BridgeGC: An Efficient Cross-Level Garbage Collector for Big Data Frameworks

YICHENG WANG, LIJIE XU*, Key Lab of System Software, State Key Laboratory of Computer Science, Institute of Software Chinese Academy of Sciences; University of Chinese Academy of Sciences, China
TIAN GUO, Worcester Polytechnic Institute, USA
WENSHENG DOU†, HONGBIN ZENG, WEI WANG†, JUN WEI†, and TAO HUANG, Key Lab of System Software, State Key Laboratory of Computer Science, Institute of Software Chinese Academy of Sciences; University of Chinese Academy of Sciences, China

Popular big data frameworks commonly run atop Java Virtual Machine (JVM), and rely on garbage collection (GC) mechanism to automatically allocate/reclaim in-memory objects. Existing garbage collectors are designed based on the hypothesis that most objects are short-lived. However, big data frameworks usually generate many long-lived data objects, which can cause heavy GC overhead. Recent approaches have reduced GC overhead in big data frameworks but still suffer from heavy human efforts, additional runtime overhead, or suboptimal GC efficiency.

This paper describes the design of `BridgeGC`, a big-data-friendly garbage collector that significantly reduces GC overhead introduced by long-lived data objects. `BridgeGC` follows a cross-level co-design. At the big data framework level, `BridgeGC` provides two annotations for framework developers to denote the creation and release of data objects. Based on the annotations, `BridgeGC` tracks the life cycles of annotated data objects and optimizes their allocation/reclamation at the GC level. At the GC level, we design a label-based allocator that stores data objects separately from other objects and balances their memory usage in the same JVM, leading to fewer GC cycles. We further design an efficient collector to eliminate unnecessary marking and copying of data objects during GC cycles, lowering the GC time. We have integrated `BridgeGC` into OpenJDK ZGC. The extensive evaluation, using two popular big data frameworks (Flink and Spark) and a key-value database (Cassandra), shows that `BridgeGC` achieves 31%-82% GC time reduction compared to the baseline ZGC. `BridgeGC` also outperforms other traditional and academic garbage collectors in end-to-end performance.

CCS Concepts: • **Software and its engineering → Garbage collection**; *Runtime environments*; • **Information systems → Data management systems**.

Additional Key Words and Phrases: Garbage collector, big data framework, memory management

1 Introduction

Big data frameworks, e.g., Apache Spark [8] and Apache Flink [1], are widely used for big data processing and analytics. These frameworks rely on garbage collectors to manage (i.e., allocate and reclaim) a large number of objects generated during data processing. Garbage collectors free big data frameworks from explicitly managing in-memory objects. However, big data applications often suffer from *heavy GC overhead*, up to 50% of the total execution time [18, 43], due to non-optimal memory management of the large number of objects generated by

*Corresponding author.
†Affiliated with Nanjing Institute of Software Technology, University of Chinese Academy of Sciences, Nanjing, China.

ACM 1544-3973/2025/3-ART
https://doi.org/XXXXXXX.XXXXXXX

big data frameworks [18, 35, 37, 38, 42, 45, 57]. Therefore, improving GC efficiency is considered a key point for improving overall framework performance [12, 16, 17, 24, 29, 31, 38, 41, 42, 45, 49].

Gap between frameworks and garbage collectors The GC inefficiency is essentially caused by the *information gap* between big data frameworks and garbage collectors—the memory usage pattern of big data frameworks mismatches the design hypothesis of traditional garbage collectors. Big data frameworks exhibit a unique memory usage pattern of *numerous long-lived data objects* [57]. That is, modern frameworks typically persist large volumes of data in memory to speed up application execution. This data consists of a large number of small records, with each record typically stored as a separate object in memory. As a result, the large volumes of data processed by frameworks lead to numerous data objects. These data objects have different life cycles but are generally long-lived, for the data is kept in memory long-term. Moreover, the volume of these long-lived data objects make up the majority of objects generated by frameworks [18, 24, 38, 41].

However, traditional garbage collectors manage objects in the JVM heap based on *weak generational hypothesis*, assuming that most objects survive for a short period. For long-lived data objects that survive multiple GC cycles, these collectors repeatedly copy them within the young generation before eventually copying and promoting them to the old generation, leading to significant copying overhead. Moreover, the old generation can quickly be filled with these objects, triggering frequent GC cycles targeting the whole heap. During these GC cycles, long-lived data objects are repeatedly marked and copied to compact memory. However, these data objects cannot be reclaimed for a long time, resulting in significant but unnecessary marking and copying overhead.

To alleviate the GC inefficiency problem, framework developers have proposed memory optimizations [2, 15, 27, 31, 33, 45, 58]. However, a significant number and volume of data objects will still be generated during large-scale data processing, hence these optimizations cannot change the *numerous long-lived data objects* memory usage pattern, the root cause of GC inefficiency. Meanwhile, new GC-level optimizations that target general Java applications [10, 11, 54, 56] are still designed for applications that assume *weak generational hypothesis* and are therefore non-optimal for big data frameworks. Some garbage collectors use reference counting algorithms to avoid repeatedly scanning long-lived objects [20, 44, 64]. While this approach reduces marking overhead, it still incurs copying overhead, as long-lived objects must be relocated during memory compaction to avoid fragmentation. Additionally, these collectors allocate a counter for each object, leading to substantial memory overhead and performance degradation when managing the numerous data objects generated by big data frameworks. In short, intra-level optimizations, i.e., targeted *solely* at the framework or GC level, are inadequate to bridge the information gap between frameworks and garbage collectors.

Challenges and limitations of existing approaches To narrow the gap, a promising idea adopted by existing big-data-friendly GC is to manage long-lived data objects differently [17, 21, 24, 38, 40–42]. To this end, an ideal big-data-friendly GC should satisfy three requirements: (1) **precise**, being able to precisely identify long-lived data objects and obtain their lifetimes, which are the basis of GC optimization for data objects; (2) **easy-to-use**, requiring limited human efforts; (3) **efficient**, properly managing long-lived data objects without incurring throughput and memory overhead.

Unfortunately, existing works cannot fulfill *all* these requirements simultaneously. For example, the epoch-based approach [24, 40, 41], represented by Yak [41], treats all objects created within code epochs as data objects, which could mistake short-lived objects allocated within code epochs as long-lived data objects, leading to suboptimal GC efficiency and memory waste. The profiling-based approach [14, 16, 17], represented by ROLP[17], allocates the profiled long-lived objects together, which cannot obtain the life end point of objects. Hence it still suffers from unnecessary marking costs when determining the reclamation timing of objects. The region-based data object management [38, 41, 42], represented by Facade [42], allocates data objects into special *data space* on heap or off-heap, which requires a manual partition of *data space* to utilize memory properly, i.e., an inflexible, hard boundary. The copying-based data object management [28–30], represented by TeraHeap [28], separates

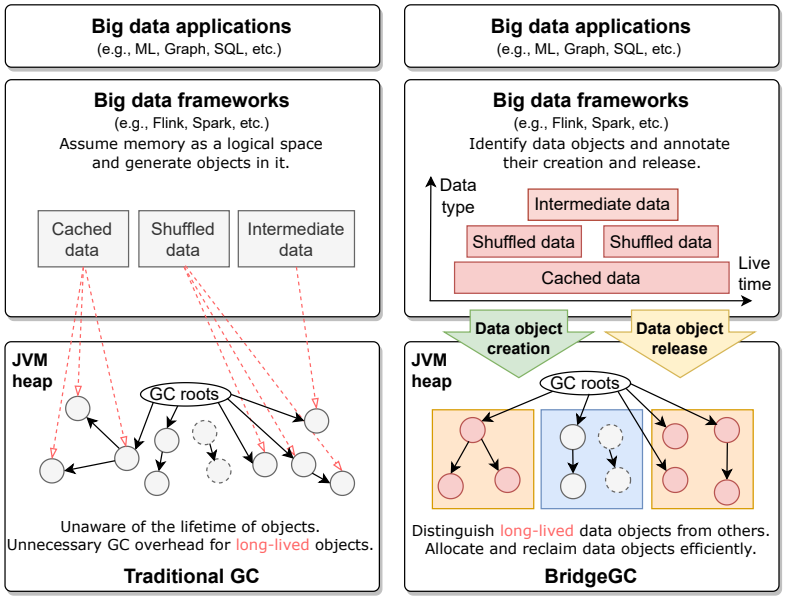

Fig. 1. Comparison between BridgeGC and traditional GC.

the long-lived objects referenced by key objects to a second heap during GC cycles, which still requires copying overhead thus more suitable for hybrid memory architectures. Besides, existing works suffer from additional throughput overhead (~6%) [17] and memory overhead (e.g., ~12.2%) [41].

Our approach We propose BridgeGC, a garbage collector that adopts a *cross-level co-design* between the big data framework and the garbage collector to narrow the gap. As shown in Figure 1, BridgeGC employs limited manual annotations offline at the exact creation/release points of data objects to profile their life cycles at the *framework level*. Then, at the *GC level*, BridgeGC leverages the life cycle information of these annotated data objects online to allocate and reclaim them in the JVM heap without unnecessary marking/copying overhead.

To implement our cross-level co-design that fulfills all the requirements above, we design three components in BridgeGC. (1) A *data object profiler* that provides two annotations for framework developers to annotate data objects, through which BridgeGC can obtain their **precise** life cycles (allocation and release time) at the GC level. As data objects in modern frameworks are abstracted as special types sharing centralized life cycles, our annotations can be **easily used** to instrument framework codes. (2) A **memory-efficient** *label-based allocator* that separates the storage of data objects and normal objects into data and normal pages to avoid memory waste, and tackles the problem of space imbalance with dynamic page allocation. To distinguish data objects quickly at the GC level, the allocator labels them using colored pointers. (3) A *marking/copying-conservation collector* that skips marking labeled data objects and excludes data pages from reclamation in GC cycles where data objects are known to be live. The collector **effectively** reclaims data pages only after a batch of data objects is released at the framework level.

We integrate BridgeGC into **ZGC** in OpenJDK 17, which is the state-of-the-art GC in OpenJDK [61], and is officially recommended by modern big data frameworks due to its efficiency in managing large heap [9]. To the best of our knowledge, BridgeGC is the first big-data-friendly GC integrated with ZGC, and BridgeGC has been

open-sourced.[1] BridgeGC extends the heap layout of ZGC for our object allocation and improves the barrier and colored pointers of ZGC for our object reclamation without introducing additional runtime overhead. We conduct an evaluation of BridgeGC's performance with two popular big data frameworks Apache Flink and Apache Spark, as well as a key-value database Apache Cassandra. We compare BridgeGC with baseline ZGC and other garbage collectors in OpenJDK, including Shenandoah, G1, and Parallel. We also compare BridgeGC with other solutions, including *off-heap* implementation of frameworks and two state-of-the-art research work ROLP [17] and LXR [64]. Based on our evaluation of representative big data workloads, we observe that BridgeGC reduces GC time by 41%-82% for Flink, 31%-46% for Spark, 33%-47% for Cassandra compared to ZGC, and outperforms all evaluated collectors in end-to-end performance.

In short, we make the following three main contributions.

- We design a new garbage collector BridgeGC to efficiently manage data objects generated by big data frameworks. BridgeGC follows a cross-level co-design and significantly reduces the GC overhead for big data applications.
- We integrate BridgeGC with advanced garbage collector ZGC in OpenJDK. BridgeGC has been open-sourced which could be easily applied to real-world big data systems without requiring application developer efforts.
- We perform an extensive evaluation of BridgeGC using two popular frameworks Flink and Spark with commonly used big data workloads. BridgeGC achieves significant GC improvement compared to baseline ZGC and outperforms both traditional garbage collectors in OpenJDK and other research works.

We organize the paper as follows. In §2, we introduce the GC overhead in big data frameworks and existing optimizations. §3 presents the BridgeGC design and its components. In §4, we describe how BridgeGC is implemented in OpenJDK. We evaluate BridgeGC's performance and compare it with other solutions in §5. We review related work in §6 and discuss generality and required efforts of BridgeGC in §7. Finally, we conclude with our contributions in §8.

2 GC and Existing Optimization

2.1 GC Overhead in Big Data Frameworks

When big data frameworks generate in-memory objects during data processing, these objects are allocated by the JVM in the heap memory. Once specific GC conditions are met, such as heap usage exceeding a certain threshold, the garbage collector in the JVM starts a GC cycle to reclaim the dead (unreferenced) objects. A GC cycle in traditional collectors mainly contains two time-consuming phases, *tracing* and *reclamation*. In the tracing phase, the collectors **mark the lived objects** through reachability analysis, which starts from GC roots, as shown in Figure 1, and traverses the *object reference graph*. In the reclamation phase, the collectors generally **copy marked live objects** to another heap space, sweep the dead objects, and compact heap space. During these phases, the collector inevitably needs to pause the application to obtain consistent reference graph or object locations, referred to as *GC pause time*. To reduce GC time, traditional garbage collectors use the *generational* GC strategy to collect a subset of the heap only in most GC cycles—they divide heap space into a *young generation* for newly created objects that are frequently collected, and a *old generation* for long-lived objects that are occasionally collected. Objects in the young generation that survive several GC cycles are promoted (copied) to the old generation. This strategy is efficient for general Java applications that satisfy weak generational hypothesis, where most objects are short-lived and can be quickly reclaimed in the young generation with a small cost.

However, as reported in previous studies [18, 24, 57], big data frameworks can generate millions of long-lived objects related to cache data and intermediate data, that will occupy large memory space and trigger frequent GC cycles. To understand the GC overhead in big data frameworks, we perform a case study of *PageRank* application

[1] https://github.com/BridgeGC/BridgeGC

in Hibench [25] executed in Apache Spark, using 32 GB heap for each executor. According to the Spark log, the application runs ~210 seconds with ~380 GC cycles. Objects related to cached data survive in memory for ~180 seconds (over 320 GC cycles), and objects related to intermediate data, such as shuffled data, survive ~20 seconds (over 35 GC cycles) before being released. Since an object will be promoted to the old generation in at most 15 GC cycles, these data objects are considered long-lived.

From the GC logs, we further observe that the survivor rate of objects in the young generation reached 28%-62% during minor/young GCs, significantly higher than the typical 0%-5% seen in conventional generational applications [26, 64]. Additionally, 29% of total objects (~ 90 GB) were eventually promoted to the old generation. We further analyze the demographics of objects that lived after the full GC cycles. We found objects related to cache data (e.g., chunked byte buffer) and shuffle data (e.g., memory block) accounted for more than 61% in number and 78% in total size. This indicates that long-lived objects in big data frameworks are mostly data objects, they take significant memory space in the old generation. Moreover, due to the accumulation of these long-lived data objects in the old generation, a significant number of GC cycles targeting the entire heap are triggered, but only 27% of these long-lived objects are reclaimed in these GC cycles. Therefore, long-lived data objects may experience multiple marking and copying during whole heap GC cycles, leading to unnecessary GC overhead since they cannot be reclaimed. Only after the framework releases these objects such as releasing the cached data, these objects can be reclaimed in the following whole heap GC cycle. Therefore, how to eliminate the unnecessary marking/copying overhead spent on long-lived objects is the key problem to solve in this paper.

2.2 Framework-Level Optimization

Since GC (marking/copying) overhead is related to the number and volume of lived objects, recent frameworks adopt the **data object abstraction** strategy to reduce the number of data objects. The key idea is to store data in binary form instead of Java objects and merge them into unified abstracted data objects. For example, Flink [27] serializes data ($\langle K, V \rangle$ records) into arrays of primitive type (e.g., byte[]), and each array is referenced by an abstracted data object *MemorySegment*. Spark [2] and other big data systems [5, 15, 18, 45, 45] also adopt similar optimizations. Although this strategy can reduce the number of $\langle K, V \rangle$ data objects, the number of abstracted data objects is still large for many big data applications. Moreover, this strategy does not change the GC behavior, so it can only reduce but cannot eliminate the unnecessary GC overhead.

Furthermore, frameworks can store the (abstracted) data objects in off-heap memory [4, 7, 23, 38, 42], which is not in the management scope of garbage collectors. Using off-heap memory, frameworks can directly allocate/reclaim data. However, off-heap memory is unsafe and slower than on-heap memory [23]. Moreover, configuring the appropriate size of off-heap memory is challenging. Improper configuration may result in memory waste, as off-heap memory cannot be used for normal objects. More importantly, frameworks still need to keep header objects in the heap to access the off-heap data objects, hence still causing GC overhead [16].

While data object abstraction can only partially alleviate GC overhead, its widespread adoption in modern big data frameworks has **centralized** the data objects generated by frameworks. Specifically, these data objects have unified types, common creation/release paths, and simple, centralized reference relationships (i.e., data objects rarely reference normal objects), which presents an opportunity for more efficient centralized management.

2.3 GC-Level Optimization

Concurrent GC algorithms. To reduce GC pause time described in §2.1, the OpenJDK community proposed mostly-concurrent garbage collectors such as ZGC [11] and Shenandoah GC [10] algorithms. Academic researchers also proposed the LXR [64] algorithm that combines concurrent marking and reference counting, which can manage long-lived objects at a lower cost. These garbage collectors perform partial or most GC work concurrently with the application, leading to shorter GC pause times. However, previous study [57] showed that concurrent GC

work could slow down the execution of CPU-intensive big data applications, indicating both **concurrent GC time** (time span of concurrent GC works) and *GC pause time* are important GC metrics. In addition, mostly-concurrent collectors introduce **load barrier**, which is a piece of code executed before every load of an object reference. Taking the large number of data objects in big data applications into account, the heavy overhead of the load barrier may also slow down the application. Besides, mostly-concurrent collectors may suffer from allocation stalls when the object allocation speed in big data applications exceeds the object collection speed. In summary, existing concurrent GC algorithms do not work well for big data frameworks either.

Big-data-friendly GC algorithms. Researchers tried to optimize traditional collectors to adapt to big data frameworks [17, 41]. For example, Yak [41] optimized the *Parallel GC* in OpenJDK 8. It first requires manual efforts to label epochs (i.e., data operations start/end), then it treats all objects within epochs as data objects and allocates them in *data space*. However, we observe that long-lived data objects and short-lived normal objects are created simultaneously, hence Yak can mistakenly allocate short-lived objects into *data space*, leading to memory fragmentation. Yak inserts extra bytes in object headers to distinguish data objects, which also increases memory overhead. Meanwhile, Yak requires a hard boundary for *data space*, making it hard to balance the memory partition. TeraHeap [28, 29] optimized the *Parallel GC* in OpenJDK 8 and OpenJDK 17, it separates the long-lived data objects from normal objects by relocating the long-lived objects to a second high-capacity heap H2 region during full GC cycles, and eliminate the GC cost for H2 region. The separation process in TeraHeap still requires object copying overhead, making it more suitable for hybrid memory architectures with a high-capacity NVMe device. The ROLP [17] and its related works [14, 16] optimized *G1 GC* in OpenJDK 8. They add more generations for objects with different life cycles. Those objects with a common life cycle are allocated in the same generation, which reduces the overhead of object promotion. However, ROLP leverages dynamic profiling to obtain object life cycles, leading to throughput overhead. Moreover, profiling cannot determine when the objects become reclaimable, they still rely on repeated marking to determine the timing of reclamation. In summary, existing solutions are implemented based on traditional collectors in older OpenJDK versions and suffer from additional **memory and throughput overhead**.

3 `BridgeGC` Design

Inspired by previous work, we propose `BridgeGC`, a cross-level garbage collector tailored for big data frameworks. Here, by *cross-level*, we refer to annotating the creation and release points of long-lived data objects offline at the framework level and leverage life cycle information profiled from annotations online to facilitate the management of them at the GC level. Figure 2 presents an overview of `BridgeGC`. Our design has the key benefit of saving the unnecessary marking and copying of data objects without introducing extra memory overhead, throughput overhead, or extensive human efforts. In designing `BridgeGC`, we answer the following key questions.

- **How to precisely identify data objects and track their life cycles?** As illustrated in §1, existing approaches have difficulty obtaining accurate information critical to GC of data objects. We infer data objects by leveraging advances in modern frameworks that abstract data objects as specific types, and we instrument their creation via annotations at the framework level. To track these annotated data objects' complete life cycles, we also instrument their common release points, at which some data objects become reclaimable. Since this work is done at the framework level, we can achieve lightweight tracking of data objects without application developers' efforts. We explain these features in §3.1.
- **How to store data objects in memory to improve GC efficiency?** As data objects and normal objects are generated simultaneously while having different life cycles, it is critical but challenging to allocate them in the *proper* place, aiming to minimize object copying while keeping memory efficient. Our solution is a label-based allocator, which splits the heap into equal-sized normal/data pages for the allocation of normal/data objects

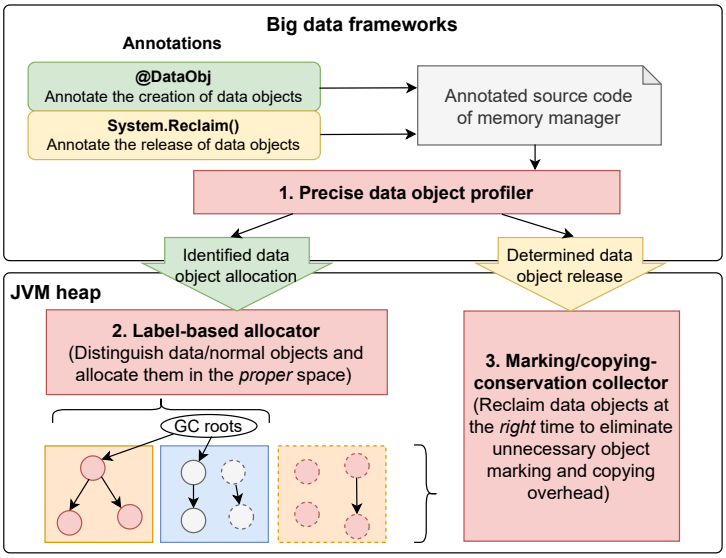

Fig. 2. BridgeGC overview.

and dynamically balances their spaces. The allocator labels the data objects at a small cost, which will facilitate their collection. We illustrate the details of object storage and labeling in §3.2.

- **When and how to reclaim data objects effectively?** As data objects are long-lived and properly placed in JVM heap until being released by the framework, marking/relocating them before that is unnecessary. We propose a marking/copying-conservation collector to eliminate these overheads, which distinguishes data objects through their labels and excludes them in the GC cycles in which they are known to be alive. The collector reclaims data objects at the *right* time, during which survived data objects are marked, and pages without survived objects could be reclaimed. We illustrate the details of object reclamation in §3.3.

3.1 Annotation-Based Data Object Profiler

To facilitate the GC of long-lived data objects, we design a profiler to track data objects' life cycles. We propose two annotations that the framework developers can use to denote the creation and release of data objects in the source code of frameworks. With these object-level annotations, we can not only distinguish the creations of data objects from that of normal objects, but also capture the lifetimes of data objects. Our annotations are designed as follows.

- **@DataObj**: The annotation `@DataObj` can be used along with the keyword `new` in the framework memory management code to denote the creation of long-lived data objects (e.g., creation of cached/shuffled data). Different from annotating a whole class or code epochs, our allocation-based annotation can precisely inform the garbage collector of the data object creation when JVM parses the annotation at runtime.
- **Reclaim()**: The interface `System.Reclaim()` is used to denote the release of data objects at the framework level. `Reclaim()` is invoked to inform the garbage collector that a batch of data objects is reclaimable, as data objects used for the same data operation generally share a common release point at the end of the operation. After JVM processes the `Reclaim()` at runtime, the collector will try to reclaim the released data objects.

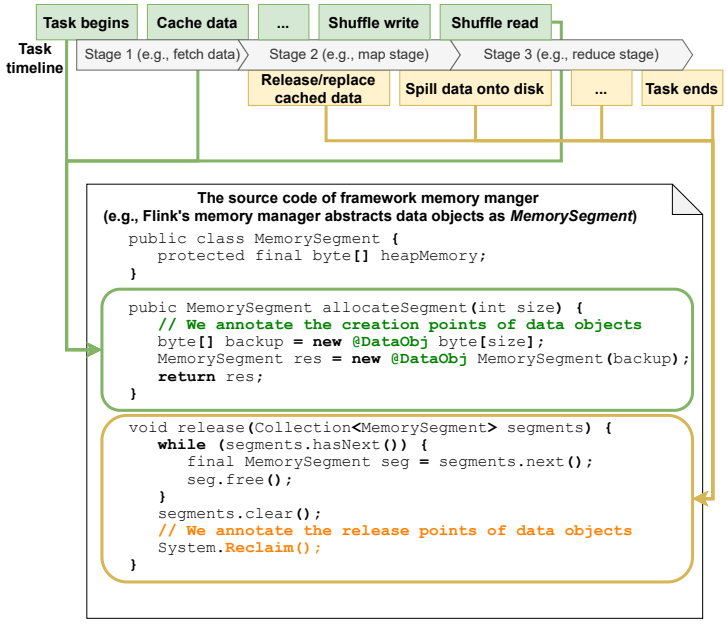

Fig. 3. Example of framework-level annotations in Flink.

We have applied our annotations to popular frameworks Apache Flink and Spark. Two characteristics of modern frameworks simplify the annotation efforts to cover most data objects. First, as described in §2.2, data objects in modern frameworks are abstracted into specific types which can be clearly inferred. For example, Flink provides an object type `MemorySegment` to host in-memory $\langle K, V \rangle$ data records, making it the main type of data object in Flink. Second, modern frameworks exhibit clear processing phases when executing applications, they centrally create/release data objects at the beginning and end of phases by invoking specific creation/release functions. For instance, when executing ML applications, Flink invokes the creation function (e.g., `allocateSegment(...)` in Figure 3) to create data objects for training data; after training, Flink releases data objects collectively by invoking the release function (e.g., `release(...)` in Figure 3).

Hence, when we annotate the creation of `MemorySegment` object and the byte array it owns inside the function `allocateSegment(...)`, it is sufficient to cover the creation of most long-lived data objects in Flink. Also, when we invoke `Reclaim()` inside the release function of `MemorySegment` after the framework frees a batch of them explicitly, it is sufficient to annotate all dead points of data objects related to `MemorySegment`.

Similarly, we can apply these annotations to other big data frameworks like Apache Spark, which also has abstracted data objects `MemoryBlock` used for aggregation and shuffle operations, and chunked `ByteBuffer` used for serialized cached RDD. As the example shown in Figure 4a, we annotate these data objects in their corresponding creation/release functions.[2]

Furthermore, our annotations can be applied to a broader range of big data systems that also generate numerous long-lived data objects, like key-value database Cassandra [3] and HBase [6]. Figure 4b shows the example of Cassandra, we annotate the `ByteBuffers` and `BufferCells` in `Memtable`, which buffer the recent writes and are generally long-lived. For systems and general Java applications without well-wrapped long-lived data objects, `BridgeGC` can also be used with a deeper understanding of the code logic and structure.

[2]Some of the annotations are applied in Java Standard Library.

Management of chunked ByteBuffer in Spark

```
public class ByteBuffer{
   final byte[] hb;
}

public ByteBuffer allocate(int size, OpOrder.Group opGroup){
   // creation points of data objects
   byte[] buf = new @DataObj byte[size];
   HeapByteBuffer result = new @DataObj HeapByteBuffer(buf, size);
   return result;
}

public void removeBlockInternal(blockId: BlockId){
   memoryStore.remove(blockId);
   blockInfoManager.removeBlock(blockId);
   // Release point of data objects
   System.Reclaim();
}
```

(a) ByteBuffer for Spark

Management of BufferCell in Cassandra

```
public class BufferCell{
   final ByteBuffer value;
}

public Cell<?> clone(ByteBufferCloner cloner){
   // creation points of data objects
   byte[] buf = new @DataObj byte[size];
   HeapByteBuffer buffer = new @DataObj HeapByteBuffer(buf, size);
   return new @DataObj BufferCell(buffer);
}

public void flushMemtable(Memtable memtable){
   cfs.replaceFlushed(memtable, sstables);
   memtable.discard();
   // Release point of data objects
   System.Reclaim();
}
```

(b) BufferCell for Cassandra

Fig. 4. Example of framework-level annotations in Spark and Cassandra.

Our proposed annotation method incurs limited framework developer efforts (e.g., with only 10 LoC modifications for each type of data object) and is transparent to application developers/users. We describe how JVM parses the annotations and obtains object life cycles in §4.1.

3.2 Memory-Efficient Label-Based Allocator

Our profiler described above distinguishes between the creation of data objects and normal objects. This section describes our design of the allocator that properly stores data objects in the JVM heap at the allocation to avoid unnecessary copying. Recall that data objects and normal objects have different life cycles, allocating them together can lead to internal fragmentation because normal objects are often short-lived. To handle the fragmentation, the long-lived data objects may be relocated to compact memory, introducing object copying.

To improve memory efficiency, we adopt a *page-based heap layout* and explicitly label the pages as one of the two types: the data page and the normal page. As the overview of our heap design presented in Figure 5, the heap is divided into fixed-sized pages and maintained as a pool of memory pages to serve the allocation/free requests for data and normal pages. The data page is exclusively used for storing data objects, and the normal page is

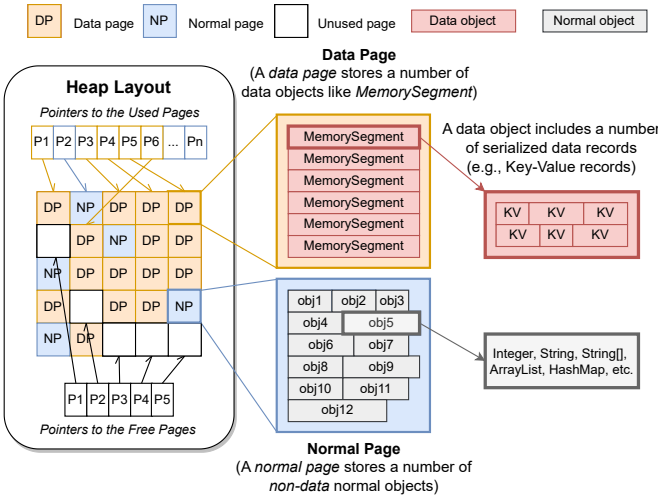

Fig. 5. An overview of BridgeGC heap layout design.

used for normal objects. The shared page pool reduces potential memory waste caused by unbalanced memory separation for data objects and normal objects, as data and normal pages are allocated only when requested.

To further improve the lifetime similarity of objects in the same data page, BridgeGC allocates one data page to a single thread only. Because data objects used for the same task generally share the same lifetimes and one task in a framework is processed by one thread, they are more likely to be released simultaneously, which provides the opportunity to reclaim them as a whole.

Additionally, we label all data objects to enable quick distinction of them during GC cycles. This allows us to skip the time-consuming process of inferring a data object through the label of the page it lies on. Instead of labeling data objects through additional bytes in the object header, which incurs memory overhead, we label data objects with *the colored pointer*. BridgeGC can therefore infer a data object using a single logic operation on its pointer without extra memory access. We describe the workflow of object allocation in §4.2.

3.3 Marking/Copying-Conservation Collector

This section describes how BridgeGC leverages the labeled data objects/pages to reduce the object marking and copying overhead during GC cycles. At the high level, based on the life cycle information of data objects provided by the annotations, BridgeGC skip marking and copying data objects until the framework releases them. Specifically, when GC conditions (e.g., high memory usage) are met, but not annotated data objects are released, our collector only reclaims the objects in normal pages, referred to as *normal GC*. Once annotated data objects become reclaimable, BridgeGC tries to reclaim both data and normal pages in the upcoming GC cycle, referred to as *full GC*. Figure 6 shows an example workflow of these two types of GC cycles.

Normal GC behavior. In normal GC cycles, our collector skips marking data objects and reclaiming data pages. This can be easily achieved because BridgeGC has labeled data objects and data pages in the allocation step. Specifically, our collector will stop tracing the object graph once a reference is found pointing to a data object. For example, as shown in Step 3 of normal GC in Figure 6, the GC marking threads will stop marking once it finds the referents of HashMap and ArrayList objects are labeled as data objects, which eliminating marking overhead of data objects related to MemorySegments. This stopping condition is based on the observation that data objects are centralized in reference relationship, they often point to other data objects only (e.g., MemorySegment reference byte[]), instead of normal objects. While to guarantee safety, we also propose a *write barrier* (detailed in §4.3) to

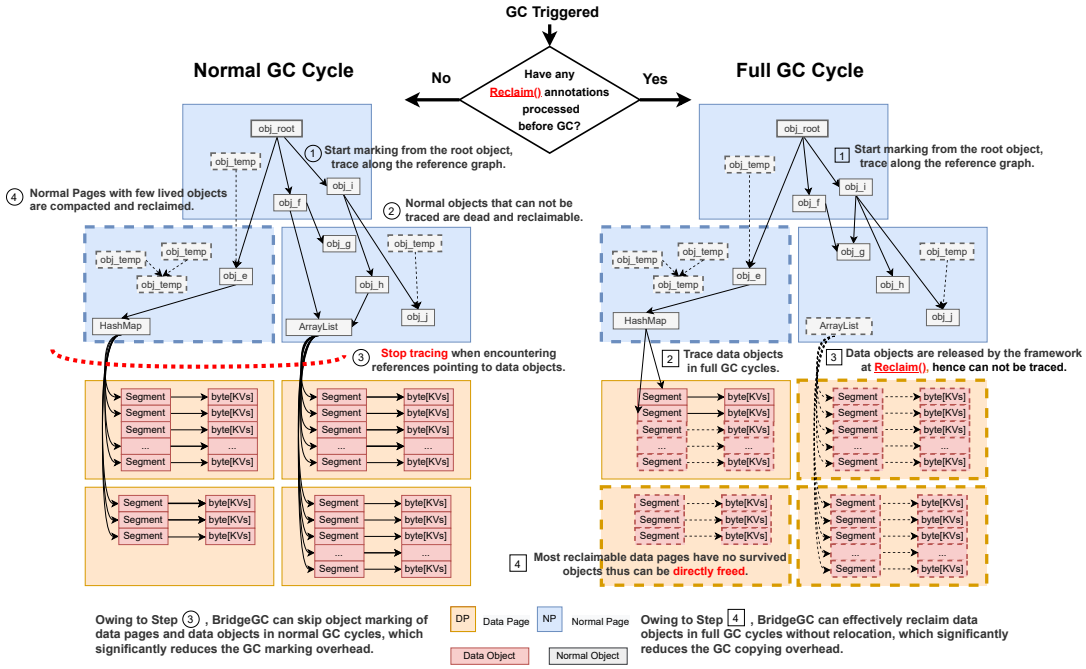

Fig. 6. Workflow of normal GC and full GC.

record all the references pointing from data objects to normal objects, and mark the recorded references in the normal GC cycles to avoid reclaiming lived normal objects. Additionally, the collector excludes all data pages from the candidate reclamation set, so no data object will be relocated in normal GC cycles.

Full GC triggering conditions. To determine when to perform full GC, the basic rule is to check whether `Reclaim()` is invoked during the last GC interval, which explicitly indicates that a batch of data objects is reclaimable. We also provide an additional *checker* to deal with the cases in which annotation `Reclaim()` can not be properly placed. Specifically, in the tracing phase of each normal GC cycle, the checker maintains a global list of *dominant* normal objects at which tracing is stopped (e.g., `HashMap` and `ArrayList` objects in Figure 6), and checks their liveness in the upcoming GC cycle. As the liveness of dominant normal objects determines the liveness of the batch of data objects they are directly or indirectly referencing, if any of these dominant objects are not marked lived in the current GC cycle, the checker concludes that a batch of data objects are reclaimable and will perform full GC in the next GC cycle.

Full GC behavior. When full GC cycles are performed, the collector can reclaim the data pages without surviving data objects. For example, Figure 6 shows that data pages comprised of all dead `MemorySegments` are directly reclaimed after marking. This whole data page reclamation eliminates the object copying overhead and the harm to the locality during the memory compaction, and is partially enabled by our allocator design. That is, the data objects located on the same data page are allocated for the same task, sharing the same release points, hence the collector can reclaim the entire data page when the task is finished. Additionally, we will not skip the data object marking during full GC cycles because not all data objects are guaranteed dead. Essentially, the collector will mark over the entire object graph to ensure correctness.

The collector uses the traditional GC algorithm for normal objects in both types of GC cycles. This is because normal objects still follow the *weak generational hypothesis*, and we can manage them with modest overhead.

4 Implementation

We implement BridgeGC in OpenJDK 17 HotSpot JVM. Theoretically, BridgeGC can be implemented on every collector in OpenJDK like Parallel, G1, and ZGC, for all of them follow the mark/copy steps and we can eliminate the unnecessary GC efforts in these steps. We decided to integrate BridgeGC to ZGC because it is the latest advanced GC in OpenJDK [61] and is officially recommended by the modern big data framework [9], it has native support for page-based layout, ideal performance for normal object collection, and colored pointers that can be improved to distinguish data objects with minimal memory and throughput overhead.

Next, we detail how we implement BridgeGC atop ZGC. Our main improvements include: (1) In the class parsing phase, BridgeGC implements the parsing logic of annotations for the creation/release of data objects (§4.1). (2) In the object allocation phase, BridgeGC separates the data/normal pages with a memory-efficient allocation process and expands the colored pointers of objects for object labeling (§4.2). (3) In the GC phase, BridgeGC updates the mark barrier to skip the marking of annotated objects in normal GC, improves the load barrier to speed up the data objects loading, and introduces the write barrier to ensure the safety (§4.3).

4.1 Annotation Processing in JVM

To enable JVM to receive the annotation of data object creation, BridgeGC lets the class parser of JVM keep the creation annotations at the metadata of the method descriptor, and checks the metadata of the current method descriptor when processing the *new* keyword, allocates data object when annotation is found. When bytecodes are further compiled to native code by C1 and C2 Just-in-Time (JIT) compilers due to frequent execution, BridgeGC binds the direction of an allocation site to data object allocation if it is annotated. Considering that the annotations are applied to a few codes only, and most allocations are done by compiled native code, the performance impact of creation annotation interception is trivial.

To enable data objects release annotation, BridgeGC creates a signal `Full` in JVM to determine whether to perform full GC in the next triggered GC cycle. Specifically, when the `System.Reclaim()` method is invoked, BridgeGC sets `Full` to be *true*, which indicates that the next triggered GC will be performed as a full GC cycle. At the end of the full GC cycle's marking phase, BridgeGC resets the `Full` signal to *false*, and any subsequently invoked `Reclaim()` will lead to another full GC cycle. Compared with performing full GC right after `Reclaim()` invoked, using `Full` signal avoids repeated full GC resulting from continuous release annotations.

4.2 Workflow of Data Object Allocation

Data page allocation. BridgeGC inherits the page-based heap layout of ZGC and dynamically allocates data pages for data object allocation. For each page in BridgeGC, we add a flag to indicate its type, the flag is set to true when pages are allocated as data pages and set to false for normal pages and reclaimed pages. Furthermore, pools of available pages are maintained, when the current page lacks sufficient free memory to satisfy the current allocation, a new page of the same type is allocated from the page pool. Dynamic data page allocation helps balance the memory usage of data objects and the overhead is minimal, because the number of pages is significantly smaller than the number of objects, and modifications to the metadata are infrequent.

TLAB allocation. Beneath page level, every application thread in ZGC holds a *Thread Local Allocation Buffer (TLAB)* to accommodate object allocation locally first. To handle the allocations of data objects, we assign another TLAB to each application thread besides the one used for normal objects. Unlike normal TLAB which allocates small memory pieces from a common normal page, TLAB for data objects allocates a whole data page at a time, this ensures each data page is exclusively used by one application thread, resulting in data objects allocated on the same data page having similar life cycles, consistent with our design in §3.2. Meanwhile, bigger TLAB also reduces the memory waste caused by TLAB replacement. Because data objects are big in size, bigger TLAB for data objects leads to less TLAB replacement and less waste of remaining memory in old TLAB.

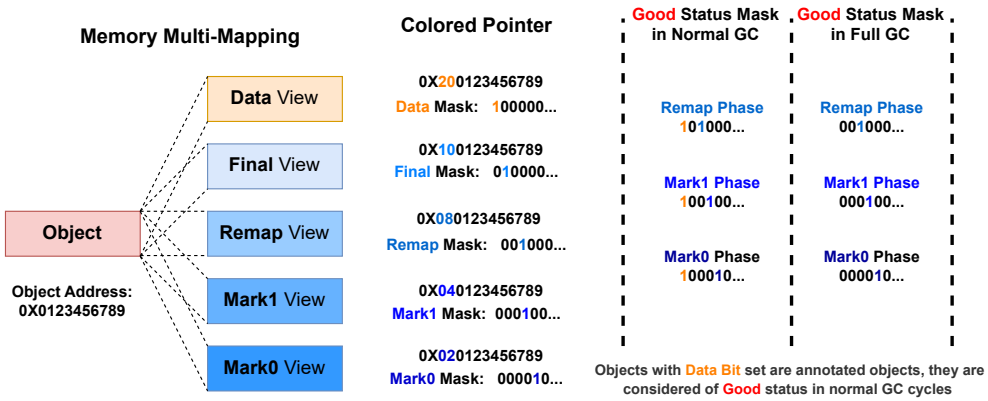

Fig. 7. BridgeGC expands colored pointer of ZGC, uses an extra metadata bit to label data objects.

Data object label. ZGC utilizes multi-mapped memory to add 4 metadata bits in a pointer without changing which object it points to, and leverages these 4 bits to represent 4 *colors* (status) of an object [46]. As shown in Figure 7, BridgeGC uses an extra metadata bit to represent a newly created *Data (D)* status, used to label data objects in the JVM heap. With the colored pointer, data objects can be inferred by a single logic *AND* operation between the object pointer and address mask of *Data* without incurring additional access overhead. As the extra bit for data objects is achieved by one more heap memory mapping, no additional memory overhead is needed.

4.3 Mark and Load/Store of Data Objects

BridgeGC achieves different marking policies of data objects in normal GC cycles and full GC cycles by modifying the *mark barrier* in ZGC, and it modifies the *load barrier* to achieve a fast load of data objects. In addition, BridgeGC proposes a new *write barrier* to record the references from data objects to normal objects. We implement the modification of these barriers both in the interpreter and the JIT compilers of JVM.

Mark barrier modification. The *mark barrier* in ZGC is executed before an object is pushed into mark stacks for further tracing. In normal GC cycles, BridgeGC intercept data objects in the *mark barrier* by comparing their pointer with the *Data Mask*. For intercepted data objects, BridgeGC stops pushing them into mark stacks for further tracing. Meanwhile, to support the *checker* described in §3.3, the mark barrier records the dominant normal object at which markings are stopped, and appends it to the global list of the dominant normal objects if it is not already present. After the entire marking phase, BridgeGC iterates the global list, and sets the signal `Full` in §4.1 to true if the pointer of any dominant object is not in *Marked* status, which indicates batches of data objects can be released, and BridgeGC will perform full GC in the next triggered GC. In full GC cycles, BridgeGC considers *Data (D)* status as *bad* status in the marking phase, then both data objects and normal objects will be marked, the living condition of data pages will be updated if survived data objects are found.

Load barrier modification. The *load barrier* in ZGC is executed before an object is loaded, the loaded object in *bad* status will enter the slow path of the load barrier to update the pointer and status. BridgeGC stops data objects loaded by application threads from entering the slow path of the load barrier by excluding the *Data (D)* status from the address mask of *bad* status in normal GC cycles. Doing so allows data objects to pass the load barrier in the fast path after the pointer check at the assembly level, as data objects are guaranteed not to be relocated in GC cycles and need no pointer update in the load barrier. Only in full GC cycles, objects with *Data (D)* status are considered not marked, and any data objects will enter the load barrier during the marking phase and be marked lived as necessary.

Algorithm 1: Write Barrier of BridgeGC

```
Input: reference, newValue
begin
    const DataMask                                          // Mask of data view
    if (reference & DataMask) ≠ 0 then
        return                                  // If not a reference from data object, return
    end
    if (newValue & DataMask) = 0 then
        return                                   // If not referencing a normal object, return
    end
    GlobalStorage.Append(reference)                 // Add the reference to global root storage
end
```

Write barrier. ZGC has no write barrier. The *write barrier* proposed in BridgeGC is executed when an object is written as a reference of another object. BridgeGC leverages the *write barrier* to record the references from data objects to normal objects and it marks these references during normal GC cycles to avoid omitted marking of lived normal objects. As shown in Algorithm 1, in the *write barrier*, BridgeGC checks the types of the referencing object and the referenced object through their pointers, and quickly eliminates non-target write operations. For target write operations, the write barrier appends the reference to a global storage. BridgeGC considers the references in the storage as pointers to GC root objects, marks and updates the references in every normal GC cycles, and rebuilds the storage in the full GC cycles. Given the small number of target write operations, the cost of the *write barrier* is negligible.

5 Evaluation

We evaluate BridgeGC with the two most popular big data frameworks, i.e., Flink 1.9.3 and Spark 3.3.0. We choose Flink 1.9.3 for it supports both on-heap and off-heap implementations, which benefits our comparison with off-heap solutions. In Flink, we annotate the objects related to MemorySegments, which are used for operations like sort, shuffle, join, and cache. In Spark, we annotate MemoryBlocks and chunked ByteBuffers that are used for similar operations. We also evaluate BridgeGC's latency performance in a key-value database Cassandra 4.0.6. We annotate the objects related to BufferCells and ByteBuffers, which are used for the write cache in Cassandra. We apply our annotations as described in §3.1.

Experimental methodology. In §5.1, we first evaluate the performance benefits of BridgeGC compared with the baseline **ZGC** that BridgeGC was implemented atop. Then in §5.2, we compare BridgeGC with other available collectors in OpenJDK 17, including **Shenandoah**, **G1**, and **Parallel GC**. In §5.3, we compare BridgeGC with other solutions for managing long-lived data objects. These include **off-heap** implementation of data objects, and also two state-of-the-art research works **ROLP** [17] and **LXR** [64]. Finally in §5.4, we evaluate BridgeGC under different memory pressures by varying heap sizes and data object numbers.

Application workloads. We use representative ML, graph, and query applications from official test suites of frameworks [13] and widely used big data benchmarks [22, 25]. The selected workloads cover the most typical operation types of evaluated systems, including cache, sort, aggregation, shuffle, and iteration. All of them will make the systems generate numerous annotated data objects. For Flink, we use 5 applications from official Flink-examples [13], including Linear Regression (**LR**, 800 million points), KMeans (**KM**, 600 centers and 240 million points), PageRank (**PR**, 40 million nodes and 385 million edges), ConnectedComponents (**CC**, 40 million

nodes and 385 million edges), and WebLogAnalysis (**WA**, 50 million documents and 100 million visits). For Spark, we also use 5 applications from popular big data benchmark HiBench [25], including Linear Regression (**LR**, 240 thousand points and 20 thousand features), Support Vector Machine (**SVM**, 120 thousand points and 100 thousand features), Gaussian Mixture Model (**GMM**, 240 million points and 15 centers), KMeans (**KM**, 240 million points and 10 centers), and PageRank (**PR**, 7 million pages). For Cassandra, we use 2 workloads from a popular NoSQL database benchmark YCSB [22], including write-intensive workload (**WI**, 3 million records with 2.4 million updates and 0.6 million reads), and read-intensive workload (**RI**, 3 million records with 0.6 million updates and 2.4 million reads). We provide more information in our code repository.[3]

Experimental setup. The evaluation environment is a cluster with four nodes. Each node has 20-core Intel(R) Xeon(R) Gold 5215 CPU, 128 GB memory, and running the CentOS release 8.0.1905. One node in our cluster serves as the master, and the others serve as workers. We set the parallelism of the frameworks equal to the number of total CPU cores in workers, to make full use of CPU resources. We store the input data of workloads with HDFS 3.2.4, execute each workload three times, clear the memory cache between each run, and report the average result. The relative standard error of our evaluation ranged from 0.43% to 3.15%.

Evaluation questions. We answer the following key questions in our evaluation.

- **How much does `BridgeGC` improve the performance compared to baseline ZGC for evaluated frameworks? (§5.1)** We measure `BridgeGC` with common GC metrics and application execution time. The result shows that `BridgeGC` performs better in all tested workloads and configurations, with up to 82% GC time reduction, up to 29% execution speedup and up to 23% p99.99 latency reduction.
- **How does `BridgeGC` perform compared to traditional collectors and other solutions? (§5.2 and §5.3)** We use the metrics of application execution time for evaluated workloads and collectors. The result shows that `BridgeGC` also outperforms other official collectors and academic collectors for most tested workloads, with up to 26% execution speedup and up to 78% p99.99 latency reduction compared to the default collector G1 in OpenJDK.
- **How well does `BridgeGC` perform under different memory pressures? (§5.4)** We analyze both GC and overall performance improvement of `BridgeGC` under different heap sizes and data object numbers. The results show that `BridgeGC` generally has greater performance benefits under increased memory pressure with more long-lived data objects.

5.1 Comparison with Baseline

Configurations. For Apache Flink, we use the default on-heap implementation, with the default 70% used as managed memory. We adopt the default size of each `MemorySegment` with 32 KB and preallocate byte arrays used by `MemorySegments` before submitting applications, which will relieve GC overhead during execution. For Spark, we adopt the default threshold for data objects with 60% of the heap, set the storage level of RDD to `memory_and_disk_ser` to cache serialized RDD with `chunked ByteBuffer` in memory first, and set the size of each `chunked ByteBuffer` and `MemoryBlock` to 32 KB. For Cassandra, we adopt the `unslabbed_heap_buffers` implementation of `Memetable`, and adopt the threshold for `Memetable` to 50% of the heap. We do not explicitly set GC parameters, instead relying on OpenJDK's heuristics to determine values, such as GC thread count and the page size in ZGC, thus page size in ZGC is by default 2 MB for small-sized objects and 32 MB for middle-sized objects. We configure the heap size of each worker to 32 GB for Flink and Spark and 12 GB for the server of Cassandra, which is generally adequate for workloads to hold more data in memory and reduce data spills onto the disk. We also vary the heap size to study the effectiveness of `BridgeGC` under different memory pressures.

[3]https://github.com/BridgeGC/BridgeGC

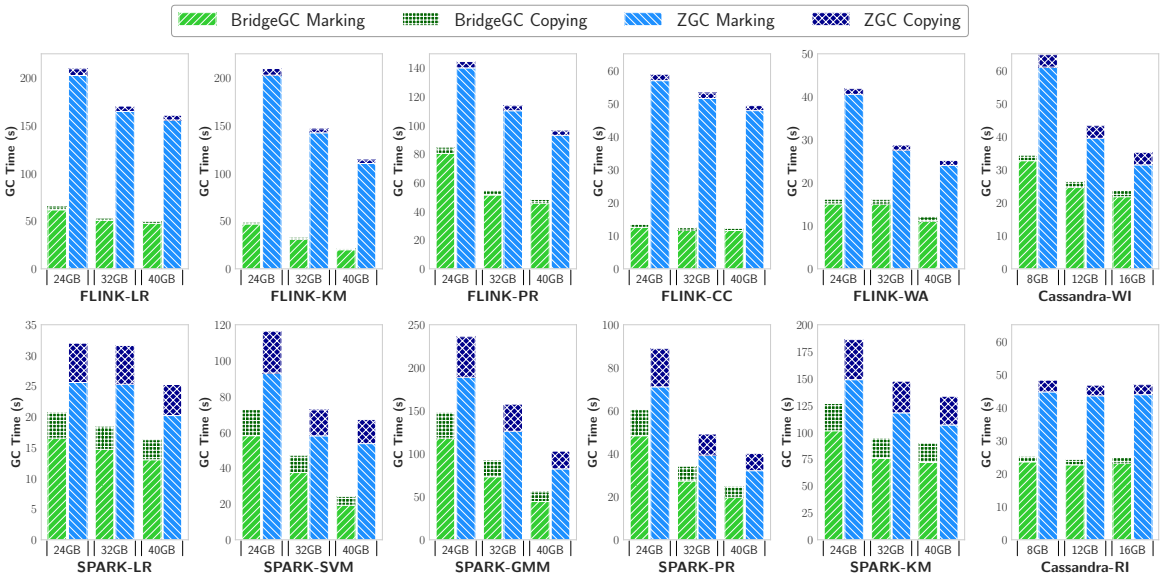

Fig. 8. The total concurrent GC time that BridgeGC and baseline ZGC spend when running Flink, Spark, and Cassandra applications with different heap sizes.

5.1.1 GC Performance. Because both BridgeGC and ZGC achieve *negligible* GC pause time, we leverage the metric of concurrent GC time to measure their GC overhead. Figure 8 illustrates the concurrent GC time reduction of BridgeGC compared to baseline ZGC. The observed GC time reduction ranged from 42% to 82% for Flink workloads, 31% to 46% for Spark workloads, and 30% to 48% for Cassandra workloads. Our in-depth analysis shows that the lower GC time achieved by BridgeGC can be attributed to having fewer GC cycle counts, a lower concurrent GC time in each GC cycle, shorter allocation stalls, and less load barrier slow path execution, as detailed below.

Reductions of heap consumption and GC count. We compare the average after-GC heap memory consumption of BridgeGC and ZGC in Figure 9, which shows that BridgeGC consumes 3%-13% less memory than ZGC. According to logs, the number of objects generated by frameworks is consistent under BridgeGC and ZGC, with annotated data objects taking 82% of total objects lived after GC. The memory efficiency under BridgeGC is due to two design aspects of our allocator. First, separated data object storage eliminates the memory fragmentation caused by the quick death of short-lived objects and inconsistent life cycles within the same page. Second, as described in §4.2, BridgeGC directly adopts a whole data page as TLAB for data objects, which reduces the memory waste associated with TLAB replacement. Due to less heap consumption, BridgeGC triggers 2%-52% fewer GC cycles than ZGC, shown as thin bars in Figure 9. Moreover, it is evident that 87% triggered GC cycles of BridgeGC are normal GC cycles, during which annotated data objects are not redundantly processed. This leads to a further reduction in GC overhead compared to the ZGC, which processes all live objects in every GC cycle.

GC time reduction. We find BridgeGC spends less concurrent GC time for marking and copying phases in a single GC cycle. For both BridgeGC and baseline ZGC, we find that marking is the most time-consuming phase. As shown in Figure 8, the marking phase of ZGC generally takes over 80% of the total concurrent GC time in evaluated applications. BridgeGC excludes the annotated data objects from the marking phase in normal GC cycles and introduces no additional marking process in full GC cycles, significantly reducing the overhead of the

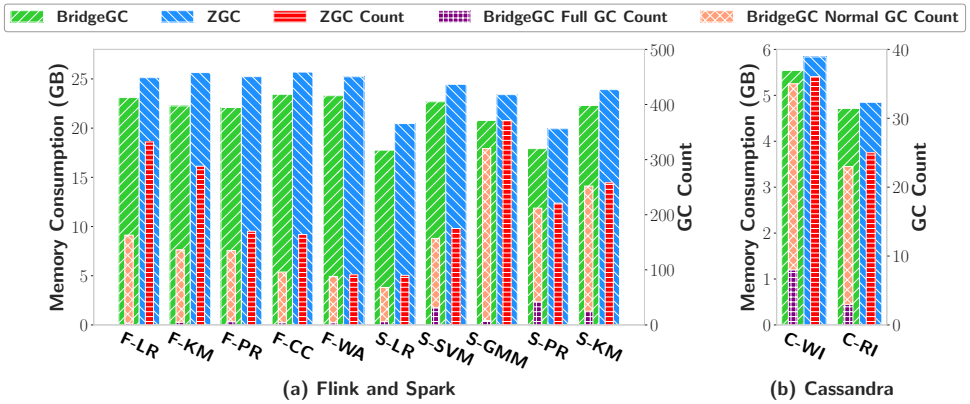

Fig. 9. Average after-GC heap consumption and GC count comparison between BridgeGC and ZGC. Flink and Spark applications are under 32 GB heap, Cassandra applications are under 12 GB heap. Most of the GC cycles triggered by BridgeGC are normal GC, during which unnecessary GC overhead spent on data objects is eliminated.

GC marking phase. Consequently, compared with the average of a single GC cycle in ZGC, BridgeGC spends 46% less marking time in a normal GC cycle. Meanwhile, BridgeGC spends 38% less time per GC cycle in the object relocation phase compared to baseline ZGC, indicating that BridgeGC spends less GC efforts in copying lived objects. In full GC cycles of BridgeGC, BridgeGC spends similar GC time as GC cycle of ZGC, and reclaims 62% data pages and 41% of heap space on average. In contrast, ZGC reclaims only 13% heap space on average in GC cycles. This indicates that BridgeGC is more effective at reclamation, for it reclaims long-lived data objects at the appropriate time.

Allocation stall reduction. We find BridgeGC triggers fewer allocation stalls, the allocation stall time under BridgeGC is 21%-100% less than baseline ZGC. An allocation stall is caused by heap memory exhaustion when a GC cycle is in progress, which lasts until enough memory is freed by the current GC cycle. We observe severe allocation stall under memory-intensive applications (e.g., Flink-PR and Spark-SVM), and we find that BridgeGC triggers less allocation stall and has a shorter average allocation stall time. This is because BridgeGC has lower memory consumption and completes a GC cycle faster than baseline ZGC. Fewer and shorter allocation stalls in turn reduce the unexpected pause in application threads.

Barrier overhead reduction. The load barrier of ZGC was found expensive and degrading overall performance due to its extremely high execution frequency, while BridgeGC reduces the number of times that the slow path of load barrier is executed. We count the number of times that BridgeGC and ZGC execute the slow path of load barrier by code instrumentation. The result shows that BridgeGC reduces 17%-23% executions of load barrier slow path per GC cycle. This is because BridgeGC alleviates the need to update the annotated objects' pointers in normal GC cycles as we illustrated in §4.3. Less load barrier slow path execution in turn accelerates the execution of application threads. We also evaluate the overhead of the write barrier introduced in BridgeGC by comparing the performance of BridgeGC when the write barrier is enabled and off, the result shows the performance difference is around 1%, which is relatively small.

5.1.2 Overall Performance. Figure 10 depicts the execution speedup of BridgeGC compared to ZGC under different heap sizes. We see that BridgeGC reduces the execution time for all Flink, Spark, and Cassandra workloads under all heap memory sizes. BridgeGC speedups applications' execution compared to ZGC mainly due to reduced GC overhead spent on data objects, the reduced GC overhead alleviates the contention between GC

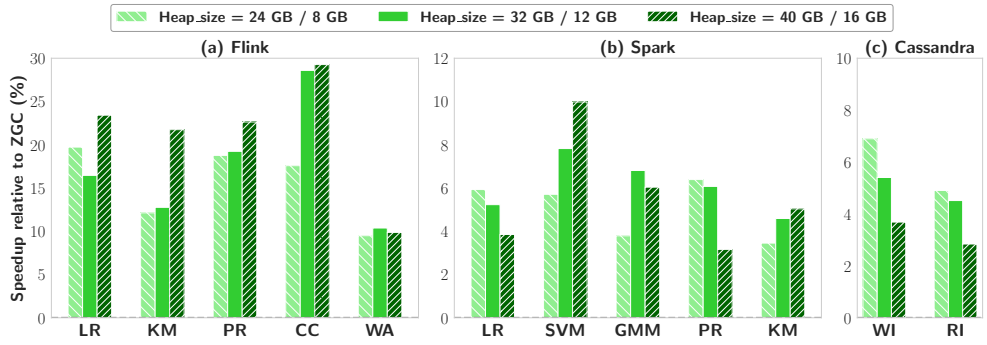

Fig. 10. The speedup in execution time when running (a) Flink applications, (b) Spark applications, and (c) Cassandra applications, with BridgeGC in comparison to the baseline ZGC, using different heap sizes (24 GB, 32 GB, and 40 GB for Flink and Spark applications, 8 GB, 12 GB, and 16 GB for Cassandra applications).

threads and application threads during execution. Therefore, we find the end-to-end performance improvement under BridgeGC is generally consistent with the BridgeGC performance enhancement, and the performance improvement is more significant under applications and configurations where more data is kept in memory. In such cases, long-lived data objects account for a larger proportion, leading to higher performance gains. We also analyze the throughput overhead introduced by the implementation of BridgeGC, by evaluating ZGC with and without the annotation processing. Our analysis shows that the throughput difference is less than 1%. In terms of latency performance, as shown in Table 1, BridgeGC outperforms ZGC in both average and tail latency for Cassandra workloads. This improvement is also attributed to reduced concurrent marking and copying overhead.

5.2 Comparison with Traditional Collectors

In this section, we compare BridgeGC with other available collectors in the same release of OpenJDK. These include *(i)* Shenandoah, another mostly-concurrent collector that also has low pause times; *(ii)* G1, a partially-concurrent collector that is OpenJDK's default; *(iii)* Parallel, a throughput-oriented collector of which all GC phases pause the application. We use the same framework configurations and execute the same workloads for this evaluation. We rely on OpenJDK to set other GC parameters to default or heuristic values, including generational ratio and compressed pointers. Therefore, collectors that support compressed pointers are enabled by default. Because evaluated garbage collectors suffer from varying degrees of GC pauses, and their concurrent GC processes have diverse levels of impact on the execution of the application, we illustrate the superior performance of BridgeGC by comparing the end-to-end execution time in Table 2 and latency metrics in Table 1.

In terms of execution time, we see that BridgeGC has the shortest execution time for all applications. Specifically, BridgeGC always performs better than Shenandoah. We have observed that Shenandoah GC exhibits more pronounced GC pauses and significantly longer concurrent GC times compared to BridgeGC. BridgeGC also outperforms G1 and Parallel in all applications. G1 and Parallel experience substantial object promotion overhead with large volumes of long-lived data objects. After being promoted to the old generation, these data objects still suffer from unnecessary marking and copying overhead in G1 and Parallel GC cycles that target the whole heap. Therefore, in applications where the whole heap GC cycles are more frequently triggered (e.g., Spark-KM), the advantages of BridgeGC are more apparent. In contrast, in applications where whole heap GC cycles are less triggered (e.g., Flink-PR), the benefits of BridgeGC are less pronounced.

Table 1. GC and application performance comparisons with *Cassandra* under 12 GB heap. Green cells highlight the best performance per metric.

Workload	Collector	Pause(ms)	Throughput	Avg. Latency(μs)	p99.9 Latency(μs)	p99.99 Latency(μs)
WI	BridgeGC	0.02	64886	287.6	2642	8291
	ZGC	0.02	61127	305.8	3356	10810
	ZGC-off	0.02	62492	302.1	2919	9635
	Shenandoah	0.16	57812	324.6	6257	14345
	G1	22.2	63109	295.6	4874	37819
	Parallel	83.1	63589	292.9	3201	60429
RI	BridgeGC	0.02	71149	261.4	2622	8463
	ZGC	0.02	67877	277.2	2863	10281
	ZGC-off	0.02	68857	272.8	2635	9185
	Shenandoah	0.16	63064	294.9	5613	13481
	G1	20.3	69193	267.5	4057	35739
	Parallel	83.9	69031	269.9	4056	72543

Table 2. Execution time of Flink and Spark applications under baseline ZGC and execution time of other collectors normalized to ZGC under 32 GB heap size. Green cells highlight the best performance.

Application	Baseline	Normalized to Baseline				
	ZGC	Parallel	G1	Shenandoah	ZGC-off	Bridge
Flink-LR	307.71 s	0.907	0.978	1.047	0.926	0.858
Flink-KM	359.81 s	0.911	1.024	1.139	0.937	0.889
Flink-PR	311.37 s	0.887	0.951	1.214	1.043	0.838
Flink-CC	269.41 s	0.849	0.974	1.317	1.018	0.773
Flink-WA	231.41 s	1.068	1.053	1.161	0.981	0.906
Spark-LR	99.98 s	1.105	1.036	1.063	1.235	0.950
Spark-SVM	291.34 s	1.211	0.945	1.277	1.189	0.927
Spark-GMM	359.84 s	0.970	1.005	1.123	1.098	0.936
Spark-PR	203.45 s	1.180	1.049	1.048	1.055	0.942
Spark-KM	172.83 s	1.198	1.117	1.306	1.133	0.953

In terms of latency metrics, we find that `BridgeGC` achieves the highest throughput and the lowest average latency, as well as the lowest p99.9 and p99.99 tail latency, across two Cassandra workloads. As shown in Table 1, `BridgeGC` outperforms other collectors in tail latency metrics due to its negligible GC pause time inherited from ZGC. The higher throughput of `BridgeGC` also contributes to its superior average latency performance.

5.3 Comparison with Other Solutions

Off-heap. Both Flink, Spark, and Cassandra provide the solution to store abstracted data objects *off-heap*. We compare `BridgeGC` to ZGC running on-heap and off-heap (labeled as **ZGC-off**) memory with the same memory budget. That is, the memory capacity for data objects remains consistent no matter being allocated on-heap or off-heap. For the example of Flink with a 32 GB memory budget, we set the `taskmanager.memory.off-heap` in

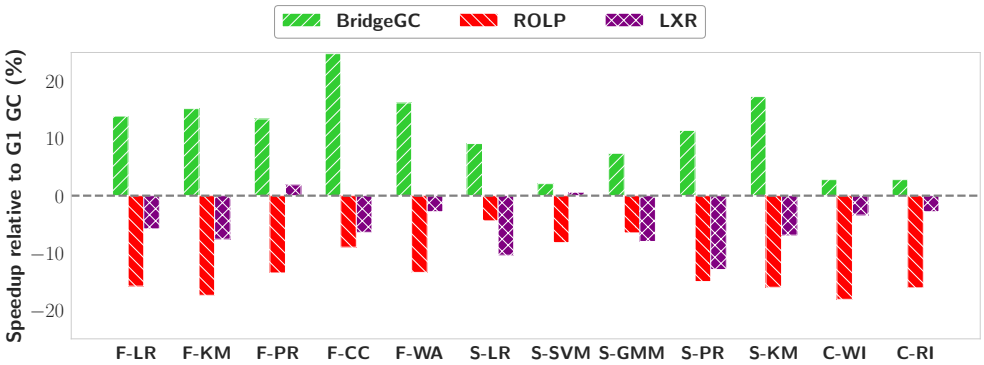

Fig. 11. Regarding G1 as the baseline, BridgeGC outperforms ROLP and LXR in application execution time. Flink and Spark applications are under 32 GB heap, Cassandra applications are under 12 GB heap.

Flink to true, making the default 70% (22.4 GB) of the memory used for data allocated off-heap and the rest (9.6 GB) used for JVM heap. We similarly configure the Spark and Cassandra.

Table 1 and Table 2 show that BridgeGC outperforms ZGC-Off in application execution time and latency metrics. The reason is that the off-heap option can only place the data used to be stored in the primitive type arrays (e.g., byte[]) of abstracted data objects off-heap, while abstracted data objects (e.g., ByteBuffer) are still allocated on-heap to reference off-heap memory. These objects are still long-lived, numerous, and contribute to unnecessary marking and copying overhead. In contrast, BridgeGC eliminates marking and copying for objects used in both data storage and control. In addition, the fraction of off-heap size is fixed during execution, and off-heap memory can not be used for heap objects. When off-heap memory is not fully used, this part of memory is wasted, and the left heap memory will face greater memory pressure, resulting in even longer concurrent GC time and application execution time. In contrast, the allocation of data and normal pages in BridgeGC is more balanced and flexible. Therefore, running the garbage collector with off-heap memory does not provide GC benefits and flexibility comparable to those of BridgeGC.

ROLP. We compare BridgeGC with ROLP [17], the state-of-the-art big-data-friendly GC integrated into the G1 algorithm as illustrated in §2.3. We obtained and built ROLP's source code from the commit number 56f471d, based on OpenJDK 1.8.0_301.[4] To fairly compare ROLP and BridgeGC given the fact that BridgeGC is integrated with OpenJDK 17, we leverage the G1 as the intermediate. That is, we compare ROLP and BridgeGC to G1 in OpenJDK 8 and OpenJDK 17, respectively.

We observe that ROLP achieves 28% shorter average GC pauses than G1, leading to 18% reduction in p99.9 latency. However, as shown in Figure 11, the application execution speed under ROLP is 11% inferior to that of G1 on average. As BridgeGC outperforms G1, we conclude that BridgeGC works better than ROLP in terms of execution time, and the reasons are two-fold. First, BridgeGC leverages static annotations, which have no throughput overhead at runtime, in contrast to ROLP's dynamic profiling. Second, BridgeGC can obtain the precise lifetime of long-lived data objects, hence eliminating both unnecessary marking and copying costs. Since ROLP achieves shorter GC pauses than G1, it is more suitable for latency-sensitive applications.

LXR. We also compare BridgeGC with LXR [64], which is another state-of-the-art GC algorithm that aims to improve throughput and reduce GC pause time on diverse contemporary workloads. LXR uses reference counting

[4]https://github.com/rodrigo-bruno/rolp

Table 3. GC time and execution time comparison between BridgeGC and ZGC when running Flink-PR under different configurations. Each configuration is represented as heap size + MemorySegment size.

Configuration	**GC Time** (BridgeGC / ZGC)	**Execution Time** (BridgeGC / ZGC)
32 GB + 32 KB	54.427 s / 116.623 s	261.133 s / 311.37 s
32 GB + 16 KB	58.780 s / 125.781 s	264.802 s / 318.798 s
32 GB + 8 KB	64.208 s / 145.321 s	270.269 s / 344.451 s

and judicious copying to collect long-lived objects efficiently. We obtained and built LXR's sourcecode, from the commit number e34dce3 for mmtk-core and 61f3be5 for mmtk-openjdk, based on OpenJDK 11.0.19.[5]

We observe that LXR triggers fewer GC pause cycles than G1, and LXR reduces p99.99 latency by 76% compared to G1 in Cassandra applications. However, as shown in Figure 11, LXR takes 5% longer execution time than G1 on average. In comparison, BridgeGC executes 11% faster than G1 on average. This indicates that BridgeGC also performs better than LXR in managing big data applications. The reason that BridgeGC performs better can be two-fold. Firstly, In Spark and Flink applications, we find LXR still triggers many GC pauses, and the average pause time of LXR is 36% longer than G1. During these pauses, LXR relocates the long-lived objects to compact memory, resulting in significant copying overhead. In contrast, BridgeGC eliminates unnecessary copying of long-lived objects by allocating them together at their allocation. Secondly, LXR still requires periodic tracing of numerous long-lived data objects to identify cyclic garbage, while BridgeGC eliminates unnecessary marking overhead of these long-lived objects in normal GC cycles. Therefore, BridgeGC performs better than LXR in execution time.

5.4 Comparison under Different Pressures

Different memory pressure. According to Figure 10 and Figure 8, the total GC time for both BridgeGC and ZGC decreases as the heap size increases (i.e., the memory pressure decreases). Although the average time of a GC cycle increases with heap size, the larger memory space generally reduces the frequency of GC triggers. For applications where the proportion of annotated data objects decreases when the heap size grows (e.g., Spark-LR), the speedup of BridgeGC compared to baseline ZGC decreases. Conversely, for applications that continue to generate a large proportion of data objects when the heap size grows (e.g., Spark-SVM), the performance improvement of BridgeGC is more significant. Nevertheless, across all workloads, BridgeGC consistently outperforms ZGC in GC and application metrics regardless of the heap size. This is because the data objects generated by frameworks always account for a significant proportion of memory usage, and BridgeGC can eliminate the unnecessary marking and copying costs associated with them.

Different GC pressure. We switch the size of each abstracted data object (e.g., MemorySegment) from 8 KB to 32 KB to study the effectiveness of BridgeGC with different GC pressures. As the total memory usage of data objects is fixed (e.g., 70% of the heap memory in Flink), smaller data object sizes result in a larger number of data objects, thereby increasing GC overhead due to the need for marking and copying more data objects. We find the improvement of BridgeGC is more significant under higher GC pressure since more data objects are generated and efficiently managed by BridgeGC. The example of the Flink-PR application can be found in Table 3. It can be observed that the growth rates of both execution time and GC time metrics for BridgeGC are smaller compared to the baseline ZGC when the GC pressure increases.

[5]https://github.com/wenyuzhao/mmtk-core/tree/lxr

6 Related Work

Big-data-friendly GC. Previous works tried to narrow the gap between big data frameworks and garbage collectors [14, 16, 17, 21, 24, 41, 42]. Broom [24] proposes three types of regions, it allocates objects with similar life cycles in the same region to achieve efficient collection of regions. Facade [42] modifies the compiler to use off-heap regions for allocating most data objects and reclaims off-heap memory at the end of the iteration. DSA [21] allocates objects belonging to a data structure in a separate area and considers them as root objects to benefit heap tracing. Yak [41] divides the heap into a control space and a data space, allocates objects within a code epoch into the data space, and manages objects in the data space based on code epochs. NG2C [16] extends 2 generations heap in G1 to N generations and pre-tenures data objects with the same life cycle into the same generation, eliminating most object promotion overhead. ROLP [17] uses dynamic profiling to obtain object life cycles for NG2C. `BridgeGC` leverages framework-level annotations to eliminate unnecessary GC overhead and addresses limitations of prior work. Here, we compare `BridgeGC` with ROLP, as it is the state-of-the-art big-data-friendly GC and is open-sourced.

General GC optimization. Previous works also optimized the existing garbage collectors to improve their performance for general applications [32, 47, 54, 59, 60, 63]. Suo et al. [47] enhances GC thread affinity for Parallel GC. Platinum [56] enhances Parallel GC to a concurrent model. LXR [64] uses reference counting and brief stop-the-world pauses to reclaim most memory. Jade [54] uses a group-wise collection mechanism to minimize pre-reclamation cycles and reduce GC frequency in G1 GC. There are also works that optimize ZGC. ThinGC [59] divides the heap into hot and cold subheaps. HCSGC [60] dynamically reorganizes objects based on mutator access patterns. Works are also being made to enhance the performance of existing garbage collectors on new architectures [19, 34, 36, 49–51, 53, 62]. TeraHeap [29] extends JVM to a high-capacity second heap (H2) over the NVM devices. RiskRelief [34] proposes a garbage collector for hybrid HBM-DRAM systems and uses allocation site prediction to place hot objects in HBM. Ye et al. [62] develop solutions to preserve the addressability of objects during GC-triggered movements in NVM. MemLiner [51] aligns memory accesses from application and GC threads. Mako [36] enables efficient pointer updates and synchronization for memory-disaggregated data centers. We may leverage the above work for further optimization.

Big data framework optimization. Researchers optimized the big data frameworks to reduce the overhead of data management [15, 18, 33, 35, 37, 39, 45, 48, 48, 52, 58]. Phronesis [33] efficiently predicts system outcomes based on configuration parameters of frameworks to enable optimal settings. Bu et al. [18] merge small objects into larger byte buffers and access data at the binary level. Deca [35, 45] analyzes the lifetimes of data in frameworks and groups data with similar lifetimes into byte arrays. Gerenuk [38] speculatively transforms frameworks to process data directly in native bytes within off-heap buffers. Bruno et al. [15] enables compiler-assisted object inlining in a closed-world environment. Taurus [37] coordinates the GC timing of distributed nodes in frameworks. ElasticMem [52] dynamically adjusts JVM heap sizes to optimize resource allocation in frameworks. Skyway [39] eliminates the need for object serialization/deserialization by directly transferring object graphs between heaps. ZCOT [55] eliminates object serialization/deserialization using a globally shared exchange space. The above work is orthogonal to our efforts, and we share a similar goal of improving application performance.

7 Discussion

Generality. We have applied our annotations to Flink, Spark, and Cassandra in this paper. Our annotations can potentially be extended to more big data systems, such as HBase [6] and Apache Ignite [7]. For HBase, we can annotate the *MemStore*, which buffers recent writes in memory and serves a similar purpose to the *Memtable* in Cassandra, as these structures are generally long-lived. We leave further exploration of these use cases as future

work. Moreover, for frameworks or systems without well-encapsulated long-lived data objects, BridgeGC can still be applied with a deeper understanding of the framework's internals.

Human efforts. To use BridgeGC, framework developers need to apply our annotations to data objects in the framework's source code before compiling the framework. This effort is limited, as we only modified 10 LoC for each type of data object in Flink/Spark/Cassandra. To run the framework with BridgeGC, we only need to use JVM with BridgeGC as the garbage collector, without introducing any human efforts for application developers/users.

Trade-off. BridgeGC uses colored pointers to identify data objects without adding memory overhead. However, the use of colored pointers is incompatible with compressed pointers. Compressed pointers can reduce memory consumption, making them beneficial for scenarios with smaller heap sizes. Despite this limitation, our evaluation shows that BridgeGC outperforms collectors that use compressed pointers in both throughput and latency. However, exploring methods to incorporate pointer compression into BridgeGC remains an interesting direction for future work.

8 Conclusion

Due to the gap between big data frameworks and garbage collectors, current big data frameworks tend to suffer from heavy GC overhead. This paper proposed a garbage collector named BridgeGC to narrow this gap. BridgeGC adopts a cross-level co-design, which leverages limited annotations of data objects at the framework level to optimize the allocation and reclamation of them at the GC level. BridgeGC has been integrated into the garbage collector ZGC in OpenJDK. Our evaluations show that BridgeGC can significantly improve the GC efficiency and application performance for popular big data frameworks Flink and Spark, as well as the database Cassandra.

References

[1] 2015. Apache Flink. https://flink.apache.org/
[2] 2015. Project Tungsten. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html
[3] 2024. Apache Cassandra. https://cassandra.apache.org/
[4] 2024. Apache Drill. https://drill.apache.org/
[5] 2024. Apache Geode. https://geode.apache.org/
[6] 2024. Apache HBase. https://hbase.apache.org/
[7] 2024. Apache Ignite. https://ignite.apache.org/
[8] 2024. Apache Spark. https://spark.apache.org/
[9] 2024. Flink OLAP Quickstart. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/table/olap_quickstart/#runtime-options
[10] 2024. Shenandoah GC. https://wiki.openjdk.org/display/shenandoah
[11] 2024. The Z garbage collector. https://wiki.openjdk.org/display/zgc/Main
[12] Shoaib Akram. 2021. Performance evaluation of intel optane memory for managed workloads. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 3 (2021), 1–26.
[13] Apache Flink. 2024. Flink Examples Batch at Release 1.9. https://github.com/apache/flink/tree/release-1.9/flink-examples/flink-examples-batch. Accessed: 2024-11-06.
[14] Rodrigo Bruno and Paulo Ferreira. 2017. POLM2: Automatic profiling for object lifetime-aware memory management for hotspot big data applications. In *Proceedings of ACM/IFIP/USENIX Middleware Conference (Middleware)*. 147–160.
[15] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. 2021. Compiler-assisted object inlining with value fields. In *Proc. ACM PLDI*. 128–141.
[16] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. 2017. NG2C: Pretenuring garbage collection with dynamic generations for HotSpot big data applications. In *Proc. ACM ISMM*. 2–13.
[17] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. 2019. Runtime object lifetime profiler for latency sensitive big data applications. In *Proc. EuroSys*. 1–16.
[18] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J Carey. 2013. A bloat-aware design for big data applications. In *Proc. ACM ISMM*. 119–130.
[19] Lei Chen, Jiacheng Zhao, Chenxi Wang, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, Guoqing Harry Xu, et al. 2022. Unified Holistic Memory Management Supporting Multiple Big Data Processing Frameworks over Hybrid Memories. *ACM TOCS* (2022).

[20] Jiho Choi, Thomas Shull, and Josep Torrellas. 2018. Biased reference counting: Minimizing atomic operations in garbage collection. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–12.
[21] Nachshon Cohen and Erez Petrank. 2015. Data structure aware garbage collector. In *Proceedings of International Symposium on Memory Management (ISMM)*. 28–40.
[22] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*. 143–154.
[23] Stephan Ewen. 2015. Off-heap Memory in Apache Flink and the curious JIT compiler. https://flink.apache.org/news/2015/09/16/off-heap-memory.html
[24] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping out garbage collection from big data systems. In *Proceedings of Workshop on Hot Topics in Operating Systems (HotOS)*.
[25] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Proc. IEEE ICDEW*. 41–51.
[26] 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. *ACM SIGPLAN Notices* 39, 10 (2004), 69–80.
[27] Fabian Hüske. 2015. Juggling with Bits and Bytes. https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html
[28] Iacovos G Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S Zakkak, Polyvios Pratikakis, and Angelos Bilas. 2023. TeraHeap: Exploiting Flash Storage for Mitigating DRAM Pressure in Managed Big Data Frameworks. *ACM Transactions on Programming Languages and Systems* (2023).
[29] Iacovos G Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S Zakkak, Polyvios Pratikakis, and Angelos Bilas. 2023. Teraheap: Reducing memory pressure in managed big data frameworks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 694–709.
[30] Iacovos G Kolokasis, Anastasios Papagiannis, Polyvios Pratikakis, Angelos Bilas, and Foivos Zakkak. 2020. Say Goodbye to Off-heap Caches! On-heap Caches Using Memory-Mapped I/O. In *Proceedings of USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.
[31] Mayuresh Kunjir and Shivnath Babu. 2020. Black or white? how to develop an autotuner for memory-based analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1667–1683.
[32] Haoyu Li, Mingyu Wu, Binyu Zang, and Haibo Chen. 2019. ScissorGC: Scalable and efficient compaction for Java full garbage collection. In *Proc. VEE*. 108–121.
[33] Yuhao Li and Benjamin C Lee. 2022. Phronesis: Efficient performance modeling for high-dimensional configuration tuning. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 4 (2022), 1–26.
[34] Wenjie Liu, Shoaib Akram, Jennifer B Sartor, and Lieven Eeckhout. 2021. Reliability-aware garbage collection for hybrid HBM-DRAM memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 1 (2021), 1–25.
[35] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. 2016. Lifetime-based memory management for distributed data processing systems. *Proceedings of the VLDB Endowment* 9, 12 (2016), 936–947.
[36] Haoran Ma, Shi Liu, Chenxi Wang, Yifan Qiao, Michael D Bond, Stephen M Blackburn, Miryung Kim, and Guoqing Harry Xu. 2022. Mako: A low-pause, high-throughput evacuating collector for memory-disaggregated datacenters. In *Proceedings of ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 92–107.
[37] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. *Acm SIGPLAN Notices* 51, 4 (2016), 457–471.
[38] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. 2019. Gerenuk: Thin computation over big native data using speculative program transformation. In *Proc. ACM SOSP*. 538–553.
[39] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting managed heaps in distributed big data systems. *ACM SIGPLAN Notices* 53, 2 (2018), 56–69.
[40] Khanh Nguyen, Lu Fang, Guoqing Xu, and Brian Demsky. 2015. Speculative region-based memory management for big data systems. In *Proceedings of the 8th workshop on programming languages and operating systems*. 27–32.
[41] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A high-performance big-data-friendly garbage collector. In *Proc. USENIX OSDI*. 349–365.
[42] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications. In *Proc. ASPLOS*. 675–690.
[43] Kehinde Philip. 2018. Spark executor GC taking long. Retrieved January 21, 2018 from https://stackoverflow.com/questions/38965787/spark-executor-gc-taking-long
[44] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S McKinley. 2013. Taking off the gloves with reference counting Immix. *ACM SIGPLAN Notices* 48, 10 (2013), 93–110.

[45] Xuanhua Shi, Zhixiang Ke, Yongluan Zhou, Hai Jin, Lu Lu, Xiong Zhang, Ligang He, Zhenyu Hu, and Fei Wang. 2019. Deca: A garbage collection optimizer for in-memory data processing. *ACM TOCS* 36, 1 (2019), 1–47.
[46] Karlsson Stefan. 2021. *Pointer Metadata using Multi-Mapped memory.* https://wiki.openjdk.org/display/zgc/Pointer+Metadata+using+Multi-Mapped+memory
[47] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. 2018. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of EuroSys Conference (EuroSys)*. 1–15.
[48] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. 2021. Naos: Serialization-free RDMA networking in Java. In *Proc. USENIX ATC*. 1–14.
[49] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 347–362.
[50] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A Memory-Disaggregated Managed Runtime. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 261–280.
[51] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 35–53.
[52] Jingjing Wang and Magdalena Balazinska. 2017. Elastic memory management for cloud data analytics. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*. 745–758.
[53] Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, and Haibing Guan. 2020. GCPersist: An efficient GC-assisted lazy persistency framework for resilient Java applications on NVM. In *Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. 1–14.
[54] Mingyu Wu, Liang Mao, Yude Lin, Yifeng Jin, Zhe Li, Hongtao Lyu, Jiawei Tang, Xiaowei Lu, Hao Tang, Denghui Dong, et al. 2024. Jade: A High-throughput Concurrent Copying Garbage Collector. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1160–1174.
[55] Mingyu Wu, Shuaiwei Wang, Haibo Chen, and Binyu Zang. 2022. Zero-Change Object Transmission for Distributed Big Data Analytics. In *Proc. USENIX ATC*. 137–150.
[56] Mingyu Wu, Ziming Zhao, Yanfei Yang, Haoyu Li, Haibo Chen, Binyu Zang, Haibing Guan, Sanhong Li, Chuansheng Lu, and Tongbao Zhang. 2020. Platinum: A CPU-Efficient Concurrent Garbage Collector for Tail-Reduction of Interactive Services. In *Proc. USENIX ATC*. 159–172.
[57] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. 2019. An experimental evaluation of garbage collectors on big data applications. In *Proc. VLDB*. 540–583.
[58] Luna Xu, Min Li, Li Zhang, Ali R Butt, Yandong Wang, and Zane Zhenhua Hu. 2016. Memtune: Dynamic memory management for in-memory data analytic platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 383–392.
[59] Albert Mingkun Yang, Erik Österlund, Jesper Wilhelmsson, Hanna Nyblom, and Tobias Wrigstad. 2020. ThinGC: Complete isolation with marginal overhead. In *Proc. ACM ISMM*. 74–86.
[60] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. 2020. Improving program locality in the GC using hotness. In *Proc. ACM PLDI*. 301–313.
[61] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2022), 22:1–22:34.
[62] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Hai Jin, Xiaofei Liao, and Yan Solihin. 2022. Preserving addressability upon GC-triggered data movements on non-volatile memory. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 2 (2022), 1–26.
[63] Yang Yu, Tianyang Lei, Weihua Zhang, Haibo Chen, and Binyu Zang. 2016. Performance analysis and optimization of full garbage collection in memory-hungry environments. *ACM SIGPLAN Notices* 51, 7 (2016), 123–130.
[64] Wenyu Zhao, Stephen M Blackburn, and Kathryn S McKinley. 2022. Low-latency, high-throughput garbage collection. In *Proc. ACM PLDI*. 76–91.