# Concurrent GCs and Modern Java Workloads: A Cache Perspective

Maria Carpen-Amarie
Huawei Zurich Research Center
maria.carpen.amarie@huawei.com

Georgios Vavouliotis
Huawei Zurich Research Center
georgios.vavouliotis2@huawei.com

Konstantinos Tovletoglou
Huawei Zurich Research Center
konstantinos.tovletoglou@huawei.com

Boris Grot
University of Edinburgh
Huawei Zurich Research Center
boris.grot@ed.ac.uk

Rene Mueller
Huawei Zurich Research Center
rene.mueller@huawei.com

## Abstract

The garbage collector (GC) is a crucial component of language runtimes, offering correctness guarantees and high productivity in exchange for a run-time overhead. Concurrent collectors run alongside application threads (mutators) and share CPU resources. A likely point of contention between mutators and GC threads and, consequently, a potential overhead source is the shared last-level cache (LLC).

This work builds on the hypothesis that the cache pollution caused by concurrent GCs hurts application performance. We validate this hypothesis with a cache-sensitive Java micro-benchmark. We find that concurrent GC activity may slow down the application by up to 3 × and increase the LLC misses by 3 orders of magnitude. However, when we extend our analysis to a suite of benchmarks representative for today's server workloads (Renaissance), we find that only 5 out of 23 benchmarks show a statistically significant correlation between GC-induced cache pollution and performance. Even for these, the performance overhead of GC does not exceed 10 %. Based on further analysis, we conclude that the lower impact of the GC on the performance of Renaissance benchmarks is due to their lack of sensitivity to LLC capacity.

***CCS Concepts:*** • **Software and its engineering → Runtime environments**; **Garbage collection**; • **Computer systems organization → Multicore architectures**.

***Keywords:*** JVM, garbage collection, ZGC, cache pollution

*ISMM '23, June 18, 2023, Orlando, FL, USA*

ACM ISBN 979-8-4007-0179-5/23/06…$15.00
https://doi.org/10.1145/3591195.3595269

**ACM Reference Format:**
Maria Carpen-Amarie, Georgios Vavouliotis, Konstantinos Tovletoglou, Boris Grot, and Rene Mueller. 2023. Concurrent GCs and Modern Java Workloads: A Cache Perspective. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3591195.3595269

## 1 Introduction

Automatic memory management, also known as *garbage collection (GC)*,[1] is a technique that provides memory access safety and reliability while significantly reducing developers' load. These aspects make garbage collection an essential component of managed runtime environments (*e.g.*, Java, C#, JavaScript), which are intensively used by web services (*e.g.*, Twitter), web browsers, and mobile platforms (*e.g.*, Android). For these reasons, automatic memory management continues to be a hot topic today, even after more than half a century of active research towards its optimization.

However, the benefits offered by the GC do not come for free. Prior work [2, 4, 13, 19, 27, 30] has shown that application performance is profoundly contingent on the effectiveness of the garbage collector. Historically, GCs harmed application performance due to unacceptably long stop-the-world (STW) pauses. To reduce the performance overhead and responsiveness issues created by STW events, significant effort was put into re-designing and implementing more efficient GCs [7, 12, 16, 22, 26, 28, 29]. Therefore, most modern GCs [10, 32] have increasingly shorter pauses, while performing most of the work concurrently with the application threads (*e.g.*, ZGC [32]). However, concurrency comes at a price as well [5], as the application needs to share resources with the GC (*e.g.*, cache capacity, bandwidth, CPU time) and even sometimes help with the collection itself. To the best of our knowledge, there is no work on concurrent GCs that quantifies these overhead components individually. Such information would reveal new weaknesses and opportunities and facilitate well-targeted performance improvements.

[1] For the rest of the paper we use *GC* to refer to the process of garbage collection as well as the garbage collector itself.

To this end, we embark on a study of GC effects at the cache-level, as a well-defined and self-contained overhead component specific to modern concurrent collectors. STW collectors are known to sometimes improve the cache locality for the application by relocating the objects in memory in an access-friendlier layout than when they were first allocated [8, 15, 21]. However, this implication does not hold for concurrent GCs. They access and move objects in memory that do not necessarily belong to the effective working-set of the application, potentially polluting the cache. Therefore, concurrent collectors are commonly blamed in the literature for harming application locality [31].

Starting from this idea, we devise a methodology on how to quantify the effect of concurrent GCs on the performance and cache locality of the application. Our work focuses on the shared last-level cache (LLC),[2] for the following reasons: (i) the LLC can be large enough to hold (a part of) the working-set of an application, and (ii) GC-induced LLC pollution may impact performance regardless of the core where the workload runs. From a runtime perspective, we focus on OpenJDK 17 and its newest concurrent collector, ZGC [32]. We also consider another concurrent GC in OpenJDK 17, Shenandoah [10], to reproduce our results.

We conduct two separate sets of experiments. First, we implement a cache-sensitive micro-benchmark and evaluate it in an environment where we can steer GC activity in a deterministic manner. The results confirm that concurrent GCs pollute the cache hierarchy and thus harm the performance of *cache-sensitive* applications. This finding holds for both ZGC and Shenandoah. The second set of experiments aims to validate the results obtained in the controlled environment on benchmarks that represent real-world workloads, such as the Renaissance suite [24]. In contrast to the micro-benchmark, we find that only a small subset of benchmarks follow the same trend. For the rest of the benchmarks no correlation can be established between GC activity and cache effects. These results are further backed up by a statistical analysis of the significance of the observed correlation. We explain this divergence in behavior in a real-world scenario compared to the artificial micro-benchmark based on the lack of temporal reuse in the LLC for these Java workloads. In other words, regardless of the memory management solution, the working-set of the benchmarks is either unable to fit in the LLC or is very small compared to the LLC size. This makes any GC impact on the cache negligible.

In summary, this work makes the following contributions:

- We analyze the correlation between application performance and LLC misses during GC activity in a controlled environment. To this end, we build a cache-sensitive synthetic micro-benchmark. We show that for this micro-benchmark the marking phase of ZGC alone leads to a 3 × slow-down in execution time and a simultaneous 1700 × increase in LLC misses. We reproduce the behavior with Shenandoah.
- We extend the analysis to the Renaissance benchmarks. For these workloads, ZGC activity causes at most 10 % slow-down and up to 56.3 % more LLC misses, as opposed to the micro-benchmark scenario.
- We investigate the discrepancy between the experimental results for artificial and real workloads through a statistical analysis and a cache-sensitivity study. We conclude that the negligible GC effect on application performance for Renaissance is due to a lack of sensitivity to the LLC size for these Java workloads.

## 2 Background and Motivation

This section elaborates on the main technologies and mechanisms that are further employed in this work. More precisely, we focus on concurrent garbage collectors in the HotSpot Java Virtual Machine (JVM) of OpenJDK.

### 2.1 Concurrent Java GCs

A garbage collector is called *concurrent* if it is capable of collecting the memory without stopping the application threads (also known as *mutators*). Since existing concurrent GCs still need STW pauses for correctly preserving the GC algorithm invariants (*e.g.*, mutators must never see pointers to regions of memory marked for evacuation), they are also called *mostly-concurrent*. The main difference compared to STW collectors is that concurrent GCs only need to pause the application for very short intervals, instead of for the whole collection cycle. The two mostly-concurrent GCs currently implemented in OpenJDK are *ZGC* and *Shenandoah*.

A concurrent GC cycle can be roughly split into two main parts: *marking* and *relocation* (also called *compaction*). During the marking phase, the collector iterates over the reference graph of the objects in the heap, starting from the roots (local variables, threads, JNI references). During this traversal, the GC marks all live objects, the *live-set* of the application. Everything that remains unmarked on the heap is considered garbage and reclaimed. During the relocation phase, the collector moves the live objects in order to compact the heap and fixes the references to the new address of the relocated objects. For concurrent collectors, the mutators may help with relocation and/or reference fixing.

### 2.2 Case Study: Z Garbage Collector (ZGC)

This work focuses on ZGC for two reasons: (i) ZGC is the newest concurrent collector in OpenJDK, and (ii) ZGC has all STW pauses under 1 ms and independent of the heap, live-set or root-set size (they are virtually constant [20]).

A ZGC collection cycle has several interwoven STW and concurrent phases, as illustrated in Figure 1. ZGC cycles cover a marking phase and a relocation phase, which roughly

[2] For the rest of the paper, if the cache level is not specified, the generic term *cache* refers to the LLC.

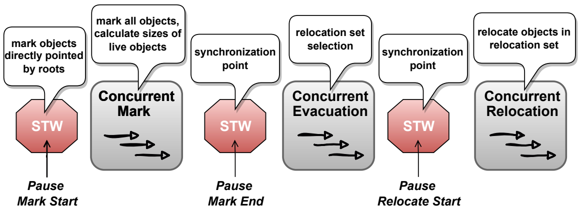


**Figure 1.** ZGC collection cycle in steps

correspond to the general description of concurrent collection presented in Section 2.1. Note that the object graph traversal to perform the marking invariably takes place in every cycle. Its cost directly depends on the live set of the application. Conversely, the work of the relocation phase depends on the state of the heap after identifying the live-set, the amount and location of the heap memory that can be reclaimed, the heap fragmentation, heap size, etc. It is therefore possible that no or very little relocation is needed, depending on the workload and other factors.

From an experimental perspective, ZGC has very few widely-used configuration options, relying on many internal heuristics to make the collection efficient without any help from the user. For initial calibration, ZGC executes several "warm-up" collections when the application starts. Then, in order to reduce memory pressure and application stalls, ZGC proactively collects parts of the heap when this extra work is tolerated by the system. Otherwise, ZGC monitors the ongoing allocation rate of the application and estimates how fast the heap would fill up if current allocation rate is maintained. GC cycles are triggered based on these predictions. There are options to turn off the proactive GC and to further tweak the timing and heuristics of the allocation rate-induced GCs.

### 2.3 Motivation

Concurrent GCs successfully address the problem of long pause times for the application, as explained in Section 2.2. On the flip side, concurrent execution also means that GC threads and application threads share resources (*e.g.*, CPU, bandwidth, cache capacity). Intuitively, allowing GC threads to compete with application threads for resources, would harm the performance of the application.

This work focuses on *cache pollution* as a specific overhead component of concurrent GCs and aims to quantify to what extent it can affect application performance. Figure 2 illustrates one potential manner of interaction between GC and mutators running inside a JVM. In the figure, the mutator and the GC run concurrently on separate cores, each with private L1 and L2 caches, but contend for LLC capacity with cache blocks fetched by the application (in orange) and by the GC (in blue) potentially evicting each other.

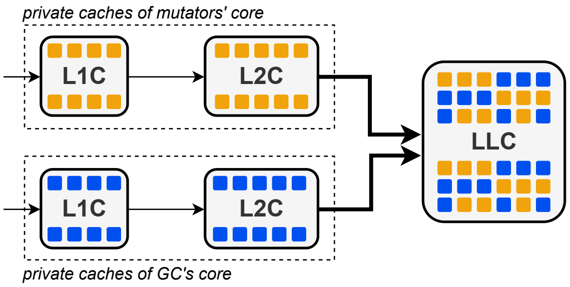


**Figure 2.** GC and mutators competing for LLC capacity when there is no contention for the private per core caches.

There are also other important considerations to a GC-induced cache pollution analysis. Figure 2 assumes that a GC cycle is in progress. When the GC is not actively working, the LLC contains only memory blocks fetched by the application threads. Moreover, the figure suggests that completely different memory blocks are requested by the GC and the mutators. This is a valid scenario, in which the GC live-set and application working-set are fully disjoint. However, most of the times the mutators and the GC threads share the working-set. Therefore, a fraction of the objects accessed by the GC may later be used by the application threads without the need to fetch the corresponding memory block any more. Finally, it is possible that the live-set and working-set completely overlap and also fit in the last-level cache, in which case GC-induced LLC pollution would not occur. In this case, the GC may even help improve application performance by reorganizing the layout of objects in the heap.

## 3 Challenges and Methodology

This section presents the obstacles we faced towards quantifying the impact of concurrent GCs on application performance and cache locality. It then describes the methodology we apply in our experiments to overcome these challenges.

### 3.1 Challenges

***GC cannot be disabled.*** GC overheads are notoriously difficult to quantify. The main challenge lies in the fact that applications running in managed runtimes do not work properly in the absence of garbage collection. Because of this, no GC on/off option and no JVM version without GC is ever provided. As such, the GC itself cannot be simply disabled. Applications with very small memory footprint could potentially terminate successfully without GC, but even then the execution is still not directly comparable – object access and allocation would behave differently, as the memory fills up and no cleaning or relocation is provided. In most recent OpenJDK versions, a no-operation GC is available, called EpsilonGC [23]. This collector offers the same interface as the others but does not collect the memory. It serves precisely as a tool for comparing and diagnosing GC problems. However, in some cases, applications may end up being slower

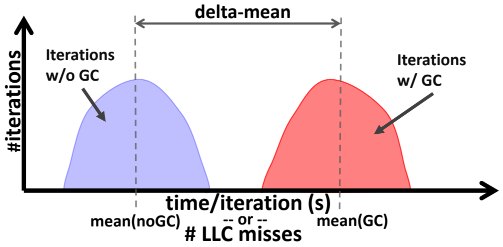


**Figure 3.** Expected distribution of iteration duration (or LLC misses) according to our methodology

without memory management than with any GC [5]. This issue prevents performance comparisons with and without GC in order to measure overheads.

***Applications threads might assist the GC.*** A challenge specific to *concurrent GCs* is the fact that the application is seamlessly involved in the collection work. In other words, the application threads execute instructions that do not belong to the application's code in order to help the GC perform the collection. However, this additional work is so deeply interleaved within the execution, that identifying the overhead that results from it is particularly difficult [5].

***Benchmark suites.*** Another crucial dimension of quantifying the GC impact on application performance is represented by what benchmarks are available for this type of research. The established benchmark suites, *i.e.*, DaCapo [3] and Renaissance [24], are considered representative of real-world applications. However, benchmarks belonging to the DaCapo suite have very small memory footprints, posing no challenge for today's GCs that are able to handle terabyte-sized heaps. The Renaissance benchmarks stress the GC more than those of DaCapo, but they do not represent long-running applications with a large live-set – which are more likely to be affected by the GC. If the GC spends very little time processing objects concurrently with the application execution, then regardless of how much it affects the application performance, the overall overhead remains negligible.

### 3.2 Methodology

To overcome the challenges presented in Section 3.1, we propose the following steps: first, we eliminate the GC cycles that are not absolutely necessary for a correct and complete execution (*e.g.*, warm-up GCs and proactive GCs in ZGC, as described in Section 2). Second, we select applications that consist of multiple iterations, each performing the same operations. Then, depending on the application characteristics we apply one of the following strategies:

- For applications for which we can deterministically control the GC activity (Section 4), we trigger explicit GCs at the beginning of some of the iterations. We use these explicit triggers to distinguish between iterations with and without GC activity. We consider all iterations that fall between the start and the end of the GC cycle as iterations with GC activity.
- For applications for which we cannot control the GC behavior (Section 5), we expect to see iterations with GC activity and iterations without any GC involvement. This allows for direct comparison between the two iteration types (with and without GC), based on metrics like execution time and the number of LLC misses encountered by the mutators.

Intuitively, we expect that, regardless of our ability to control GC activity, the application threads will still experience more LLC misses and higher execution time for iterations with GC than for iterations without GC. However, it is also important to note that for larger-scale benchmarks where collections are naturally triggered by memory pressure (as opposed to being programmatically triggered), the iterations cannot be strictly split into two groups. Rather, the GC presence covers a variable number of iterations, some of them completely falling under GC execution, while others only partially overlapping with GC activity. For simplicity, we refer in the rest of the paper to an iteration as "iteration with GC (activity)" if it falls into one of the following situations:

- GC starts before the end of the iteration (regardless of whether the GC ends in the same iteration or not).
- GC ends after the beginning of the iteration (regardless of when it started).
- GC is fully contained in the respective iteration (*i.e.*, GC starts after the iteration and ends before it).
- The iteration is fully contained in the GC cycle (*i.e.*, GC cycle starts before the iteration and ends after it).

To account for the duration variations across iterations in the real-world environment, we present results in the shape of histograms illustrating the distribution of iteration duration. We expect this distribution to be centered in two clusters, one for the duration of iterations with GC activity and one for the duration of iterations with no GCs (*i.e.*, a bimodal distribution). We compute the distance between the two clusters as the difference between the mean duration for each cluster (delta-mean), as shown in Figure 3, and use this difference to quantify the GC impact on cache locality.

## 4 Cache-Sensitive Micro-Benchmark

Given the complexity of the JVM machinery and the fundamental difficulty in following GC activity, we first develop a synthetic micro-benchmark that provides a clear, deterministic, and easily controllable environment. We use this micro-benchmark with the goal of defining a baseline to help us understand more complex scenarios.

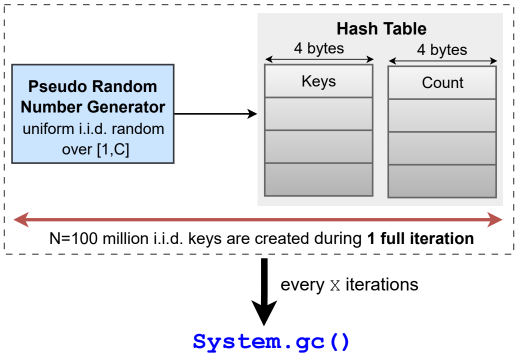

Hash Table
4 bytes
4 bytes
Keys
Count
Pseudo Random Number Generator
uniform i.i.d. random over [1,C]
N=100 million i.i.d. keys are created during 1 full iteration
every X iterations
System.gc()

**Figure 4.** Count-group-by micro-benchmark

### 4.1 Controlled Environment

***Micro-benchmark description and operation.*** We devise a synthetic micro-benchmark that computes a count-group-by aggregate. We implement the group-by aggregation as a simple hash table using open-addressing with linear probing for collision resolution. Keys and counts are 32-bit integers. Furthermore, we choose the cardinality of the set of uniformly randomly generated keys such that the hash table is fully LLC-resident, *i.e.*, both keys and counts.

Figure 4 shows the components and functionality of the micro-benchmark. The micro-benchmark consists of one or more iterations during which $N = 100$ million uniformly independent and identically distributed (i.i.d.) keys in the interval $[1, C]$ are created. The number of distinct keys are counted in the hash table. We size it to hold $C$ entries. We use Java's 32-bit hash function, implemented as the identity function for `Integer`. As a result, there are no bucket collisions and consequently no linear probes in the hash table. To artificially increase the cache sensitivity of the benchmark we use two separate `int` arrays for keys and counts. A thread responsible for triggering a GC runs in parallel with the main thread. The latter controls when the GC should start by notifying and waking up the GC-triggering thread. The explicit GCs should be sufficiently far apart to allow the application to return to a baseline with respect to cache effects. At the same time, the iteration duration should be long enough, for the GC impact on the cache to be visible.

Although simple and artificial, this single-threaded micro-benchmark provides the necessary infrastructure to fully control the activity of a concurrent GC and deterministically measure its impact on the application's cache locality. In other words, this micro-benchmark reflects the overhead of concurrent collectors on cache-sensitive applications since the hash table is fully LLC resident by design.

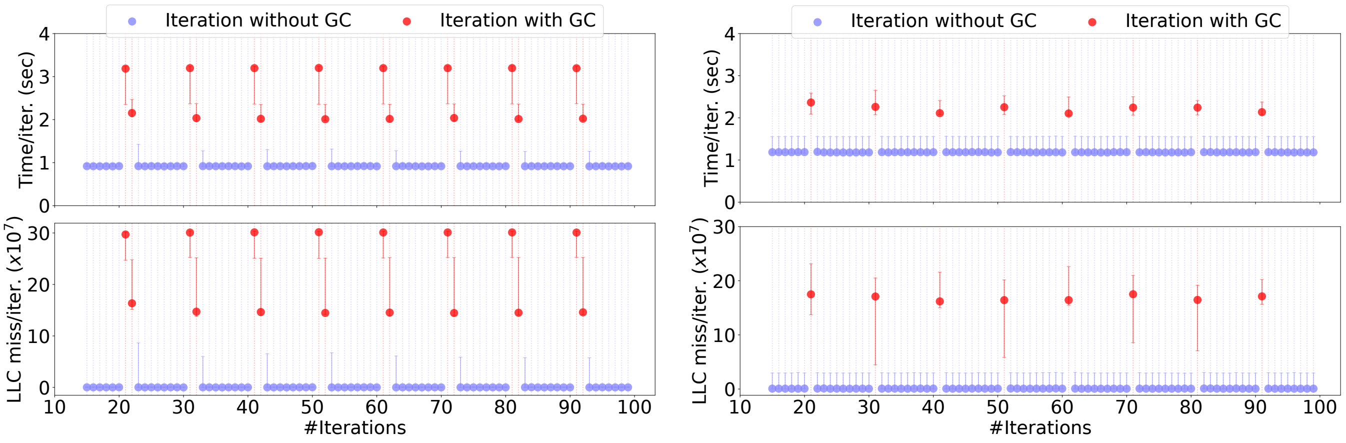
***Controlling the GC.*** The small working-set size of the micro-benchmark creates a negligible amount of work for the garbage collector. As the GC effort depends on the amount of references in the live-set that it needs to mark and potentially on the set of objects selected for relocation (as explained in Section 2.2), we artificially increase the live-set of the micro-benchmark. Doing this in a manner that does not change in any way the targeted functionality of the micro-benchmark and that also keeps the same deterministic behavior when a GC is triggered, is particularly challenging. To address both these issues, we load a large and complex data structure in memory before launching the workload and avoid operating on it afterwards. This structure will represent a constant body of live objects that need to be visited and marked in every GC cycle. Importantly, the set of live objects traversed by the GC and the working-set of the application only intersect on the headers of the two array objects that form the hash table. This means that the number of referenced objects that are brought into the cache by the GC and can be used by the application is negligible. In order to have sufficiently many references to mark, we use a directed random graph, in which each node has an array of references to its randomly chosen neighbors. By modifying the size of the graph, we can easily adjust the amount of effort spent by the GC during the *marking phase*, independently of the working-set size of the application, *i.e.*, the size of the hash table.

The downside of having a fixed graph with objects that are never deallocated is that the *relocation phase* is not exercised. One way to introduce relocation effort would be to delete some of the graph nodes and let the GC relocate live objects. However, in this scenario it is difficult to maintain a deterministic behavior across runs for several reasons, such as: traversing the graph to delete nodes would contribute to the cache pollution, as it is not part of our carefully designed application working-set; consistently having a similar number of relocated objects is a difficult problem, as it depends on the collector's heuristics; having a randomly-generated graph makes the problem of choosing which nodes to delete a challenging one. In conclusion, in this experiment, we consider only the impact of the marking phase on application locality, which gives us a lower bound over the effect of a complete GC cycle (*i.e.*, including relocation). We leave the task of redesigning the micro-benchmark such that it creates a constant amount of relocation effort for future work.

### 4.2 Experimental Setup

The machine we use for experiments is an 8-core Intel Xeon W-1270 @3.4 GHz with a 64 GiB DRAM and 16 MiB of LLC, running Ubuntu 20.04 on a 5.14.0 kernel. We disable the Intel Turbo feature and enable the `performance governor` on all cores in order to minimize noise. We run our experiments on the HotSpot JVM in OpenJDK, using the latest LTS version 17.0.5. As an adequate representative for concurrent GCs, we focus on ZGC. We reproduce our results with Shenandoah.

We pin the main Java thread on one core and the GC-triggering thread on a separate core. The latter is configured to call the `System.gc()` function every 10 iterations, starting from iteration 11. The hash table size is 14.5 MiB. The static



**(a)** ZGC

**(b)** Shenandoah

**Figure 5.** Impact of ZGC (left) and Shenandoah (right) on application performance. The data points represent the median with min/max error-bars over 11 runs.

graph that controls the amount of marking work for the GC is created at the beginning of the execution. We generate a graph having $5 \times 10^6$ nodes and $5 \times 10^8$ edges, which results in a live-set size of around 5 GiB or 32 % of the heap size.

We configure ZGC to use 2 MiB large pages to avoid performance degradation due to TLB misses. In addition, we disable proactive GC collections to have better control over the GC activity, as explained in Section 3.2. We do not explicitly set the heap size. The JVM allocates by default a 15.6 GiB heap. For low-level measurements we use the `perf_event_open` system call in the Linux kernel to access the raw performance-monitoring counters. We measure the number of LLC misses (`LONGEST_LAT_CACHE.MISS` [9]: `event=0x2E, umask=0x41`) and that of retired instructions (`INST_RETIRED.ANY_P` [9]: `event=0xC0, umask=0x0`) for the application thread. The GC timing information is provided by the built-in JVM GC logs that can be enabled with the `-Xlog:gc*` option. For Shenandoah, we force explicit GCs to run as concurrent instead of the typical STW full GC implementation. ZGC executes explicit GCs concurrently by default.

### 4.3 Evaluation

Figures 5a and 5b show the impact of the GC on the application performance and the extent to which this overhead is correlated with cache pollution for ZGC and Shenandoah, respectively. The x-axis of both figures represents the iteration number. The micro-benchmark was run for 100 iterations, out of which we drop the first 15 in order to avoid performance artifacts from the start-up of the JVM. The y-axis of each figure in the first row corresponds to the duration of each iteration. The y-axis of each figure in the second row presents the absolute number of LLC misses encountered within an iteration. Moreover, each bullet (blue or red) per sub-plot corresponds to the median across 11 different runs; the plots also present min/max error bars per data point to show the variation across runs. We show the measurements corresponding to iterations that contain GC activity and those that do not in different colors in order to better outline the impact of the GC on some iterations.

Figure 5a reveals a correlation between GC cycles, LLC misses and iteration duration for ZGC. Note that the number of LLC misses for the iterations with no GC activity is in a magnitude of $10^5 - 10^6$, appearing close to zero at the shown scale. The low number of misses is due to the hash table being fully LLC resident, as explained in Section 4.1. Both LLC misses and iteration duration experience spikes for every GC collection. More precisely, we note a 3 × increase in iteration duration during concurrent GC cycles. This corresponds to a 1700-fold increase in LLC misses when the concurrent GC is active. The result supports the hypothesis that the concurrent GC not only impacts the application performance but also causes cache pollution, which is observable through the increase of LLC misses. We also note that the ZGC collections encompass 2 iterations of the micro-benchmark each time, affecting the iteration duration and LLC misses in both of them. Finally, an interesting observation is that only the iterations with GC activity and the one immediately following the GC cycle exhibit very high variation across multiple runs, both with respect to execution time and to LLC misses. This behavior suggests that the GC effects on application performance extend beyond the end of the GC cycle.

We repeat the same experiment with the Shenandoah collector, with the goal of verifying whether our finding holds for another mostly-concurrent collector of the same JVM. Figure 5b shows the impact of Shenandoah on LLC misses and iteration duration, similar to Figure 5a. We observe that Shenandoah-enabled runs are characterized by a slightly higher iteration duration in general. We confirm the same trend as before during GC activity: correlated spikes in both LLC misses and iteration duration, albeit less prominent than for ZGC. In the case of Shenandoah, each collection encompasses only one iteration and results in 2 × longer iterations and 225 × more misses in the LLC.

# 5 Typical Java Workloads

To validate the findings in Section 4 on well-established applications, we turn towards the Renaissance benchmark suite as a reasonable proxy for real-world applications.

## 5.1 Benchmark Suite Selection and Setup

We study Renaissance because it is a recent and actively maintained benchmark suite that contains a wide variety of workloads. It includes multiple categories of modern applications made for the JVM (*e.g.*, Apache Spark [11] and actor-based applications). At the same time, the workloads illustrate a diversity of programming paradigms, including: concurrent and parallel applications, functional and object-oriented programming. It is especially developed for testing JIT compilers, GCs, and other JVM components.

Renaissance benchmarks use a harness that handles measurements, policies, and benchmark execution. We use the term *iteration* to denote one individual benchmark execution and measurement.[3] Each benchmark run has at least one iteration. The default number of iterations varies between 10 and 90 across the Renaissance suite. Unless otherwise noted, we follow the methodology used for the officially published Renaissance performance results [1]: we run each benchmark for 10 minutes and discard the measurements for the first half of the iterations. We execute the entire Renaissance suite with the default set of inputs for each benchmark (*i.e.*, input files, structure sizes, etc.).

The Renaissance benchmarks present important differences from a GC perspective as well, allocating between $10^7$ and $10^9$ objects [25] with different allocation rates. By default, the Renaissance harness triggers explicit GCs between iterations of the benchmarks, in order to have each iteration start with the same memory state as all others. Since we want to measure and compare the iteration duration when a GC takes place and in the absence of GC, we disable the explicit GCs between iterations. This allows the GC to only collect when necessary according to its internal heuristics and, thus, increase the probability of having both behaviors of interest in the same run, *i.e.*, iterations with and without GC. Again, for ZGC, we disable proactive GC collections and discard the warm-up GC results at the beginning of every benchmark run. We also set the heap size to 28 GiB. However, even with these measures in place, there are benchmarks with such a high allocation rate that multiple GCs are requested per iteration or the collections span all iterations in a run; and, conversely, benchmarks for which GCs are such rare events that the results are not statistically relevant.

Similarly to the experiments in Section 4, we measure the iteration duration and the number of LLC misses and aim to correlate them. The main challenge compared to the micro-benchmark scenario is that the Renaissance iteration duration per benchmark varies significantly across a run, regardless of GC activity. Therefore it is more difficult to gauge the impact of GC and to differentiate it from other sources of noise. Another difficulty comes from the limited control one has over the GC and application behavior when running real benchmarks, as opposed to a synthetic micro-benchmark created with a specific purpose in mind. Finally, in these experiments object relocation is more prevalent than in the micro-benchmark scenario, also contributing to the GC effects on application performance.

## 5.2 Evaluation

Figure 6 illustrates the iteration duration for iterations with GC activity and iterations without any GC activity for all Renaissance workloads, following the methodology defined in Section 3.2. The iteration duration is represented as a histogram for a clear visualization of the variation across all iterations in a run. We say that an iteration has *GC activity* if there is a partial or total overlap between a GC cycle and said iteration, as defined in Section 3.2.

Figure 6 contains 23 subplots, one per Renaissance benchmark. We note the difference between the means of the two distributions, relative to the mean duration of iterations without GC activity in each subgraph along with the fraction of iterations that contain GC activity. The x-axis represents the iteration duration in seconds. The y-axis presents the number of iterations in logarithmic scale. The subplots are sorted in the decreasing order of the mean difference.

The first observation is that there is significant variability across the benchmarks with respect to GC activity. On the one hand, we have `akka-uct` and `naive-bayes` with 100 % and, respectively, 98 % of iterations exhibiting GC activity. On the other hand, there are benchmarks where less than 2 % of iterations show any GC activity, *e.g.*, `scala-kmeans`, `dotty`, `rx-scrabble`. Any conclusions from the latter benchmarks need to be drawn with caution, as they contain too few samples in one of the two distributions.

The main takeaway is that, overall, the GC has no major impact on iteration duration for the Renaissance benchmarks. Extrapolating from our micro-benchmark results, we expected to observe a pronounced bimodal distribution, indicating that the iterations overlapping with a concurrent GC take on average longer than those without GC interference. Several benchmarks, such as `scrabble`, `page-rank` or `fj-kmeans` do exhibit this trend. However, for other benchmarks, the two distributions fully overlap ( `future-genetic`) or even show an opposite trend to what we expected (as in the case of the two `finagle` benchmarks). We observe a

[3] Not to be mistaken for the iterations that some of the workloads may execute internally for their own computation.

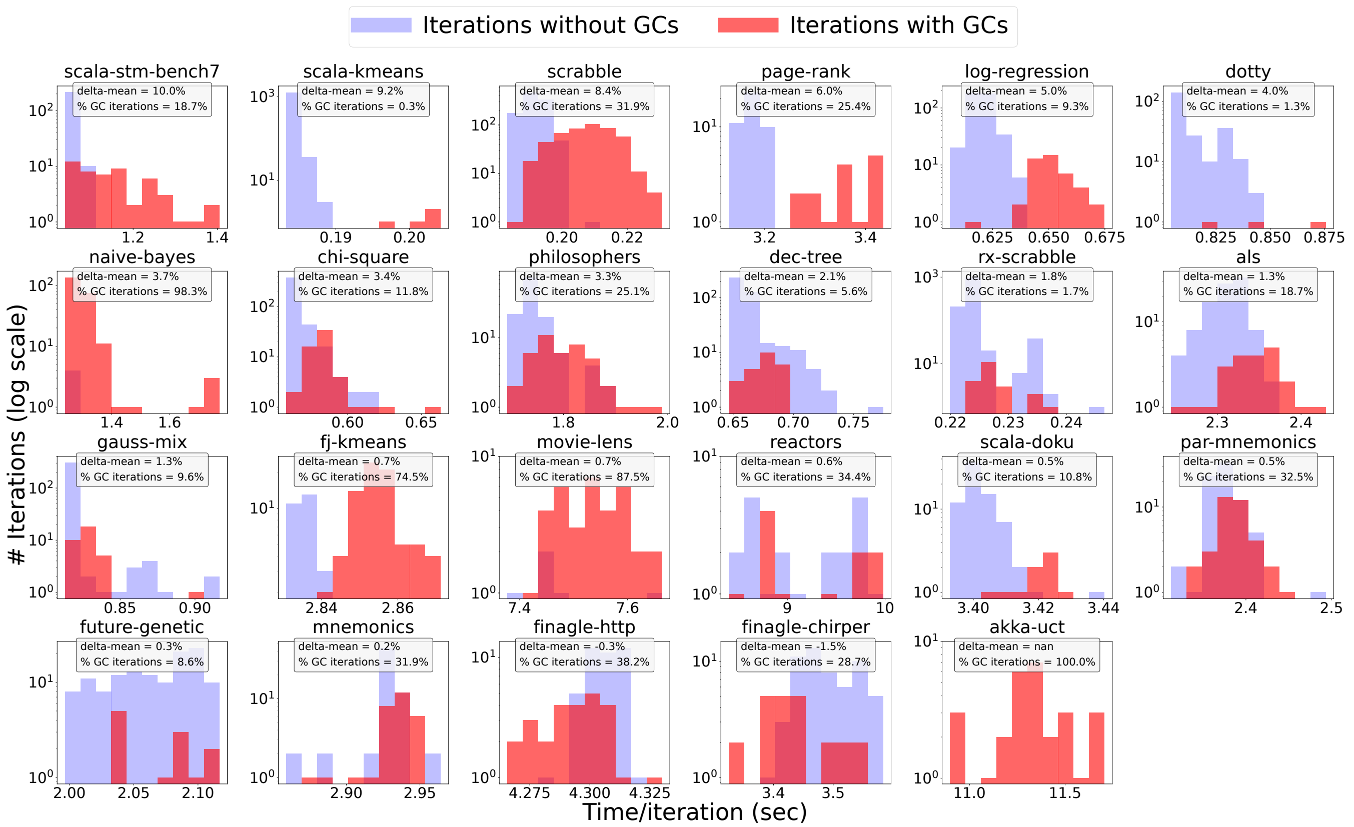


**Figure 6.** Impact of ZGC activity on the iteration duration of the Renaissance benchmarks (one subplot per benchmark). The x-axis represents the iteration duration and the y-axis, the number of iterations (log scale).

maximum difference of 10 % between the two iteration clusters, for `scala-stm-bench7` (with an average of 2.8 % across benchmarks). The GC impact is, thus, significantly weaker than for our micro-benchmark experiments in Section 4.

We illustrate in the same manner the distributions of LLC misses across iterations for the two cases, with and without concurrent GC activity. Figure 7 shows the results. The subplots are in the same order as in Figure 6 to facilitate the comparison between results with the two metrics. When looking at the measured LLC misses, we observe a similar trend to that of execution time per iteration. Only a few benchmarks (*e.g.*, `scala-kmeans` and `page-rank`) have a distribution with two clearly visible clusters. For the other benchmarks the two distributions partially overlap. The effect of GC activity is slightly more pronounced for LLC misses than in the case of iteration duration, with a delta-mean of 13.4 % on average across benchmarks, going up to 56.3 % (`dotty`).

We repeat the experiments with the Shenandoah collector. Because of space limitations, we do not include the figures here. The results confirm the observations made with ZGC, albeit with an even more diminished GC effect on the iteration duration: the delta-mean reaches at most 5.7 % (`scrabble`). However, from an LLC miss standpoint, three benchmarks present a delta-mean of over 70 % between the two distributions, indicating increased cache pollution: `chi-square`, `dec-tree` and `gauss-mix`.

## 6 GC Effect on Cache and Performance

This section provides further insight into the correlation between GC-induced cache pollution and application performance, through a statistical analysis of the observed results and a study of the working-set of Renaissance applications.

### 6.1 Statistical Analysis

To have a better understanding of the impact of GC activity on application performance and whether this correlates with cache pollution, we examine the data from a statistical perspective. We first consider how the iteration duration and, respectively, the number of LLC misses independently correlate with GC activity. Table 1 shows the results for the Renaissance suite. Note that we use the exact same data presented in Section 5 and illustrated in Figures 6 and 7. We exclude the `akka-uct` benchmark from this analysis, as all its iterations include GC activity and thus no differentiation can be made between iterations with and without GC impact.

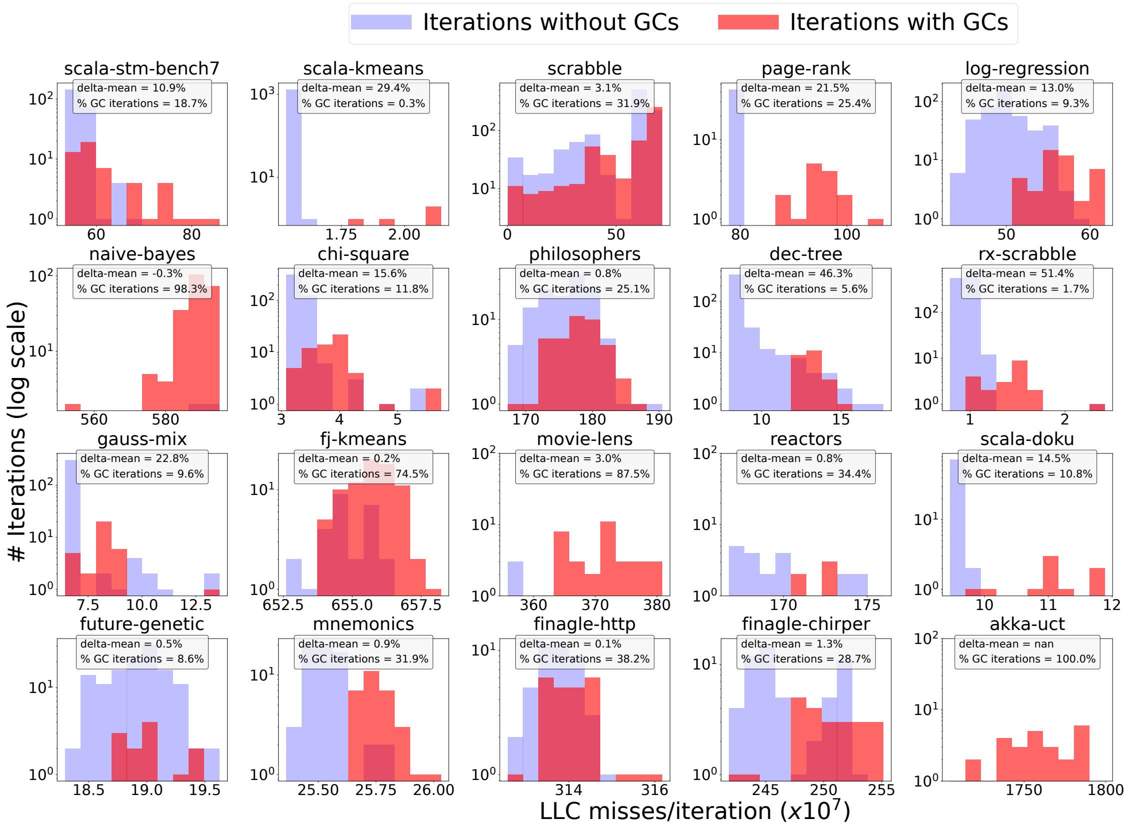


**Figure 7.** Impact of ZGC activity on LLC misses for the Renaissance benchmarks (one subplot per benchmark). The x-axis represents the absolute number of misses in the LLC and the y-axis, the number of iterations (log scale).

We consider two metrics: the *Levene homogeneity test* [18] for the two distributions, and the *point biserial correlation* [17]. The former examines whether the given samples come from populations with equal variances. In order to measure the correlation between GC activity (binary variable, GC present or not) and iteration duration or, respectively, LLC miss count (continuous variables) we use the point biserial correlation. The correlation results are considered valid if the continuous variable has equal variances for each category of the binary variable, as tested with the Levene test. The computed coefficient varies between -1 and 1, where 0 means no correlation. A value significantly different from zero implies a significant difference in means between the two distributions. Therefore, the results in Table 1 compact and consolidate the observations in Section 5. For each of the two target variables, *i.e.*, iteration duration and misses in the LLC, we indicate with a check mark (✓) if the distributions with and without GC have equal variances and with an × otherwise. We use the same notation for marking whether a correlation exists between GC activity and the respective variable. In addition, we also list the computed coefficient value for the point biserial correlation and the corresponding $p$-value, based on which we consider the correlation between the target variable and GC activity statistically significant. A $p$-value under 0.05 indicates a statistically significant correlation.

We find that the considered distributions for the iteration duration are not homogeneous for half of the Renaissance benchmarks and in the case of LLC misses for 14 out of 22 benchmarks. We further observe that for 5 benchmarks there is no statistically relevant correlation between GC activity and iteration duration. The same can be concluded about the correlation between GC activity and LLC misses in the case of 4 benchmarks. There are 3 benchmarks which fail both correlations and the homogeneity tests: `future-genetic`, `naive-bayes`, and `reactors`. Finally, we identify 5 benchmarks that have homogeneous distributions and pass both correlation tests (highlighted in Table 1). We focus on these benchmarks for a more detailed analysis.

Figure 8 shows the correlation between LLC misses (y-axis) and iteration duration (x-axis) for the selected subset of 5 benchmarks. The scatter plots have one data point per iteration. The iterations are further split into iterations with GC activity or without GC activity, as defined in Section 3.2. We observe that the iterations in the two categories tend to cluster together, with the no-GC iterations closer to the

**Table 1.** Statistical analysis of the GC impact on execution time (left) and LLC misses (right), showing: Levene's homogeneity test ("Equal Var.") and point biserial correlation ("Corr.") w.r.t. GC activity (correlation coefficient and $p$-value).

| **Benchmark** | **Iteration Duration** | | | | **LLC Misses** | | | |
|---|---|---|---|---|---|---|---|---|
| | Equal Var. | Corr. | Corr. Coeff. | $p$-value | Equal Var. | Corr. | Corr. Coeff. | $p$-value |
| **als** | ✓ | ✓ | 0.4 | $4.11 \times 10^{-6}$ | × | ✓ | 0.66 | $7.53 \times 10^{-17}$ |
| **chi-square** | ✓ | ✓ | 0.6 | $4.94 \times 10^{-51}$ | ✓ | ✓ | 0.59 | $5.31 \times 10^{-48}$ |
| **dec-tree** | × | ✓ | 0.22 | $4.91 \times 10^{-6}$ | × | ✓ | 0.62 | $9.76 \times 10^{-48}$ |
| **dotty** | × | ✓ | 0.33 | $4.52 \times 10^{-7}$ | ✓ | ✓ | 0.76 | $6.50 \times 10^{-44}$ |
| **finagle-chirper** | × | ✓ | -0.41 | $1.79 \times 10^{-4}$ | × | ✓ | 0.4 | $2.37 \times 10^{-4}$ |
| **finagle-http** | ✓ | ✓ | -0.5 | $1.73 \times 10^{-5}$ | × | ✓ | 0.33 | $5.38 \times 10^{-3}$ |
| **fj-kmeans** | ✓ | ✓ | 0.83 | $3.98 \times 10^{-28}$ | × | ✓ | 0.45 | $1.34 \times 10^{-6}$ |
| **future-genetic** | × | × | 0.05 | $5.65 \times 10^{-1}$ | × | × | 0.11 | $2.02 \times 10^{-1}$ |
| **gauss-mix** | × | ✓ | 0.27 | $3.10 \times 10^{-7}$ | ✓ | ✓ | 0.48 | $5.56 \times 10^{-22}$ |
| **log-regression** | ✓ | ✓ | 0.85 | $4.00 \times 10^{-135}$ | × | ✓ | 0.57 | $2.57 \times 10^{-41}$ |
| **mnemonics** | × | × | 0.14 | $1.75 \times 10^{-1}$ | × | ✓ | 0.78 | $2.01 \times 10^{-20}$ |
| **movie-lens** | × | × | 0.26 | $1.01 \times 10^{-1}$ | × | ✓ | 0.55 | $2.32 \times 10^{-4}$ |
| **naive-bayes** | × | × | 0.09 | $1.77 \times 10^{-1}$ | × | × | -0.04 | $5.03 \times 10^{-1}$ |
| **page-rank** | ✓ | ✓ | 0.93 | $1.21 \times 10^{-26}$ | ✓ | ✓ | 0.95 | $4.48 \times 10^{-30}$ |
| **par-mnemonics** | × | ✓ | 0.23 | $1.60 \times 10^{-2}$ | × | ✓ | 0.21 | $2.80 \times 10^{-2}$ |
| **philosophers** | ✓ | ✓ | 0.52 | $4.15 \times 10^{-13}$ | × | ✓ | 0.16 | $3.69 \times 10^{-2}$ |
| **reactors** | × | × | 0.05 | $7.66 \times 10^{-1}$ | × | × | 0.25 | $1.72 \times 10^{-1}$ |
| **rx-scrabble** | ✓ | ✓ | 0.21 | $2.54 \times 10^{-14}$ | ✓ | ✓ | 0.59 | $9.85 \times 10^{-119}$ |
| **scala-doku** | × | ✓ | 0.62 | $3.77 \times 10^{-10}$ | ✓ | ✓ | 0.88 | $1.71 \times 10^{-28}$ |
| **scala-kmeans** | ✓ | ✓ | 0.87 | 0.00 | ✓ | ✓ | 0.94 | 0.00 |
| **scala-stm-bench7** | ✓ | ✓ | 0.72 | $1.38 \times 10^{-44}$ | ✓ | ✓ | 0.52 | $3.96 \times 10^{-20}$ |
| **scrabble** | ✓ | ✓ | 0.81 | 0.00 | × | × | 0.05 | $7.36 \times 10^{-2}$ |

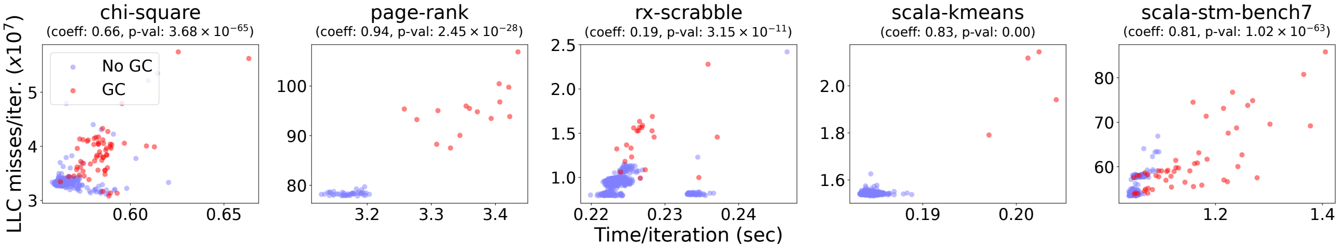


**Figure 8.** Correlation between LLC misses and iteration duration for 5 Renaissance benchmarks. The Pearson correlation coefficient ("coeff." in the figure) and $p$-value are listed for each benchmark.

bottom-left side of the plots and the GC iterations closer to the upper-right quadrant. This trend suggests that GC activity correlates with both higher execution time and higher LLC misses. Page-rank illustrates this observation best. For completeness, the plot also shows the *Pearson correlation* statistics computed for each benchmark. The coefficient has a value between -1 and 1; a value of 0 indicates no correlation. Notably, all 5 benchmarks pass the test, according to a $p$-value smaller than the reference value of 0.05.

In conclusion, we find that a subset of the Renaissance benchmarks show, with reasonable confidence, a correlation between GC-induced cache pollution and negative effects on application performance. However, even for these benchmarks, the measured GC impact is very small: as already evaluated in Section 5 we are looking at an overhead of at most 10 %. For the other 17 Renaissance benchmarks, no correlation between LLC misses and application performance can be inferred. We suspect that the inconsistency in results between the synthetic micro-benchmark (Section 4) and the Renaissance benchmarks, representative of real-world workloads, comes from the fact that typical Java applications are either not LLC-friendly or have very small working-sets. To validate our hypothesis, we analyze the sensitivity of the Renaissance benchmarks to the LLC size in Section 6.2.

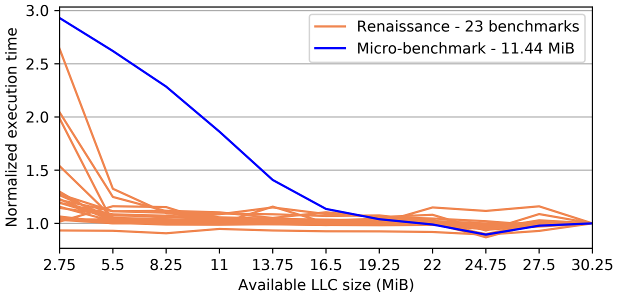


**Figure 9.** Sensitivity of Renaissance benchmarks and micro-benchmark to LLC capacity. The execution time is normalized to the baseline (using the full 30.25 MiB LLC).

### 6.2 LLC Sensitivity Analysis

This section quantifies the sensitivity of the Renaissance benchmarks and micro-benchmark (Section 4) to the LLC size. To do so, we restrict the available LLC capacity of specific processes using the Cache Allocation Technology (CAT) from the Intel Resource Director Technology [9].

We conduct this set of experiments on a different machine from the one described in Section 4.2 since only specific Intel server processors have Intel CAT enabled. We use a dual-socket system with a 22-core Intel Xeon Gold 6238T and 30.25 MiB of LLC per socket, running Ubuntu 20.04 on a 5.15.0 Linux kernel. The processor has 11 slices of LLC which can be enabled or disabled independently on demand. Thus, we can split the LLC at a granularity of 2.75 MiB. We control the available cache size with the `resctrl` Linux interface. We limit the execution of the Renaissance benchmarks and that of the cache-sensitive micro-benchmark to a single socket and its local memory (NUMA node) using `numactl`, and we control the LLC size available for all the cores of the socket.

Figure 9 presents the results of the LLC sensitivity analysis for the Renaissance benchmarks and the micro-benchmark. The x-axis shows the available LLC sizes. The y-axis presents the execution time of the benchmarks for specific LLC sizes normalized to the execution time when the full 30.25 MiB LLC is used. The Renaissance benchmarks are executed as described in Section 5.1, but with the default number of iterations and discarding the first half of the execution. As we study the normal behavior of the benchmarks themselves, we use the default configuration of Renaissance benchmark suite in which explicit GCs are triggered before each iteration. Because this increases the number of deterministic GCs, the variability of the iteration durations is also reduced. Furthermore, we run each experiment three times and take the mean of the results for each workload. For the baseline execution, *i.e.*, with the full LLC, we take the result with the shortest execution time of the three runs. For the cache-sensitive micro-benchmark, we use a 11.44 MiB hash table, which is fully LLC-resident in the baseline run. Similarly to Section 5.1, we use 2 MiB huge pages. Figure 9 shows all Renaissance benchmarks in the same color (orange), as we want to emphasize the trends related to the execution of these benchmarks under reduced LLC size, rather than the individual trend for each benchmark.

We see in Figure 9 that for the cache-sensitive micro-benchmark (in blue) the performance scaling with varying LLC sizes is close to our expectations. For an LLC size of 11.44 MiB and higher, we expected the performance of the micro-benchmark to be stable as its working-set can fit in the LLC. However, we observe that for LLC sizes from 13.75 MiB up to 19.25 MiB the execution time is still above the baseline execution time. We associate the performance overheads at those LLC sizes with the memory overheads of the JVM itself and the system threads running concurrently with our micro-benchmark. When the available LLC capacity further decreases below the hash table size (*i.e.*, the size of the micro-benchmark's working-set), we see the expected behavior. The execution time increases linearly with the miss rate in the hash table due to the fact that all accesses are uniformly random and only a smaller part of the hash table fits in LLC.

Figure 9 also reveals that all Renaissance benchmarks experience a minimal slow-down, *i.e.*, less than 15 %, when we reduce the available LLC size by 8/11, from 30.25 MiB to 8.25 MiB. Even the benchmarks identified in Section 6.1 as having a correlation between GC-induced cache pollution and the negative effect on application performance, show less than 12 % performance degradation. When we reduce the available LLC size from 8.25 MiB to 2.75 MiB, some of the benchmarks (`scrabble`, `fj-kmeans`, `finagle-http` and `finagle-chirper`) experience a considerable performance drop (higher than 50 %). This effect can be attributed to having hit a working-set size cliff (*e.g.*, working-set of instructions or data that would have been LLC-resident otherwise).

The cache sensitivity analysis indicates that the working-set size of the Renaissance benchmarks is either larger than the LLC capacity or smaller than 2.75 MiB, which is the minimum we are able to measure with Intel CAT. We would otherwise expect to see a steep performance degradation when the available LLC capacity is close to the working-set size, as demonstrated by the micro-benchmark. This result is well aligned with our previous observations. Even if we restrict the LLC for the Renaissance benchmarks to just a fraction of its full capacity, by artificially reducing its size, there is no noticeable difference in performance. A concurrent GC polluting the LLC would have the same effect. Therefore, we can conclude that GC-induced cache pollution can only have a limited impact on the performance of Renaissance benchmarks, which, indeed, is what we have observed.

## 7 Related Work

Garbage collection is a rich source of scientific work. While generally addressing the most evident problems of automatic memory management, such as pause times or memory footprint, there is little literature aimed at analyzing the impact of concurrent GC on applications' cache locality.

Among the earliest efforts in this direction, Blackburn *et al.* [2] aim to illustrate the costs of GC and its impact on application performance and locality. However, as GC algorithms evolved over the years, the observations extracted in this work are not directly applicable to today's ultra-low latency concurrent GCs. Similarly, Ha *et al.* [14] perform a microarchitectural characterization of GCs, together with other JVM components, such as the JIT compiler. In order to obtain accurate results, they limit their evaluation to non-concurrent GCs running in a uniprocessor environment. By contrast, our work overcomes this limitation and adapts our tools and methodology for quantifying the impact of concurrent GCs on application performance and correlating it with microarchitectural events.

Cao *et al.* [6] attempt to measure the overhead introduced by concurrent GCs as well, among other JVM services. They highlight two main challenges in evaluating concurrent collectors: (i) the difficulty to obtain data per thread for some metrics (*e.g.*, energy), and (ii) the intricacy of the GC housekeeping work needed within the application execution, as opposed to the case of STW collectors. The authors propose separately measuring the GC and application work in an STW setting and then deducing the concurrent GC overhead from the difference between a concurrent run and a STW run. They use Jikes Research VM for evaluation. We argue that it is not straightforward to apply this approach to a production VM, such as HotSpot, as the higher complexity of the VM implies more effort to obtain the same GC algorithm implemented both as stop-the-world and concurrent.

Another recent study [5] recognizes the complexity of measuring the overhead of concurrent GCs. The authors propose a methodology that employs a lower bound for the cost of GC. They argue that absolute GC overheads are not clear to this day and that typical GC evaluation results can be misleading. Their study compares the collectors in OpenJDK 17 and finds that the newer GCs, *i.e.*, ZGC and Shenandoah, are more costly than the default GC, G1. This work is orthogonal to our study, which focuses on the cache-level effects. However, it strongly supports our motivation for quantifying the extent to which modern concurrent GCs affect applications' locality in the cache hierarchy.

The work of Sareen and Blackburn [27] explores the cost differences between manual and automatic memory management. In their paper, the authors also propose a methodology for quantifying the impact of GC-induced cache pollution on application performance, but they use stop-the-world home-brewed GCs. In contrast, our analysis extends to concurrent collectors and their impact on application locality and aims to show the effects of widely-used production GCs on application performance.

Yang *et al.* [31] also conduct an investigation focused on GC impact on application performance from a cache locality perspective. They evaluate ZGC in this context and propose a new heap organisation that improves application locality. However, the authors of [31] measure cache-level events for the whole JVM runs, instead of the application thread alone, warning that these results should be taken "with a grain of salt". Our extensive and targeted analysis complements this work with precise insights regarding the GC effect on application data locality in the caches.

## 8 Conclusions and Future Work

Automatic memory management is a double-edged sword: on the one hand it offers important run-time correctness guarantees, eliminating any risk of dangling pointers, memory leaks, or use-after-free hazards. On the other hand, a relatively high price has to be paid for the provided memory management. Modern concurrent garbage collectors solved the original problem of long application pauses introduced by old-style STW GCs. However, the changes needed for removing the pauses prompted new sources of overhead. There are several possible components to this overhead: resource sharing, non-application work done by application threads to help the GC, and memory footprint for GC bookkeeping.

To this end, our work aims to shed light on the overhead contributed by one single component, namely the effect of GC on the cache. We experiment with a cache-sensitive micro-benchmark and the Renaissance benchmark suite. On the former, specifically tailored for demonstrating GC-induced cache-pollution, we find a combined effect of a 3× increase in application execution time (as measured per iteration of the micro-benchmark) and a 3-order of magnitude increase in LLC misses. However, the experiments on Renaissance change the perspective. We show that for 17 out of 22 benchmarks, no statistically significant correlation can be found between GC-induced LLC misses and execution time. A further study on the working-set size of Renaissance benchmarks reveals that these workloads lack LLC-sensitivity. As such, we conclude that the impact of concurrent GCs on the application's data locality in the cache is a non-problem in real-world scenarios where applications themselves are oblivious to cache locality.

This work represents just a small step towards more clearly understanding the impact of concurrent GCs on application performance. Our study on cache locality can be further consolidated by quantifying the contributions of the GC and application towards cache pollution at a memory-access level. Another research direction could focus on other individual overhead components of concurrent GCs. A clear breakdown on the relative contribution of each component to the total overhead of concurrent GCs would represent an important advancement towards a more targeted improvement of the Java memory management system.

## References

[1] Renaissance Benchmark Suite: Measurement results. https://github.com/renaissance-benchmarks/measurements Last accessed Feb. 2023.

[2] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and Realities: The Performance Impact of Garbage Collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA) *(SIGMETRICS '04/Performance '04)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1005686.1005693

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. 2006. Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications. *ACM Transactions on Programming Languages and Systems* 28, 5 (Sep 2006), 908–941. https://doi.org/10.1145/1152649.1152652

[5] Zixian Cai, Stephen M. Blackburn, Michael D. Bond, and Martin Maas. 2022. Distilling the Real Cost of Production Garbage Collectors. In *Proceedings of the 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '22)*. IEEE Computer Society, Los Alamitos, CA, USA, 46–57. https://doi.org/10.1109/ISPASS55109.2022.00005

[6] Ting Cao, Stephen M. Blackburn, Tiejun Gao, and Kathryn S. McKinley. 2012. The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '12)*. IEEE Computer Society, USA, 225–236. https://doi.org/10.1109/ISCA.2012.6237020

[7] Maria Carpen-Amarie, Yaroslav Hayduk, Pascal Felber, Christof Fetzer, Gaël Thomas, and Dave Dice. 2017. Towards an Efficient Pauseless Java GC with Selective HTM-Based Access Barriers. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes* (Prague, Czech Republic) *(ManLang '17)*. Association for Computing Machinery, New York, NY, USA, 85–91. https://doi.org/10.1145/3132190.3132208

[8] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. 2006. Profile-Guided Proactive Garbage Collection for Locality Optimization. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 332–340. https://doi.org/10.1145/1133981.1134021

[9] Intel® Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html Last accessed Feb. 2023.

[10] Christine H. Flood, Roman Kennke, Andrew 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) *(PPPJ '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/2972206.2972210

[11] Apache Software Foundation. Apache Spark. https://spark.apache.org Last accessed Apr. 2023.

[12] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. 2007. Lock-Free Parallel and Concurrent Garbage Collection by Mark & Sweep. *Science of computer programming* 64, 3 (Feb 2007), 341–374. https://doi.org/10.1016/j.scico.2006.10.001

[13] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2013. A Study of the Scalability of Stop-the-World Garbage Collectors on Multicores. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 229–240. https://doi.org/10.1145/2451116.2451142

[14] Jungwoo Ha, Magnus Gustafsson, Stephen M. Blackburn, and Kathryn S. McKinley. 2008. Microarchitectural Characterization of Production JVMs and Java Workloads *(IBM CAS Workshop)*.

[15] 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* (Vancouver, BC, Canada) *(OOPSLA '04)*. Association for Computing Machinery, New York, NY, USA, 69–80. https://doi.org/10.1145/1028976.1028983

[16] Balaji Iyengar, Gil Tene, Michael Wolf, and Edward Gehringer. 2012. The Collie: A Wait-Free Compacting Collector. In *Proceedings of the 2012 International Symposium on Memory Management* (Beijing, China) *(ISMM '12)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2258996.2259009

[17] Joseph Lev. 1949. The point biserial coefficient of correlation. *The Annals of Mathematical Statistics* 20, 1 (1949), 125–126.

[18] Howard Levene. 1960. Robust tests for equality of variances. *Contributions to probability and statistics* (1960), 278–292.

[19] Haoyu Li, Mingyu Wu, and Haibo Chen. 2018. Analysis and Optimizations of Java Full Garbage Collection. In *Proceedings of the 9th Asia-Pacific Workshop on Systems* (Jeju Island, Republic of Korea) *(APSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 18, 7 pages. https://doi.org/10.1145/3265723.3265735

[20] Per Lidén. ZGC: What's new in JDK 16. https://malloc.se/blog/zgc-jdk16 Last accessed Feb. 2023.

[21] David Lion, Adrian Chiu, Michael Stumm, and Ding Yuan. 2022. Investigating Managed Language Runtime Performance: Why JavaScript and Python Are 8x and 29x Slower Than C++, yet Java and Go Can Be Faster?. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC '22)*. USENIX Association, Carlsbad, CA, 835–852. https://www.usenix.org/conference/atc22/presentation/lion

[22] Jonas Norlinder, Erik Österlund, and Tobias Wrigstad. 2022. Compressed Forwarding Tables Reconsidered. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 45–63. https://doi.org/10.1145/3546918.3546928

[23] OpenJDK. Epsilon: A No-Op Garbage Collector. https://openjdk.org/jeps/318 Last accessed Feb. 2023.

[24] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, 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* (Phoenix, AZ, USA) *(PLDI '19)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[25] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Wuerthinger, and Walter Binder. On Evaluating the Renaissance Benchmarking Suite: Variety, Performance,

and Complexity. https://doi.org/10.48550/ARXIV.1903.10267

[26] Carl G. Ritson, Tomoharu Ugawa, and Richard E. Jones. 2014. Exploring Garbage Collection with Haswell Hardware Transactional Memory. In *Proceedings of the 2014 International Symposium on Memory Management* (Edinburgh, United Kingdom) *(ISMM '14)*. Association for Computing Machinery, New York, NY, USA, 105–115. https://doi.org/10.1145/2602988.2602992

[27] Kunal Sareen and Stephen M. Blackburn. 2022. Better Understanding the Costs and Benefits of Automatic Memory Management. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 29–44. https://doi.org/10.1145/3546918.3546926

[28] Fridtjof Siebert. 2010. Concurrent, Parallel, Real-Time Garbage-Collection. In *Proceedings of the 2010 International Symposium on Memory Management* (Toronto, Ontario, Canada) *(ISMM '10)*. Association for Computing Machinery, New York, NY, USA, 1–20. https://doi.org/10.1145/1806651.1806654

[29] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the 2011 International Symposium on Memory Management* (San Jose, California, USA) *(ISMM '11)*. Association for Computing Machinery, New York, NY, USA, 79–88. https://doi.org/10.1145/1993478.1993491

[30] Bochen Xu, Eliot Moss, and Stephen M. Blackburn. 2022. Towards a Model Checking Framework for a New Collector Framework. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 128–139. https://doi.org/10.1145/3546918.3546923

[31] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020. Improving Program Locality in the GC Using Hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI '20)*. Association for Computing Machinery, New York, NY, USA, 301–313. https://doi.org/10.1145/3385412.3385977

[32] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Transactions on Programming Languages and Systems* 44, 4, Article 22 (Sep 2022), 34 pages. https://doi.org/10.1145/3538532

Received 2023-03-03; accepted 2023-04-24