Jade: A High-throughput Concurrent Copying Garbage Collector

Mingyu Wu[1], Liang Mao[2], Yude Lin[2], Yifeng Jin[2], Zhe Li[1], Hongtao Lyu[1],
Jiawei Tang[2], Xiaowei Lu[2], Hao Tang[2], Denghui Dong[2], Haibo Chen[1,3], Binyu Zang[1,3]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
[2]Alibaba Group
[3]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

Garbage collection (GC) pauses are a notorious issue threatening the latency of applications. To mitigate this problem, state-of-the-art concurrent copying collectors allow GC threads to run simultaneously with application threads (mutators) in nearly all GC phases. However, the design of concurrent copying collectors does not always lead to low application latency. To this end, this work studies the behaviors of mainstream concurrent copying collectors in OpenJDK and mainly focuses on long application pauses under heavy workloads. By analyzing the design of those collectors, this work uncovers that lengthy pre-reclamation cycles (including GC phases before actual memory release), high GC frequency, and large metadata maintenance overhead are major factors for long pauses. Therefore, this work proposes *Jade*, a concurrent copying collector aiming to achieve both short pauses and high GC efficiency. Compared with existing collectors, *Jade* provides a group-wise collection mechanism to shorten pre-reclamation cycles while controlling GC frequency. It also embraces a generational heap layout and a single-phase algorithm to maximize young GC's throughput. The evaluation results on representative latency-critical applications show that *Jade* can reach sub-millisecond-level pauses even under heavy workloads and significantly improve applications' peak throughput compared with state-of-the-art concurrent collectors.

CCS Concepts: • **Software and its engineering** → **Garbage collection**.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3650087

Keywords: Language Runtime, Garbage Collection

ACM Reference Format:
Mingyu Wu, Liang Mao, Yude Lin, Yifeng Jin, Zhe Li, Hongtao Lyu, Jiawei Tang, Xiaowei Lu, Hao Tang, Denghui Dong, Haibo Chen, Binyu Zang. 2024. Jade: A High-throughput Concurrent Copying Garbage Collector. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627703.3650087

1 Introduction

Garbage collectors are one of the most critical modules in managed runtimes like Java Virtual Machine (JVM), JavaScript V8, and Go runtime. By automatically detecting dead objects (or *garbage*) and reclaiming memory resources, collectors remove the burden of manual memory management but also induce performance overhead. Pauses are a notorious issue introduced by collectors: to perform garbage collection (GC), language runtimes may require application threads (namely *mutators*) to stop running. Collectors pausing mutators during GC are usually called *stop-the-world (STW) collectors*. As the memory demand of applications increases, the duration of STW pauses becomes longer since GC threads need to scan and collect a larger heap, which affects application performance, especially for latency-critical ones. To this end, recent collectors are designed to reach short and controllable pause times regardless of the heap size [8, 10, 15, 18, 25, 27, 28, 33]. In those *concurrent copying collectors*, GC threads are running simultaneously with mutators to find live objects and copy them to new addresses, which can reach nearly pauseless collections.

However, short pauses come at a cost. Prior work [7, 39, 41] has shown that the design of low-pause concurrent copying collectors does not always translate to short latency for applications due to their overhead like costly read/write barriers (code instrumentation), poor GC efficiency, and frequent interferences with mutators. Nevertheless, it is still not clear what exactly causes unsatisfying latency, especially under heavy workloads (e.g., serving a large number of requests within a short period of time). To this end, this work provides a detailed study of the memory behaviors of

two state-of-the-art concurrent copying collectors in OpenJDK (ZGC [25] and Shenandoah [15]). We mainly focus on the behaviors when running applications with heavy workloads, since those collectors introduce long pauses to mutators (similar to pauses in STW collectors), which significantly affect the throughput and latency of applications.

To understand those pauses, we explore the designs inside mainstream concurrent collectors and analyze their performance issues under heavy workloads. For Shenandoah, its generation-wise collection releases memory resources only after all chosen live objects are evacuated and corresponding references are updated, which induces lengthy pre-reclamation cycles and thus causes long STW pauses due to free memory shortage. Meanwhile, the number of objects collected in each GC cycle is also restricted by the remaining free space size, which contributes to higher GC frequency and more pauses. As for ZGC, although its region-wise collection allows incremental memory reclamation and lazy reference updating, its reference memorization mechanism (called *color pointers*) introduces significant performance overhead and even longer *mutator stalls* (the same effect as pauses).

According to limitations found in existing concurrent collectors, this work introduces *Jade*, a collector to achieve high GC efficiency while maintaining low pauses even under heavy workloads. Although sharing similarities in design with ZGC (region-wise) and Shenandoah (generation-wise), *Jade* leverages *groups* (a set of regions) as the basic unit of collection. It further divides the collection into multiple *rounds* where each round collects and releases memory resources consumed by one group. Thanks to groups, *Jade* achieves per-group incremental reclamation and avoids overhead introduced by color pointers. However, it needs to handle performance issues brought by grouping. To this end, *Jade* provides a simulation-based algorithm to finish grouping in less than one millisecond and proposes *group-based remembered sets* and *cross-region discover table* (CRDT) for low-cost reference memorization.

To reduce the pre-reclamation phase, *Jade* also embraces a generational design and leverages young GC to collect newly created objects. Instead of reusing the group-wise collection, *Jade* provides a specialized single-phase mechanism to maximize the young GC throughput. When facing inevitable pauses or application stalls, *Jade* also provides a *chasing mode* to fully leverage idle CPU cores to complete them as soon as possible.

Jade has been implemented in OpenJDK and evaluated against state-of-the-art production garbage collectors (G1, ZGC, Shenandoah) and recent efforts (LXR [41]). The evaluation uses a series of representative latency-sensitive applications, including a widely used online service deployed in *Alibaba*. The results show that *Jade* can significantly improve the application's maximum throughput while still inducing controllable low pauses and satisfying application latency.

To summarize, the contributions of this work include:

- A detailed analysis of pauses in concurrent collectors to uncover their deficiencies in design (§2);
- *Jade*, a concurrent generational collector leveraging group-wise collection to achieve both high GC efficiency and low pauses (§3-4);
- Comprehensive evaluation with representative latency-sensitive applications (including commercial online services) to show *Jade*'s improvement over state-of-the-art collectors (§5).

2 Analysis on Concurrent Copying GC

2.1 Concurrent copying garbage collectors

Although garbage collectors are helpful by automatically reclaiming memory resources for later reuse, they also introduce performance overhead to applications. Stop-the-world (STW) pauses are one of the most severe problems introduced by collectors since they require application threads (known as *mutators*) to pause until GC finishes. Although GC threads can leverage STW pauses to monopolize CPU resources for high-throughput collection, the duration of pauses becomes intolerable for applications with large memory demands and strict latency requirements. Therefore, nowadays *concurrent collectors* are trying to minimize pauses by allowing GC threads to run concurrently with mutators. *Concurrent copying collectors* are one of the most important categories in concurrent collectors. The collection in concurrent copying collectors mainly consists of two phases: *concurrent marking* to locate live objects, and *concurrent evacuation* to copy live objects into new addresses in a compact way. Various language runtimes have embraced concurrent copying for their garbage collection mechanisms. Taking OpenJDK, the mainstream runtime for Java, as an example, it has introduced two concurrent copying collectors, Shenandoah [15] and ZGC [25], and both of them only induce sub-millisecond-level pauses. Other language runtimes like Azul JDK [33] and Android Runtime (ART) [29] also provide concurrent copying collectors.

2.2 Characterizing pauses in concurrent copying collectors

Interestingly, although existing concurrent copying collectors are designed to control the duration of pauses, they still induce large pauses especially when under heavy workload. We have analyzed the performance of two representative concurrent copying collectors in OpenJDK: ZGC and Shenandoah. The evaluated application derives from the DaCapo benchmark [2], which uses the TPC-C workload [34] to evaluate a relational database, H2 [17]. The application

runs with eight physical cores to simulate a common web service scenario, which is widely deployed in *Alibaba*. The maximum Java heap size is configured as 8GB, which is quite generous compared with the live data size (about 2GB). Meanwhile, the number of concurrent GC threads is set to two for a fair comparison (the default number for ZGC is one while the others are two). Since JVMs require a warm-up phase to improve application performance, we evaluate H2 with six iterations while the first one is used for warm-up. Each iteration finishes when H2 has processed a fixed number of TPC-C transactions (256000), and the following results are average numbers among the five iterations.

We first run the application by repetitively sending requests to maximize the application's throughput and compare those concurrent copying collectors with G1 [13], which copies objects in a STW fashion and thus achieves better collection efficiency. Table 1 shows the throughput (queries per second) for three collectors under this setting. Compared with G1, both concurrent collectors cause the application's throughput to drop: they only reach 23.25% (ZGC) and 46.52% (Shenandoah) of G1's maximum throughput. When under their own maximum throughput, the p99 latency for both concurrent collectors is also worse than G1.

To further understand the performance, we analyze the memory behaviors in Shenandoah and ZGC by setting the application throughput close to the maximum (9600 and 4800 for Shenandoah and ZGC, respectively). Note that we fix the thoughput by throttling clients to control the send rate (details in Section 5.5). As shown in Table 1, both ZGC and Shenandoah induce quite long pauses to mutators when under heavy workload. For Shenandoah, it triggers *degenerated collections* when facing heavy memory pressure, which pauses all mutators. As for ZGC, although it does not directly introduce STW pauses, mutators can be stalled when no free memory is available, which has the same effect as a pause in STW collectors. As shown in Table 2, the cumulative pause time for Shenandoah and ZGC is both larger than G1 (even though its application throughput is 20800). For ZGC, the cumulative pause time is even 30.36× larger than G1 (1.81× for Shenandoah). Meanwhile, their p99 pause time is also larger than G1, which suggests that those long pauses are not rare and thus significantly affect application latency. The results indicate that reducing those pauses is vital to improving the performance of both GC and applications for existing concurrent copying collectors. To this end, we further explore the reasons for long pauses by analyzing the design of those two collectors.

2.3 Shenandoah: heap-wise collection

Shenandoah tends to treat the whole heap as a monolithic unit. Figure 1 illustrates three major phases in Shenandoah. The first *marking* phase traverses all equal-sized *regions* inside Shenandoah's heap to locate live objects and store their

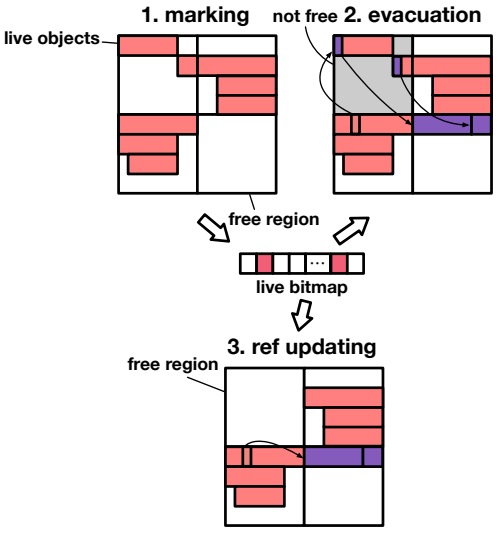


Figure 1. Concurrent phases in Shenandoah (suppose only one region is collected). Colored rectangles stand for live objects while lines with arrows stand for references.

address information in a *live bitmap*. Afterward, Shenandoah evacuates objects marked in the live bitmap to free regions (marked with a different color). During evacuation, Shenandoah stores an object's new address into the header of its old copy. Since the number of free regions might be limited, only a part of the regions can be evacuated. Finally, Shenandoah scans the live bitmap again to update references so that they all point to an object's newest address (with the help of marked headers). Only after the reference updating phase can the memory resources be recycled.

Although this design has advantages like low memory overhead (only one bitmap is used to memorize live objects), it introduces the following two performance issues.

- **Longer pre-reclamation cycle.** Since Shenandoah only maintains live objects' information in a simple bitmap, it does not know exactly which objects hold references to copied ones. Therefore, GC threads have to check all live objects for reference updates, and free regions can be reclaimed *only* when all phases are finished, which lengthens the time before the memory resources can be reused (we refer to this cycle as a *pre-reclamation cycle*). Meanwhile, each live object needs to be accessed at least three times in one GC cycle, which adds more performance overhead and also affects the duration of pre-reclamation.
- **Higher collection frequency.** Shenandoah's reclamation efficiency heavily relies on the number of free regions in its one-generation heap. If the heap only contains a few free regions, the number of reclaimed bytes is limited, and the frequency of GC would increase.

Table 1. Application and pause statistics for three mainstream collectors.

Collectors	Max Thru. (tx/s)	p99 latency (ms)	Cumulative pause time (ms)	p99 pause time (ms)
G1	21208	34.20	447.32	82.77
ZGC	4931	128.11	13581.87	1963.38
Shenandoah	9865	37.65	809.22	297.39

Those two issues make GC a severe performance bottleneck with heavy workloads. When the application's allocation rate increases, mutators may exhaust all available free space and have to wait until the pre-reclamation cycle finishes. Since Shenandoah's pre-reclamation cycle is quite long, the pause time could also become longer. Meanwhile, higher collection frequency suggests more pre-reclamation cycles are required, which induces more performance overhead and potentially more pauses. As shown in Table 2, the three phases (*Mark* and *Other*) in Shenandoah together contribute to 24.89 seconds, which means GC threads are active for over 90% of execution time and potentially block mutators from memory allocation. Worse still, although collections are long and frequent, their marking results are actually quite similar. We have dumped the live bitmap generated in Shenandoah's marking cycles and analyzed regions not moved by the subsequent evacuation phase. By comparing those regions in the live bitmaps for two consecutive cycles, we find that only 0.53% of objects are different, which suggests that the high frequency of marking is actually unnecessary. Therefore, the key to reducing Shenandoah's overhead would be decreasing both the duration of pre-reclamation and the whole-heap GC frequency.

2.4 ZGC: region-wise collection

Compared with Shenandoah, the algorithm of ZGC is more incremental and region-wise: regions can be reclaimed independently. As illustrated in Figure 2, the vanilla ZGC has a similar marking phase to Shenandoah, which marks the whole heap to find live objects. However, when all live objects in a region have been evacuated, the region can be *immediately* reclaimed without waiting for reference updating. ZGC achieves this by using *color pointers*, which encode information inside object references to determine if the referred object requires updating. Furthermore, it also maintains per-region forwarding tables storing mappings between an object's old address and the new one. With this design, reference updating can be lazily conducted during mutator execution (for example, when a stale reference is loaded) or in the next marking phase (ultimately updating all stale references to live objects).

This region-wise design seems to shorten the pre-reclamation cycle by moving evacuation and reference updating off the critical path. Unfortunately, the application stall becomes even longer as shown in Table 1. Although ZGC's pre-reclamation only contains the marking phase,

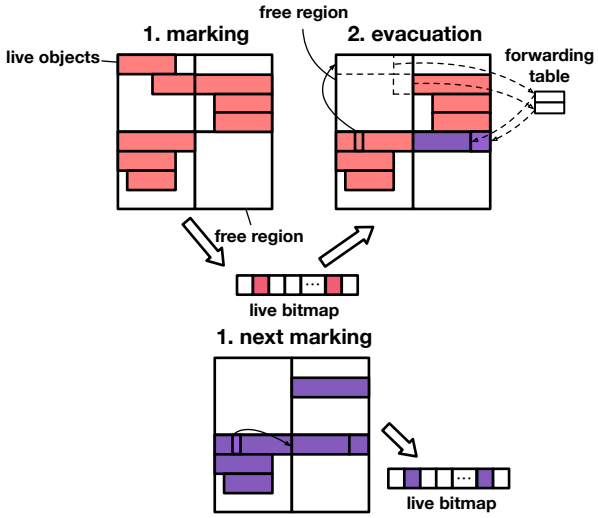


Figure 2. Concurrent phases in ZGC.

its duration is much longer than Shenandoah (2.40×) and remains active in 92.89% of the overall execution time of H2 (Table 2). The overhead mainly comes from ZGC's color pointer design. Since it requires four extra bits in virtual addresses to encode reference-related information, the virtual memory size of Java heaps is enlarged by 16× and makes the reference compression optimization [21] (encoding 64-bit heap references to 32-bit offsets from the start address of heap) impractical, which affects both GC and application performance. Meanwhile, to fix all stale references, the marking phase in ZGC needs to recolor them by modifying their encoded bits, which introduce a large number of atomic instructions. To simulate the overhead, we have disabled the reference compression optimization and added a dummy compare-and-swap instruction before marking each object in Shenandoah. With those modifications, H2's peak throughput on Shenandoah drops from 9865 to 6052, which is close to that in ZGC (note that disabling compressed references also significantly affects mutators' performance in addition to GC).

2.5 Discussion: generational variants

To achieve better collection efficiency, both ZGC and Shenandoah have developed their own generation variants (referred to as *GenZ* and *GenShen*), which typically divide the heap into two generations. The *young generation* is used

Table 2. Breakdown analysis for concurrent copying collectors on the H2 application. The first three columns show the duration for overall execution (**App.**), GC marking (**Marking**), and other phases in the pre-reclamation cycle (**Other**, only exists in Shenandoah). The latter two show the average time for marking and other phases for each GC cycle, while the last two show the cumulative pause time in marking and other phases. All results are wall-clock time in seconds.

Collectors	App.	Marking	Other	Avg. Marking	Avg. Other	Cumulative Pause Marking	Cumulative Pause Other
ZGC	56.95	52.90	-	2.40	-	13.39	-
Shenandoah	27.40	13.56	11.33	1.00	0.81	0.12	0.68

to serve memory allocation requests, and a separate *young GC* cycle is provided to only collect the young generation. Meanwhile, a full GC cycle is only triggered when memory resources in the whole heap become exhausted. Since the young generation is relatively smaller, the pre-reclamation cycle is largely shortened, which helps reduce application stalls and pauses. Nevertheless, the design of GenZ's and GenShen's young GC algorithm is similar to the corresponding single-generation one. For GenZ, the young GC algorithm still contains the overhead of color pointers. As for GenShen, it turns into a *generation-wise* collector which still collects the young generation in three phases. Therefore, they still induce considerable overhead. We will show the detailed results of GenZ and GenShen in Section 5.

2.6 Summary

According to the performance analysis, we find that reducing the duration of pre-reclamation cycles is critical to control pauses when under a heavy workload. Meanwhile, the collector should also avoid introducing large runtime overhead. We therefore build our own collector to achieve both goals.

3 Group-wise design in *Jade*

3.1 Overview

This work introduces *Jade*, a high-throughput concurrent copying collector achieving both low GC pauses and satisfying collection efficiency. Compared with prior concurrent copying collectors (heap/generation-wise or region-wise), *Jade* provides a new abstraction named *group* as a unit of reclamation, which is the core of *Jade*'s design. Thanks to the group-wise collection mechanism, *Jade* can achieve incremental reclamation while inducing moderate performance overhead.

Analogous to prior concurrent copying collectors, *Jade* also adopts equal-sized regions to organize its heap. When GC is triggered, *Jade* also contains a concurrent marking phase which generates a live bitmap to memorize live objects in its regions. The live bitmap uses each bit to denote if a memory range of 8 bytes contains the header of a live object, so the bitmap's memory consumption is 1.56% of the heap size. Marking is followed by a special *grouping* phase, which divides the heap into *collection groups* where each group contains several regions. The reclamation phase is then divided into multiple *rounds*. Each round only evacuates regions in one group, i.e., copying live objects in those regions to free regions in the heap. *Jade* releases memory resources consumed by a group immediately after the corresponding round ends. To coordinate concurrent operations from mutators, *Jade* adopts loaded value barriers (also used in ZGC) to ensure mutators always hold references to an object's latest copy. With the group-wise design, *Jade* achieves group-level incremental reclamation since free regions can be recycled when each group is evacuated. Meanwhile, it also leverages the same marking results for all rounds in one collection cycle, which reduces collection frequency and performance overhead compared with an algorithm like Shenandoah which treats the whole heap/generation as the reclamation unit.

Nevertheless, the results on ZGC suggest that an incremental reclamation algorithm is not enough. The design of a group-wise algorithm should also control its runtime overhead, which mainly consists of two parts: (1) the *grouping algorithm* to assign regions to groups and (2) *reference memorization* for subsequent reference updating. To this end, *Jade* proposes designs for those two parts separately.

3.2 Simulation-based hand-over-hand grouping

As introduced before, the group abstraction can be seen as a middle ground between generation and region: when each group only contains one region, the evacuation is similar to ZGC; when only one group is generated in a GC cycle, the behavior is close to Shenandoah. Therefore, the grouping algorithm is important for *Jade*'s collection efficiency. Meanwhile, the time required for grouping should be short as it is included in the pre-reclamation cycle. To this end, *Jade* proposes a *simulation-based hand-over-hand grouping* algorithm.

After marking is finished, *Jade* simulates a hand-over-hand compaction [20] that assumes a former group's released memory is directly reused by the evacuation of a latter one. As illustrated by Algorithm 1, *Jade* first constructs a *tracked list* to include regions to be evacuated in the current GC cycle (line 1-6). This step filters out regions whose live bytes exceed a preset threshold (85% by default) to avoid inducing large memory copying overhead. The tracked list

is then sorted by the number of live bytes so that the evacuation can start with those containing the fewest live bytes. This part of the algorithm is similar to the construction of collection sets in collectors like G1 and Shenandoah, but *Jade* needs to further split the list into multiple groups to achieve group-wise collection. It starts by estimating the free space size available for evacuation (line 9, more details in Section 4.2) and adds regions to the first group until the accumulated live bytes exceed the overall free bytes (line 13-22). *Jade* uses the size of the first group for all subsequent groups as well (line 23). This design is used to control the group size: since *Jade* compacts live objects in multiple used regions to free ones, the number of free regions gradually increases every time a group is reclaimed. If all released regions are used for a per-group evacuation, the pre-reclamation time becomes longer, which could hinder mutators from memory allocation. For subsequent groups, since the number of regions is fixed, *Jade* only needs to continuously fill them up until the tracked list is drained (line 26-33). Lastly, *Jade* sets a maximum number of groups (16 by default) to avoid a long collection cycle with too many groups, and regions in the tracked list would not be collected when *Jade* has constructed enough groups (line 34-36). Note that since *Jade*'s grouping mechanism is based on a simulation, it requires no data copying and thus only introduces trivial (microsecond-level) overhead. Other advanced policies (e.g., controlling a group's size by setting the maximum number of regions) are also possible to tune *Jade*'s performance.

Algorithm 1 Constructing collection groups

```
1:  for region in old_regions do
2:      if region.live_bytes/region.size_bytes <
3:        threshold then
4:          tracked_list.add(region)
5:      end if
6:  end for
7:  # Sorting the tracked regions (organized as a list) by the
    size of live bytes
8:  tracked_list.sort()
9:  free_bytes ← estimate_free_space()
10: while !tracked_list.is_empty() do
11:     group ← empty
12:     if groups.is_empty() then
13:         # Constructing the first group
14:         while !tracked_list.is_empty() do
15:             region ← tracked_list.take_first()
16:             free_bytes ← free_bytes−
17:                          region.live_bytes
18:             if free_bytes < 0 then
19:                 break
20:             end if
21:             group.add(region)
22:         end while
23:         group_size ← group.size()
24:     else
25:         # Constructing subsequent groups
26:         while group.size() < group_size do
27:             if tracked_list.is_empty() then
28:                 break
29:             end if
30:             group.add(tracked_list.take_first())
31:         end while
32:     end if
33:     groups.add(group)
34:     if groups.size() >= MAX_GROUP then
35:         break
36:     end if
37: end while
38: # Output groups
```

3.3 Group-wise remembered sets

After grouping, *Jade* can start following its simulation process to evacuate live objects in each group. The remaining challenge is how to guarantee regions in a group are free to release after live objects have been moved to free regions. Considering the overhead of color pointers, *Jade* embraces *remembered sets* to memorize references at a coarser granularity.

Remembered sets are usually used to memorize incoming references to a given memory range (e.g., a region). Taking regions as an example, the collector can maintain a remembered set for each region (G1 has a similar design for its remembered sets), which is usually implemented with a bitmap where each bit corresponds to a 512-byte memory range (referred to as a *card*) in the heap. After the marking phase, GC threads can scan the live bitmap to locate cross-region references. Suppose a reference r in region x points to an object in another region y, GC threads need to (1) acquire the remembered set for region y and (2) set the bit for the corresponding card where r resides. When a region is evacuated, GC threads only need to use its remembered set, scan memory ranges where the corresponding bits are set, and update incoming references therein.

Although enabling incremental reclamation, remembered sets have three disadvantages compared with color pointers. First, since the references are memorized in a coarse (card-level) granularity, GC threads have to scan more objects to find and update cross-region references. Second, the memory overhead is proportional to the number of regions, which can be quite large (usually more than thousands). Third, remembered sets need to be rebuilt after each marking phase, which potentially lengthens the pre-reclamation cycle. To mitigate the first two problems, *Jade* provides *group-wise remembered sets*, where regions in the same group share a remembered set. Since regions in the

same group are released together, we do not need to memorize inter-region references inside them, which reduces the re-scanning overhead. Meanwhile, the memory overhead is also reduced as the number of groups is much smaller. Since each bitmap consumes only 1/4096 of the heap size, the overall memory overhead is only 0.39% for 16 groups.

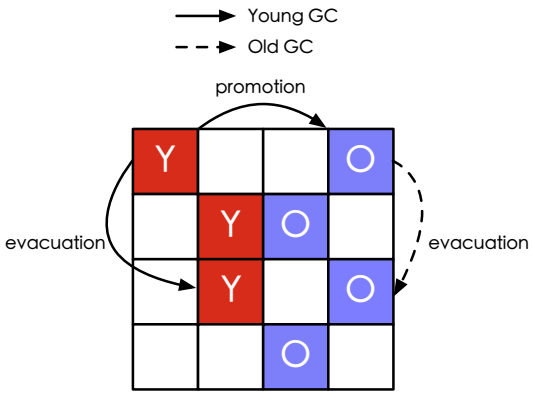


Figure 3. Heap layout in *Jade*.

Optimization: piggyback with marking. Nevertheless, rebuilding group-based remembered sets is still time-consuming as it needs to scan the remembered sets. Fortunately, this step can be further improved considering the fact that the number of inter-region references is usually limited. For example, our analysis on the Specjbb2015 workload [12] shows that 83.13% of dirty cards (containing inter-region references) in the whole memory range contain references to only one or two regions (excluding the region the card resides in), so this region-related information can be memorized with acceptable overhead. To this end, *Jade* proposes to collect the information in the preceding marking phase. This is achieved by a global *cross-region discover table (CRDT)*, which memorizes inter-region references also in a card granularity (512 bytes). CRDT maintains a mapping between cards and an integer (4 bytes). In contrast to the remembered set, CRDT maintains *outgoing references* to other regions by storing region IDs in the integer. For example, suppose *Jade* discovers an address in card x from region y which stores a reference to another region z, then the region ID z should be stored into x's corresponding integer in CRDT. For each card, two outgoing references to the same region are only stored once. Since the number of regions is usually in the thousands, 4 bytes are enough to store two region numbers. If the number of discovered regions exceeds two, *Jade* marks the corresponding integer in the CRDT with a special value (currently -1) so that it needs to be rescanned in the remembered set building phase. With CRDT, the work required by building remembered sets is greatly reduced. First, cards not containing cross-region references do not need scanning since it is impossible for them to contain cross-group references. For those recording region IDs, *Jade* finds the group containing the regions and marks the bit for the card in the group-wise remembered set. Because both remembered sets and CRDT use the same granularity for reference memorization, this step does not need card scanning. It also eliminates unnecessary scanning if the recorded regions are in the same group as the card. Lastly, only cards marked with -1 should be thoroughly scanned, so the scanning frequency is greatly reduced (evaluated in Section 5.6). Meanwhile, the memory consumption of CRDTs is also moderate: Since one CRDT is enough to maintain cross-region references for all groups, its memory consumption is only 0.78% of the heap size.

4 Generational collection

4.1 Single-phase young GC

To break long collection cycles into smaller ones, *Jade* also embraces a generational design and provides young GC to collect its young generation. As shown in Figure 3, the regions in *Jade*'s heap can be free (unmarked), used by the young generation (marked as *Y*), or consumed by the old generation (marked as *O*). Compared with a multi-generation approach with more than two generations [5], *Jade*'s two-generation design is simpler and enough to collect short-lived objects. When young GC is triggered, objects in young regions can be copied (or promoted) to the old generation or still reside in the young one. As for the old GC (the group-wise collection introduced before), objects are only evacuated to other old regions. Those two can run together as they reclaim different parts of memory resources.

Generational concurrent collectors like GenZ and C4 use the same algorithm for young and old GC, so the young GC still contains two phases: marking and incremental evacuation. This two-phase design is helpful to (1) prioritize regions with the most garbage for collection and (2) immediately reclaim a region once it is evacuated. However, both two advantages are somewhat unnecessary for young GC. First, since the weak generational hypothesis [35] still holds for many modern applications, the survival rate for young regions is usually low and all young regions will be selected for reclamation, rendering the prioritization useless. Second, reclamation immediacy is not that important for young GC because it is much faster than old GC and hardly becomes a bottleneck for memory allocation. Therefore, *Jade* instead adopts a single-phase young GC algorithm to maximize its collection efficiency and thus saves more CPU cycles for the execution of mutators and old GC.

The single-phase design finishes the marking, evacuation, and reference updating in the same phase: *Jade* traverses the young generation and immediately copies an object to a survivor region once it is marked live. Instead of using a marking bitmap like the old GC, *Jade* directly stores an object's

new address in its old header as the marking result and uses atomic instructions to avoid repetitive copying of the same object. Since young GC does not traverse old regions, *Jade* also needs to remember inter-generation references from them. This is achieved by an *old-to-young remembered set*, where each bit maintains if a corresponding 512-byte memory range contains inter-generation references. Meanwhile, references inside live objects are pushed into a GC-thread-local marking stack so that they can be updated immediately after their referents have been evacuated. By coalescing multiple operations into one single phase, *Jade* reduces the number of memory accesses and significantly improves the GC efficiency.

4.2 Free space estimation

To avoid blocking mutator allocation during old GC, *Jade* allows the co-running of both GC cycles. Therefore, the free space estimation in Algorithm 1 should consider two factors: (1) allocatable free bytes in the heap and (2) memory behaviors in the young generation. The first part is simple to calculate: since only completely free regions can be allocated to mutators, we can get the value by multiplying the number of free regions by the region size (Line 2). Meanwhile, since the young GC promotes objects to the old generation (Figure 3), *Jade* calculates its size with the historical promotion rate and the estimated remaining GC time (Line 3-4). As for the young generation, since its memory behaviors are complicated (perhaps including multi-round memory allocation and collection), *Jade* conservatively leaves a part for all memory-related activities within the young generation, whose size is determined by an empirical ratio (*young_ratio*, 85% by default). Only the remaining free bytes are treated as free space and used as destinations for evacuation in old GC.

Algorithm 2 Estimating free space size for collection

```
1: function ESTIMATE_FREE_SPACE
2:     free_space ← free_region_count * region_size
3:     free_space ← free_space − promotion_ratio*
4:                  estimated_gc_time
5:     free_bytes ← free_space * (1 − young_ratio)
6:     return free_bytes
7: end function
```

4.3 Chasing mode and full GC

When the allocation rate becomes too high, *Jade* may face inevitable mutator stalls due to free memory shortage. Since the number of running mutators becomes smaller, the CPU resources can be reaped to run more GC threads. However, since the default number of GC threads in ZGC and Shenandoah is usually small, they fail to leverage idle CPU cores in a mutator stall (e.g., ZGC only occupies 26.80% of overall CPU resources during application stalls in Table 1's experiment). Meanwhile, if the user manually increases the thread number, GC threads would compete for more CPU resources and affect application performance when no stall happens. To this end, *Jade* introduces a *chasing mode*, which launches more GC threads to concurrently evacuate groups only when mutators begin to stall. In the default setting, the number of concurrent GC threads is equal to the number of cores, which maximizes the GC throughput to finish processing the current collection group. *Jade* also supports full GC but only triggers it when extreme cases occur (e.g., consecutive long application stalls), and it also sufficiently utilizes all available CPU resources to improve the collection efficiency.

4.4 Weak references handling

Jade supports handling weak references in both young and old GC. In the marking phase (mark-and-copy phase in the young GC), *Jade* puts objects into a discover list if they are pointed to by weak references. After all concurrent phases finish, *Jade* leverages an extra stop-the-world phase to check if objects in the list are marked through other strong references. If an object is not marked, *Jade* tries to reclaim it and invokes its corresponding callback function (if any). Since the number of weak references is not large for many applications, the induced pause time is trivial. We plan to modify this phase into a concurrent one in our future work.

5 Evaluation

5.1 Experiment setup

Jade is implemented on OpenJDK (the version is 11.0.17.13) with about 30,000 lines of code. During implementation, some components of *Jade* derive from existing collectors in OpenJDK, such as snapshot-at-the-beginning (SATB) marking and card tables.

We mainly evaluate *Jade* on a bare-metal instance in the public cloud environment. The instance has dual Intel Xeon Platinum 8369B CPUs (32 physical cores with SMT enabled) and 512GB DRAM. To avoid NUMA issues, we bind applications to separated physical cores on the same socket. We leverage the following applications to evaluate *Jade*.

- **Specjbb2015** [12] is the de facto standard for Java server performance, especially for garbage collectors. It simulates an online supermarket serving various kinds of requests. Both ZGC and Shenandoah adopt this application to show their improvement against prior collectors upon their release [15, 22]. Meanwhile, Oracle leverages it as the primary application to show the progress of GC in OpenJDK [19, 30].

- **HBase** [1] is a distributed big-data store for non-relational data. We mainly use it to study the single-machine performance, and the evaluated version is 2.4.14.
- **Shop** is a real-world online service used in *Alibaba*'s e-commerce platform. This service is used by tens of millions of users every day to serve their requests to access online shops.
- **DaCapo** [2] is a benchmark suite containing various Java workloads. Compared with other evaluated applications, most workloads in DaCapo have smaller memory demand, so we mainly use them to show the performance of *Jade* with tight memory budgets.

With those applications, we mainly compare *Jade* with four different concurrent garbage collectors: G1, ZGC, Shenandoah, and LXR. G1 and LXR evacuate objects in an STW fashion, so it is helpful to improve application throughput. However, their STW pauses may cause worse application latency especially when the size of live data grows larger. Both ZGC and Shenandoah concurrently evacuate objects and thus have larger interferences on application throughput. All those collectors are also evaluated under OpenJDK-11.0. Since G1 allows users to control its pause time by setting a soft limit (*-XX:MaxGCPauseMillis*), we manually set it to 10ms for better application latency, in addition to a default setting (200ms). We also evaluate the generational variants of ZGC and Shenandoah (namely GenZ and GenShen), and the respective versions are OpenJDK-21 and OpenJDK-22[1] since they do not backport to older versions. As GenZ and GenShen are implemented in different JDK versions, the evaluation results would be affected by other factors other than GC (e.g., efficiency of JIT optimizations). LXR is built from its latest commit (the commit number for mmtk-core and mmtk-openjdk is 4ab99bb and cdbc8de, respectively), which is also based on OpenJDK-11.0 (commit 7caf8f7). The *mmtk/opt* cargo feature is also enabled to avoid performance issues. C4 is not included as it uses a similar algorithm to GenZ but a much different JDK implementation (Azul JDK).

For all evaluated applications, we mainly concentrate on two metrics: application throughput and latency. To this end, we collect the latency statistics when applications are under different throughput levels. Meanwhile, we also leverage various heap configurations for all applications but the online Shop service since we have no permission to adjust its heap size. To configure the heap size, we first calculate a minimum heap size for ZGC and simulate three scenarios: tight (1.5×), medium (2×), and large (4×). The minimum heap size for Specjbb2015 and HBase is 1941MB and 1100MB, respectively. For Shop, we directly use its fixed configuration (8GB heap, approximately 4× of the live data size).

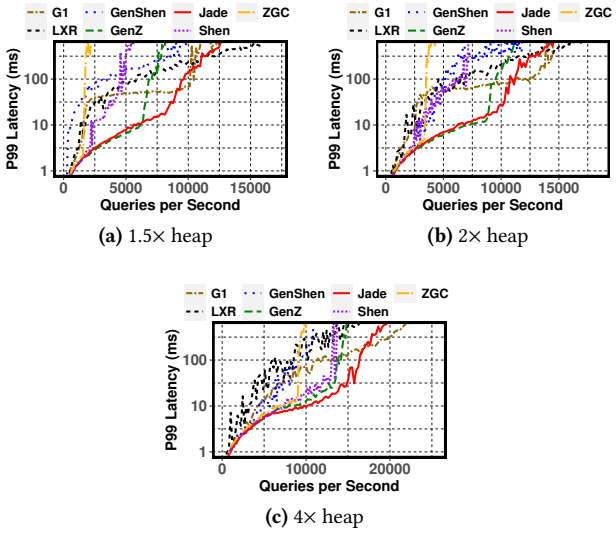
[1]Generational Shenandoah is still under development, so we download the latest version (commit f3c9eda) for evaluation.

As for DaCapo, we use a tighter memory configuration (details in Section 5.5). All applications use 8 physical cores regardless of heap size.

5.2 Specjbb2015

We mainly evaluate Specjbb2015 with its default mode (*HBIR_RT*), which reports two scores for each execution: *max-jops* stands for the peak throughput while *critical-jops* stands for the maximum throughput satisfying p99 latency requirements.

As shown in Table 3, *Jade* has worse max-jops compared with the default setting of G1 (3.98% smaller in arithmetic mean) and LXR (6.81%), but the result is much better than other concurrent copying collectors. Meanwhile, its critical-jops score of *Jade* is the best for all settings. The evaluation suggests that the critical-jops score of other concurrent collectors is significantly restricted by their maximum throughput, especially under tight heap configurations. As for G1, although setting a smaller soft limit (10ms) helps improve the critical-jops score, the max-jops is significantly decreased due to more frequent collections and larger overhead.



(a) 1.5× heap **(b)** 2× heap

(c) 4× heap

Figure 4. P99 latency results under various throughput settings in Specjbb2015.

Figure 4 further shows the p99 latency when under various throughput settings. The latency statistics are collected when Specjbb2015 gradually increases the application throughput to find the peak one. The results show that other concurrent copying collectors have smaller p99 latency than G1 when the throughput is moderate, but the latency soon becomes larger due to their poor GC efficiency. In contrast, *Jade* has optimized its collection phases and thus reaches satisfying latency even under heavy workloads. As for other percentiles, *Jade*'s worst latency in the 2x heap

Table 3. Maximum throughput numbers for various collectors. For Specjbb2015, we also show the *critical-jops* score (the first of the two numbers).

Application	Collector	1.5× heap	2× heap	4× heap
Specjbb2015	Jade	8299/13101	9497/15257	14149/21454
	G1	4462/13433	4788/14926	7942/24297
	G1-10ms	4122/11761	5058/12936	8811/19786
	ZGC	1618/2774	3443/4563	8576/9931
	Shenandoah	3771/6120	4602/7245	11371/13599
	LXR	3170/16302	4038/17694	5310/18989
	GenZ	6675/8387	8654/11346	12258/15308
	GenShen	1605/11345	4313/10218	7212/10931
HBase-Insert	Jade	1128	1164	1284
	G1	970	1141	1304
	G1-10ms	866	931	1294
	ZGC	668	690	1096
	Shenandoah	840	873	969
	GenZ	874	890	1045
	GenShen	726	528	777
HBase-Mixed	Jade	1334	1577	1929
	G1	1384	1489	1820
	G1-10ms	1332	1313	1954
	ZGC	737	811	1302
	Shenandoah	843	734	944
	GenZ	1085	1146	1666
	GenShen	861	663	1218
Shop (8GB)	Jade	-	-	400
	G1	-	-	350
	ZGC	-	-	150
	Shenandoah	-	-	300

configuration is better than GenZ when QPS exceeds 9000 and GenShen for almost all QPS settings.

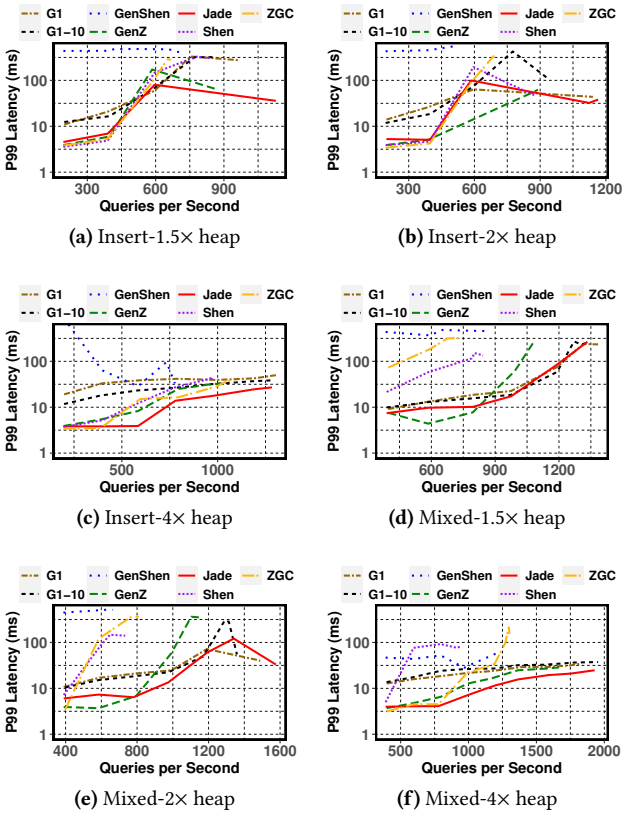
5.3 HBase

HBase is evaluated using YCSB [11] with two workloads: an insert-only one and a mixed one (50% read and 50% insert). Furthermore, we use the throttle option to control the request rate and generate results under various throughputs. LXR is not included since it is based on MMTk, whose current version cannot run HBase [38]. As shown in Figure 5, when compared with other concurrent copying collectors, *Jade* has better latency results than Shenandoah and GenShen for nearly all configurations and remains comparable with ZGC and GenZ under moderate throughput. Meanwhile, *Jade*'s peak throughput is 1.63×, 1.63×, 1.27×, and 1.82× against ZGC, Shenandoah, GenZ, and GenShen, respectively. When setting the pause soft limit to 10ms, G1 has better p99 latency under moderate workload, but the maximum throughput decreases.

5.4 Shop

To evaluate the online Shop service, we use four nodes to generate concurrent requests and simulate a stressful workload. Since the online service only supports JDK11, GenZ is not evaluated in this experiment. If the p99 latency is larger than a second, the Shop service automatically reports itself as unavailable for user-experience considerations. As shown in Figure 6, *Jade* can achieve the best peak throughput among all collectors. Although G1 can endure high throughput, it triggers too many long pauses and thus induces a large p99 latency, so its peak throughput is worse than *Jade*. In contrast, *Jade* reaches comparable p99 latency with Shenandoah and ZGC under moderate throughput while outperforming G1 by 3.94× on average (from 100 to 350 QPS).

We also collect the average CPU utilization during application execution. The results in Figure 6b show that the CPU utilization under moderate throughput is similar among all collectors except ZGC. However, when the throughput is close to the maximum, both Shenandoah and ZGC's CPU utilization quickly increases. As analyzed before, they introduce long pre-reclamation cycles, which consume much



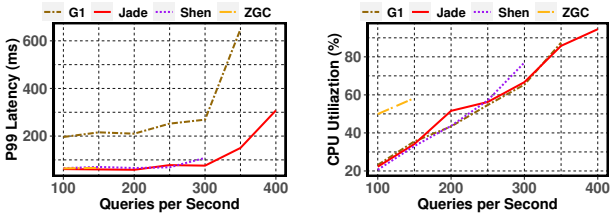
(a) Insert-1.5× heap **(b)** Insert-2× heap

(c) Insert-4× heap **(d)** Mixed-1.5× heap

(e) Mixed-2× heap **(f)** Mixed-4× heap

Figure 5. P99 latency results under various throughput settings in HBase.

more CPU resources and put heavier burdens on mutators. Meanwhile, the p99 latency of *Jade* is relatively stable even when the CPU utilization reaches 85.78% (QPS is 350), which confirms that *Jade* can retain high GC efficiency and short pauses even under heavy CPU-intensive workloads.



(a) P99 Latency **(b)** CPU utilization

Figure 6. Evaluation results for Shop.

5.5 DaCapo

Table 4 first shows the normalized application execution time under two heap configurations (1.5x and 2x of G1's minimal heap [37]) for all workloads in DaCapo. The results are averaged over ten times of execution, where each execution reports the numbers in the fifth iteration for each application (i.e., the other four iterations are used as warm-up execution). The relative standard error ranged from 0.002% to 51.49%, with 98.18% of data points having a standard error of 10% or less. Due to the tight memory configuration, ZGC and Shenandoah (including their generational variants) fail to execute some applications due to out-of-memory errors or inducing large performance overhead. In contrast, *Jade* can execute all applications and its performance remains relatively stable. Compared with them, LXR and G1 show better performance on DaCapo. Those two collectors both introduce STW pauses for evacuation to reach better collection efficiency. Since the memory budget for most applications in DaCapo is relatively small (less than 1GB), the pause time is also short. As for concurrent copying collectors, their runtime overhead like load barriers becomes more significant under those scenarios.

For latency results, in applications like tomcat (142MB heap size), LXR has 10.31% better p99 latency against *Jade*. However, for those with larger heap sizes, the application latency is largely affected by LXR's pauses. Figure 7 shows the results for two different size configurations in H2 from DaCapo, normal (by default) and large (the corresponding minimal heap size is 4099MB). To collect latency statistics in various throughput levels, we modify the metered latency measurements from the Chopin version of Dacapo [16] to model request queuing with adjustable QPS configurations (we refer to this application as *H2-throttle*). For both settings, *Jade* shows better p99 metered latency than G1 and LXR although its maximum throughput is smaller. By analyzing the GC log, we find LXR's average pause time under the large configuration and moderate throughput (8000 QPS) is 46.30ms, while G1 is 40.41ms. In contrast, *Jade*'s average pause time is only 0.52ms, which can explain its better latency than the other two collectors.

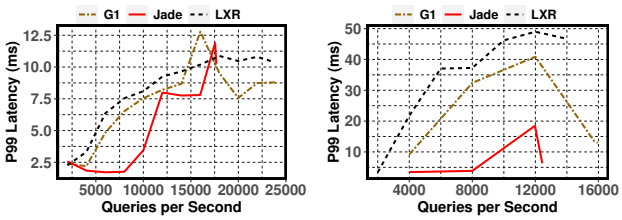
5.6 Breakdown analysis

Group-related parameters. Figure 8 shows the p99 latency when varying the maximum group and region number. The evaluated application is Specjbb2015 with its preset mode, which runs with a fixed QPS (2000) for 10 minutes. When *Jade* is only allowed to evacuate one group in each collection cycle, the collection efficiency is affected, so the tail latency becomes worse. As for the region number, different configurations lead to similar results, which suggest *Jade*'s performance is not sensitive to them.

Collection efficiency. We also compare the GC performance of GenZ and *Jade* since they both use a generational design and GenZ has better performance than GenShen (Table 3). The workload is the same as Figure 8. To achieve a relatively fair comparison, we disable the compressed pointers in *Jade* and fix the number of GC threads to two (one for young and another for old, chasing mode disabled) and

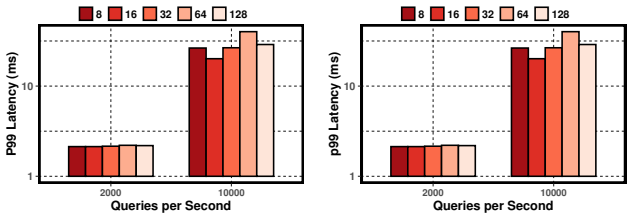
Table 4. Application execution time for the DaCapo benchmark (normalized to G1), under 1.5x (left) and 2x (right) minimal heap configurations. N/A means the applications cannot run due to unsupported JDK versions.

App	G1	G1-10ms	Shen.	ZGC	GenShen	GenZ	LXR	Jade
avrora	2902/2811	0.994/0.972	1.092/1.037	OOM	1.734/1.437	OOM	0.992/1.027	1.747/1.770
batik	1707/1735	1.004/0.978	1.152/0.976	OOM	1.389/1.269	1.314/1.044	OOM/1.048	1.741/1.712
biojava	7487/7334	1.000/0.992	2.912/2.062	OOM	3.105/2.581	OOM/3.781	0.954/0.969	1.825/1.803
cassandra	8812/7760	1.009/0.989	1.275/1.177	OOM	N/A	N/A	0.961/1.011	1.793/1.801
eclipse	12473/12215	0.999/0.996	1.068/1.047	OOM/0.980	1.418/1.309	1.119/0.978	1.023/1.038	1.770/1.767
fop	1041/860	0.998/0.989	4.035/1.592	OOM	6.049/4.609	OOM/8.059	0.731/0.829	1.655/1.700
graphchi	4252/4188	0.998/0.997	1.909/1.636	OOM/1.403	4.414/3.987	1.405/1.227	0.949/0.956	1.825/1.831
h2	4972/3790	0.989/1.005	7.104/5.642	OOM/11.47	2.304/2.373	2.212/1.914	0.904/1.109	1.941/2.130
h2o	4573/3793	1.015/0.998	2.259/1.750	OOM	N/A	N/A	1.068/1.110	1.913/1.857
jme	6873/6873	1.000/1.000	1.008/1.005	OOM	1.088/1.097	1.006/1.005	0.999/1.000	1.733/1.734
jython	5890/5393	0.999/1.002	3.323/2.022	OOM	6.548/5.238	OOM/1.954	0.958/1.031	1.883/1.829
kafka	5186/5200	0.999/0.996	1.000/0.996	OOM/0.995	2.441/3.588	0.993/0.989	0.991/0.992	1.728/1.725
luindex	4290/4283	0.994/0.989	1.173/1.089	OOM	18.20/20.93	1.054/0.976	0.969/0.988	1.780/1.782
lusearch	5398/4688	1.021/0.987	OOM	OOM	OOM	OOM/6.914	0.981/1.101	2.313/2.240
pmd	2549/2407	1.002/1.013	1.338/1.243	OOM/1.432	10.95/31.76	1.794/1.469	0.959/0.987	1.793/1.797
spring	4414/3077	1.005/1.043	9.403/5.243	OOM	7.621/5.314	OOM	0.804/0.995	1.683/1.853
sunflow	8285/8100	1.000/0.967	8.371/2.729	OOM	OOM/13.10	3.009/2.234	0.699/0.705	2.143/1.945
tomcat	13377/13330	1.001/0.999	1.494/1.198	OOM	4.510/3.245	OOM/4.019	1.005/1.003	1.750/1.747
tradebeans	5984/5691	1.004/0.997	4.842/2.626	OOM	OOM	OOM/9.141	1.089/1.072	2.081/2.076
tradesoap	4615/3031	0.989/1.007	3.022/2.480	OOM	OOM	OOM/13.68	0.757/1.095	1.664/1.999
xalan	2747/1753	0.988/1.008	26.03/25.83	OOM	37.19/43.93	6.657/7.631	0.737/0.962	2.077/2.312
zxing	2432/2404	1.003/0.960	1.023/1.009	1.454/1.017	1.794/1.386	0.995/0.958	0.917/0.957	1.680/1.680
geomean	-	1.000/0.995	2.450/1.873	1.454/1.685	4.047/4.263	1.597/2.468	0.919/0.995	1.835/1.860



(a) Nomal **(b)** Huge

Figure 7. P99 latency results under various throughput and workload settings in H2-throttle.



(a) Maximum group number **(b)** Maximum region number

Figure 8. P99 latency results with varying group-related parameters (the application is Specjbb2015).

Table 5. Breakdown GC statistics for *Jade* and GenZ, including average time (in milliseconds) and GC throughput (reclaimed megabytes per second) for both young and old GC. Again, GenZ and *Jade* are running on different JDK versions.

Cycle	Collector	Phase	Avg.	Thru.
Young	Jade	Mark+Evac.	111.90	8579.47
		Total	111.90	
	GenZ	Mark	366.29	2259.70
		Evac.	86.87	
		Total	453.16	
Old	Jade	Mark	1124.64	1248.36
		Build	218.88	
		Evac.	763.23	
		Total	2106.75	
	GenZ	Mark	2525.76	760.53
		Evac.	932.31	
		Total	3458.07	

the young generation size to 1GB. Note that in this configuration, Jade's application throughput is also affected (e.g., the max-jops for 2x heap is 11761). According to the results

in Table 5, *Jade* reaches 3.80× larger GC throughput (calculated by released memory size over collection time) in the young GC cycle. The speedup mainly comes from (1) the single-phase algorithm in *Jade* that avoids repetitive object graph traversals, (2) the inherent color pointer overhead and (3) the performance loss due to disabling compressed pointers in the presence of color pointers in GenZ. As for the old GC cycle, *Jade* performs better in both marking and evacuation than GenZ and thus reaches 64.14% improvement on GC throughput. The results further confirm that *Jade*'s group-based design outperforms prior concurrent copying collectors even with similar generational layouts.

Duration of different phases. Table 6 shows different phases' execution time under various heap configurations and maximum throughput for the H2 application. The workload is the same as Table 2, making the statistics comparable with ZGC and Shenandoah. Note that when the heap configuration is more than 2×, *Jade* only has young GC, so we evaluate it with two tighter heap configurations: 1× and 1.2× of ZGC's minimum heap size. As for the smallest heap size, although GC threads are mostly active, each pre-reclamation cycle is much shorter (averaging 0.09s and 1.09s for young and old GC), which allows *Jade* to quickly reclaim memory for mutators. Meanwhile, the accumulated pause time contributes to less than 1% of overall execution time, while the average pause time is less than 1ms even with a tight heap size, which induces little interference on application latency. When the heap becomes larger, the pause is even shorter and does not restrict the application latency and throughput like ZGC and Shenandoah.

Table 6. GC-related statistics in *Jade*, including time for applications (**App.**) and different GC phases.

	Time	**1×**	**1.2×**	**1.5×**	**2×**
Total	App. (s)	20.28	18.15	16.37	16.22
	Mark (s)	17.68	2.41	0	0
	Build (s)	11.82	0.62	0	0
	Pause (s)	0.13	0.08	0.04	0.03
	Young Evac. (s)	6.87	7.20	6.89	6.44
	Old Evac. (s)	1.39	0.47	0	0
Avg.	Mark (s)	0.85	0.60	0	0
	Build (s)	0.24	0.62	0	0
	Pause (ms)	**0.72**	**0.71**	**0.55**	**0.66**
	Young Evac. (s)	0.09	0.13	0.18	0.26
	Old Evac. (s)	0.20	0.26	0	0
p99	**Pause (ms)**	**3.21**	**1.64**	**1.08**	**1.39**

CRDT. To show how CRDT helps reduce *Jade*'s concurrent GC time, we break down *Jade*'s GC cycle and compare it against G1, which uses region-wise remembered sets for its mixed collections (reclaiming both young and old generation) and thus also includes a concurrent marking and remembered set building phase. The workload is also the same as Figure 8. The results in Table 7 show that *Jade* improves the remembered set building time by 67.81%. This is mainly because CRDT reduces the number of cards to be scanned in the building phase by 64.63%. Meanwhile, although CRDT can introduce overhead in the marking phase, *Jade* still outperforms G1 by 24.95% in marking. The improvement mainly comes from *Jade*'s co-running design: during an old GC cycle, young GC threads can help by pushing young-to-old references into marking stacks. In contrast, since G1 conducts young GC in an STW fashion, it has to temporarily store those references in the live bitmap (similar to that in concurrent collectors), which needs rescanning in a future old marking cycle. Due to those two optimizations, *Jade* achieves a 40.48% improvement on the two concurrent phases together even compared with a high-throughput collector like G1, which further confirms *Jade*'s GC efficiency. As for the H2 benchmark analyzed before, CRDT also reduces the average number of scanned cards by 61.11%.

Table 7. Remembered set-related time breakdown in milliseconds, which mainly divides into two phases: marking (**Mark**) and remembered set re-building (**Build**).

Collectors	**Mark**	**Build**	**Total**	**No. of cards**
G1	1369.02	777.93	2146.95	1215774
Jade	1027.36	250.43	1277.78	430041

Chasing mode. Thanks to *Jade*'s efficient marking and collection phases, we do not observe application stalls in most configurations. Therefore, we run Specjbb2015 with high throughput (13,000 for 1.5× heap) for 15 minutes and find the average pause time introduced by application stalls is 40.05ms (p99 is 97.96ms), which suggests that *Jade* does not induce large pauses even under extreme configurations. Meanwhile, the average CPU utilization within the chasing mode is 90.75%, showing that *Jade* sufficiently leverages CPU resources when mutators are stalled.

6 Related work

6.1 Concurrent collectors

As applications' memory demands constantly grow, concurrent collectors are becoming popular to provide controlled GC pause time regardless of heap sizes. The Garbage-First (G1) collector introduces soft limits and a concurrent marking phase, but its evacuation phase is still stop-the-world. Pauseless GC [10] divides the collection into three phases (marking, relocating, remapping) and each phase allows co-execution with mutators, which inspires the design of today's concurrent collectors. C4 [33] extends Pauseless GC with a generational design while Collie [18] proposes to use hardware transactional memory (HTM) to atomically relocate objects. Compressor [20] also uses hand-over-hand compaction to retain low physical memory overhead, but its

reference calculation algorithm is costly. Block-free GC [26] introduces non-block handshakes for concurrent stack scanning and object copying. OpenJDK also introduced two concurrent copying collectors, Shenandoah and ZGC, which have been studied in this work. Cai et al. [7] also find ZGC and Shenandoah can introduce long pauses when under heavy workload, but they do not explore the design of collectors to explain those pauses. *Jade* summarizes the deficiencies inside existing concurrent collectors and provides group-based evacuation and single-phase young GC to improve GC efficiency and application performance.

6.2 Reference counting

In contrast to tracing collectors, reference counting (RC) collectors record incoming references for objects and can immediately reclaim them when the number reaches zero. Immediacy is an appealing feature in RC, but it also has two limitations: (1) the inability to handle cyclic references and (2) the large overhead for maintaining the per-object counter. The first one is the inherent limitation for RC, so prior work mainly focuses on optimizing the maintenance overhead. Biased reference counting (BRC) [9] observes most objects are only accessed by a single thread (namely *owner*) and thus allows the owner to modify the counters without atomic instructions. Deferred RC [14] introduces a collection phase to RC, which only focuses on objects updated since the last collection and updates those objects' counters. RCImmix [31, 32] further combines the deferred RC design with Immix's heap layout [3] to reach comparable performance with trace-based collectors. LXR [41] finds that RC can be elegantly integrated with the concurrent marking algorithm of G1 and thus proposes to still use RC-based STW pauses for both GC efficiency and low application latency. *Jade* instead focuses on improving the performance of concurrent collectors.

6.3 GC optimizations

Another line of work proposes optimizations to collectors so that they can be adapted to various scenarios. Yak [24] provides an epoch-based GC design for big-data applications while NG2C [4–6] pre-tenures long-lived objects for similar workloads. Yang et al. [40] provide NVM-friendly GC designs according to the bandwidth characteristics of non-volatile memory devices. Mako [23] and MemLiner [36] optimize the performance of concurrent GC on a far memory scenario. Our work mainly focuses on optimizing the performance of concurrent copying GC when under heavy workload and thus orthogonal to those prior efforts.

7 Conclusion

Garbage collectors (GC) are among the most important modules in language runtimes. Recent concurrent collectors claim to reach *pauseless* by allowing concurrent execution of mutators and GC threads, but they still induce long pauses when under heavy workloads. To this end, this work proposes *Jade*, which provides corresponding designs to improve GC efficiency and reduce the duration of pauses. The evaluation results show that *Jade* can significantly improve the peak application throughput while remaining comparable tail latency with mainstream concurrent collectors.

Acknowledgments

We sincerely thank our shepherd Martin Maas and the anonymous EuroSys'24 reviewers for their insightful comments and feedback. We also thank Wenyu Zhao for helping us evaluate LXR. This work was supported in part by the National Natural Science Foundation of China (No. 62202295, 62172272, 61925206), and in part by Alibaba Group through the Alibaba Innovative Research Program. Corresponding author: Liang Mao (maoliang.ml@alibaba-inc.com).

References

[1] Apache. Welcome to apache hbase. https://hbase.apache.org/, 2022.
[2] 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. The dacapo benchmarks: java benchmarking development and analysis. In *OOPSLA*, pages 169–190. ACM, 2006.
[3] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI*, pages 22–32. ACM, 2008.
[4] Rodrigo Bruno and Paulo Ferreira. POLM2: automatic profiling for object lifetime-aware memory management for hotspot big data applications. In *Middleware*, pages 147–160. ACM, 2017.
[5] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. NG2C: pretenuring garbage collection with dynamic generations for hotspot big data applications. In *ISMM*, pages 2–13. ACM, 2017.
[6] Rodrigo Bruno, Duarte Patrício, José Simão, Luís Veiga, and Paulo Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. In *EuroSys*, pages 28:1–28:16. ACM, 2019.
[7] Zixian Cai, Stephen M. Blackburn, Michael D. Bond, and Martin Maas. Distilling the real cost of production garbage collectors. In *ISPASS*, pages 46–57. IEEE, 2022.
[8] Maria Carpen-Amarie, Yaroslav Hayduk, Pascal Felber, Christof Fetzer, Gaël Thomas, and Dave Dice. Towards an efficient pauseless java GC with selective htm-based access barriers. In *ManLang*, pages 85–91. ACM, 2017.
[9] Jiho Choi, Thomas Shull, and Josep Torrellas. Biased reference counting: minimizing atomic operations in garbage collection. In *PACT*, pages 35:1–35:12. ACM, 2018.
[10] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In *VEE*, pages 46–56. ACM, 2005.
[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154. ACM, 2010.
[12] Standard Performance Evaluation Corporation. The specjbb2915 benchmark. https://www.spec.org/jbb2015/, 2021.
[13] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *ISMM*, pages 37–48. ACM, 2004.
[14] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526, 1976.

[15] Christine H. Flood, Roman Kennke, Andrew E. Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *PPPJ*, pages 13:1–13:9. ACM, 2016.

[16] Dacapo Group. The dacapo benchmark suite (chopin development). https://github.com/dacapobench/dacapobench/tree/dev-chopin, 2022.

[17] H2. H2 database engine. https://www.h2database.com/html/main.html, 2022.

[18] Balaji Iyengar, Gil Tene, Michael Wolf, and Edward F. Gehringer. The collie: a wait-free compacting collector. In *ISMM*, pages 85–96. ACM, 2012.

[19] Stefan Johansson. Gc progress from jdk 8 to jdk 17. https://kstefanj.github.io/2021/11/24/gc-progress-8-17.html, 2021.

[20] Haim Kermany and Erez Petrank. The compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 354–363, 2006.

[21] Chris Lattner and Vikram S. Adve. Transparent pointer compression for linked data structures. In *Memory System Performance*, pages 24–35. ACM, 2005.

[22] Per Lidén and Stefan Karlsson. The z garbage collector - low latency gc for openjdk. http://cr.openjdk.java.net/ pliden/slides/ZGC-Jfokus-2018.pdf, 2018.

[23] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D. Bond, Stephen M. Blackburn, Miryung Kim, and Guoqing Harry Xu. Mako: a low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *PLDI*, pages 92–107. ACM, 2022.

[24] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *OSDI*, pages 349–365. USENIX Association, 2016.

[25] OpenJDK. Zgc - the z garbage collector. https://openjdk.org/projects/zgc/, 2022.

[26] Erik Österlund and Welf Löwe. Block-free concurrent GC: stack scanning and copying. In *ISMM*, pages 1–12. ACM, 2016.

[27] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM*, pages 159–172. ACM, 2007.

[28] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *PLDI*, pages 33–44. ACM, 2008.

[29] Android Open Source Project. Art gc overview. https://source.android.com/docs/core/runtime/gc-debug#art_gc_overview, 2022.

[30] Thomas Schatzl. Java garbage collection: The 10-release evolution from jdk 8 to jdk 18. https://blogs.oracle.com/javamagazine/post/java-garbage-collectors-evolution, 2022.

[31] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. Down for the count? getting reference counting back in the ring. In *ISMM*, pages 73–84. ACM, 2012.

[32] Rifat Shahriyar, Stephen M. Blackburn, Xi Yang, and Kathryn S. McKinley. Taking off the gloves with reference counting immix. In *OOPSLA*, pages 93–110. ACM, 2013.

[33] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: the continuously concurrent compacting collector. In *ISMM*, pages 79–88. ACM, 2011.

[34] TPC. Tpc-c is an on-line transaction processing benchmark. https://www.tpc.org/tpcc/, 2022.

[35] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Software Development Environments (SDE)*, pages 157–167. ACM, 1984.

[36] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. Memliner: Lining up tracing and application for a far-memory-friendly runtime. In *OSDI*, pages 35–53. USENIX Association, 2022.

[37] wenyuzhao. Dacapo minheap values. https://gist.github.com/wenyuzhao/29e3e0e10bb68c4f2862851c874e0275, 2023.

[38] wenyuzhao. Incorrect heap usage reporting. https://github.com/mmtk/mmtk-openjdk/issues/270, 2024.

[39] Mingyu Wu, Ziming Zhao, Yanfei Yang, Haoyu Li, Haibo Chen, Binyu Zang, Haibing Guan, Sanhong Li, Chuansheng Lu, and Tongbao Zhang. Platinum: A cpu-efficient concurrent garbage collector for tail-reduction of interactive services. In *USENIX Annual Technical Conference*, pages 159–172. USENIX Association, 2020.

[40] Yanfei Yang, Mingyu Wu, Haibo Chen, and Binyu Zang. Bridging the performance gap for copy-based garbage collectors atop non-volatile memory. In *EuroSys*, pages 343–358. ACM, 2021.

[41] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. Low-latency, high-throughput garbage collection. In *PLDI*, pages 76–91. ACM, 2022.