the workload used in the evaluation are dated. The LBO methodology measures GC running on real hardware and does not rely on an oracle to construct a baseline.

Some work (such as [17]) has measured the cost of conservative GC in the context of explicitly managed languages. We focus on precise GC in managed languages.

Numerous studies (such as [4]) measure GC pauses to quantify the GC costs. We show that this approach is problematic because some GC costs happen outside of GC pauses and are tightly coupled with mutator activities. In particular, for concurrent GCs, the GC pauses are shorter, and the majority of the work is shifted to run concurrently with mutators.

D. Garbage Collection Algorithms in HotSpot

Table I shows the production GCs studied in this work. The first of these, Epsilon, does not actually collect garbage, but is used as part of our LBO methodology. The others can be divided into three groups: *stop-the-world collectors* (Serial and Parallel), *concurrent tracing collector* (G1), and *concurrent copying collectors* (Shenandoah and ZGC). A stop-the-world

[2]In development, see https://openjdk.java.net/jeps/404.
[3]Brooks prior to JDK 13, Baker prior to JDK 14.

collector requires all mutators to be stopped while it is running, *i.e.*, it does not exhibit any concurrency with respect to the mutator. A concurrent tracing collector performs garbage identification via a trace which executes concurrently with the mutator. The trace does not modify the heap, but marks reachable objects as live. The correctness of the concurrent trace is typically protected by write barriers. A concurrent copying collector performs reclamation concurrently with the mutator by copying objects. This involves modifying the heap and ensuring that the concurrently executing mutator maintains a coherent view of the heap even when objects it references are moved. The correctness of concurrent copying is typically protected by read barriers.

The production GCs we study are all based on tracing to establish liveness of objects. Apart from the STW collectors, where a mark-sweep-compact policy is used for the mature space, the concurrent collectors strictly rely on copying to reclaim memory, with G1 performing the copying in STW pauses, while Shenandoah and ZGC perform copying concurrently.

Since the main design goals of the two concurrent copying collectors are to reduce the GC pause times and improve the responsiveness for latency-sensitive applications, we also refer to them as the *low-pause collectors* in the rest of the paper.

III. The Lower-bound Overhead Methodology

As discussed in Section I, it is important to quantify the absolute overhead of GCs relative to a zero-cost baseline (with respect to some metrics). Knowing the overhead can, for example, help us assess whether GC optimizations can yield fruitful improvements to the overall performance of the application. In this section, we introduce the lower-bound overhead (LBO) methodology that gives a lower bound on the absolute overhead of GCs using empirical data. First, we sketch the intuitions underpinning LBO using an example with three production GCs running the h2 benchmark. This also serves as a running example throughout the section. Then, we give a formal definition of the LBO methodology. Finally, we discuss the advantages and the limitations of the LBO methodology.

A. Intuitions

It is easy to measure the relative overheads of different GCs. Table II shows the total cycle usage for three production GCs, Serial, Parallel, and Shenandoah, running the h2 benchmark. When normalized to the best value (Serial), we know that Serial is the cheapest GC in terms of cycles, with Parallel 0.2 % and Shenandoah 102.3 % more expensive. However, we do not know the absolute overhead of each collector.

If an ideal GC existed, we could measure its cycle usage, and then use that to normalize the cycles of each real collector, giving us the absolute overheads. The ideal GC has all the benefits of GC (such as improved locality through defragmenting the heap) but none of the costs (such as traversing the heap to identify live objects). Of course, the ideal GC does not exist. Our key insight is that we can establish an *upper bound* on ideal cycles using a real GC by subtracting cycles that we can ascribe to GC from its total cycles.

TABLE II: The total CPU cycles consumed when running h2 with a generous $3\times$ heap (see Section IV-A) using three different collectors. Lower the better. The cycles are also normalized to the best collector (Serial) shown in green.

Collector	**Total** billion cycles	**Normalized**
Parallel	108.33	1.002
Serial	108.12	1.000
Shenandoah	218.72	2.023

TABLE III: Further attribute the CPU cycles consumed in Table II to cycles used during stop-the-world (STW) pauses, and other cycles. The best other cycles is shown in green. This best value is used for the LBO calculation in Table IV.

Collector	**STW**	**Other** billion cycles	**Total**
Parallel	4.46	103.87	108.33
Serial	2.75	105.37	108.12
Shenandoah	0.03	218.69	218.72

TABLE IV: The LBO value for each collector can be obtained by normalizing the respective total cycles by the smallest other cycles in Table III.

Collector	**Total** billion cycles	**LBO**
Parallel	108.33	108.33/103.87 = 1.043
Serial	108.12	108.12/103.87 = 1.041
Shenandoah	218.72	218.72/103.87 = 2.106

TABLE V: A collector with a smaller other cycles (shown in green) is found. This allows us to derive tighter lower bounds, *i.e.*, larger LBO values.

Collector	**Other**	**Total** billion cycles	**LBO**
Parallel	103.87	108.33	108.33/100.00 = 1.083
Serial	105.37	108.12	108.12/100.00 = 1.081
Shenandoah	218.69	218.72	218.72/100.00 = 2.187
Hypothetical	100.00	109.50	109.50/100.00 = 1.095

As shown in Table III, one example of measuring the apparent GC cycles is to measure the cycles spent in stop-the-world (STW) pauses, where no mutator activities happen, and the cycles spent outside STW pauses (other).

By definition, the other cycles of each GC is strictly greater than the ideal cycles. In other words, they yield upper bounds of the ideal cycles. The *minimum* of the other cycles among the three collectors gives *the tightest upper bound* of the ideal cycles for that set of collectors. We obtain the minimum using the 103.87 billion other cycles of Parallel.

When normalizing the total cycles of each collector by this tightest upper bound on ideal cycles, we get a lower bound on the overheads (LBO) of the respective collector (see Table IV).

The estimative nature of the LBO methodology means we can gradually refine the estimation and obtain tighter lower bounds (numerically larger). As shown in Table V, when a collector (a hypothetical collector in this case) with a smaller other cycles is found, we have a better estimation of the ideal cost, which in turn allows us to have a better estimation of the absolute overheads of each GC, which is reflected as the larger LBO values for all collectors compared with Table IV.

B. Definition

Even though cycles is used in the previous section as an example, the LBO methodology does not depend on the choice of cost metric. Any notion of cost metrics, such as CPU cycles, the wall-clock time, or energy consumed, can be used as the Cost in the formulae below. We use tilde to denote estimations. For example, $\widetilde{\text{Cost}}_{\text{ideal}}$ is an estimation of $\text{Cost}_{\text{ideal}}$.

Let G be the set of GC algorithms that we study. In the previous example, $G = \{\text{Parallel}, \text{Serial}, \text{Shenandoah}\}$. For some fixed workload W and hardware H, we have the ideal cost $\text{Cost}_{\text{ideal}}$ intrinsic to running W on H. Then, for each $g \in G$, we define its absolute overhead by normalizing its total cost (*i.e.*, Cost measured throughout the program execution) by the ideal cost: $\text{Overhead}(g) = \text{Cost}_{\text{total}}(g) / \text{Cost}_{\text{ideal}}$.

In reality, we do not know $\text{Cost}_{\text{ideal}}$, and therefore, cannot calculate $\text{Overhead}(g)$. However, for each GC $g \in G$, we can measure the apparent GC cost $\widetilde{\text{Cost}}_{\text{GC}}(g)$, *e.g.*, by only counting the Cost during STW pauses. The insight is that $\text{Cost}_{\text{total}}(g)$ and $\widetilde{\text{Cost}}_{\text{GC}}(g)$ allow us to place an upper bound on $\text{Cost}_{\text{ideal}}$.

By definition, $\text{Cost}_{\text{total}}(g) - \widetilde{\text{Cost}}_{\text{GC}}(g) > \text{Cost}_{\text{ideal}}$. That is, the difference yields an upper bound of $\text{Cost}_{\text{ideal}}$. For the set of GCs G we study, the tightest upper bound on $\text{Cost}_{\text{ideal}}$ can be found by $\min_{g \in G} \text{Cost}_{\text{total}}(g) - \widetilde{\text{Cost}}_{\text{GC}}(g)$. We use $\widetilde{\text{Cost}}_{\text{ideal}}$ to denote this tightest upper bound. Note that the choice of g that achieves the minimum depends on the workload W and hardware H. In the previous example, we obtain the minimum when $g = \text{Parallel}$ (see Table II).

Then, for each $g \in G$, we define its lower bound overhead by $\text{LBO}(g) = \text{Cost}_{\text{total}}(g) / \widetilde{\text{Cost}}_{\text{ideal}}$, analogous to the definition of $\text{Overhead}(g)$. Since $\widetilde{\text{Cost}}_{\text{ideal}}$ is an upper bound of $\text{Cost}_{\text{ideal}}$, $\text{LBO}(g)$ is a lower bound on $\text{Overhead}(g)$. For the previous example, this calculation is detailed in Table IV.

C. Discussion

The biggest advantage of the LBO methodology is its simplicity. It is agnostic to the particular GC algorithms and runtimes. The only requirement is to measure the apparent GC cost, which is conceptually easy, and should be possible with simple, non-intrusive instrumentations. In fact, for OpenJDK, this can be implemented using JVMTI callbacks [18], and for .NET, implemented on top of GCRealTimeMon [19].

However, the quality of the LBO estimation greatly depends on the ability to approach the ideal by identifying and attributing as much Cost to GC as possible. For example, if one only attributes the Cost incurred using STW pauses to the GC, then the LBO estimation based only on concurrent collectors will be too low since the costs of concurrent GC tasks will not be attributed to the GC. To improve the estimation, the approach needed depends on the cost metric and how the apparent GC cost is captured. We give two specific examples.

1) When measuring the CPU cycles used, one can get per-thread readings from the performance monitoring units (PMUs), and then combine the activities of GC threads both during pauses and when running concurrently. This gives better estimations than measuring the whole-process cycles in and outside of STW pauses.
2) When measuring the wall-clock time, one can try to include GCs that are cheap outside STW pauses where possible. For example, a STW GC with no barrier is likely to give better estimation of the ideal time than a STW GC with a write barrier (*e.g.*, a generational GC), which in turn is likely better than concurrent GCs.

We also note that the magnitude of LBO values depends on the performance of GCs relative to the rest of the runtime. For example, with everything else constant, the same GC is likely to have a lower LBO value when paired with a simple interpreter than an optimizing compiler, because the application code is more expensive to execute with the interpreter (*i.e.*, higher $\text{Cost}_{\text{ideal}}$).

Furthermore, compiler optimizations that interact with allocations can also lead to lower LBO values with no change to the GC. For example, a compiler that performs escape analysis can reduce overall allocation, and therefore reduces the pressure on the GC. Such analysis allows objects to be allocated on the stack rather than on the heap when the objects do not escape.

IV. Case Study: Collectors in OpenJDK 17

Recall that we address two key problems: 1) the lack of good methodology to quantify the absolute overheads of GCs, and 2) the fact that GC evaluation results are prone to misinterpretation even when well-established methodologies are used. The previous section proposed the LBO methodology to estimate the absolute overheads of GCs. In this section, we perform a case study of GCs in OpenJDK 17, the latest release of a high-performance production JVM. First, we apply the LBO methodology on these GCs, and measure their wall-clock time LBOs (*time LBOs*) and total cycle LBOs (*cycle LBOs*). Second, we concretely show how the evaluation of these GCs can be misinterpreted when deploying only a few metrics. We then demonstrate how to mitigate this problem by using a richer set of metrics, including the LBO values.

Our approach allows us to observe the following:

1) *Garbage collection overheads:* When deployed with a modest heap size, production collectors can incur substantial absolute overheads across workloads. This is revealed by both high time and cycle LBOs. A surprising trend is that in addition to being significantly slower and cycle-intensive, the low-pause collectors fail to deliver better application latency for the evaluated workloads.
2) *Misinterpretation of evaluation results:* We highlight three important types of misinterpretation: a) not considering opportunity cost, b) not considering concurrency overhead, and c) measuring pause time instead of application latency.

Based on our observations, we recommend: 1) that the LBO methodology should be used to report the absolute overheads of GCs, and 2) that more performance and cost metrics should be used in order to evaluate GCs holistically.

TABLE VI: LBO total time overhead averaged over 16 benchmarks. The best value for each heap size is shown in green. Where a collector cannot run all benchmarks at a particular heap size, the entry is left blank. Parallel outperforms other collectors, except at 1.4×, where G1 has the lowest cost.

GC	1.4×	1.9×	2.4×	3.0×	3.7×	4.4×	5.2×	6.0×
Ser.	1.42	1.17	1.14	1.13	1.11	1.10	1.09	1.09
Par.	1.41	1.09	1.07	1.06	1.05	1.04	1.04	1.03
G1	1.24	1.16	1.11	1.09	1.08	1.07	1.07	1.06
Shen.	*	1.94	1.64	1.43	1.37	1.30	1.25	1.23
ZGC	*	*	1.82	1.54	1.39	1.32	1.27	1.23

TABLE VII: LBO cycle overhead averaged over 16 benchmarks. The best value for each heap size is shown in green. Where a collector cannot run all benchmarks at a particular heap size, the entry is left blank. Serial consistently outperforms other collectors for all heap sizes.

GC	1.4×	1.9×	2.4×	3.0×	3.7×	4.4×	5.2×	6.0×
Ser.	1.22	1.08	1.06	1.06	1.04	1.04	1.04	1.04
Par.	1.70	1.15	1.12	1.10	1.07	1.06	1.06	1.05
G1	1.54	1.34	1.17	1.14	1.10	1.09	1.09	1.09
Shen.	*	1.75	1.54	1.47	1.42	1.39	1.36	1.33
ZGC	*	*	1.92	1.68	1.55	1.46	1.40	1.34

A. Methodology

a) Benchmarks and latency measures: We use a snapshot release of the forthcoming Chopin release of DaCapo benchmark suite [20]. We exclude the benchmarks cassandra, h2o, and kafka, because they include depreciated Java features that are not compatible with JDK 17.

DaCapo's Chopin snapshot release includes a number of latency-sensitive benchmarks. Apart from the benchmark time, which is reported for all benchmarks, DaCapo reports two measures of latency, *simple* and *metered*, for the latency-sensitive benchmarks. Latency-sensitive services handle remotely issued requests (*e.g.*, over a network) that arrive at some remotely determined *rate*. When such a system is unable to process a request immediately, it is placed in a queue. The latency of a request is impacted by three major sources of delay: the uninterrupted time taken to compute the request; the time taken inclusive of interruptions such as GC and scheduling; and the time taken inclusive of interruptions and queuing. DaCapo's *simple* latency ignores queuing, while *metered* latency models requests coming at a metered rate with an arbitrary-sized queue. When an interruption such as a GC pause occurs, the metered measure reflects the delay that this imposes not only on the currently executing requests, but also on those that are enqueued during the delay. We use metered latency here because it more accurately models latency-sensitive services.

b) Cost metrics: We use the Java Virtual Machine Tool Interface (JVMTI) [18] to measure the GCs. The interface provides callbacks for when a GC pause starts and when it ends. This allows us to break down the cost for the execution into the costs incurred in stop-the-world (STW) pauses and outside STW pauses. Our JVMTI agent captures metrics including the wall-clock time, CPU cycles, instruction counts, cache misses, and Intel RAPL energy measurements. The performance counter readings are obtained from the `perf_events` subsystem in the Linux kernel. In this section, we focus on two important metrics, the wall-clock time and CPU cycles.

c) JVM parameters: In this paper, we focus on the out-of-the-box performance characteristics of GCs, and *deliberately do not* set any GC-related parameters except for the heap size. GC tuning is often specific to particular (classes of) workloads, whereas our concern is how well each GC handles a diverse set of workloads. Also, in general, GC tuning is an open-ended problem, outside the scope of this paper. We report the performance of each GC for different heap sizes, because the performance of GC is sensitive to the heap size [20]. Apart from Epsilon, which does not perform GC, we set the heap size relative to the minimum heap size required to run each benchmark (*e.g.*, 2.0× means that the heap size is set to be twice as big as the minimum heap required for a particular benchmark). The minimum heap size for each benchmark is measured using G1 because it is the most space-efficient GC among the ones we study. The only other JVM parameters we set are `-server -XX:-TieredCompilation -Xcomp`, to speed up the warmup of the JVM and reduce the experimental noise due to JIT compilation. We omit parameters `-XX:-TieredCompilation -Xcomp` for tradebeans and tradesoap, because these parameters cause these two benchmarks to crash for the version of OpenJDK we use.

d) Execution methodology: For each configuration, we invoke each benchmark for 20 times. We interleave invocations of different configurations to minimize bias due to systemic interference. In each *invocation*, the benchmark performs five *iterations* and we report results from the last iteration, with the first four iterations serving to warm up the runtime. For each configuration, we report the mean and the 95 % confidence interval (CI) based on the 20 invocations.

e) Hardware: All machines have Intel Core i9-9900K (Coffee Lake) CPUs (8 cores, 16 threads), with 4×32G DDR4-3200 memory. We disable the dynamic frequency scaling (*i.e.*, Turbo Boost) to reduce experimental noise.

f) Software: All machines run identical Ubuntu 18.04.6 LTS images with `5.4.0-89-generic` kernels. We use the Eclipse Temurin (formerly AdoptOpenJDK)'s distribution of OpenJDK. For OpenJDK 17, we use the `Temurin-17.0.1+12` release. We focus on OpenJDK 17 in this paper, as it is the latest LTS release as of writing and is supposed to bring many GC performance improvements (such as [21], [22]) since OpenJDK 11. We also measured OpenJDK 11 using the `AdoptOpenJDK-11.0.11+9` release, and the overall results were quite similar to those of OpenJDK 17. All benchmarks are executed on an otherwise idle machine, with as many background daemons and periodic tasks disabled as possible.

B. Results: Garbage Collection Overhead

First, we estimate the absolute overheads of the five garbage collectors other than Epsilon using the LBO methodology. Epsilon is only used in calculating the LBO values in the cases where it is able to run a benchmark without exhausting the memory available on our machine.

TABLE VIII: Total time overhead at 3× heap using LBO. Lower is better. xalan is excluded from the summary statistics due to ZGC failing to run, and the corresponding row is grayed out. The best results for each benchmark are shown in green (light green for xalan). Each LBO is the mean of 20 invocations. The 95 % CIs are all less than 2 % except for 8 % for pmd/ZGC. Parallel outperforms other collectors for most benchmarks.

Benchmark	**Serial**	**Parallel**	**G1**	**Shen.**	**ZGC**
avrora	1.009	1.025	1.031	1.101	1.072
batik	1.146	1.092	1.067	1.089	1.077
biojava	1.006	1.007	1.033	1.221	1.882
eclipse	1.050	1.010	1.062	1.127	1.136
fop	1.159	1.129	1.201	1.414	2.107
graphchi	1.021	1.011	1.020	1.191	1.125
h2	1.121	1.044	1.093	1.570	1.964
jme	1.003	1.003	1.002	1.010	1.006
jython	1.037	1.022	1.065	1.609	2.204
luindex	1.007	1.011	1.044	1.150	1.097
lusearch	1.230	1.105	1.135	3.766	2.701
pmd	1.613	1.124	1.188	1.559	1.777
sunflow	1.360	1.113	1.113	2.099	2.031
tomcat	1.157	1.137	1.144	1.305	1.914
tradebeans	1.120	1.020	1.164	1.423	1.327
tradesoap	1.094	1.011	1.112	1.340	1.282
xalan	3.380	3.074	3.496	30.176	
zxing	1.030	1.058	1.028	1.274	1.235
min	*1.003*	*1.003*	*1.002*	*1.010*	*1.006*
max	*1.613*	*1.137*	*1.201*	*3.766*	*2.701*
mean	*1.127*	*1.054*	*1.088*	*1.485*	*1.584*
geomean	*1.118*	*1.053*	*1.087*	*1.404*	*1.509*

Table VI shows time LBOs of the collectors we study while Table VII shows cycle LBOs. Each table covers eight different heap sizes, ranging from a small 1.4× heap to a very generous 6.0× heap (see Section IV-A for the multiplier notation). The LBO values for each configuration (a collector at a heap size) is the geometric mean for 16 [4] DaCapo benchmarks.

The production GCs incur substantial overheads. For a modest 2.4× heap, production GCs on average spend 7–82 % more wall-clock time and 6–92 % more cycles relative to a zero-cost GC. Even for a very generous 6.0× heap, the overheads are as much as 23 % in terms of time and 34 % in terms of cycles. For all heap sizes shown, Serial achieves the lowest overheads in terms of cycles, while Parallel achieves the lowest overheads in terms of time for all but the smallest heap size.

Table VIII and Table IX take a closer look at the 3.0× heap size. This allows us to observe how different collectors behave when challenged with a diverse, modern workload.

C. Analysis of Results

We compare the performance trends among and within three groups of GCs (see Section II-D).

a) Stop-the-world (STW) GCs vs Concurrent GCs: Overall, STW collectors (Serial and Parallel) are cheaper than concurrent collectors (G1, Shenandoah, and ZGC), both in terms of the time and cycles. The exception is that Serial is uncompetitive in terms of time due to its lack of parallelism. At 3.0× heap, in terms of cycles, STW collectors are never more expensive than concurrent collectors.[5] In terms of time, STW collectors are only more expensive than concurrent collectors for two out of 17 benchmarks: batik and zxing.[6]

[4]We use 18 DaCapo benchmarks in total. However, eclipse and xalan are excluded in the geometric mean calculation because too many collectors were not able to run these two benchmarks for small heap sizes. If we were to include these two benchmarks, a lot more entries would be missing from the tables.

TABLE IX: Cycle overhead at 3× heap using LBO. Lower is better. xalan is excluded from the summary statistics due to ZGC failing to run, and the corresponding row is grayed out. The best results for each benchmark are shown in green (light green for xalan). Each LBO is the mean of 20 invocations. The 95 % CIs are all less than 2 % except for the following: 3 % for batik/ZGC, and 10 % for pmd/ZGC. Serial outperforms other collectors for most benchmarks.

Benchmark	**Serial**	**Parallel**	**G1**	**Shen.**	**ZGC**
avrora	1.007	1.014	1.047	1.201	1.209
batik	1.146	1.991	1.565	1.454	1.915
biojava	1.006	1.014	1.046	1.521	3.962
eclipse	1.054	1.109	1.317	1.390	1.474
fop	1.160	1.203	1.460	1.893	2.227
graphchi	1.008	1.031	1.048	1.226	1.226
h2	1.053	1.055	1.123	2.131	2.645
jme	1.071	1.132	1.091	1.517	1.470
jython	1.036	1.034	1.072	2.038	2.444
luindex	1.010	1.018	1.067	1.207	1.199
lusearch	1.045	1.030	1.081	1.213	1.268
pmd	1.096	1.178	1.286	1.563	1.655
sunflow	1.123	1.085	1.086	1.239	1.075
tomcat	1.010	1.015	1.022	1.119	1.119
tradebeans	1.060	1.049	1.211	1.637	2.171
tradesoap	1.052	1.029	1.117	1.710	2.155
xalan	1.211	1.633	1.783	1.744	
zxing	1.017	1.034	1.027	1.275	1.238
min	*1.006*	*1.014*	*1.022*	*1.119*	*1.075*
max	*1.160*	*1.991*	*1.565*	*2.131*	*3.962*
mean	*1.056*	*1.119*	*1.157*	*1.490*	*1.791*
geomean	*1.055*	*1.103*	*1.148*	*1.462*	*1.671*

b) Single-threaded vs multi-threaded STW GC: Between two stop-the-world collectors, Parallel is more expensive than Serial in terms of cycles. The difference is as much as 48 % for a small 1.4× heap. However, the converse is true in terms of time. This is presumably due to the synchronization overhead of a multi-threaded collector. At 3.0× heap, in terms of cycles, Parallel is only cheaper than Serial for three out of 17 benchmarks: lusearch, tradebeans, and tradesoap.[7] In terms of time, Serial is only cheaper than Parallel for avrora.[8]

c) Concurrent tracing vs concurrent copying GC: Among concurrent collectors, the newer, concurrent copying collectors (Shenandoah and ZGC) are significantly more costly than the concurrent tracing collector (G1). The difference is up to 75 % in terms of cycles (G1 vs ZGC at 2.4× heap) and 78 % in terms of time (G1 vs Shenandoah at 1.9× heap). This is presumably due to the use of costly read barriers

[5]For sunflow, Parallel is 1 % more expensive than ZGC, but the confidence intervals overlap.

[6]The differences for jme and sunflow are negligible (and also not statistically significant for sunflow).

[7]Not statistically significant for jython and sunflow.

[8]Not statistically significant for biojava, luindex, and sunflow.

TABLE X: Percent of time spent in STW pauses averaged over 16 benchmarks. Lower the better. The best value for each heap size is shown in green. Where a collector cannot run all benchmarks at a particular heap size, the corresponding entry is left blank. ZGC consistently outperforms other collectors where it runs.

GC	1.4×	1.9×	2.4×	3.0×	3.7×	4.4×	5.2×	6.0×
Ser.	9.0	4.7	3.7	3.1	2.7	2.5	2.4	2.3
Par.	9.0	2.9	2.1	1.7	1.3	1.1	1.0	0.9
G1	5.2	2.8	1.6	1.3	0.9	0.9	0.7	0.7
Shen.		0.3	0.2	0.2	0.1	0.1	0.1	0.1
ZGC			0.1	0.0	0.0	0.0	0.0	0.0

TABLE XI: Percent of cycles spent in STW pauses averaged over 16 benchmarks. Lower the better. The best value for each heap size is shown in green. Where a collector cannot run all benchmarks at a particular heap size, the corresponding entry is left blank. ZGC consistently outperforms other collectors where it runs.

GC	1.4×	1.9×	2.4×	3.0×	3.7×	4.4×	5.2×	6.0×
Ser.	4.3	2.1	1.6	1.4	1.2	1.1	1.0	1.0
Par.	13.1	5.3	4.0	3.4	2.8	2.6	2.4	2.2
G1	8.1	5.0	3.3	2.8	2.1	2.1	1.9	1.8
Shen.		0.1	0.1	0.1	0.1	0.1	0.1	0.0
ZGC			0.0	0.0	0.0	0.0	0.0	0.0

and the (un)timeliness of reclamation, which we will discuss below. At 3.0× heap, G1 consistently outperforms Shenandoah and ZGC for all benchmarks in terms of time. In terms of cycles, Shenandoah/ZGC is only cheaper than G1 for batik, and xalan (not statistically significant for sunflow).

d) Pathological Modes of Concurrent Copying Collectors: We find that the low-pause, concurrent copying collectors can deliver worse application latencies while incurring substantial costs. We observe two pathological modes of concurrent copying collectors when challenged with workloads that have high allocation rates (GB/s relative to the heap size).

One stark result is for xalan, where Shenandoah has an enormous time LBO of 30.2, about ten times that of Serial/-Parallel/G1. ZGC simply failed to run xalan with OOM errors. However, Shenandoah has a modest cycle LBO of 1.74, which is close to the 1.63 LBO of Parallel and even slightly better than the 1.78 LBO of G1. To understand this behavior, recall that Shenandoah and ZGC rely on tracing to establish liveness, and strictly rely on evacuation for reclamation (see Section II-D). The consequence is a substantial delay between when an object becomes unreachable and when the unreachable object is reclaimed. The delay's impact is amplified by the high allocation rates of benchmarks such as xalan and lusearch; because allocation would fail if reclamation did not keep up with the allocation rate, and the collector had to resort to STW collections. In this case, it might be more beneficial to use a STW collector in the first place and thus avoid the concurrency overhead (which we will discuss in Section IV-D).

By examining the logs from Shenandoah, we observe two pathological modes. First, the untimeliness of reclamation causes allocation failures, and Shenandoah requires STW collection to finish an in-flight concurrent collection (known as degenerated GCs in Shenandoah). Second, in order to avoid STW collections, Shenandoah throttles allocations by stalling the mutator at allocation sites (known as pacing in Shenandoah, or "allocation stall" in ZGC). Since sleeping threads do not contribute to the cycles consumed, but increase the wall-clock time needed to run a workload, this explains the much higher time overhead but modest cycle overhead.

D. *Misinterpretation of Evaluation Results*

Prior work [3], [4] points out common pitfalls and respective mitigations when evaluating GCs. We have applied these suggestions where possible (see Section IV-A). For example, we control the heap size relative to the minimum heap size required to run a given workload. We reaffirm the fundamental time–space tradeoff is still applicable in a modern, diverse benchmark suite (such as shown in Table VI). However, these suggestions are not sufficient to avoid misinterpretation of evaluation results, especially in light of modern hardware, concurrent GCs, and emerging latency-sensitive workloads. In this section, we highlight three important types of misinterpretation. In the next section, we make recommendations on mitigations.

a) Opportunity cost: Comparing Table VI and Table VII, we notice that all collectors have higher cycle LBO than time LBO.The only exceptions are Serial, which is a single-threaded GC, and Shenandoah at small heap sizes due to pacing, which we discussed previously. In particular, the cycle LBOs of Parallel and G1 are about 0.3 larger than the respective time LBO at a small 1.4× heap. In the extreme case, such as for batik at 3.0× heap, Parallel has a mere 1.09 time LBO but a significant 1.99 cycles LBO, the highest among all collectors studied. This difference reveals that substantial cycle overheads can go unnoticed when only the wall-clock time is reported.

Historically, garbage collector performance has been mostly measured using wall-clock time only. This implicitly assumes that on machines with free cores available, these cores may be used by GC with no repercussions. On modern, massively parallel hardware, this assumption can mean that a significant portion of the hardware is dedicated to the GC. This assumption, however, ignores the incurred opportunity cost.

On multi-tenant hosts, which are increasingly common to increase utilization in datacenters, more CPU cores taken by GC threads lead to fewer cores available for other applications on the same host. That is, when heavily relying on parallelism, a collector will not only run slower when deployed in multi-tenant settings, because fewer cores are available to run GC, but also negatively impact other applications on the same host. Even when the server has only one application, fewer cycles required to run the application can reduce energy consumption.

b) Concurrency overhead: A commonly used metric to tune GC for throughput is the fraction of time spent in GC (such as the `-XX:GCTimeRatio` suggested in the GC tuning guide from the vendor [23]). Table X and Table XI show the fraction of time and cycles spent in STW pauses for different GCs. Similar to Table VI and Table VII, the results are grouped by heap sizes, and show the geometric means over 16 benchmarks.

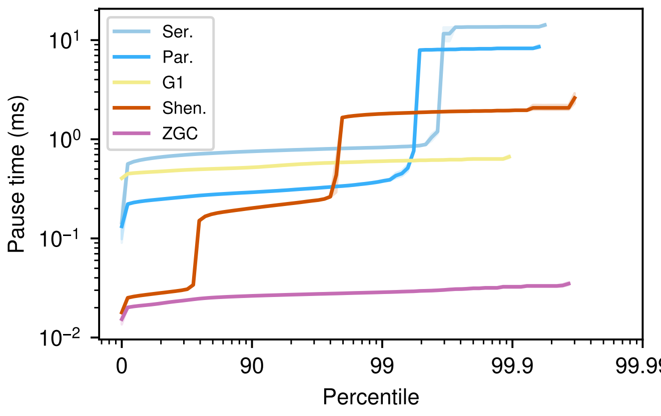


Fig. 3: GC pause time for lusearch in a 3.0× heap. Each line and its shade show the mean and 95 % CI over 20 invocations.

Compared with Table VI and Table VII, we can see that the classic methodology of estimating the GC overhead from the time/cycles spent in GC pauses is highly problematic, especially for the concurrent collectors. In particular, the two concurrent copying collectors spend a negligible fraction of time/cycles in GC pauses, but have enormous LBOs.

With fixed hardware resources, concurrent GC threads compete with mutator threads, e.g., for cache capacity and memory bandwidth; the effects are more severe effects for more parallel workloads. In other words, concurrent GC overheads are high not only from the expensive mechanisms they use, such as read and write barriers, but also from resource contention.

c) Low pause ≠ low latency: A commonly used metric to tune GC for latency-sensitive applications is the maximum GC pause time (such as the `-XX:MaxGCPauseMillis` suggested in the GC tuning guide from the vendor [23]). Here, we show that the pause time is a poor metric to assess GCs for latency-sensitive applications. Figure 3 and Fig. 4 show the distribution of pause times and query latencies (using metered latency; see Section IV-A) of different GCs running the lusearch benchmark at 3.0× heap. The low-pause collectors (Shenandoah and ZGC) indeed achieve better pause times in general, with ZGC consistently having the lowest pause time for all percentiles, while Shenandoah has lower pause times than the other three GCs under the 90th percentile. However, low pause times do not automatically confer low application latencies. Indeed, both Shenandoah and ZGC have worse (by factors of 10–100×) query latencies than the other three collectors.

Application latencies are affected by GC pauses—both by their durations and frequency. Short but frequent pauses will not impact the distribution of pause times, but can certainly impact application latency.

Importantly, application latencies are also functions of mutator performance. As discussed in previous sections, concurrent copying GC can affect mutator performance through expensive mechanisms, such as read barriers, and through the resource contention. In the case of the pathological modes we discussed, Shenandoah and ZGC throttle mutator threads when the GC cannot keep up with allocation. Such throttling indeed avoids

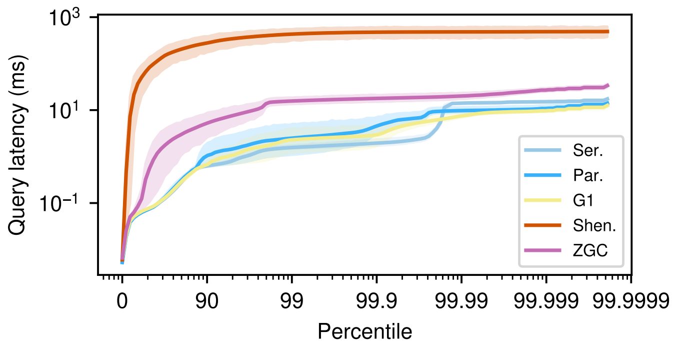


Fig. 4: Metered latency for lusearch in a 3.0× heap. Each line and its shade show the mean and 95 % CI over 20 invocations.

triggering a STW collection, and keeping each GC pause short. However, it comes at great cost: if the mutators are sufficiently throttled, both the application latency and throughput will be worse than a simple STW GC, as evidenced in the above graphs and Table VIII.

E. Recommendations

Drawing on the observations, we make two recommendations for improving GC evaluation in future research. First, the LBO methodology should be used to report the absolute overheads of GCs. This helps us better understand the scale of the impact of GC on the program execution. Second, a richer set of performance and cost metrics should be used when evaluating GCs. At a minimum, both the wall-clock time and the CPU cycles used should be reported. Any additional metric (such as application latency, or an energy consumption model like Intel RAPL) can help us understand the performance characteristics of different GCs better.

V. Conclusion

In this paper, we identify two important problems in empirical evaluation of GCs: unclear absolute overheads, and easy-to-misinterpret results presented using limited metrics. To address these problems, we first devise the lower-bound overhead (LBO) methodology to empirically estimate the absolute overheads of GCs for any given cost metric. Then, we use the LBO methodology to study five production collectors in OpenJDK 17.

We find that these GCs incur substantial overheads: at least (lower bound) 7–82 % time overhead and 6–92 % cycle overhead relative to a zero-cost baseline for a modest heap size. We identify three important types of misinterpretation: opportunity cost, concurrency overhead, and using GC pause times as a proxy metric for application latency. These types of misinterpretation can be mitigated by including more metrics in the evaluation.

Our findings reveal substantial overheads in production GCs, highlighting opportunities for future research on low-latency collectors with low overheads. We recommend using more metrics when evaluating GCs, because this helps reveal tradeoffs and limitations of GCs that often go unnoticed. The LBO methodology is language and runtime agnostic, and thus can benefit GC research on a wide range of managed languages and platforms, such as .NET (for C# and other CLI languages) and V8 (for JavaScript).

References

[1] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, D. T. Marr and D. H. Albonesi, Eds. ACM, 2015, pp. 158–169. [Online]. Available: https://doi.org/10.1145/2749469.2750392

[2] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, R. E. Johnson and R. P. Gabriel, Eds. ACM, 2005, pp. 313–326. [Online]. Available: https://doi.org/10.1145/1094811.1094836

[3] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "Wake up and smell the coffee: evaluation methodology for the 21st century," *Commun. ACM*, vol. 51, no. 8, pp. 83–89, 2008. [Online]. Available: https://doi.org/10.1145/1378704.1378723

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley, "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*, E. G. C. Jr., Z. Liu, and A. Merchant, Eds. ACM, 2004, pp. 25–36. [Online]. Available: https://doi.org/10.1145/1005686.1005693

[5] D. Detlefs, C. H. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, D. F. Bacon and A. Diwan, Eds. ACM, 2004, pp. 37–48. [Online]. Available: https://doi.org/10.1145/1029873.1029879

[6] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin, "Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK," in *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*, W. Binder and P. Tuma, Eds. ACM, 2016, pp. 13:1–13:9. [Online]. Available: https://doi.org/10.1145/2972206.2972210

[7] P. Lidén and S. Karlsson, "JEP 333: ZGC: A scalable low-latency garbage collector (experimental)," http://openjdk.java.net/jeps/333, Feb 2018, accessed: 2021-12-07. [Online]. Available: http://web.archive.org/web/20211207112317/http://openjdk.java.net/jeps/333

[8] ——, "The Z garbage collector," http://cr.openjdk.java.net/~pliden/slides/ZGC-FOSDEM-2018.pdf, 2018, accessed: 2021-12-07. [Online]. Available: http://web.archive.org/web/20211207112406/http://cr.openjdk.java.net/~pliden/slides/ZGC-FOSDEM-2018.pdf

[9] S. M. Blackburn and K. S. McKinley, "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*, R. Gupta and S. P. Amarasinghe, Eds. ACM, 2008, pp. 22–32. [Online]. Available: https://doi.org/10.1145/1375581.1375586

[10] S. M. Blackburn, P. Cheng, and K. S. McKinley, "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*, A. Finkelstein, J. Estublier, and D. S. Rosenblum, Eds. IEEE Computer Society, 2004, pp. 137–146. [Online]. Available: https://doi.org/10.1109/ICSE.2004.1317436

[11] B. G. Zorn, "Barrier methods for garbage collection," University of Colorado Boulder, Tech. Rep., 11 1990. [Online]. Available: https://scholar.colorado.edu/concern/reports/47429970d

[12] A. L. Hosking, J. E. B. Moss, and D. Stefanovic, "A comparative performance evaluation of write barrier implementations," in *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92), Seventh Annual Conference, Vancouver, British Columbia, Canada, October 18-22, 1992, Proceedings*, J. R. Pugh, Ed. ACM, 1992, pp. 92–109. [Online]. Available: https://doi.org/10.1145/141936.141946

[13] S. M. Blackburn and A. L. Hosking, "Barriers: friend or foe?" in *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, D. F. Bacon and A. Diwan, Eds. ACM, 2004, pp. 143–151. [Online]. Available: https://doi.org/10.1145/1029873.1029891

[14] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking, "Barriers reconsidered, friendlier still!" in *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, M. T. Vechev and K. S. McKinley, Eds. ACM, 2012, pp. 37–48. [Online]. Available: https://doi.org/10.1145/2258996.2259004

[15] G. E. Collins, "A method for overlapping and erasure of lists," *Commun. ACM*, vol. 3, no. 12, pp. 655–657, 1960. [Online]. Available: https://doi.org/10.1145/367487.367501

[16] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng, "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*, J. M. Vlissides and D. C. Schmidt, Eds. ACM, 2004, pp. 69–80. [Online]. Available: https://doi.org/10.1145/1028976.1028983

[17] B. G. Zorn, "The measured cost of conservative garbage collection," *Softw. Pract. Exp.*, vol. 23, no. 7, pp. 733–756, 1993. [Online]. Available: https://doi.org/10.1002/spe.4380230704

[18] Oracle, "JVM^{TM} tool interface," https://docs.oracle.com/en/java/javase/17/docs/specs/jvmti.html, Jun 2021, accessed: 2021-12-09. [Online]. Available: https://web.archive.org/web/20211209121820/https://docs.oracle.com/en/java/javase/17/docs/specs/jvmti.html

[19] M. Stephens, "GCRealTimeMon," https://github.com/Maoni0/realmon, Nov 2021, accessed: 2021-12-09. [Online]. Available: https://web.archive.org/web/20211209122302/https://github.com/Maoni0/realmon

[20] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, P. L. Tarr and W. R. Cook, Eds. ACM, 2006, pp. 169–190. [Online]. Available: https://doi.org/10.1145/1167473.1167488

[21] P. Lidén, "ZGC | what's new in JDK 17," https://malloc.se/blog/zgc-jdk17, Oct 2021, accessed: 2021-12-07. [Online]. Available: http://web.archive.org/web/20211115113228/https://malloc.se/blog/zgc-jdk17

[22] R. Kennke, "Shenandoah in OpenJDK 17: Sub-millisecond GC pauses," https://developers.redhat.com/articles/2021/09/16/shenandoah-openjdk-17-sub-millisecond-gc-pauses, Sep 2021, accessed: 2021-12-07. [Online]. Available: http://web.archive.org/web/20211207114248/https://developers.redhat.com/articles/2021/09/16/shenandoah-openjdk-17-sub-millisecond-gc-pauses

[23] Oracle, "Java platform, standard edition HotSpot virtual machine garbage collection tuning guide, release 17," https://docs.oracle.com/en/java/javase/17/gctuning/hotspot-virtual-machine-garbage-collection-tuning-guide.pdf, Sept 2021, accessed: 2021-12-13. [Online]. Available: http://web.archive.org/web/20211213041034/https://docs.oracle.com/en/java/javase/17/gctuning/hotspot-virtual-machine-garbage-collection-tuning-guide.pdf