Distilling the Real Cost of Production Garbage Collectors

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

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

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 cost. Without understanding their cost, crucial tradeoffs made by garbage collectors (GCs) go unnoticed. This can lead to misguided design constraints and evaluation criteria used by GC researchers and users, hindering the development of high-performance, low-cost GCs.**

In this paper, we develop a methodology that allows us to empirically estimate the cost of GC for any given set of metrics. This fundamental quantification has eluded the research community, even when using modern, well-established methodologies. By distilling out the explicitly identifiable GC cost, we estimate the intrinsic application execution cost using different GCs. The minimum distilled cost forms a baseline. Subtracting this baseline from the total execution costs, we can then place an empirical lower bound on the absolute costs of different GCs. Using this methodology, we study five production GCs in OpenJDK 17, a high-performance Java runtime. We measure the cost of these collectors, and expose their respective key performance tradeoffs.

We find that with a modestly sized heap, production GCs incur substantial overheads across a diverse suite of modern benchmarks, spending *at least* 7–82 % more wall-clock time and 6–92 % more CPU cycles relative to the baseline cost. We show that these costs can be masked by concurrency and generous provisioning of memory/compute. In addition, we find that newer low-pause GCs are significantly more expensive than older GCs, and, surprisingly, sometimes 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 distillation methodology together with 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.

***Index Terms*—garbage collection, OpenJDK**

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.

Because of the ubiquity of GC, the research community has extensively studied GC performance. The approaches include characterizing specific elements of GC behavior, performing comparative evaluation among garbage collectors (GCs), and deconstructing the performance of specific GCs. These aspects are addressed by a substantial literature, including [1]–[10]. They are explicitly non-goals of our work.

While this rich literature helps us understand how GCs compare, how they are designed, and what key sources of cost are, there is a surprising lack of clarity regarding the *real* costs that GC brings to a programming language. In this paper, we focus on two key problems: (a) lack of clarity about the absolute cost of GC, and (b) misinterpretations of GC evaluations due to limited cost metrics. We now offer more detail regarding these two problems and outline our contributions.

a) Unclear absolute costs: The absolute cost that garbage collectors impose on modern production runtimes is an important quantification, but it has eluded the community to date. Its importance is twofold. For programming language implementers and hardware architects, understanding the absolute cost of GC and its magnitude relative to the rest of the language runtime can help them decide where to spend research and engineering resources. For language users, knowing the absolute cost of GC can help them decide whether to use a managed language or to use alternatives such as C/C++ and Rust—a decision that often cannot be easily reversed.

Hertz and Berger [11] attempted to answer this question, but their work is limited by the fidelity of the simulation infrastructure, and by requiring invasive changes to the runtime. Due to this complexity, their method cannot be readily applied to modern workloads and production runtimes, and their particular analysis is now somewhat dated by advances in language implementation and computer architecture, and substantial changes to workloads.

In this paper, we develop a language- and runtime-agnostic methodology to empirically place a lower bound on the absolute cost of GC for any cost metric. The intuition behind our methodology is simple. If we knew the intrinsic cost of running an application without any of the costs of a garbage collector, then we could use that as a baseline for understanding the absolute cost imposed by real garbage collectors. However, in

We thank the anonymous reviewers and Ben Titzer for their detailed feedback and insightful suggestions for improving the paper. This material is based upon work supported by the Australian Research Council under Grant No. DP190103367 and National Science Foundation under Grant No. CSR-2106117. Zixian Cai is supported by an Australian Government Research Training Program Scholarship.

Z. Cai, S. M. Blackburn, M. D. Bond and M. Maas, "Distilling the Real Cost of Production Garbage Collectors," 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2022, pp. 46-57, doi: 10.1109/ISPASS55109.2022.00005.

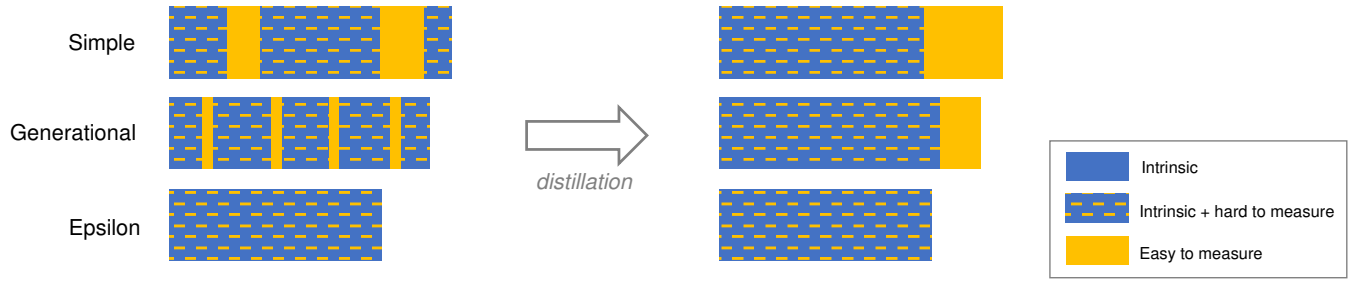

Fig. 1: Application execution is comprised of the intrinsic application cost shown in blue and GC costs shown in orange. Some of the GC costs, shown with dashed lines, tightly couple with the execution and are hard to measure. The distillation process subtracts the easy-to-measure GC costs from the total cost, and what remains approximates the intrinsic application cost. The minimum distilled cost can be used to estimate the absolute cost of each GC.

practice, some of the GC costs permeate the execution process, and are hard to tease out (Fig. 1). The key insight is that we can approximate this baseline by running an application with real collectors, and distilling out explicitly identifiable GC costs from the total execution costs. The *minimum distilled cost* overestimates the baseline, and can then be used to derive an empirical lower bound on the absolute cost of each GC.

We analyze all five native GCs in OpenJDK 17, the latest release of an industrial-strength, high-performance JVM. Our methodology is simple yet effective: even an underestimate of the GCs' absolute costs reveals that they impose substantial costs to program execution across a diverse set of workloads. We used a wide range of cost metrics including CPU cycles, cache misses, and RAPL energy. Due to space constraints, we focus on two important metrics, the wall-clock time and CPU cycles, in our case study. Using a modest heap size, our methodology estimates that by using GC, applications spend 7–82 % more wall-clock time and 6–92 % more CPU cycles relative to their intrinsic costs.

b) Misinterpretation of evaluation results: The second problem we address is that GC evaluations are prone to misinterpretation, even when modern, well-established methodologies (*e.g.*, varying the heap size) are used. Such misinterpretation is dangerous: a costly GC technique understood as cheap can mislead the community to overuse it and dismiss cheaper alternatives, while discouraging future optimizations. Unless a cost is properly measured, it will not be properly addressed.

This danger is acutely clear in the new low-pause collectors. These algorithms are popularly understood [12], [13] to make applications more responsive at low cost to throughput. Our case study (Section IV-D) shows this to be unfounded. These algorithms impose substantial costs on application execution and do not achieve good application responsiveness. Although they deliver low *pause times*, the application overheads are so great that responsiveness is often *no better than simple stop-the-world GCs*.

In this paper, we analyze three important types of misinterpretation of GC evaluations arising from overly focusing on limited metrics. We show how key performance tradeoffs made by different GCs can go unnoticed. We recommend mitigating this problem by using more cost and performance metrics, including wall-clock time, CPU cycles, and application latency.

To summarize, our novel methodology allows the community to grasp the substantial costs incurred by widely used production GCs in real applications. We recommend empirically estimating the absolute cost of GC, and using richer sets of metrics when evaluating GCs to mitigate common misinterpretations. These steps are important for revealing the tradeoffs and limitations of different collectors, and for assisting language users in choosing the most appropriate GC for a plethora of existing and emerging workloads. Our findings invite future research on developing low-latency, low-cost collectors.

The code we used in this work is publicly available: https://github.com/caizixian/distillation [14].

II. Background and Related Work

This section discusses a few attempts to measure the absolute cost of GC, and points out their limitations. We then highlight two categories of GC costs that are hard to measure: costs tightly coupled with application execution, and indirect costs. We describe the established methodologies to measure such costs, but they are insufficient to adequately characterize the absolute GC cost on their own. Finally, we give an overview of the production garbage collectors that we study in this paper, and how they present such hard-to-measure sources of GC cost.

A. Absolute Costs of Garbage Collection

Hertz and Berger [11] attempted to quantify the cost of garbage collection over an explicit memory management regime. Their work shares the same basic intuition as ours (comparing against a hypothetical "zero-cost" baseline), but is an entirely different approach.

By tracing the program execution, they construct a database of object liveness information. This database allows them to create an "oracular GC" that inserts `malloc` and `free` calls into the program execution trace, which is processed by an architectural simulator (SimpleScalar [15]). To do so, they need to make invasive changes to the runtime as well as integrate into SimpleScalar specific information about the behavior of the runtime's collector and allocator.

Although it is an interesting study, it is limited by the fidelity of the simulation infrastructure (a problem whenever using architectural simulation) and by the requirement of being able to make such changes to the runtime and the appropriate modeling within the simulator. The particular analysis has other limitations including that it is now dated by changes in architecture, workloads and algorithms.

In contrast, our methodology is very simple. It measures any production collector running on native hardware, making minimal assumptions about the GC implementation, and without access to source code, entirely avoiding the issues in prior work [11].

Some work (e.g., [8]) measures the cost of conservative GC in the context of explicit memory management. In this paper, we focus on precise GC in managed languages.

Numerous studies (e.g., [2]) measure GC pauses to quantify GC costs. We show that this approach is problematic because some GC activities (such as barriers) are carried out by mutators on behalf of the GC, and are hard to separate. Moreover, for concurrent GCs, while the GC pauses are shorter, the majority of the work is shifted to run concurrently with the mutator threads.

B. Costs Tightly Coupled with Application Execution

A GC can be considered in terms of three principal activities: *allocation*, *identification*, and *reclamation* [16]. Allocation is performed by the mutator using space provided by the GC. Identification establishes which part of the heap is live and which may be reclaimed. Reclamation makes space occupied by unreachable objects available for reuse.

While some GC activities are performed directly by the collector, others are performed by the mutator; for example:

1) Mutators can allocate on their respective local allocation buffers, and only require periodic synchronization with the GC.
2) Mutators can assist identification and reclamation via *barriers*—code snippets that mediate mutators' heap operations on behalf of the GC [6].
3) Mutators can reclaim unreachable objects. For example, in the case of naïve reference counting (RC) [17], the code that decrements the reference count of an object to zero can immediately reclaim the memory.

For performance reasons, the above mechanisms, such as allocation sequences and barriers, are tightly coupled with mutator execution, often implemented using a "fast-and-slow-path" paradigm [18]. Fast paths are short but frequently executed code snippets, often inlined into the mutator code by the just-in-time (JIT) compiler. Because they are so tightly integrated into the mutator, measuring their costs requires carefully crafted methodologies that avoid instrumentation overheads that will affect the measurement:

1) Blackburn *et al.* [2] 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.
2) The cost of barriers has also been extensively studied [3]–[6]. These methodologies 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.
3) Blackburn *et al.* [2] measured the cost of a deferred reference counting collector in terms of the mutator time.

These specialized methodologies require deep understanding of the GC implementation and modifications to the collector and/or runtime, and consequently are not trivially applied to arbitrary language runtimes. Furthermore, they are insufficient to adequately characterize the absolute GC cost on their own. In this paper, we focus on devising a methodology that encompasses all of these costs instead of measuring individual components.

C. Indirect Costs and Benefits

Apart from GC's direct costs, it can also bring indirect costs and even benefits. A common source of indirect cost (or benefit) is reduced (increased) 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 improve mutator locality through rearranging objects. A compacting or evacuating GC can improve spatial locality by moving objects that are frequently accessed together to be spatially closer to each other [7].

Some of the locality impact might be observable through hardware performance counters, such as cache miss events (*e.g.*, LLC and TLB misses). 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 in the process. It may be possible to achieve such an effect under simulation, but it would likely require significant compute resources to evaluate and would suffer from the same limitations as prior work [11].

In this paper, we are not concerned with teasing out the indirect impact of GC. Our distillation methodology measures the absolute cost of GC, including its indirect cost.

D. Garbage Collection Algorithms in OpenJDK

In our case study, we measure the cost of production GCs in OpenJDK 17, and reveal common misinterpretations in their evaluations. We highlight their key properties in Table I. Note that analyzing the performance of a particular collector and proposing improvements to its design are explicitly not our objectives. There exists a substantial literature addressing those, including a recent paper by Zhao and Blackburn that gives a detailed performance analysis of a number of collectors, including ones we study in this paper [10].

The first of these collectors, Epsilon, does not actually collect garbage. Just like any other collector, its absolute cost—which

TABLE I: Garbage collectors that this paper studies. SATB = snapshot at the beginning [19]. LVB = loaded value barrier [20].

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 [21]	2009	Yes	Yes	Yes (tracing)	Write (card-marking and SATB)
Shenandoah [12]	2019	No[1]	Yes	Yes (tracing and copying)	Write (SATB) and read (LVB[2])
ZGC [13], [22]	2018	No[3]	Yes	Yes (tracing and copying)	Read (LVB)

we find to be nonzero—can be measured by our methodology. The other five collectors can be divided into three groups:

1) Stop-the-world collectors (Serial and Parallel): the collector requires all mutators to be stopped while it is running, *i.e.*, it does not exhibit any concurrency with respect to the mutator.
2) Concurrent tracing collector (G1): the collector performs garbage identification concurrently via a trace. This trace does not modify the heap, but marks reachable objects as live. The correctness of the concurrent trace is typically protected by write barriers.
3) Concurrent copying collectors (Shenandoah and ZGC): in addition to concurrent tracing, the collector performs reclamation concurrently 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.

All collectors (except Epsilon) we studied use read and/or write barriers to allow a part of the heap (a generation or a region) to be independently collected. 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.

These collectors all include hard-to-measure sources of GC cost discussed previously in this section, hindering the quantification of their absolute costs. For example, the allocation sequence and the barriers are tightly integrated with the application by inline compilation. Moreover, when collector threads run concurrently with mutators, it is hard to attribute overheads due to resource contention to the originating threads. These collectors (including Epsilon) therefore motivate our methodology, with which we are able to measure and place a lower bound on their implicit and explicit costs.

III. Distilling the Absolute Cost of Garbage Collection

Due to the prevalence of GC, its cost is a hot—and sometimes contentious—topic. The extensive existing literature focuses on *comparative evaluation*, such as shown in Table II, which helps us understand how GCs compare. In this section, we address a distinctly different goal: measuring the absolute cost of GCs. First, we describe our distillation methodology together with an example. Then, we discuss the applicability, advantages, and limitations of the methodology.

A. Definition with Examples

Section I sketched the intuitions underpinning our distillation methodology (Fig. 1). To deepen the understanding, we now use concrete evaluation results from three production GCs—Serial, Parallel, and Shenandoah—running a real-world application, the H2 database. This also serves as a running example throughout this section.

Note that in the examples, we use one particular cost metric—total CPU cycles used by all JVM threads—and one particular way of excluding explicitly identifiable GC cost—costs incurred during stop-the-word (STW) pauses. We emphasize that these choices are orthogonal to the mechanism of our distillation methodology. Our methodology can operate on other cost metrics and other ways to isolate GC costs.

It is easy to measure the total costs of real collectors and see how they compare. Table II shows that when using Serial, the total cycle usage is the smallest, with Parallel and Shenandoah being 0.2 % and 102.3 % more expensive, respectively. This *comparative analysis* shows which GC is the best to use if you are concerned about total CPU cycle consumption, a well-understood methodology in the literature. However, it does not measure the absolute CPU cycle cost of each *collector alone*.

TABLE II: The total CPU cycles consumed when running the DaCapo benchmark h2 with a 3× heap (see Section IV-A) using three different collectors. Lower is 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

Definition 3.1 (Intrinsic application cost): We define the intrinsic application cost as the *theoretical ideal* cost of running an application. This intrinsic cost includes the best GC benefit (such as improved locality) any configuration could bring (*i.e.*, a collector with tuning parameters) but none of the GC costs (such as barriers).

For example, if we knew the intrinsic cost of the application H2 (inclusive of the best GC benefit but exclusive of the GC costs) in terms of CPU cycles, we could subtract that from the

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

total cycles used by each real collector, giving us the absolute cost of the respective collector.

Definition 3.2 (Distilled application cost): Of course, we do not know the intrinsic application cost of a given workload. Our key insight is that we can approximate the intrinsic application cost by excluding the costs that we can easily ascribe to GC from its total costs. This insight can perhaps be better understood using the distillation analogy. The total cost is a mixture of the intrinsic application cost (after considering the best GC benefit) and the absolute GC cost. Some of the GC costs can be explicitly identified (such as the cost incurred during STW pauses), while others are hard to measure (such as barriers). We cannot remove all hard-to-measure GC costs from the mixture, but we can easily distill out explicitly identifiable GC costs. This gives us an overestimate of the intrinsic application cost.

$$\text{Distilled cost} \equiv \text{Total cost} - \text{Explicit GC cost} \geq \text{Intrinsic cost}$$

As shown in Table III, we can easily distill out explicitly identifiable GC cost by excluding the cost during STW pauses, where no mutator activities happen and costs are strictly from GC. Repeating the distillation process for each of three collectors, we obtain three distilled application costs.

TABLE III: Distilling out cycles used during stop-the-world (STW) pauses from the total cycles in Table II. The minimum distilled application cost (MDC) in cycles is shown in green. The MDC value is used to calculate the empirical lower bound on the cost of each of the collectors in Table IV.

Collector	Total	STW	Distilled
	billion cycles		
Parallel	108.33	4.46	$108.33 - 4.46 = 103.87$
Serial	108.12	2.75	$108.12 - 2.75 = 105.37$
Shenandoah	218.72	0.03	$218.72 - 0.03 = 218.69$

Definition 3.3 (Minimum distilled application cost): The *minimum distilled cost* (MDC) is the minimum of the distilled application costs from running the application with each of the collectors. Since we define the intrinsic application cost to include the best benefit any GC configuration could bring, the set of collectors used to derive the MDC can include the same collector with *different* tuning parameters (*e.g.*, heap size, the number of collector threads). The MDC is the best overestimate of the intrinsic cost.

$$\left(\text{MDC} \equiv \min_{g \in \text{GCs}} \text{Distilled cost}_g\right) \geq \text{Intrinsic cost}$$

In Table III, the MDC is 103.87 billion cycles (Parallel, shown in green).

Definition 3.4 (Lower Bound Overhead of Garbage Collectors): The MDC in turn allows us to place an empirical lower bound on the absolute cost of the collectors, which we call the LBO. Since Absolute GC cost = Total cost − Intrinsic cost and MDC ≥ Intrinsic cost, for each GC g, we have

$$(\text{LBO}_g \equiv \text{Total cost}_g - \text{MDC}) \leq \text{Absolute GC cost}_g$$

As shown in Table IV, we subtract the MDC from the total cycles of each of the GCs, yielding a lower bound on the absolute cycle cost of each GC.

Definition 3.5 (Normalized LBO): We can normalize the LBO to the MDC, which we use in the rest of the paper:

$$\text{Normalized LBO (NLBO)} \equiv (\text{LBO} + \text{MDC})/\text{MDC}$$

In the above example, we can see that in the context of the H2 database running on OpenJDK, we spend at least 4.1 %, 4.3 %, and 110.6 % more CPU cycles than the intrinsic costs for Serial, Parallel, and Shenandoah respectively. We note that even though Shenandoah is not useful when calculating the MDC, the absolute cost of Shenandoah is still accurately captured.

TABLE IV: The empirical lower bound (LBO) on the cycle cost for each collector can be obtained by subtracting the MDC from the respective total cycles in Table III. The lowest LBO in green.

Collector	LBO	Normalized LBO
	Total − MDC	(LBO + MDC)/MDC
Parallel	$108.33 - 103.87 = 4.46$	1.043
Serial	$108.12 - 103.87 = 4.25$	1.041
Shenandoah	$218.72 - 103.87 = 114.85$	2.106

We emphasize that the distilled cost of any collector can be calculated, and it is *not* the same as the cost of a no-GC scheme (such as Epsilon in OpenJDK). In the above example, all three collectors perform collection, and we estimate their respective LBOs without using a no-GC scheme.

B. Discussion

The biggest advantage of our distillation methodology is its simplicity. The only requirement of our methodology is to exclude explicitly identifiable GC cost.

This simplicity makes our methodology generally applicable on different runtimes and GC algorithms. In the above example, we exclude the costs incurred during STW pauses, where the costs are strictly from GC. This is conceptually easy, and can be simply implemented as callbacks to delineate STW phases. In fact, for OpenJDK, this instrumentation can be implemented using JVMTI callbacks [23], and for .NET, implemented on top of GCRealTimeMon [24].

This simplicity also allows us to evaluate GCs running on native hardware. As a result, any hardware performance counter can trivially be used as a cost metric when using our methodology. This includes RAPL energy readings, cache misses, and other metrics.

However, language implementers need to be careful when applying our methodology to GCs in different runtimes. For example, a compiler that performs escape analysis can lower the GC costs despite no change to the GCs. Such analysis allows objects to be allocated on the stack rather than on the heap when the objects do not escape. This reduces overall allocation, and therefore reduces the pressure on the GC.

We also note that to get a useful MDC, our methodology requires *at least* one GC where the distilled cost is close

to the intrinsic cost. For example, if all evaluated collectors were concurrent *and* if explicitly identifiable GC cost were from STW pauses, the distilled costs from *all* collectors would include costs from the concurrently running GC threads. In this scenario, the MDC would be much higher than the intrinsic cost, and would lead to a very poor empirical lower bound on GC cost.

For hardware performance counters, we can address the above deficiency by measuring the costs on a per-thread basis. Instead of excluding STW costs, one can exclude costs originating from GC threads.[4] This measurement mode is easily supported by, *e.g.*, Linux's perf event subsystem.

In our case study, we do not use this engineering optimization. Because we have two STW collectors (Serial/Parallel) in the set we studied, taking the whole-process reading at STW points is sufficient to factor out the explicitly identifiable GC cost to establish an MDC that closely approximates the intrinsic application cost.

IV. Case Study: Collectors in OpenJDK 17

Recall that we address two key problems in this paper: 1) lack of clarity about the costs imposed by GC, and 2) misinterpretation of GC evaluations due to limited cost metrics. In the previous section, we proposed the distillation methodology to estimate the cost of GCs for any given set of metrics. In this section, we perform a case study of GCs in OpenJDK 17 (Section II-D), the latest release of a high-performance production JVM.

First, we apply our distillation methodology on these GCs, and measure their costs using the lower bound overhead (LBOs) defined in Section III. We focus on two cost metrics—the wall-clock time (*time LBOs*) and the total CPU cycles (*cycle LBOs*). We reveal that with a modest 2.4× heap and across workloads, production collectors can incur substantial absolute costs, evident in both high time LBO and high 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.

Second, we concretely show how simplistic evaluations of these GCs can be misinterpreted. We highlight three important types of misinterpretation: 1) not considering opportunity cost, 2) not considering overheads due to concurrency, and 3) measuring pause time instead of application latency. We then demonstrate how to mitigate the misinterpretation by using a richer set of cost and performance metrics.

Based on our observations, we recommend that evaluations of GCs should 1) use the distillation methodology to report the costs of GCs, and 2) use more performance and cost metrics in order to evaluate GCs holistically.

A. Methodology

a) Benchmarks and latency measures: We use a snapshot release of the forthcoming Chopin release of the DaCapo benchmark suite [25]. We exclude the benchmarks cassandra, h2o, and kafka, because they use deprecated Java features that are not compatible with JDK 17.

DaCapo's Chopin snapshot release includes a number of latency-sensitive benchmarks. In these 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. For these latency-sensitive benchmarks, DaCapo reports two measures of latency, *simple* and *metered*, 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.

We also investigate the SPECjbb2015 benchmark, which is often used in the literature when evaluating low-pause GCs. However, unlike the DaCapo suite, SPECjbb2015 fixes the workload to a constant amount of time rather than to constant work, preventing us from easily measuring the costs of different GCs executing the same amount of work. Therefore, SPECjbb2015 is not included in our analysis.

b) Cost metrics: We implement our distillation methodology as a Java Virtual Machine Tool Interface (JVMTI) [23] agent. We implement the distilled cost in the methodology by excluding costs incurred in stop-the-world (STW) pauses. The STW points are provided by the JVMTI callbacks for the starts and the ends of GC pauses.

In addition to the wall-clock time, our JVMTI agent captures a wide range of hardware performance counters, including CPU cycles, instruction counts, cache misses, and Intel RAPL energy measurements. To read these counters, we use 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 the real cost of each GC when handling 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 [9]. Except for Epsilon, which does not perform GC, we set the heap size relative to the minimum heap size (Table V) 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.

[4]Excluding costs of GC threads is not the same as excluding all GC costs. For example, the cost of barriers will still be in the distilled cost.

TABLE V: Minimum heap size required to run each benchmark.

Benchmark	Heap size (MB)
avrora	7
batik	189
biojava	95
eclipse	411
fop	15
graphchi	255
h2	773
jme	29
jython	25
luindex	42
lusearch	21
pmd	156
sunflow	29
tomcat	21
tradebeans	131
tradesoap	103
xalan	8
zxing	97

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: While microarchitectural sensitivity of a GC algorithm is important, it is orthogonal to this work. We use two identical machines with 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 `Temurin-17.0.1+12` release of OpenJDK, which is Eclipse Temurin's (formerly AdoptOpenJDK's) distribution of OpenJDK. 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 [26], [27]) 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: Costs of Garbage Collection

First, we estimate the costs of each *configuration* (a collector at a heap size) using our distillation methodology. The distilled cost for each configuration is the total cost excluding the STW cost. Note that Epsilon does not actually collect garbage, and therefore, its distilled cost is the same as the total cost.

TABLE VI: Time LBOs 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: Cycle LBOs 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

Though Epsilon is not a practical GC, its distilled cost could potentially help us obtain an MDC that better estimates the intrinsic cost of a workload. However, in practice, Epsilon has high absolute costs because the spatial locality of objects is poor, and the allocation sequence will always need to request more physical memory from the operating system. This is evident in Tables VIII and IX, showing the LBOs of Epsilon beside those of standard collectors, for cases in which Epsilon is able to run a benchmark without exhausting the machine's available memory. From the tables, we can see that Epsilon only affects the MDC of one benchmark—sunflow—indicated by the 1.000 time LBO and the 1.000 cycle LBO. For the rest of the section, we focus on the absolute costs of collectors other than Epsilon.

Table VI shows the time LBOs of the collectors we study while Table VII shows the cycle LBOs. Each table covers eight different heap sizes, ranging from a small 1.4× heap to a generous 6.0× heap (see Section IV-A for the multiplier notation). The LBO value for each configuration is the geometric mean over 16 DaCapo benchmarks.[5]

The production GCs incur substantial costs. For a modest 2.4× heap, production GCs on average spend at least 7–82 % more wall-clock time and 6–92 % more cycles relative to the intrinsic costs. Even for a generous 6.0× heap, the costs are as much as 23 % in terms of time and 34 % in terms of cycles.

[5]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 VIII: Time LBOs at 3× heap (except Epsilon) using distillation. 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, with its 95 % CI shown below in gray. Parallel outperforms other collectors for most benchmarks.

Bench.	Serial	Para.	G1	Shen.	ZGC	Eps.
avrora	1.009	1.025	1.031	1.101	1.072	1.010
	±0.7%	±0.5%	±0.6%	±0.8%	±0.7%	±0.7%
batik	1.146	1.092	1.067	1.089	1.077	1.087
	±0.2%	±0.4%	±1.0%	±0.4%	±0.3%	±0.2%
biojava	1.006	1.007	1.033	1.221	1.882	1.157
	±0.1%	±0.1%	±0.2%	±0.2%	±1.0%	±0.1%
eclipse	1.050	1.010	1.062	1.127	1.136	1.180
	±0.4%	±0.1%	±0.2%	±0.2%	±0.1%	±0.3%
fop	1.159	1.129	1.201	1.414	2.107	1.175
	±0.2%	±1.6%	±1.2%	±0.8%	±0.6%	±0.3%
graphchi	1.021	1.011	1.020	1.191	1.125	1.054
	±0.3%	±0.3%	±0.4%	±1.5%	±0.4%	±0.7%
h2	1.121	1.044	1.093	1.570	1.964	1.483
	±0.3%	±0.3%	±0.6%	±0.3%	±0.6%	±0.2%
jme	1.003	1.003	1.002	1.010	1.006	1.006
	±0.0%	±0.0%	±0.0%	±0.0%	±0.1%	±0.0%
jython	1.037	1.022	1.065	1.609	2.204	1.231
	±1.7%	±0.2%	±0.2%	±0.9%	±0.5%	±0.3%
luindex	1.007	1.011	1.044	1.150	1.097	1.055
	±0.2%	±0.3%	±0.4%	±0.7%	±0.3%	±0.8%
lusearch	1.230	1.105	1.135	3.766	2.701	
	±0.2%	±0.3%	±0.4%	±1.6%	±0.4%	
pmd	1.613	1.124	1.188	1.559	1.777	1.165
	±0.2%	±0.2%	±0.3%	±0.6%	±8.7%	±0.4%
sunflow	1.412	1.156	1.156	2.180	2.109	1.000
	±2.0%	±2.0%	±1.2%	±2.2%	±0.6%	±0.4%
tomcat	1.157	1.137	1.144	1.305	1.914	1.145
	±0.2%	±0.2%	±0.1%	±0.2%	±0.6%	±0.1%
tradebeans	1.120	1.020	1.164	1.423	1.327	1.415
	±0.6%	±0.6%	±0.7%	±0.8%	±0.5%	±0.5%
tradesoap	1.094	1.011	1.112	1.340	1.282	1.376
	±0.4%	±0.2%	±0.3%	±0.2%	±0.2%	±0.2%
xalan	3.380	3.074	3.496	30.176		1.111
	±0.3%	±0.4%	±0.3%	±1.7%		±0.6%
zxing	1.030	1.058	1.028	1.274	1.235	1.062
	±1.1%	±1.8%	±1.3%	±1.1%	±1.1%	±1.3%
min	*1.003*	*1.003*	*1.002*	*1.010*	*1.006*	
max	*1.613*	*1.156*	*1.201*	*3.766*	*2.701*	
mean	*1.130*	*1.057*	*1.091*	*1.490*	*1.589*	
geomean	*1.121*	*1.055*	*1.089*	*1.407*	*1.513*	

For all heap sizes shown, Serial achieves the lowest costs in terms of cycles, while Parallel achieves the lowest costs in terms of time for all but the smallest heap size.

Note that even though Serial is single-threaded, the rest of the VM is still multithreaded, hence the difference in between the time and cycle LBOs for Serial. Unsurprisingly, this difference is more pronounced on benchmarks with more mutator parallelism.

Tables VIII and 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 a 3.0× heap, in terms of cycles, STW collectors are never more expensive than concurrent collectors.[6] In terms of time, STW collectors are only more expensive than concurrent collectors for 2 out of 17 benchmarks: batik and zxing.[7]

b) Single-threaded vs. multi-threaded STW GC: Between two stop-the-world collectors, Parallel is more expensive than Serial in terms of cycles (as much as 48 % for a small 1.4× heap), and the reverse is true in terms of time. This effect occurs presumably because a multi-threaded collector exploits the available parallelism to shorten pauses but introduces synchronization overhead among collector threads. At 3.0× heap, in terms of cycles, Parallel is only cheaper than Serial for 2 out of 17 benchmarks: lusearch, tradebeans, and tradesoap.[8] In terms of time, Serial is only cheaper than Parallel for avrora.[9]

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

d) Pathological Modes of Concurrent Copying Collectors: We observe two pathological modes of concurrent copying collectors when challenged with workloads that have high allocation rates but low survival rates. In such scenarios, the concurrent copying collectors often fall back to STW collections and/or stall the mutators to keep up with the allocation, resulting in poor performance.

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. Despite the high time LBO, 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

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

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

[8]The differences for jython and sunflow are not statistically significant.

[9]The differences for biojava, luindex, and sunflow are not statistically significant.

TABLE IX: Cycle LBOs at 3× heap (except Epsilon) using distillation. 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, with its 95 % CI shown below in gray. Serial outperforms other collectors for most benchmarks.

Bench.	Serial	Para.	G1	Shen.	ZGC	Eps.
avrora	1.007	1.014	1.047	1.201	1.209	1.017
	±0.6 %	±0.7 %	±0.7 %	±0.6 %	±0.7 %	±0.4 %
batik	1.146	1.991	1.565	1.454	1.915	1.087
	±0.2 %	±2.1 %	±0.8 %	±0.5 %	±3.3 %	±0.2 %
biojava	1.006	1.014	1.046	1.521	3.962	1.157
	±0.1 %	±0.1 %	±0.3 %	±0.7 %	±1.0 %	±0.1 %
eclipse	1.054	1.109	1.317	1.390	1.474	1.196
	±0.4 %	±0.1 %	±1.2 %	±0.3 %	±0.5 %	±0.1 %
fop	1.160	1.203	1.460	1.893	2.227	1.175
	±0.3 %	±1.5 %	±1.9 %	±1.1 %	±1.1 %	±0.3 %
graphchi	1.008	1.031	1.048	1.226	1.226	1.099
	±0.3 %	±0.3 %	±0.3 %	±2.2 %	±0.7 %	±0.6 %
h2	1.053	1.055	1.123	2.131	2.645	1.365
	±0.1 %	±0.2 %	±0.4 %	±0.3 %	±0.7 %	±0.2 %
jme	1.071	1.132	1.091	1.517	1.470	1.133
	±0.3 %	±0.4 %	±0.3 %	±0.5 %	±1.0 %	±0.3 %
jython	1.036	1.034	1.072	2.038	2.444	1.225
	±1.7 %	±0.2 %	±0.2 %	±0.7 %	±0.4 %	±0.3 %
luindex	1.010	1.018	1.067	1.207	1.199	1.065
	±0.2 %	±0.2 %	±1.3 %	±0.8 %	±0.5 %	±0.9 %
lusearch	1.045	1.030	1.081	1.213	1.268	
	±0.3 %	±0.2 %	±0.2 %	±0.3 %	±0.3 %	
pmd	1.096	1.178	1.286	1.563	1.655	1.266
	±0.5 %	±0.4 %	±0.6 %	±0.5 %	±10.1 %	±0.2 %
sunflow	1.125	1.087	1.088	1.240	1.076	1.000
	±2.4 %	±2.1 %	±1.2 %	±0.8 %	±0.8 %	±0.4 %
tomcat	1.010	1.015	1.022	1.119	1.119	1.035
	±0.1 %	±0.1 %	±0.1 %	±0.1 %	±0.1 %	±0.1 %
tradebeans	1.060	1.049	1.211	1.637	2.171	1.293
	±0.4 %	±0.3 %	±0.7 %	±0.3 %	±0.5 %	±0.6 %
tradesoap	1.052	1.029	1.117	1.710	2.155	1.251
	±0.4 %	±0.4 %	±0.8 %	±0.4 %	±0.4 %	±0.2 %
xalan	1.211	1.633	1.783	1.744		1.139
	±0.4 %	±0.3 %	±0.3 %	±0.4 %		±0.3 %
zxing	1.017	1.034	1.027	1.275	1.238	1.061
	±1.1 %	±1.8 %	±1.4 %	±1.2 %	±1.1 %	±1.4 %
min	*1.006*	*1.014*	*1.022*	*1.119*	*1.076*	
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*	

impact is amplified by the high allocation rates and low survival rates of benchmarks such as xalan and lusearch;[10] because allocation would fail if reclamation did not keep up with the allocation rate, the collector resorts to STW collections. In this case, it would be more beneficial to use a STW collector in the first place and thus avoid the concurrency overhead (as we 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 cost but modest cycle cost.

[10]Both benchmarks allocate multiple GBs per second, and their minimum heap sizes are merely 8 MB and 21 MB, respectively.

D. Misinterpretation of Evaluation Results

Prior work [1], [2] 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 that the fundamental time–space tradeoff is still applicable in a modern, diverse benchmark suite (Tables VI and VII). 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.

In this section, we present the misinterpretation by comparing the absolute costs of different GCs shown earlier. We would like to emphasize that this misinterpretation is also applicable in classic comparative analysis, and using more metrics still improves the evaluations in that context.

a) Opportunity cost: Comparing Table VI and Table VII, we notice that 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 × MDC 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 costs can go unnoticed when only the wall-clock time is reported.

Historically, GC performance has been mostly measured using wall-clock time only. This methodology implicitly assumes that on machines with free cores available, the free 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 mean 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, using fewer cycles to run the application can free up CPU cores and 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` flag suggested in the GC tuning

TABLE X: Percent of wall-clock time spent in STW pauses averaged over 16 benchmarks. Lower is 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 total cycles spent in STW pauses averaged over 16 benchmarks. Lower is 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

guide from the vendor [28]). Tables X and XI show the fraction of time and cycles spent in STW pauses for different GCs. Similar to Tables VI and VII, the results are grouped by heap sizes, and show the geometric means over 16 benchmarks. Compared with Tables VI and VII, we can see that the classic methodology of estimating the GC costs 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, resulting in more severe effects for more parallel workloads. In other words, concurrent GC costs 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 [28]). It is well known that the pause time is a poor metric to assess GCs for latency-sensitive applications [29]. We reaffirm this here, and highlight the importance of using more metrics. The following example shows that fixating on limited metrics (*e.g.*, pause times) can even make a language implementer unintentionally work against the motivating goal (the responsiveness of the application).

Figures 2 and 3 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.

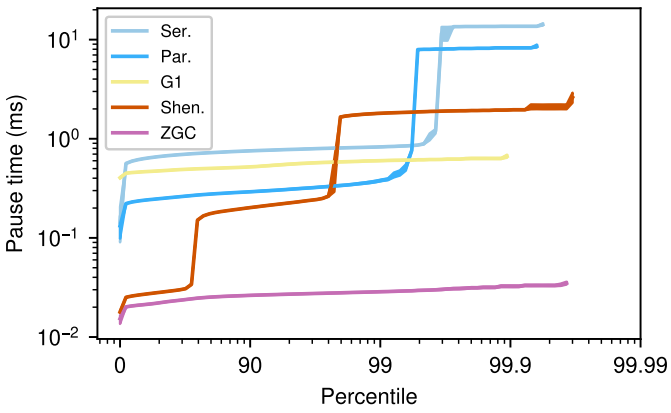

Fig. 2: GC pause time for lusearch in a 3.0× heap. Each line and its shade show the mean and 95 % CI over 20 invocations.

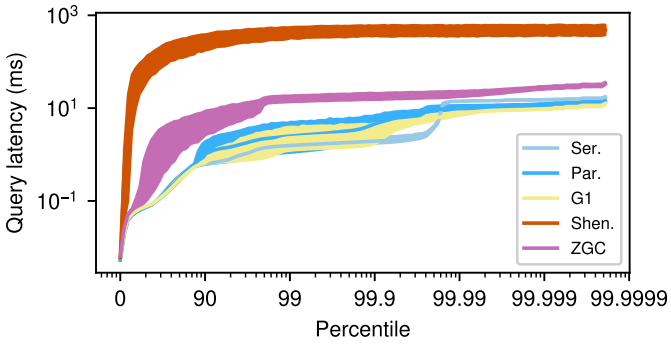

Fig. 3: Metered latency for lusearch in a 3.0× heap. Each line and its shade show the mean and 95 % CI over 20 invocations.

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 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 triggering a STW collection, 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 these observations, we make two recommendations for improving GC evaluation in future research. First, the distillation methodology should be used to report the costs 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 can help us understand the performance characteristics of different GCs better. This includes measuring application latency for applications with latency requirements, instead of using pause times as a surrogate.

V. Conclusion

In this paper, we identify two important problems in empirical evaluation of GCs: unclear costs, and easy-to-misinterpret results presented using limited metrics. To address these problems, we first devise the distillation methodology to place an empirical lower bound on the costs of GCs for any given cost metric. Then, we use the distillation methodology to measure the costs of five production collectors in OpenJDK 17. We find that these GCs incur substantial costs: (as an underestimate) at least 7–82 % more time and 6–92 % more cycle overhead relative to the intrinsic application cost for a modest 2.4× heap.

Our results also show how a lack of diverse cost/performance metrics can lead to misinterpretation of GC evaluations, hindering GC development. We identify three important types of misinterpretation: neglecting opportunity cost, neglecting 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 costs in production GCs, highlighting opportunities for future research on low-latency collectors with low costs. We recommend using more metrics when evaluating GCs, because this helps reveal tradeoffs and limitations of GCs that often go unnoticed. The distillation methodology is language and runtime agnostic, and thus can benefit GC research on a wide range of managed languages and platforms, such as Go, .NET (for C# and other CLI languages) and V8 (for JavaScript).

References

[1] 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

[2] 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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] W. Zhao and S. M. Blackburn, "Deconstructing the garbage-first collector," in *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, S. Nagarakatte, A. Baumann, and B. Kasikci, Eds. ACM, 2020, pp. 15–29. [Online]. Available: https://doi.org/10.1145/3381052.3381320

[11] 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

[12] 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

[13] P. Lidén and S. Karlsson, "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

[14] Z. Cai, "ISPASS 2022 artifact: Distilling the real cost of production garbage collectors," Apr. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6476821

[15] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997. [Online]. Available: https://doi.org/10.1145/268806.268810

[16] 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

[17] 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

[18] 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

[19] T. Yuasa, "Real-time garbage collection on general-purpose machines," *J. Syst. Softw.*, vol. 11, no. 3, pp. 181–198, 1990. [Online]. Available: https://doi.org/10.1016/0164-1212(90)90084-Y

[20] G. Tene, B. Iyengar, and M. Wolf, "C4: the continuously concurrent compacting collector," in *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, H. Boehm and D. F. Bacon, Eds. ACM, 2011, pp. 79–88. [Online]. Available: https://doi.org/10.1145/1993478.1993491

[21] 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

[22] 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

[23] 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

[24] 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

[25] DaCapo Group, "DaCapo benchmarks evaluation snapshot 29a657f," Oct. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.6475255

[26] 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

[27] 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

[28] 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

[29] P. Cheng and G. E. Blelloch, "A parallel, real-time garbage collector," in *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, M. Burke and M. L. Soffa, Eds. ACM, 2001, pp. 125–136. [Online]. Available: https://doi.org/10.1145/378795.378823