Rethinking Java Performance Analysis

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

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

Rui Chen
chenrui.crc@bytedance.com
Independent
Beijing, China

Xi Yang
xi@iop.systems
IOP Systems
Sydney, Australia

John Zhang
john.z@canva.com
Canva
Sydney, Australia

John Zigman
john.zigman@sydney.edu.au
The University of Sydney
Sydney, Australia

Abstract

Representative workloads and principled methodologies are the foundation of performance analysis, which in turn provides the empirical grounding for much of the innovation in systems research. However, benchmarks are hard to maintain, methodologies are hard to develop, and our field moves fast. The tension between our fast-moving fields and their need to maintain their methodological foundations is a serious challenge. This paper explores that challenge through the lens of Java performance analysis. Lessons we draw extend to other languages and other fields of computer science.

In this paper we: i) introduce a complete overhaul of the DaCapo benchmark suite [6], characterizing 22 new and/or refreshed workloads across 47 dimensions, using principal components analysis to demonstrate their diversity, ii) demonstrate new methodologies and how they are integrated into an easy to use framework, iii) use this framework to conduct an analysis of the state of the art in production Java performance, and iv) motivate the need to invest in renewed methodologies and workloads, using as an example a review of contemporary production garbage collector performance.

We highlight the danger of allowing methodologies to lag innovation and respond with a suite and new methodologies that nudge forward some of our field's methodological foundations. We offer guidance on maintaining the empirical rigor we need to encourage profitable research directions and quickly identify unprofitable ones.

CCS Concepts: • **Software and its engineering → Software performance**.

Keywords: Performance Analysis; Java; Garbage Collection

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707217

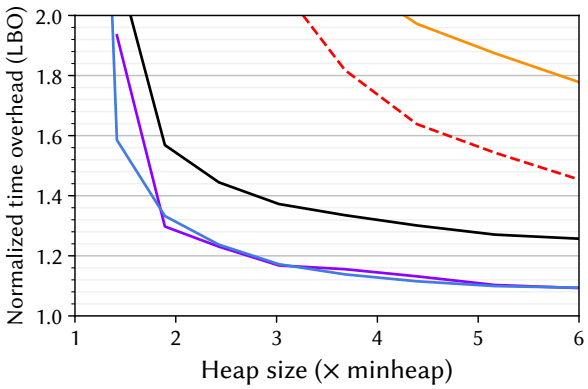
ACM Reference Format:
Stephen M. Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3669940.3707217

1 Introduction

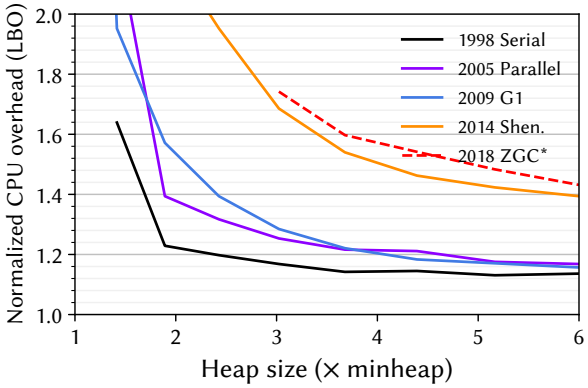
Building and maintaining benchmarks is a Sisyphean task, yet our field depends critically on them. Methodological innovation seems to be uncommon, yet our field is fast moving, so demands it. These two contradictions pose an enduring risk that we misdirect our field, abandoning promising ideas while pursuing ideas that we should have abandoned [5].

DaCapo is a broadly-focussed benchmark suite for Java heavily used in diverse domains within academia and industry [7, 19, 25, 40]. We motivate this work with a case study using DaCapo Chopin to explore overheads of contemporary production garbage collectors. We find that garbage collectors are consuming 15 % of CPU cycles even in the most favorable situations and that newer garbage collectors incur even higher overheads — as high as 17× in small heaps and 63 % in generous ones. What is interesting and relevant to our paper is not these overheads, but that they've largely gone unnoticed. Given the scale at which Java is deployed, the likely impact is substantial. We suggest that this lack of awareness is an example of collective methodological inattention.

One contribution of this paper is that we highlight the importance of the systems community continuously improving and evolving our methodologies. The heart of contribution, though, is DaCapo Chopin, a major release of the DaCapo benchmark suite for Java [6] that took fourteen years to develop, with eight entirely new workloads, all other workloads fully refreshed, a new methodology for measuring and reporting latency, nine latency-sensitive workloads, novel integrated workload characterization, and applications targeting the phone and the server, with minimum heap sizes from



(a) Lower bound wall clock time overheads.



(b) Lower bound total CPU overheads (Linux TASK_CLOCK).

Figure 1. Lower bounds on the overheads of five OpenJDK 21 production garbage collectors with their default settings, as a function of heap size, showing the geometric mean of overhead over all 22 DaCapo Chopin benchmarks. We only plot data points where the respective collector can run all 22 benchmarks to completion. In the best case, wall clock overheads are 9 % (G1 and Parallel) and total CPU overheads are 15 % (Serial). At smaller heaps, overheads exceed 2×.

5 MB to 20 GB. We offer methodological guidance, including a description of DaCapo Chopin's new latency metrics, and offer insights that we glean from evaluating DaCapo Chopin using OpenJDK 21.

We hope that the methodological guidance we offer will be used, that it might fuel a spirit of methodological critique and development, and that it might also inspire others to develop diverse, open, standardized workloads and methodologies for Java, for other languages, and in other areas.

2 Motivation

Figure 1 shows the overhead of five OpenJDK 21 garbage collectors as a function of heap size, taking the geometric mean over all 22 DaCapo Chopin benchmarks, using the lower bound overhead (LBO)[1] methodology [10].[2] The graphs show the time-space tradeoff inherent to garbage collection: CPU resources consumed by garbage collection rise as the available memory shrinks. Figure 1(a) shows the wall clock time overhead while Figure 1(b) uses Linux perf TASK_CLOCK, which sums the running time of all threads in the process, indicating the total computational overhead. All of the collectors except ZGC use compressed pointers by default. Because ZGC does not support compressed pointers, care should be taken when comparing it with the other collectors.

First, note that the CPU overhead of garbage collection is 15 % *in the best case*. Second, note that there is a regression when we consider collector designs in terms of when they were introduced into the JVM: Serial (1998), Parallel (2005), G1 (2009) [14], Shenandoah (2014) [16, 17, 38], and ZGC (2018) [27, 28, 46]. Comparison with the previous analysis by Cai et al. [10] indicates that these results are robust across JVM versions and benchmark versions. If we accept for a moment that our analysis is sound and that these figures accurately reflect the state of the art, this should give researchers pause.

What's going on? The relevant context is the rise of parallelism and latency as foremost concerns in application domains from mobile to the data center; parallelism because of the ubiquity of parallel hardware, latency because of the increasing use of garbage collection in latency-sensitive settings. The evolution in collector designs reflects this. Serial uses a single collection thread, while Parallel uses all of the available hardware parallelism, so runs faster than Serial. However, parallelism is never perfectly efficient, so Parallel tends to have larger total overhead when considering the task clock (Figure 1(b)). G1 performs work concurrently with the application and works in smaller regions at a time, so offers better latency than Parallel, and sometimes incurs a performance overhead in doing so. Shenandoah and ZGC go a step further and perform almost all collector work concurrently with the application, promising to offer better latency still, but incurring additional overhead in doing so. The problem depicted in Figure 1 was raised in 2022 by Cai et al. [10].

How did this happen? We don't improve what we don't measure. Specifically: i) **Failure to expose garbage collection's time-space tradeoff.** Despite the importance of systematically exploring this most basic tradeoff having been laid down more than twenty years ago [8, 9, 24], work routinely ignores this advice, neither systematically identifying minimum heap sizes [8], nor varying the available memory [9, 24] when evaluating collectors.[3] We respond to this in Section 4.2. ii) **Failure to appropriately evaluate latency.** Two decades ago, Cheng and Blelloch [12] clearly

[1] Pronounced *elbow*.
[2] Details of the methodology are in Section 6.1 and Section 6.2.
[3] We're pointing to the broader research community (the authors included) and its collective culture, not to individual researchers.

explained why garbage collection pause times should never be used as a proxy for user-experienced latency, but GC pauses continue to be (mis)used this way. We respond to this in Section 4.4. iii) **Failure to expose total computational overheads.** Despite the ubiquity of hardware parallelism and the importance of multi-tenanted platforms such as mobile devices, browsers, and data centers, evaluations rarely measure the total computational cost of systems, focusing instead on wall clock time. The situation is worse still with garbage collection, where costs can be hard to attribute [10]. We respond to this in Section 4.5. iv) **Failure to evaluate using diverse, appropriate workloads.** The cost of creating and maintaining benchmarks means that often there are not good, representative, workloads available. DaCapo Chopin is our response.

Objections to our motivating analysis might include:

Q: Shouldn't the latency advantages of the various collectors be presented here too?
A: We investigate latency in Section 4.4 and Section 6.3.
Q: Weren't some of these collectors designed (only) to be used with generous heaps?
A: Our analysis extends to 6× the minimum required memory. Given the cost of memory across mobile, desktop and data centers, a 6× memory overhead is generous. Consider Figure 6(d), showing h2 at 4 GB.
Q: What about an application domain that is not memory or compute-constrained?
A: In this situation, it may be best not to garbage collect at all [31, 44]. Otherwise, the time and space overheads introduced by a system remain important.

Although our motivating example is concretely based on garbage collection, it need not be. Garbage collection is an example of a methodologically challenging subject. Our point here is not to paint a negative picture of production garbage collectors, but to cast a light on overheads that have gone largely unnoticed yet affect production systems. We don't attribute this to the designers and engineers, but to our broader research community and our methodological inattention.

Our hope is that the new workloads, new methodologies, and new tooling we provide will nudge the field forward again, making it easier for researchers to better measure, analyze, and understand the likely impact of their work.

3 Background and Related Work

3.1 Modern Virtual Machines and OpenJDK 21

Modern language virtual machines are large and complex. According to estimates published by Synopsys Open Hub [41], the OpenJDK runtime includes 12 M lines of code, took 3193 person years to develop, and cost approximately $215 M to build. The complexity of these runtimes and their wide use compounds the methodological challenges of measuring them well while raising the stakes of not doing so.

We focus our attention on OpenJDK 21 and the compilers and collectors that ship with it. OpenJDK 21 was released in September 2023, a recent stable release of the most widely used runtime for Java [33], grounding our analysis in a state of the art production environment. It is not our goal to create benchmarks or methodologies for other languages, although some of our findings apply broadly to garbage-collected languages, and others to performance analysis more broadly. Nor is it our goal to evaluate other Java runtimes or garbage collectors, although the work we present should be directly applicable to them.

OpenJDK 21 was built over more than two decades, with a sophisticated multi-tier compiler [34] and a suite of highly-tuned production garbage collectors [14, 17, 23, 28]. The system is constantly under development, with contributions from the research community and industry. This analysis is not a commentary on developers of OpenJDK 21 but a critique of the broader community from which it emerged, and particularly our research community, which includes the authors.

3.2 Benchmarks and Benchmark Suites

The history of using benchmarks to evaluate systems performance dates at least to the early 1970's, when Curnow and Wichmann [13] discuss the use of a *"clearly defined task"* to compare the speed of various CPUs in their paper describing the Whetstone FORTRAN benchmark. They note that

> *unless such a program is carefully constructed it is unlikely to be typical of the many thousands of programs run at an installation.*

This observation cuts to heart of the problems outlined in Section 2. Ensuring benchmarks are representative makes them expensive to create and maintain, yet an absence of representative benchmarks is typically the justification researchers give for their use of ad hoc workloads and all the methodological problems that follow.

Benchmarks such as Whetstone and more recent examples such as gcbench [15], tests for Java Concurrency JSR166 [26], and the computer language benchmark game [2] are examples of micro benchmarks. These are easy to use, easy to measure, but far from realistic. They are nonetheless valuable tools. Simple, deterministic workloads can be particularly helpful in identifying and attributing specific performance regressions with high fidelity.

The DaCapo benchmark suite [6] was developed nineteen years ago to provide a realistic setting for JVM development and performance analysis. It was the result of a broad collaboration among industrial and academic researchers. The first-order design goals were diverse real-world applications, and ease of use. These led to the following criteria: i) open source workloads, ii) maximizing coverage of application

domains and behaviors, iii) easy to measure self-contained workloads, iv) exclusion of GUI workloads, and v) provision of a range of inputs. The authors also outlined a series of methodological recommendations, with a particular focus on JIT compilation and garbage collection, which differentiated Java performance analysis from that of C and FORTRAN which had dominated the decades prior to DaCapo's release [7].

The Renaissance suite [36] was developed to fill an absence of Java workloads that adequately exercised Java's newer parallel programming abstractions and concurrency primitives. The authors built a rich suite of programs following similar principles to DaCapo and used the suite to evaluate the Graal compiler [45] against the HotSpot C2 compiler [34], evaluating four new compiler optimizations and a number of other existing optimizations. In addition to Renaissance, there are other broadly-focussed suites developed with similar principles to DaCapo [37], but none target Java. There are many other suites with more narrow objectives, such as JaConTeBe which targets concurrency bugs [29].

SPECjvm and SPECjbb are produced by SPEC, a non-profit corporation guided by a desire to provide industry-standard benchmarks with which products can be fairly compared. SPECjvm was last released in 2008 and is not widely used by researchers. SPECjbb2015 [39] is a synthetic benchmark that executes requests over a simple model of a business, reporting both throughput and latency statistics. The business model can be scaled, generating large workloads. The benchmark measures *jOPS* (operations), and reports a *critical-jOPS* metric which is the geometric mean of the number of jOPS across different service-level agreements (SLAs) where the 99th percentile latency meets the respective SLA.

We present the new DaCapo Chopin benchmark suite. It differs from the prior work in significant ways: i) it is comprised entirely of real world workloads, ii) it includes workloads with application domains all the way from mobile to server, iii) it introduces a novel integrated latency measure and nine latency-sensitive workloads which report rich latency statistics, iv) its workloads have minimum heap sizes ranging from 5 MB to 20 GB, v) it includes 47 per-benchmark statistics characterizing and ranking the workloads, including nominal minimal heap sizes and various performance metrics, to help researchers understand workload behavior and to facilitate sound methodology, and vi) it introduces eight completely new workloads and updates all others. All benchmarks run on OpenJDK 21. We rely on community input to gain assurance of the representativeness of the suite. The composition of DaCapo Chopin was heavily guided by feedback from industry, with more than half of the workloads proposed by and/or co-developed with industrial users.

3.3 Empirical Evaluation

There is a large body of work on empirical evaluation [4, 5, 7, 20, 22, 43]. The SIGPLAN Empirical Evaluation Checklist [4] provides seven checklist items and 22 counter-examples designed to guide researchers to conduct sound evaluations. It presents both principles (seven checklist items) and concreteness (via counterexamples). The principles include that a paper's claims must be explicit and supported by their evaluation, and that there must be an appropriate and clear experimental design. A group of SIGPLAN researchers developed a pragmatic guide to assessing empirical evaluations, which includes extensive guidance and references [5]. Their opening sentence resonates with our goals:

> *An unsound claim can misdirect a field, encouraging the pursuit of unworthy ideas and the abandonment of promising ideas.*

The original DaCapo release came with methodological recommendations, including how to control for JIT compiler warmup and how to evaluate the time-space tradeoff of garbage collectors [6, 7]. We build on that work in Section 4.

There are numerous papers on how to improve fidelity in empirical evaluations of managed languages. Huang et al. [21] developed a methodology for *replaying* dynamic optimization plans to reduce measurement noise. Georges et al. [18] proposed refinements to this approach and made recommendations on how many measures should be taken in any given experiment. Cai et al. [10] describe the lower bound overhead (LBO) methodology evaluating garbage collector overheads. We use LBO throughout this paper and discuss it further in Section 4.5. Mytkowicz et al. [32] highlight alarming pitfalls for those conducting empirical evaluations. Papadakis et al. [35] conducted a broad study of the memory sensitivity of a range of Java benchmarks. We include some similar metrics here, but our goal is a broader characterization of workloads and we focus on the new OpenJDK 21 and DaCapo Chopin. Carpen-Amarie et al. [11] use cache coloring to artificially restrict the size of the last level cache in order to study the sensitivity of concurrent garbage collectors to last level cache size. We apply the same technique in our workload characterization (Section 5.1). Others such as Barrett et al. [3] have focussed on how to understand minute changes in performance, techniques which can be invaluable when attempting to identify and attribute small performance regressions. Our focus is in evaluating large production runtimes such as OpenJDK 21 in the face of large multi-threaded workloads with complex time-varying inputs that often exhibit complex behaviors.

4 Methodology

We made the case at the start of this paper that upholding sound methodological principles is important to the health of the field, and we summarized some of the wealth of resources available to researchers [3–5, 7, 10, 11, 18, 20, 22, 32, 35, 43].

Here we focus on: i) aspects of existing empirical evaluation advice that we think need renewed attention, and ii) new methodological features that DaCapo Chopin makes available. We believe that together, the following offers a strong response to each of the four major points of methodological failure that we outlined in Section 2.

4.1 Methodological Principles

While concrete recommendations are often most helpful, their concreteness means that their utility has a limited lifetime. We therefore start with some principles what provide a higher level framework from within which researchers can view methodology for empirical evaluation.

Researchers should **follow established methodology** right up to the very point where their evaluation moves beyond what the state of the art can support. Then they must **identify new methodologies** that correctly measure the subject of their evaluation. The SIGPLAN empirical evaluation checklist captures this by noting that the checklist *is meant to* support *informed judgement, not* supplant *it* [4]. Doing this well is hard. It requires the **integrity** to earnestly want to reveal the empirical truth, in the full knowledge that the results may not support an idea that has been years in development. It requires a high degree of **rigor**, because attention to detail is essential when evaluating complex systems in complex environments. Finally, it requires **discernment**, since one must be able to discern when existing methodologies are adequate (and should be rigorously adhered to), and when new methodologies must be developed.

4.2 The Time–Space Tradeoff

Figure 1 clearly illustrates the time–space tradeoff underpinning garbage collection, something all evaluations of garbage collected languages should either explore or control for. Although this has been widely understood for two decades or more, it remains commonplace for this aspect of managed language performance evaluation to be ignored.

Recommendation H1. Garbage collectors should be evaluated across a range of heap sizes to demonstrate the sensitivity of the collector to the time–space tradeoff [7].

We use a generous 6× upper bound in Figure 1 but a smaller upper bound might be more reasonable in many circumstances. Because the time-space tradeoff is not linear (see Figure 1), larger heap sizes yield smaller and smaller amounts of information. We therefore suggest selecting heap sizes in a distribution that gives more resolution to small heap sizes. Because the heap size requirements of different benchmarks vary greatly,[4] it is essential that the heap sizes used in an evaluation are chosen on a benchmark-by-benchmark basis, rather than applying the same heap sizes to all benchmarks.

Recommendation H2. Heap sizes should be expressed in terms of multiples of the minimum heap size in which a baseline collector can run that workload [7, 8].

To assist with this, DaCapo Chopin includes nominal statistics[5] which capture minimum heap sizes for its small, default, large and vlarge benchmark configurations with compressed pointers, as well as for the default configuration when compressed pointers are disabled. These all use the baseline OpenJDK 21 configuration we describe in Section 6.1.

Note that methodologies like this which control the memory available to the garbage collector (e.g. via `-Xmx`) *do not* necessarily provide a clear measure of how efficiently a collector reclaims space. This is because the minimum heap size in which a workload can run reflects the workload's *peak* memory usage, not its average usage. A metric which reflected the 'area under the memory use curve' might better reflect the net memory footprint of a workload.

4.3 Compilers, Warmup and Performance Analysis

Aside from the challenges that garbage collection brings to measuring a runtime, the just-in-time compiler and the broader experimental environment can create confounding effects. Mytkowicz et al. explain how small details such as the length of strings in environment variables can confound findings, advice researchers should heed [32].

While some researchers (implicitly) take the position that more iterations and more invocations are better, we make two contrary points: i) resources are finite and there is an opportunity cost associated with running more executions—it is quite possible that an experiment that tests more features is empirically stronger than one that yields very high precision on fewer dimensions, and ii) contrived experimental environments run a risk of irrelevance by compromising realism.

Recommendation P1. Researchers should be cautious of naïvely following methodological prescriptions. Instead they should be guided by: i) the coherence of their experimental design with respect to the claims they plan to make (which, for example, may determine whether to time the first iteration capturing JIT overheads, class loading, etc., or one that is well warmed up), and ii) the statistical significance of their findings (ensuring that there are sufficient data points such that a statistically sound conclusion can be drawn).

[4] For example, the minimum heap sizes range from 5 MB (avrora) to 681 MB (h2) in the default benchmark sizes settings, and up to 20 GB in the vlarge setting (h2).

[5] We use the term 'nominal' in the sense of *'being, or relating to a designated or theoretical size that may vary from the actual'* [30]. We use the word to emphasize that these measures are only intended to provide broad characterizations of the workloads in some default context and should not be viewed as a concrete, definitive measures defining the workload.

Figure 2. Cheng and Blelloch [12] used a figure like this to illustrate the problem with using GC pauses as a measure of responsiveness and proposed the minimum mutator utilization (MMU) metric as a response. GC pause time continues to be widely (mis)used as a proxy for responsiveness more than twenty years later.

The DaCapo Chopin suite comes with detailed measures of warmup time for each workload. In practice we found that the fifth iteration (-n 5) for default workload sizes and the first iteration for large and vlarge sizes exhibit well-warmed up behavior for our baseline configuration of OpenJDK 21.

4.4 User-Experienced Latency

Latency-sensitive applications are an increasingly important consideration for developers of managed languages, since garbage collectors and JIT compilers are capable of generating latency that is visible to application users. This applies to mobile devices, where refresh rates and smooth scrolling affect the user experience, to the desktop where browser responsiveness is important, and in request-based services that run on servers. DaCapo Chopin includes nine latency-sensitive workloads. These include jme, which is based on the jMonkey Engine, a popular video game engine, spring, which is a microservices workload built on the Spring web framework, and seven other request-based services.

A naïve approach to measuring latency is to simply measure the length of pauses created by the runtime that lock out the application (*stop the world* pauses). However, as Cheng and Blelloch [12] pointed out, this is a poor measure since several short pauses may have a similar or worse effect than a long pause (Figure 2). They proposed the notion of *minimum mutator utilization* (MMU) metric to reflect how much CPU was available to the mutator over a sliding window of time, for various window sizes. Despite this clear insight and guidance two decades ago, it remains common for GC pauses to be used as a proxy for user-experienced latency. Even so, MMU is not ideal since it is a single-threaded measure, cannot capture throughput reductions due to expensive barriers embedded within the mutator, and requires instrumenting the garbage collector. Zhao et al. [47] show that reliance on simple GC pause times has led to designs with surprisingly poor latency responsiveness even in modern collectors that specifically target latency. DaCapo Chopin addresses the problem by *directly reporting user-experienced latency*.

Simple Latency. DaCapo times every event: frame renders for jme and client requests for the other eight latency-sensitive workloads. As the workload progresses, DaCapo stores event start and end times in an array. Careful engineering ensures that the cost of recording these measurements is low. Once the workload completes, DaCapo determines the distribution of latencies, reporting the distribution in terms of percentiles, from median to 99.99, as well as optionally saving the complete data to file for offline analysis. We call this metric *Simple Latency*.

Request Queuing and Metered Latency. Most real-world request-based services implement a queuing system. Requests enter the queuing system at some externally determined rate, dictated by factors such as when customers make purchases or when people launch queries. These systems typically adjust server capacity as demand changes. One of the DaCapo design principles is that each benchmark will run in a single JVM on a single machine. Thus, attempting to implement a realistic distributed load balancing system was out of scope for DaCapo Chopin. Sacrificing some realism for determinism, the DaCapo request-based workloads are driven by a pre-determined set of requests, with each worker consuming consecutive requests until all have been completed. Within each thread, the start time of each request is thus dictated by the completion of the request before.

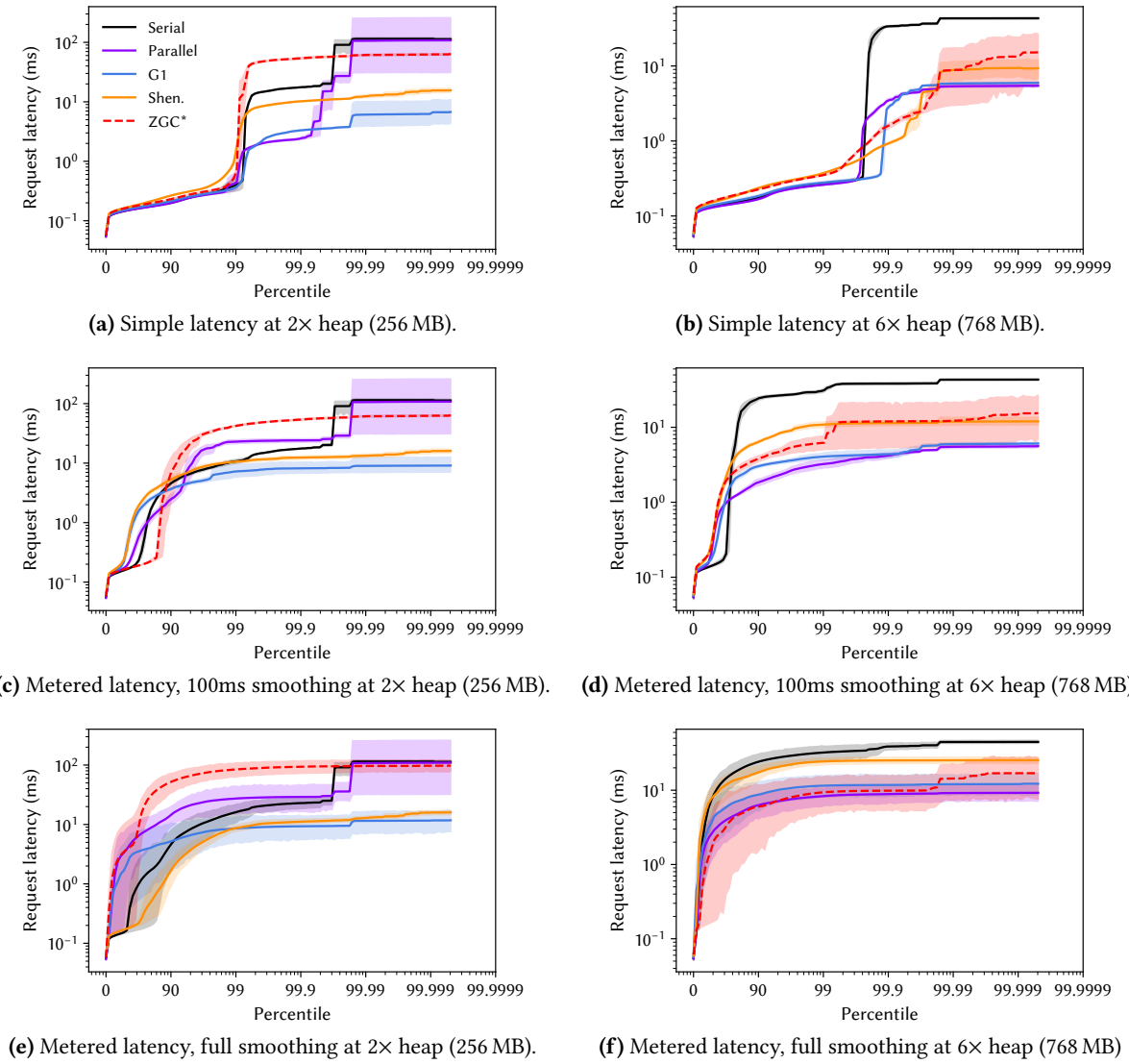
In a real system, request/event start times are *externally defined*, so a delay will affect not only all running events, but all subsequent events that are forced to wait in the queue due to the backlog of work. Without a queue, DaCapo's workloads cannot directly model the cascading effect of delays.

Instead, we model a similar effect with what we call *Metered Latency*. At the completion of the workload, we assign each event an assumed start time based on all events having been hypothetically received at *uniform* intervals throughout the execution of the benchmark. We then determine the metered latency for each event as the time between its end time and the earlier of its actual and assumed start times. Thus, when the application is paused, the effect of the pause is not felt just by those events that were running when the pause occurred, but also by those whose end time was delayed.

We implement the uniform synthetic start times by applying a smoothing function to the actual start times, using a sliding average. A window size of one affords no smoothing, so is identical to simple latency, reflecting no queueing effect. On the other hand, an arbitrarily large window gives all events uniformly distributed synthetic start times. DaCapo reports metered latency using window sizes from 1 ms up to the length of the benchmark execution, in powers of ten. We suggest that a smoothing window of 100 ms is a reasonable middle ground, allowing for variation in request completion rate over the whole execution (e.g. due to compiler or file cache warm up), while exposing effects of disruptions due garbage collection etc.

Recommendation L1. Researchers should report user-experienced latency, not weak proxies such as GC pauses.

Recommendation L2. Researchers should report distribution statistics and/or plot CDFs as illustrated in Figure 3,



(a) Simple latency at 2× heap (256 MB).

(b) Simple latency at 6× heap (768 MB).

(c) Metered latency, 100ms smoothing at 2× heap (256 MB).

(d) Metered latency, 100ms smoothing at 6× heap (768 MB).

(e) Metered latency, full smoothing at 2× heap (256 MB).

(f) Metered latency, full smoothing at 6× heap (768 MB)

Figure 3. DaCapo Chopin records the time for each event for its latency-sensitive workloads, avoiding the need for users to resort to using misleading proxies such as GC pause times. These figures plot the distribution of request latencies for cassandra for each of OpenJDK 21's five production collectors, with the 95th percentile indicated by the shaded area. Even at the generous 6.0× heap, the newer collectors do not deliver better latency than G1 on this workload.

rather than reporting singular latency metrics.

The inclusion of a realistic and diverse set of latency-sensitive workloads based on modern widely-used frameworks such as Spring, Cassandra, Kafka, Lucene, Tomcat, and Wildfly, and built-in latency metrics allow the community to easily and systematically measure user-experienced latency.

4.5 Lower Bound Garbage Collection Overheads

Understanding the real cost of garbage collection is a long-standing problem. The root of the problem is that garbage collection costs (and benefits) can be hard to measure. The difficulty of attribution is due to some costs being finely woven into the fabric of the application, such as the cost of the allocator or the cost of read and write barriers, while other costs are indirect, such as the locality effects of various allocation strategies or copying orders.

Cai et al. [10] exploit two simple observations to develop an easy-to-use and transparent measure of GC overheads: 1. If a perfect zero-cost GC existed, it could be used as the baseline with which to measure the overhead of concrete collectors. The overhead of a collector would simply be the difference between a benchmark run using that collector and the benchmark using the ideal collector. 2. Although the ideal

collector by definition does not exist, it can be *approximated* by running with a real garbage collector and subtracting costs easily attributable to the GC from the total costs.

The methodology thus requires taking measurements using multiple collectors, subtracting the costs attributable to the GC in each case. The system with the lowest net cost is the best approximation to ideal, and thus forms the baseline. The difference between the total cost for a concrete system and the baseline is thus an approximation to its overhead. Since the baseline is always an overestimate of the ideal, this overhead measure is always an *underestimate* of the overhead, and is thus a *lower bound on overhead*. This methodology is transparent and simple to implement, yielding a clear insight into the real overheads of garbage collection.

Recommendation O1. Researchers should report GC overheads when evaluating garbage collectors, using a methodology such as LBO [10], as illustrated in Figure 1.

Recommendation O2. Researchers should report both wall clock and total CPU overheads.[6]

5 DaCapo Chopin Benchmarks

The DaCapo Chopin suite replaces DaCapo Bach, adding eight new benchmarks and removing one. The composition of the suite, the retirement of old benchmarks and the addition of new ones is driven by community engagement, with the goal of keeping the suite relevant, diverse, and representative. Section 5.2 explains how we use principal components analysis (PCA) to quantify the diversity of the benchmarks we include.

5.1 Nominal Statistics

DaCapo Chopin comes with a large and diverse set of precomputed analyses and statistics, including bytecode execution and allocation size statistics as well as various performance metrics, such as ones measuring sensitivity to heap and cache size.[7] The statistics are included as part of the suite because they are methodologically and computationally non-trivial to calculate, yet provide considerable insight into *how* each of the benchmarks behave and *why* they behave that way.

DaCapo's *nominal statistics*[5] are derived from these precomputed metrics and are available, with brief descriptions, at the command line (`-p`). Their purpose is to provide benchmark users with a rich *qualitative* characterization of each workload with respect to a fixed hardware and software setting. Each benchmark is scored out of ten against each metric. The score is a simple linear mapping of the benchmark's rank among all benchmarks. 1 indicates the lowest ranked, while 10 indicates the highest ranked. We characterize each benchmark in the DaCapo Chopin suite across at least 35 dimensions[8].

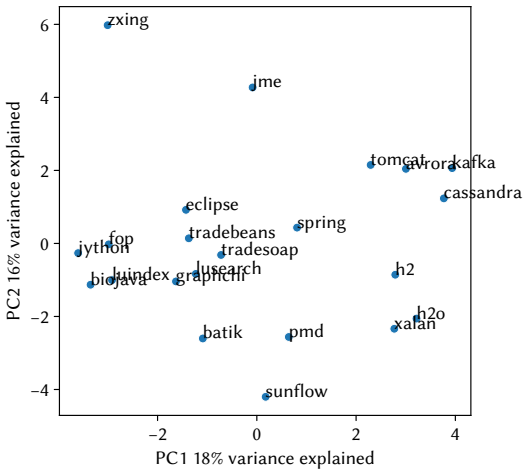
The inclusion of such metrics in a benchmark suite is novel as far as we know. We believe that they will help improve methodology (for example, nominal minimum heap sizes are among the statistics), and help researchers readily reason about behaviors they observe (such as why a compiler optimization appears to be less effective on some benchmarks).

The original DaCapo paper made the choice to only characterize statistics using JVM-neutral measures, such as total bytes allocated and total objects allocated [6]. We found that approach overly constraining as it precluded using metrics such as architectural sensitivity, which require measurements on a real JVM. Instead we chose to use OpenJDK 21 with its most basic configurations (such as the default G1 garbage collector), and describe the measures we made as *nominal*. This is to make clear that we were not attempting to *evaluate* the benchmarks or the JVM but to *characterize* the benchmarks within the suite in meaningful ways that would help users of the suite better understand the benchmarks and their sensitivity to various aspects of the execution environment. Our metrics include measures such as sensitivity to last level cache size, sensitivity to compiler configuration, sensitivity to heap size, etc. The focus of the nominal statistics is the *rank* among the benchmarks. We gather the statistics using a variety of techniques including time-consuming bytecode instrumentation and (separate) performance measurements. The bytecode instrumentation tools are included as part of the suite, allowing others to reproduce our measurements.

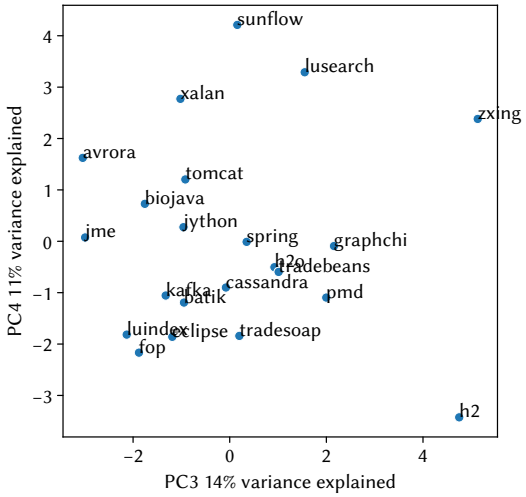
We give each nominal statistic a three-letter acronym and assign each benchmark a rank and a score from 1 to 10 on every metric, according to its sensitivity. For example, the lusearch workload has a nominal allocation rate (ARA) of 23556 MB/sec based on its total allocation and execution time. This places it first in the suite, yielding a score of 10. On the other hand, its sensitivity to aggressive (-comp) C2 compilation (PCC), is close to average, yielding a score of 4. These scores hold no meaning beyond allowing users to assess the *relative* sensitivities of the workloads.

We cluster the statistics into five groups, indicated by the first letter in the metric's acronym. The first four allocation metrics (AOA, AOL, AOM & AOS) are based on data gathered via bytecode-instrumented executions of the benchmarks, and the fifth (ARA) combines a measure of bytes allocated with a separately measured, uninstrumented time for benchmark execution. The seven bytecode metrics (BAL, BAS, BEF, BGF, BPF, BUB & BUF) are also gathered via bytecode instrumentation. Four of these (BAL, BAS, BGF & BPF) combine

[6] Note that naïvely counting cycles on a heterogenous platform unsound.
[7] From DaCapo Chopin MR1 onward, these statistics are available within the `stats` folder, e.g.: `dacapo-23.11-MR1-chopin/stats`.
[8] Most benchmarks have 47 dimensions, but not every dimension is available or relevant to each benchmark. tradebeans and tradesoap have the fewest, at 35, while h2 has the most at 47.



(a) PC1 and PC 2



(b) PC3 and PC4.

Figure 4. Principal components analysis of the 22 DaCapo workloads with respect to the 33 nominal statistics which had non-null results for all benchmarks.

bytecode counts with uninstrumented execution time to produce a rate. The twelve garbage collection metrics (GCA, GCC, GCM, GCP, GLK, GMD, GML, GMS, GMU, GMV, GSS, & GTO) use telemetry from the runtime's garbage collector. The eleven performance metrics (PCC, PCS, PET, PFS, PIN, PKP, PLS, PMS, PPE, PSD, & PWU) measure the execution time of the benchmark under various hardware and software configurations. Eleven of the microarchitectural metrics (UBM, UBR, UBS, UDC, UDT, UIP, ULL, USC, & USF) use hardware performance counters to measure performance characteristics, while the remaining two, UAI and UAA, measure the sensitivity of the benchmarks to running on entirely different processor designs (Intel and ARM).

5.2 Principal Component Analysis

We use the nominal statistics for each benchmark to conduct a principal component analysis of the workloads in the suite. In the analysis we use the 33 nominal metrics where all benchmarks have data points. We use raw values rather than scores, and apply standard scaling (linear scaling with 0 mean and unit variance). Figure 4 shows scatter plots of the twenty two workloads with respect to the top four principal components, with PC1 being the most determinative component (18 %) and PC4 being the least (11 %). Together, these four principal components account for over 50 % of the variance between benchmarks. Intuitively, the further apart the workloads are in the scatter graph, the greater the difference between them with respect to the nominal statistics. When designing a suite, diversity is important for coverage, and to avoid implicit duplication and thus over-emphasizing certain features. Figure 4 shows that the workloads that make up the DaCapo Chopin suite are well distributed, exhibiting substantial variation.

6 Analysis

We now use the new DaCapo Chopin and the methodologies we have discussed in the previous sections to conduct a detailed analysis of the workloads. The detailed results and statistics that underpin the following analysis are available within the benchmark suite[7] and as an appendix to this paper. In this section we will explain the methodology we've used throughout our analysis and highlight key results.

6.1 Methodology

The large number of dimensions available to our analysis prevents us from exploring the cross product of all variations across each dimension. Instead we identify a single baseline and explore variations with respect to it. We chose as our baseline the default configuration of the most recently shipping OpenJDK release, running on a recent x86 processor.

6.1.1 Benchmark Suite. We use version 23.11-chopin-MR2 of the DaCapo benchmark suite [1, 6], the most recent release of the suite at the time of writing.

6.1.2 JVM, Compilers and Garbage Collectors. We use the OpenJDK 21 runtime 2024-07-16 LTS shipped as the Temurin-21.0.4+7 distribution. Unless otherwise stated: for compatibility and consistency when evaluating across multiple OpenJDK versions, we used -server to select the runtime's compiler behavior; we ran 5 iterations of each benchmark, timing the last; and we used a 2× the benchmark's GMD nominal statistic, which is the minimum heap size in which that application will run 5 iterations with the default collector and the -server flag. When controlling for heap size, we used the -Xms and -Xmx flags. We run 10 invocations of each benchmark and show or plot the 95 % confidence intervals.

In practice, 10 invocations is sufficient to produce results with sufficiently tight confidence intervals.

6.1.3 Hardware and Operating System. Our default hardware configuration is an AMD Ryzen 9 7950X Zen4 with 16 cores and 32 hardware threads, a 4.5 GHz base clock (with frequency scaling turned off), and 64 MB of last level cache. The system has 2 × 32 GB of DDR5-4800 with the standard JEDEC 40-39-39-77 timing profile.

We used the Ubuntu 22.04.4 distribution with the Linux 6.8.0-40 kernel, with the scaling governor set to performance. We turned off NMI watchdog to fully use all 6 performance monitor counters on the CPU. The system firmware is AMD AGESA 1.1.0.0.

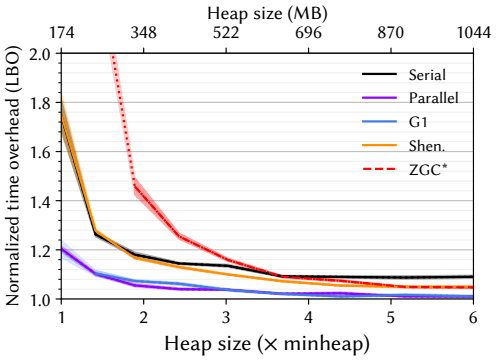
When testing benchmarks' sensitivity to memory speed, we configure the memory to the equivalent of DDR5-2000 32-32-32-64. When testing benchmarks' sensitivity to frequency scaling, we enable Core Performance Boost. When testing benchmarks' sensitivity to LLC size, we use AMD's PQOS L3 Cache Alloction Enforcement through Linux's resctrl interface.

6.2 Lower Bound Overheads

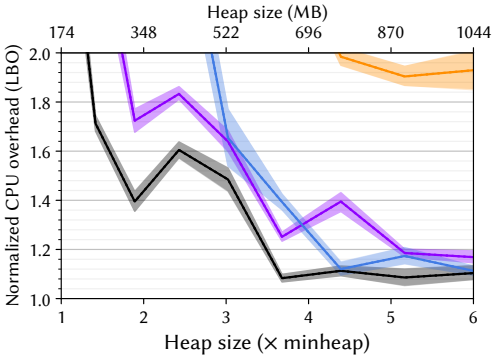
We now use the lower bound overhead (LBO) methodology introduced by Cai et al. [10]. The key to LBO is that it exposes the *total overheads* of a garbage collector relative to a conservative approximation to the ideal. LBO is methodologically straightforward, yet captures difficult-to-attribute overheads which had previously gone largely unmeasured. We discussed the geometric mean of LBO results in Section 2, and per-benchmark LBO results are included in the appendix. Here we analyze cassandra and lusearch as examples.

LBO Methodology. The key idea is to 'distill' a baseline that conservatively approximates the ideal GC. The distilled baseline is then used as the denominator in the LBO graphs, while the measured system forms the numerator [10]. We use Java's JVMTI interface to capture the easily-attributable stop-the-world periods of the collectors. The remainder is an approximation to the application costs. We then find the lowest approximated application cost from among all collectors and all heap sizes, and use that as the distilled cost, our denominator. Note that the simpler the collector, the more likely it is that the stop-the-world period captures most of the collector's cost. The simplest of the OpenJDK 21 collectors are Serial and Parallel, but even these have write barriers embedded within the application. So our distilled cost is clearly short of the ideal, and as a consequence the LBO overheads are *systematically conservative* estimates.

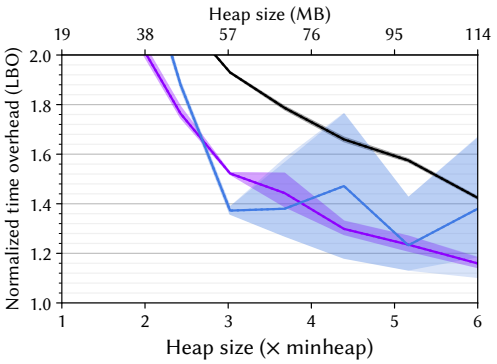
We use all of the garbage collectors that ship with OpenJDK 21 in our analysis. We evaluate them with respect to wall clock and task clock, at heap sizes from 1–6× the minimum heap size. The task clock captures total CPU use, across all threads. We plot each curve and indicate 95 % confidence intervals with shading.



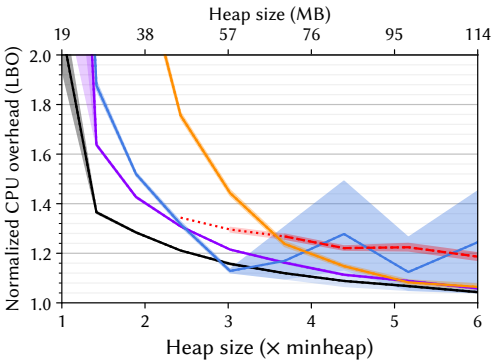
(a) Wall clock overheads for cassandra.



(b) Total CPU overheads (task clock) for cassandra.



(c) Wall clock overheads for lusearch.



(d) Total CPU overheads (task clock) for lusearch.

Figure 5. LBO overheads for cassandra and lusearch.

LBO Analysis. Figure 5(a) and Figure 5(b) show overheads for cassandra. The wall clock and task clock results are strikingly different. Above 4× the minimum heap size, all collectors have modest wall clock overheads, and right down to a modest 2× heap, Shenandoah's overhead remains below 20 %, while G1 and Parallel remain below about 10 %. However, the task clock tells a different story. G1 has a 60 % overhead even at a moderate 3× heap, while other collectors have overheads greater than a factor of two. This is most likely due to the collectors successfully making use of unused cores, since cassandra itself is not fully utilizing the available hardware. Although in the case of our experimental setup, the cores not being used by cassandra were free, in general there is an opportunity cost associated with using computational resources like this. This highlights the importance of taking both wall clock and task clock measures.

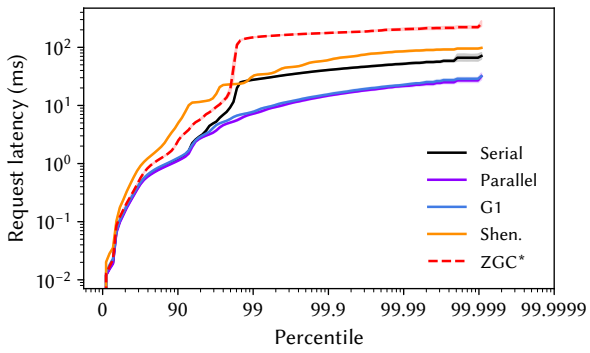
Figure 5(c) and Figure 5(d) show the overheads for lusearch. Wall clock overheads for Shenandoah are very high, greater than the 2.0× y-axis limit for all values of x. However, task clock overheads are significantly *lower*. This is counter intuitive since Shenandoah is a concurrent collector. However, note that lusearch has a very high allocation rate (ARA). Collectors like Shenandoah throttle the application in cases where the collector can't free memory fast enough to satisfy the application's demand. In this case, Shenandoah is throttling lusearch's 32 rapidly allocating client threads. This has the effect of much worse wall clock time (Figure 5(c)), and the side effect that the application can run efficiently due to less synchronization and contention (Figure 5(d)).

6.3 User-Experienced Latency

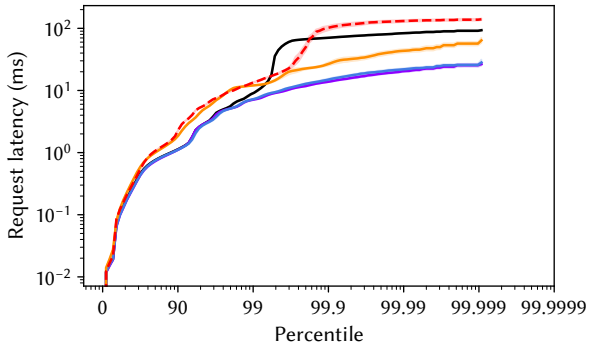
In Section 4.4 we described how DaCapo Chopin directly measures user-experienced latency, and discussed simple and metered latency for cassandra.

Figure 6 shows user-experienced latency for the h2 workload. The graphs are remarkably consistent. Above the 99.9th percentile the graphs are almost identical, with plateaus that reflect pauses in the range of 10–200 ms. The effect of the larger heap size is for most collectors to push the curves to the right. These graphs raise four questions: 1. Why are metered and simple latency almost identical at 2×? 2. Why do the latency-sensitive collectors (Shenandoah, ZGC and GenZGC) perform worse than Parallel and G1 in all cases, and worse than Serial in the 2× heap? 3. Why do all collectors have slightly *worse* tail latency when the heap is larger? 4. Why does Shenandoah's metered latency get *worse* at the larger heap size?

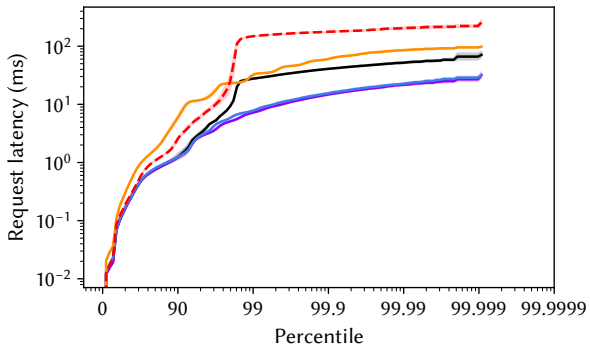
Answering these questions requires more understanding of the h2 workload. h2 is a database benchmark. It first creates a large in-memory database and then conducts queries over that database using the TPC-C workload [42]. It is the latency distributions of those queries that is depicted Figure 6. As a result of its design, h2 has a large heap size (GMD) but a low memory turnover (GTO), and each GC tends to



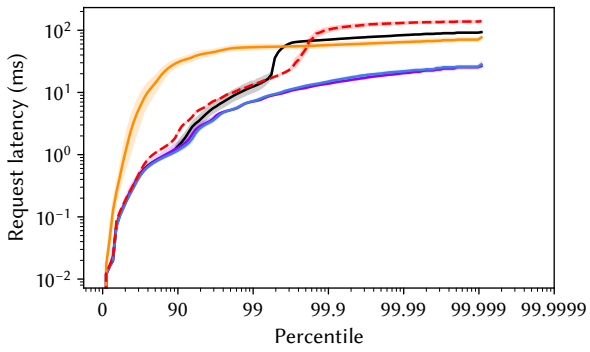
(a) Simple latency, 2× heap (1.36 GB).



(b) Simple latency, 6× heap (4 GB).



(c) Metered latency, full smoothing, 2× heap (1.36 GB).



(d) Metered latency, full smoothing, 6× heap (4 GB).

Figure 6. User-experienced latency for h2, plotting the latency distribution for 100000 requests using each collector.

yield a lot (GCM). Although its allocation rate (ARA) is high, much of that allocation occurs in the database construction phase. As a result of these factors, h2 has low sensitivity to heap size (GSS).

Recall that because metered latency takes the *earlier* of the actual and notional uniform start times, but leaves the end time unchanged, it can never be lower than the simple latency. The similarity between the metered and simple latency in Figure 6 is consistent with the pauses due to garbage collection being small relative to the query execution time. In fact the 90th percentile latency of the h2 queries is around 3 ms. The heap profile of h2 means that it needs few GCs, and those are performed very quickly and productively, having little impact on the query latency, which is why the metered latency is almost identical to the simple latency, and why a simple collector like Serial is able to perform relatively well.

The reason for the poor latency for the newer collectors can be explained by h2's LBO graph, Figure 16 of the appendix. The new collectors have task clock overheads of around 70 % at the 6×, rapidly rising above 100 %. This means that the collectors are consuming roughly half or more of the available CPU cycles, with the result that individual queries are running noticeably slower.

Looking closely at the 6× heaps, all collectors have slightly worse latency at the far right of the graph (i.e. the tail). This effect is particularly noticeable for Serial. This is because when the heap is a lot larger, although collections will be less frequent, each collection will likely take a little longer since the larger heap will likely hold more live data than the smaller heap. Thus pauses tend to be larger when the heap is larger.

The noticeably worse metered latency for Shenandoah at the 6× heap is explained by Shenandoah's mutator throttling. When it cannot collect fast enough to keep up with application's allocation needs, Shenandoah will throttle application threads to make more hardware available for concurrent garbage collection work. It does this aggressively in the 2× heap, resulting in less application parallelism, and it is able to keep up. However, at 6.0× it is less aggressive, and as a consequence exposes the application threads that run to more GC-induced overheads. This is evident from h2's time LBO, which shows time overheads well over 100 % at 2× due to the mutators being throttled. We also confirm this by reviewing Shenandoah's GC log.

This analysis of h2's latency highlights the importance of a multi-faceted performance analysis, including user-experienced latency, wall clock and task clock LBO, as well as insights into the workload's characteristics revealed by DaCapo's nominal statistics.

6.4 Architectural Sensitivity

Among the twelve nominal statistics most dominant in the PCA analysis are six microarchitectural features: instructions per cycle (UIP), level one data cache miss rate (UDC), last level cache miss rate (ULL), front and back end processor boundedness (USF, USB) and SMT contention (USC). This illustrates that the workloads in the suite exhibit microarchitectural diversity and substantially different degrees of architectural sensitivity. To explore this further and to highlight the analytical utility of the microarchitectural measures included with the DaCapo nominal statistics, we consider four workloads in more detail.

Instructions executed per clock (IPC) indicates how effectively an out of order processor is being utilized. The maximum IPC of any workload is bounded by the number of issue slots in the CPU, which is 6 on the AMD Zen4 machine we use. The IPCs of the DaCapo Chopin workloads range from biojava and jython with high IPCs of 4.76 and 2.76 to h2o and xalan with IPCs of just 0.92 and 0.94 respectively. We will use these four workloads as examples in our exploration of architectural sensitivity. The complete nominal statistics for each of these workloads is included in the `stats` folder of the benchmark suite and the appendix to this paper.

biojava. The high IPC (4.67) of biojava, which analyzes protein sequences, reflects that it is a highly-tuned computational workload. The nominal stats reveal that with the exception of pipeline restarts (UBR), the other nine 'negative' microarchitectural measures are among the lowest in the suite. biojava is fairly insensitive to memory slowdown (PMS) and last level cache size reduction (PLS). Consistent with this, it has above average sensitivity to CPU frequency scaling (PFS) and compiler configuration (PCC, PCS).

jython. The high IPC of jython (2.68) is less impressive than biojava's and has a different explanation. It is a python language implementation that implements an interpreter. It scores very well in most of the metrics, but suffers from very high stalls due to bad speculation due to misprediction (UBP & UBS). It is insensitive to memory speed (PMS) and last level cache size (PLS). This is all consistent with jython spending most of its time in a small but somewhat unpredictable interpreter loop.

xalan. The low IPC of xalan (0.98) is due to a mix of factors, but poor locality is key. It has very high data cache, last level cache, and DTLB miss rates (UDC, ULL, UDT), and is sensitive to last level cache size (PLS).

h2o. Memory performance is an even bigger contributor to h2o's low IPC (0.89). It has the highest back end stalls (USB) and last level cache misses (ULL), and very high data cache and DTLB misses (UDC & UDT). It also has high sensitivity to memory speed (PMS). These are consistent with h2o being a memory-intensive machine learning workload.

7 Conclusion

This paper outlines a problem: when methodological norms cannot keep pace with innovation, we lack the tools we need to notice important regressions. We motivate the problem concretely and then respond to it three ways. First, we contribute a benchmark suite that embodies fourteen years of work, adding eight completely new workloads and bringing existing workloads up to date. The workloads are rich, together constituting a codebase of roughly 16 MLOC. They are diverse, as shown by our principal components analysis. Second, we contribute methodological innovations as part of the suite, allowing researchers to easily analyze their workloads' characteristics, and to measure user-experienced latency. Finally, we contribute a number of methodological recommendations. Although no benchmark suite or methodology can ever be *complete* in any sense, we hope that together, these contributions will strengthen our field's methodological grounding and in doing so help address the problem we used to motivate the paper.

Acknowledgments

The development of the Bach and Chopin releases of the DaCapo benchmark suite was supported by generous donations from Google, Intel, and Oracle. This material is partially based upon work supported by the Australian Research Council under Grant No. DP190103367. The ARM microarchitectural evaluation uses the hardware donated by Ampere Computing. Zixian Cai is supported by an Australian Government Research Training Program Scholarship.

References

[1] 2023. DaCapo 23.11-chopin. https://github.com/dacapobench/dacapobench/releases/tag/v23.11-chopin

[2] Doug Bagley, Brent Fulgham, and Isaac Gouy. 2004. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame

[3] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 52:1–52:27. https://doi.org/10.1145/3133876

[4] Emery. D. Berger, Stephen. M. Blackburn, Matthias Hauswirth, and Michael Hicks. 2018. SIGPLAN Empirical Evaluation Checklist. http://www.sigplan.org/Resources/EmpiricalEvaluation/

[5] Stephen M. Blackburn, Amer Diwan, Matthias Hauswirth, Peter F. Sweeney, José Nelson Amaral, Tim Brecht, Lubomír Bulej, Cliff Click, Lieven Eeckhout, Sebastian Fischmeister, Daniel Frampton, Laurie J. Hendren, Michael Hind, Antony L. Hosking, Richard E. Jones, Tomas Kalibera, Nathan Keynes, Nathaniel Nystrom, and Andreas Zeller. 2016. The Truth, The Whole Truth, and Nothing But the Truth: A Pragmatic Guide to Assessing Empirical Evaluations. *ACM Trans. Program. Lang. Syst.* 38, 4 (2016), 15:1–15:20. http://dl.acm.org/citation.cfm?id=2983574

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. 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*, Peri L. Tarr and William R. Cook (Eds.). ACM, 169–190. https://doi.org/10.1145/1167473.1167488

[7] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2008. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM* 51, 8 (2008), 83–89. https://doi.org/10.1145/1378704.1378723

[8] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. 2001. Pretenuring for Java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001*, Linda M. Northrop and John M. Vlissides (Eds.). ACM, 342–352. https://doi.org/10.1145/504282.504307

[9] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. 2001. Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001*, Linda M. Northrop and John M. Vlissides (Eds.). ACM, 353–366. https://doi.org/10.1145/504282.504308

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

[11] Maria Carpen-Amarie, Georgios Vavouliotis, Konstantinos Tovletoglou, Boris Grot, and René Müller. 2023. Concurrent GCs and Modern Java Workloads: A Cache Perspective. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management, ISMM 2023, Orlando, FL, USA, 18 June 2023*, Stephen M. Blackburn and Erez Petrank (Eds.). ACM, 71–84. https://doi.org/10.1145/3591195.3595269

[12] Perry Cheng and Guy E. Blelloch. 2001. 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*, Michael Burke and Mary Lou Soffa (Eds.). ACM, 125–136. https://doi.org/10.1145/378795.378823

[13] H. J. Curnow and Brian A. Wichmann. 1976. A Synthetic Benchmark. *Comput. J.* 19, 1 (1976), 43–49. https://doi.org/10.1093/COMJNL/19.1.43

[14] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, David F. Bacon and Amer Diwan (Eds.). ACM, 37–48. https://doi.org/10.1145/1029873.1029879

[15] John Ellis, Pete Kovac, Hans Boehm, and Will Clinger. 1997. An Artificial Garbage Collection Benchmark. https://hboehm.info/gc/gc_bench.html

[16] Christine H. Flood and Roman Kennke. 2014. JEP 189: Shenandoah: A Low-Pause-Time Garbage Collector (Experimental). https://openjdk.org/jeps/189

[17] Christine H. Flood, Roman Kennke, Andrew E. Dinn, Andrew Haley, and Roland Westrelin. 2016. 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*, Walter Binder and Petr Tuma (Eds.). ACM, 13:1–13:9. https://doi.org/10.1145/2972206.2972210

[18] Andy Georges, Lieven Eeckhout, and Dries Buytaert. 2008. Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, Gail E. Harris (Ed.). ACM, 367–384. https://doi.org/10.1145/1449764.1449794

[19] Bhargav Reddy Godala, Sankara Prasad Ramesh, Gilles A. Pokam, Jared Stark, André Seznec, Dean M. Tullsen, and David I. August. 2024. PDIP: Priority Directed Instruction Prefetching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 846–861. https://doi.org/10.1145/3620665.3640394

[20] Phillip I. Good and James W. Hardin. 2012. *Common Errors in Statistics (And How to Avoid Them)* (4th ed.). John Wiley & Sons. https://doi.org/10.1002/9781118360125

[21] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage collection advantage: improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 69–80. https://doi.org/10.1145/1028976.1028983

[22] Schuyler W. Huck. 2009. *Statistical Misconceptions.* Routledge.

[23] Stefan Karlsson. 2021. JEP 439: Generational ZGC. https://openjdk.org/jeps/439

[24] Jin-Soo Kim and Yarsun Hsu. 2000. Memory system behavior of Java programs: methodology and analysis. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Santa Clara, CA, USA, June 18-21, 2000*, Alexandre Brandwajn, Jim Kurose, and Philippe Nain (Eds.). ACM, 264–274. https://doi.org/10.1145/339331.339422

[25] Michael Larabel. 2024. Java Throughput/Latency & Power Efficiency Tuning For AMD EPYC Turin. https://www.phoronix.com/review/java-optimizations-epyc-turin

[26] Doug Lea. 2011. JSR166 software repository. https://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/jsr166/src/test/

[27] Per Liden. 2018. JEP 377: ZGC: A Scalable Low-Latency Garbage Collector (Production). https://openjdk.org/jeps/377

[28] Per Liden and Stefan Karlsson. 2018. JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental). https://openjdk.org/jeps/333

[29] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 178–189. https://doi.org/10.1109/ASE.2015.87

[30] Merriam-Webster.com. 2023. "nominal". https://www.merriam-webster.com/dictionary/nominal

[31] Kent Mitchell. 1995. Re: Does memory leak? comp.lang.ada 1995/03/31. https://archive.legitdata.co/comp.lang.ada/3lhdjd$l6h@rational.rational.com/

[32] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing wrong data without doing anything obviously wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, Mary Lou Soffa and Mary Jane Irwin (Eds.). ACM, 265–276. https://doi.org/10.1145/1508244.1508275

[33] OpenJDK. 2023. JDK 21. https://openjdk.org/projects/jdk/21/

[34] Michael Paleczny, Christopher A. Vick, and Cliff Click. 2001. The Java HotSpot Server Compiler. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*, Saul Wold (Ed.). USENIX. http://www.usenix.org/publications/library/proceedings/jvm01/paleczny.html

[35] Orion Papadakis, Andreas Andronikakis, Nikos Foutris, Michail Papadimitriou, Athanasios Stratikopoulos, Foivos S. Zakkak, Polychronis Xekalakis, and Christos Kotselidis. 2023. A Multifaceted Memory Analysis of Java Benchmarks. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2023, Cascais, Portugal, 22 October 2023*, Rodrigo Bruno and Eliot Moss (Eds.). ACM, 70–84. https://doi.org/10.1145/3617651.3622978

[36] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 31–47. https://doi.org/10.1145/3314221.3314637

[37] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da capo con scala: design and analysis of a scala benchmark suite for the java virtual machine. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 657–676. https://doi.org/10.1145/2048066.2048118

[38] Aleksey Shipilev. 2020. JEP 379: Shenandoah: A Low-Pause-Time Garbage Collector (Production). https://openjdk.org/jeps/379

[39] SPEC Standard Performance Evaluation Corporation. 2015. The SPECjbb 2015 benchmark. https://www.spec.org/jbb2015/

[40] Phoronix Test Suite. 2025. DaCapo Benchmark. https://openbenchmarking.org/test/pts/dacapobench

[41] Synopsys. 2023. OpenJDK Estimated Cost. https://openhub.net/p/openjdk/estimated_cost

[42] TPC. 1992. TPC-C on-line transaction processing benchmark. https://www.tpc.org/tpcc/

[43] Andrew J. Vickers. 2009. *What is a p-value anyway? 34 Stories to Help You Actually Understand Statistics.* Pearson.

[44] Chenyang Wu and Min Ni. 2017. Dismissing Python Garbage Collection at Instagram. Instagram Engineering. https://instagram-engineering.com/dismissing-python-garbage-collection-at-instagram-4dca40b29172

[45] Thomas Würthinger. 2014. Graal and truffle: modularity and separation of concerns as cornerstones for building a multipurpose runtime. In *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld (Eds.). ACM, 3–4. https://doi.org/10.1145/2584469.2584663

[46] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Trans. Program. Lang. Syst.* 44, 4 (2022), 22:1–22:34. https://doi.org/10.1145/3538532

[47] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-latency, high-throughput garbage collection. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 76–91. https://doi.org/10.1145/3519939.3523440

A Artifact Appendix

A.1 Abstract

In this artifact, we provide DaCapo 23.11-Chopin release—an overhaul of the DaCapo benchmark suite [6]. As a case study, we use the new suite to measure the contemporary production Java garbage collector performance. The results from the experiments drive the methodological recommendations for performance analysis in the paper.

A.2 Artifact Checklist

- **Program:** DaCapo Chopin benchmarks, which is a contribution of this paper.
- **Run-time environment:** Recent Linux kernel (tested with Ubuntu 20.04 LTS's `5.15.0-113-generic` kernel), OpenJDK 21 (tested with Eclipse Temurin `21.0.3_9`) for running benchmarks, and Python 3.7 or newer for experiment automation (tested with Python 3.10.12). `perf_event_open` syscall access required. sudo access required for Docker.
- **Hardware:** An x86_64 host. 16 cores/32 threads, and 32 GB RAM are recommended. AMD Ryzen 9 7950X with frequency scaling off, and 2 × 32 GB DDR5-4800 DIMM usng JEDEC 40-39-39-77 timing, are required to reproduce the performance results.
- **Execution:** The machine should be otherwise idle.
- **Metrics:** Execution time, task clock, simple and metered user-experience latency in various quantiles.
- **Output:** Console logs with performance metrics, and raw latency CSVs for latency-sensitive benchmarks.
- **Experiments:** Experiments are automated via running-ng v0.4.6, which provides rich customization options.
- **How much disk space required (approximately)?:** 30 GB: a 17 GB image, which is 7 GB after compression.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 1 hour for a simiplified run, and 1 week for a full run.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache License, Version 2.0
- **Workflow framework used?:** running-ng v0.4.6
- **Archived (provide DOI)?:** 10.5281/zenodo.12682890

A.3 Description

A.3.1 How to Access. Download from the Zenodo archive.

A.3.2 Hardware Dependencies. Please refer to the above checklist.

A.3.3 Software Dependencies. Docker Engine required on the host, and all other dependencies are provided in the Docker image.

A.4 Installation

Import the Docker image using `docker load < dacapo-asplos-2025-artifact.tar.gz`. A container can be launched using `docker run -it --cap-add PERFMON --rm -v ./results:/dacapo/results dacapo`. All the below commands are to be run inside the container under `/dacapo` as the working directory.

A.5 Basic Test

`running runbms ./results/ ./experiments/kick-the-tires.yml -p "kick-the-tires" -s 2`.

A.6 Experiment Workflow

Use `running runbms` and provide a folder to store results and the path to the experiment definition file. Please refer to the `README.md` of the artifact for more details.

A.7 Evaluation and Expected Results

Note that the full experiment run is time consuming, and we provide a smaller subset (see `README.md`).

To reproduce the results for the time-space tradeoff (Section 4.2) and lower bound garbage collector overheads (Section 4.5), use `running runbms ./results/ ./experiments/lbo.yml 8 -p "lbo"`. The results can reproduce Figure 1 and Figure 5.

To reproduce the results for user-experienced latency (Section 4.4), use `running runbms ./results/ ./experiments/latency.yml -s 6,2 -p "latency"`. The results can reproduce Figure 3 and Figure 6.

To view the nominal statistics for each benchmark, pass `-p` to the benchmark.

Please refer to the `README.md` of the artifact for sample outputs, and other details.

A.8 Experiment Customization

We use `running-ng` to systematically run experiments, which provides rich customization options. Experiments are defined using composable and customizable YAML files. Please refer to the `README.md` of the artifact for more details.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html