TeraHeap: Exploiting Flash Storage for Mitigating DRAM Pressure in Managed Big Data Frameworks

IACOVOS G. KOLOKASIS and GIANNOS EVDOROU, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece and Department of Computer Science, University of Crete, Heraklion, Greece
SHOAIB AKRAM, Australian National University, Canberra, Australia
CHRISTOS KOZANITIS, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece
ANASTASIOS PAPAGIANNIS, Isovelent, Inc., Cupertino, CA, USA
FOIVOS S. ZAKKAK, Red Hat Ltd., Manchester, UK
POLYVIOS PRATIKAKIS and ANGELOS BILAS, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece and Department of Computer Science, University of Crete, Heraklion, Greece

Big data analytics frameworks, such as Spark and Giraph, need to process and cache massive datasets that do not always fit on the managed heap. Therefore, frameworks temporarily move long-lived objects outside the heap (off-heap) on a fast storage device. However, this practice results in (1) high serialization/deserialization (S/D) cost and (2) high memory pressure when off-heap objects are moved back for processing.

In this article, we propose *TeraHeap*, a system that eliminates S/D overhead and expensive GC scans for a large portion of objects in analytics frameworks. *TeraHeap* relies on three concepts: (1) It eliminates S/D by extending the managed runtime (JVM) to use a second high-capacity heap (H2) over a fast storage device. (2) It offers a simple hint-based interface, allowing analytics frameworks to leverage object knowledge to populate H2. (3) It reduces GC cost by fencing the collector from scanning H2 objects while maintaining the illusion of a single managed heap, ensuring memory safety.

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the projects AERO (Grant agreement No. 10048318). Iacovos G. Kolokasis is also supported by the Meta Research PhD Fellowship and the State Scholarship Foundation of Cyprus.
Authors' Contact Information: Iacovos G. Kolokasis (corresponding author), Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece and Department of Computer Science, University of Crete, Heraklion, Greece; e-mail: kolokasis@ics.forth.gr; Giannos Evdorou, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece and Department of Computer Science, University of Crete, Heraklion, Greece; e-mail: evdorou@ics.forth.gr; Shoaib Akram, Australian National University, Canberra, Australia; e-mail: shoaib.akram@anu.edu.au; Christos Kozanitis, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece; e-mail: kozanitis@ics.forth.gr; Anastasios Papagiannis, Isovelent, Inc., Cupertino, CA, USA; e-mail: anastasios@isovalent.com; Foivos S. Zakkak, Red Hat Ltd., Manchester, UK; e-mail: fzakkak@redhat.com; Polyvios Pratikakis, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece and Department of Computer Science, University of Crete, Heraklion, Greece; e-mail: polyvios@ics.forth.gr; Angelos Bilas, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece and Department of Computer Science, University of Crete, Heraklion, Greece; e-mail: bilas@ics.forth.gr.

ACM 1558-4593/2024/12-ART12
https://doi.org/10.1145/3700593

We implement *TeraHeap* in OpenJDK8 and OpenJDK17 and evaluate it with fifteen widely used applications in two real-world big data frameworks, Spark and Giraph. We find that for the same DRAM size, *TeraHeap* improves performance by up to 73% and 28% compared to native Spark and Giraph. Also, it can still provide better performance by consuming up to 4.6× and 1.2× less DRAM than native Spark and Giraph, respectively. *TeraHeap* can also be used for in-memory frameworks and applying it to the Neo4j Graph Data Science library improves its performance by up to 26%. Finally, it outperforms Panthera, a state-of-the-art garbage collector for hybrid DRAM-NVM memories, by up to 69%.

CCS Concepts: • **Software and its engineering → Memory management**; **Garbage collection**; **Runtime environments**; • **Information systems → Flash memory**; **Phase change memory**; *Data analytics*; • **Computer systems organization → Cloud computing**;

Additional Key Words and Phrases: Java Virtual Machine (JVM), large analytics, serialization, large managed heaps, memory management, garbage collection, memory hierarchy, fast storage devices

ACM Reference format:
Iacovos G. Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S. Zakkak, Polyvios Pratikakis, and Angelos Bilas. 2024. *TeraHeap*: Exploiting Flash Storage for Mitigating DRAM Pressure in Managed Big Data Frameworks. *ACM Trans. Program. Lang. Syst.* 46, 4, Article 12 (December 2024), 37 pages.
https://doi.org/10.1145/3700593

1 Introduction

Managed big data frameworks, such as Spark [72] and Giraph [55], are designed to analyze huge volumes of data. Typically, such processing requires iterative computations over data until a convergence condition is satisfied. Each iteration produces new transformations over data, generating a massive volume of objects spanning long computations.

Hosting a large volume of objects on the managed heap increases memory pressure, resulting in frequent **garbage collection (GC)** cycles with low yield. Each GC cycle reclaims little space because (1) the cumulative volume of allocated objects is several times larger than the size of available heap [65] and (2) objects in big data frameworks exhibit long lifetimes [12, 60, 66]. Although production garbage collectors efficiently manage short-lived objects, they do not perform well under high memory pressure introduced by long-lived objects [42].

The common practice for coping with rapidly growing datasets and high GC cost is to move objects outside the managed heap (off-heap) over a fast storage device (e.g., **non-volatile memory (NVMe)** SSD). However, frameworks cannot compute directly over off-heap objects, and thus, they (re)allocate these objects on the managed heap to process them. Although some systems support off-heap computation over byte arrays with primitive types [19], they do not offer support for computation over arbitrary objects, resulting in application-specific solutions, such as Spark SQL [17].

Moving managed objects off-heap has two main limitations. First, it introduces high **serialization/deserialization (S/D)** overhead for applications that use complex data structures [45, 54, 64]. Recent efforts [31, 35] reduce S/D but demand custom hardware extensions and do not mitigate GC overhead. Second, moving a large volume of off-heap objects to the managed heap for processing increases the GC cost. Although TMO [63] transparently swaps cold application memory to NVMe SSDs and provides direct access to device resident objects (no S/D), it cannot avoid slow GC scans over the device. Our evaluation shows that GC and S/D constitute up to 87% of the execution time in big data applications.

In this article, we propose *TeraHeap*, a system that eliminates S/D and GC overheads for a large portion of the data in managed big data analytics frameworks. *TeraHeap* extends the **Java virtual**

machine (JVM) to use a second, **high-capacity heap (H2)** over a fast storage device that coexists alongside the **regular heap (H1)**. It eliminates S/D by providing direct access to objects in H2 and reduces GC by avoiding costly GC scans over objects in H2. Frameworks use *TeraHeap* through its hint-based interface without modifications to the applications that run on top of them. *TeraHeap* addresses three main challenges, as follows.

Identifying Candidate Objects for H2. Big data frameworks move specific objects outside the managed heap on off-heap storage. For instance, Spark moves off-heap intermediate results; Giraph moves the vertices and edges of the graph and the messages sent between vertices. Frameworks organize such data (partitions) as groups of objects with a single-entry root reference [40]. *TeraHeap* provides a hint-based interface that uses *key-object opportunism* [27] and enables frameworks to mark objects and indicate when to move them to H2. During GC, *TeraHeap* dynamically identifies the transitive closure of marked objects and moves them to H2.

Eliminating GC Cost for H2. TeraHeap presents a unified heap with the aggregate capacity of H1 and H2, where *scans over H2 during GC are eliminated,* to avoid expensive device I/O. To achieve this, *TeraHeap* organizes H2 into regions with similar-lifetime objects and deals differently with liveness analysis and space reclamation. We exploit the observation that analytics frameworks [55, 72] organize groups of objects with similar lifetimes, such as cached partitions in Spark, in data structures with a single-entry root reference. For liveness analysis, *TeraHeap* identifies live H2 regions by tracking forward (H1 to H2) and cross-region (into H2) references during GC. To identify live objects in H1, *TeraHeap* explicitly tracks backward references (H2 to H1) and fences GC scans in H2. *TeraHeap* tracks backward references using a card table optimized for storage-backed heaps, minimizing I/O traffic to the underlying device during GC. For space reclamation, the collector reclaims H1 objects as usual. For H2 regions, unlike existing region-based allocators [21, 46] *TeraHeap* resolves the space-performance trade-off for reclaiming space differently. Existing allocators reclaim region space eagerly by moving live objects to another region, which would generate excessive I/O for storage-backed regions. Instead, *TeraHeap* uses the high capacity of NVMe SSDs to reclaim entire regions lazily, avoiding slow object compaction on the storage device.

Applying TeraHeap. Managed big data analytics frameworks exhibit significant diversity in handling dataset sizes exceeding memory limits. One line of frameworks, such as Spark and Giraph, reduce memory pressure by temporarily offloading excess data off-heap over a storage device. Spark users explicitly store immutable cached data on the device, while Giraph transparently (without user hints) offloads mutable objects to the device. We modify the two frameworks to use *TeraHeap*. The use of *TeraHeap* is different in each framework: Spark uses *TeraHeap* to store immutable intermediate results, whereas Giraph uses *TeraHeap* to store mutable objects, such as edges and messages. Another set of frameworks, such as the Neo4j **Graph Data Science (GDS)** library is built on the assumption that all data fits in DRAM and does not move data off-heap. To process large datasets that exceed DRAM capacity in GDS, we show how it can exploit *TeraHeap* to offload objects to the storage device without refactoring its source code.

We implement *TeraHeap* and its mechanisms in OpenJDK8 and OpenJDK17, extending the **Parallel Scavenge (PS)** garbage collector. We also extend the interpreter and the C1 and C2 **Just-in-Time (JIT)** compilers to support object updates in H2 during application execution. Our evaluation shows that *TeraHeap* improves performance by up to 73%, 28%, and 26% compared to native Spark, Giraph, and GDS, respectively. *TeraHeap* provides similar or better performance by consuming up to 4.6× and 1.2× less DRAM capacity than native Spark and Giraph, respectively. Also, it outperforms Panthera [60], a garbage collector specialized for hybrid memories by up to 69%.

Overall, our work makes the following contributions:

—We introduce a dual heap approach to reduce S/D and memory pressure in big data frameworks by adding a second, high-capacity, managed heap over a fast storage device.
—We propose a hint-based interface based on key-object opportunism that enables frameworks to mark candidate objects in a coarse-grain manner and select when to move them to the second heap.
—We show the applicability of *TeraHeap* as: (1) a large, on-heap, compute cache in Spark to store intermediate results, (2) a H2 in Giraph to store messages and edges, and (3) a H2 in GDS to offload objects during graph processing.

This article builds upon the *TeraHeap* system [33] by presenting several extensions and improvements. First, we rigorously analyze the trade-off between a large managed heap to store long-lived objects versus moving these objects off-heap. Second, we extend the PS garbage collector in OpenJDK17 to work with *TeraHeap* and provide an in-depth evaluation of its performance, showing that *TeraHeap* still provides better performance in newer JVM versions. The key challenge in supporting OpenJDK17 includes multithreaded compactions during major GC in the H1. Third, we illustrate the applicability of *TeraHeap* in enabling in-memory frameworks that do not move objects off-heap to handle larger problem sizes (beyond DRAM) without needing *ad-hoc* solutions. For this purpose, we show how *TeraHeap* applies to and improves the performance of the GDS library compared to mapping the managed heap or part of the managed heap over a storage device. Fourth, we investigate how dividing a fixed amount of DRAM between H1 and (OS) page cache for H2 affects *TeraHeap*'s performance. Our goal is to understand better the scenarios in which the size of H1 or the size of the page cache has a more significant impact on performance. Fifth, we examine the sensitivity of *TeraHeap* to different storage devices technologies (predominant practice in the data center) by allocating H2 over SATA SSD and NVMe devices. Finally, we introduce a taxonomy of prior art and discuss the critical limitations of the state of the art in depth.

2 Background and Motivation

This section provides background on GC in Java runtimes and S/D of managed objects. We also discuss the tradeoff between storing serialized objects in off-heap memory and GC overheads due to a large heap.

GC. Modern collectors exploit the generational hypothesis that many objects die young. For this reason, they divide the managed heap into a young generation for new objects and an old generation for objects that survive multiple young (minor) collections [59]. They further divide the young generation into an eden space and two survivor spaces, called *from-space* and *to-space*. Application (mutator) threads allocate new objects into the eden space. When the eden space becomes full, garbage collectors perform a minor GC. During minor GC, the garbage collector identifies live objects in the eden space and from-space. Then, it moves live objects to the to-space and the mature objects to the old generation. When the managed heap becomes full, the JVM performs a full (major) GC, which scans and compacts objects in both old and young generations.

Although JVMs are typically used with only DRAM-resident managed heaps, today, they can allocate either the entire heap or the old generation over a storage device (e.g., NVMe SSD) using memory-mapped I/O (e.g., Linux *mmap*). Applications perform load/store operations for objects that now reside on SSD. These load/store operations go to DRAM and generate page faults. The kernel (mmap path) then moves data between DRAM and SSD using block operations, transparently to application code. However, existing garbage collectors are designed for DRAM-backed heaps and incur significant overhead for storage-backed heaps [69]. DRAM is byte addressable and provides low latency and high throughput regardless of operation types (read/write) and access patterns (random/sequential). On the other hand, fast-storage devices (e.g., NVMe SSDs) can only be accessed in page granularity. Page granularity accesses cause a significant increase in I/O traffic by transferring the entire page even if only a small portion of that page is required, resulting in a performance penalty [1, 52].

Object Serialization. Java serialization enables the conversion of a memory-resident object into a form that is convenient for storing it off-heap (memory, storage, or network) and can even be shared across JVMs. *Serialization* transforms Java objects in the managed heap into a byte stream, and *deserialization* reconstructs the Java objects from byte streams into heap representations (with references). During S/D, the serializer traverses the object graph to identify all objects that need to be serialized, starting from the root object selected for off-heap placement.

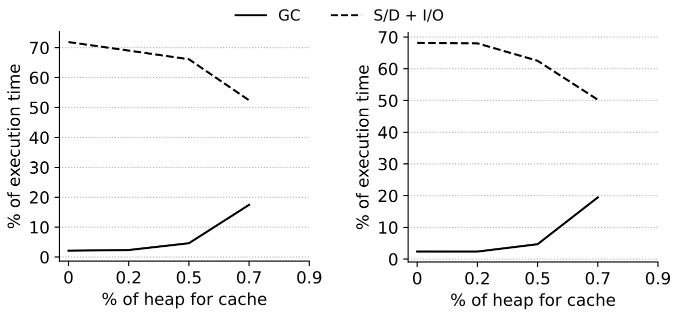
Java serialization is a complex process that introduces four significant limitations and overheads during execution. First, when serializing an object, the serializer omits fields marked with the *transient* modifier. Transient fields are initialized to a default value during deserialization based on the serializer implementation. Thus, serialization limits the objects that can be moved off-heap, as it requires self-contained entities without references to and from the managed heap, i.e., only serializable objects [24, 26, 48]. Second, extracting and recreating the object state requires mechanisms that bypass constructors and ignore class and field accessibility. Third, performance-wise, traversing the object graph requires effort proportional to the volume of objects in the transitive closure of the root object. Fourth and most relevant to our work, S/D generates many temporary objects while transforming objects into byte streams and vice-versa. Temporary objects put more pressure on the heap and lead to more frequent GC cycles. Recent work identifies S/D as a significant performance bottleneck in big data analytics frameworks [44, 45, 54, 58].

2.1 Tradeoff between S/D and GC Overheads

The datasets that applications need to process are growing continuously. However, the expectation is that only a small percentage of data fits in memory. Therefore, in such cases, frameworks offload objects off-heap over a fast storage device. On the one hand, serializing objects and moving them in off-heap memory reduces the heap requirements. However, it results in excessive S/D overhead. On the other hand, using a large managed heap to avoid S/D costs increases the GC overhead. Therefore, today, frameworks pay the overhead from S/D or GC. To show this tradeoff quantitatively, we demonstrate how Spark copes with large datasets by presenting experimental results from two representative Spark applications, **Linear Regression (LR)**, and **Logistic Regression (LgR)**. The two applications are representative of two popular computational patterns: data aggregation with varying space demands and iterative computations. Spark provides a compute cache and application written on top of Spark benefit from caching large amounts of intermediate results.

For our investigation, we vary the portion of memory (heap) allocated for execution and on-heap cache to investigate the effect on performance. Spark divides the available heap into memory required for task execution and on-heap memory for cached intermediate results. To determine the size of execution and on-heap cache memory for a given heap size, Spark uses F, a tunable parameter defined as the portion of the heap to be used as an on-heap cache ($F = C/H$). F is a soft boundary and varies between 0 and 1. The two functions, caching and execution, can exceed their share if the other function does not fully utilize its partition. We explore several values for F.

Figure 1 shows the tradeoff between GC and S/D as we vary F. The total heap size in LR and LgR is 64 GB. When $F = 0$, Spark serializes and offloads all the cached data to the storage device without keeping any cached data on the heap. As a result, when $F = 0$, S/D cost is higher by 24% and 30% compared to $F = 0.7$ in LR and LgR, respectively. However, when $F = 0.7$, GC time increases by



(a) Linear Regression (b) Logistic Regression

Fig. 1. Tradeoff between S/D and GC overhead for (a) LR and (b) LgR workloads in Spark.

88% compared to $F = 0$ in both workloads because the GC needs to scan more objects in each GC cycle. The GC is mainly affected by the increased memory pressure when the application keeps more data in the on-heap cache. The JVM triggers frequent GC cycles to free space for the newly allocated objects of execution memory.

3 TeraHeap Design

3.1 Overview

The key idea of *TeraHeap*, as shown in Figure 2, is to extend the JVM to use a second, high-capacity managed heap (H2) over a fast storage device that coexists with the regular managed heap (H1). Unlike DRAM-backed H1, H2 is memory-mapped over a storage device, allowing direct access to deserialized objects without S/D. Memory-mapped I/O eliminates the need to use a custom reference lookup mechanism in the JVM to identify objects on the device, as the OS virtual memory mechanism performs this translation. *TeraHeap* manages the two heaps differently and hides their heterogeneity, providing to big data applications the abstraction of a single managed heap.

Although *TeraHeap* is agnostic to the specific device that backs H2, the intention is to map H2 over fast storage devices, either block-addressable NVMe SSDs or byte-addressable NVM. Such devices are amenable to memory mapped I/O due to their high throughput and low latency for small request sizes (4 KB) regardless of the access pattern [52]. NVMe SSDs are particularly attractive as datasets grow because they provide high density (capacity) and lower cost per bit compared to DRAM and NVM [63].

We design *TeraHeap* based on our observations about objects and their management in big data analytics frameworks, as follows.

Which Objects to Move to H2 and When? We observe that different managed big data frameworks maintain off-heap stores to move (specific) long-lived objects that are reused across computation stages outside the managed heap. They organize groups of objects in data structures, such as arrays with a single-entry root reference (key objects). However, their off-heap objects have diverse access and update patterns. For example, Spark only moves immutable objects off-heap, and Giraph moves objects that are *eventually* immutable.

TeraHeap exposes a novel hint-based interface that uses key-object opportunism [27] to identify specific objects to move to H2, typically objects that would be moved off-heap. Frameworks use that interface to (1) tag with a label the root key object appropriate for placement to H2 and (2) advise *TeraHeap* when to move objects to H2. Decoupling the selection of the candidate root key objects from their transfer to H2 enables *TeraHeap* to cope with expensive read-modify-writes over the storage device. These hints are translated into two native function calls at runtime. The hint-based

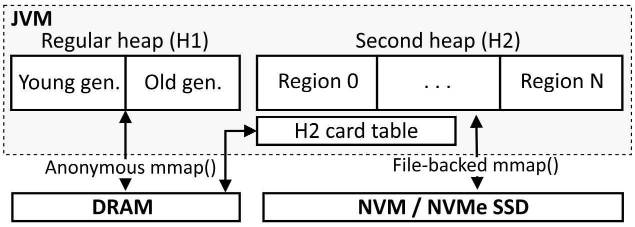


Fig. 2. TeraHeap design overview.

interface works at the framework level and is entirely transparent to applications written on top of such frameworks (Section 3.2).

How to Reclaim Dead Objects in H2 without GC Scans? Scanning the storage-backed H2 for liveness analysis and compacting objects for space reclamation incurs a high GC overhead due to excessive device I/O traffic and page faults. Liveness analysis is a graph traversal operation over a huge graph of live objects connected by references. Object graph traversal often suffers from random accesses with poor page locality, so the garbage collector potentially triggers a page fault as it follows each reference. Also, object compaction in H2 incurs high I/O traffic due to excessive read-modify-writes operations.

TeraHeap reduces the high GC overhead by organizing H2 in virtual memory as a region-based heap. Each region hosts object groups with similar lifetimes to reclaim dead objects in bulk. Big data frameworks operate on a distributed computing model where data are divided into partitions, and each partition is processed independently by a single task. All objects in a partition have the same lifespan. For this purpose, most objects that are reachable by root key-objects representing partitions exhibit a similar lifetime [21]. *TeraHeap* leverages the high capacity of NVMe SSDs to resolve the space-performance trade-off differently than existing work [21, 46]. Previous work targets DRAM-backed heaps and focuses on freeing address space eagerly by scanning regions and moving live objects with cross-region references to other regions. However, *TeraHeap* reclaims H2 space lazily with low overhead by freeing whole regions and their objects in bulk. We demonstrate the effectiveness of space reclamation in H2 through our evaluation. To ensure memory safety while reclaiming dead H2 regions, *TeraHeap* must take into account forward (H1 to H2) and cross-region references (Section 3.3).

How to Prevent Reclamation of Backward References (H2 to H1)? Fencing GC scans in H2 further requires tracking backward references from H2 to H1, as the garbage collector must not reclaim H1 objects referenced by live H2 objects. The key difficulty is that H2 objects can reference objects in both generations of H1 and need to be tracked differently. Young objects in H1 change location during minor GC, while old objects move only during major GC. Scanning H2 to identify backward references may incur significant overhead, depending on the size of H2 and its backing device. Instead, we use an extended card table for H2 to track backward references, optimized for storage-backed heaps (Section 3.4).

Next we discuss how *TeraHeap* solves the three main challenges related to: (1) identifying and moving candidate objects to H2, (2) reclaiming dead objects in H2 without GC scans and I/O traffic, and (3) tracking backward references (H2 to H1) with low GC cost and I/O overhead.

3.2 Identifying and Moving Objects to H2

TeraHeap provides a hint-based interface, enabling frameworks to tag root key-objects with a label for H2 movement, via a new field (eight bytes for alignment purposes) in the Java object header.

Alternatively, we can avoid this field by using additional JVM metadata for storing the address of each object that needs to be moved to H2. However, this would increase GC time because it requires updating object accesses in every GC cycle until they are moved to H2. The *TeraHeap* interface consists of the following function calls.

h2_tag_root(obj, label): The framework uses `h2_tag_root()` to tag a root key-object with a label.

h2_move(label): The framework uses `h2_move()` to advise *TeraHeap* to move all objects with the specified label to H2. During the next major GC, the garbage collector identifies the root key-objects tagged with the same label as the specified label. Then it detects and marks for moving to H2 objects in the transitive closure of the root key-object by tagging these objects with the same label as the root-key object.

Typically, frameworks can use `h2_move()` once their object group becomes immutable. However, immutability is not a strict requirement for movement to H2 and partly depends on storage device characteristics [30]. For instance, in Spark, all objects can be moved when marking the root key-object, whereas, in Giraph, objects are best moved at the end of each computation stage, possibly much later than when marking the root key-object.

Delaying the move to H2 runs the danger of creating **out-of-memory (OOM)** errors because H1 may fill before `h2_move()` is called. To avoid this, *TeraHeap* monitors the space that live objects occupy at the end of each major GC. If the live objects occupy more space in H1 than a high threshold (e.g., 85% of H1), *TeraHeap* will move marked objects to H2 during the next major GC without waiting for `h2_move()`.

At this point, if *TeraHeap* moves all marked objects to H2, it may incur excessive device traffic, e.g., in case some of these objects may be updated frequently prior to the application using `h2_move()`. To mitigate this effect, *TeraHeap* uses a low threshold mechanism as well, which limits how many marked objects will move to H2 when *TeraHeap* detects high H1 pressure prior to seeing an `h2_move()` hint. In our evaluation, we also examine the alternative of not using `h2_move()` and relying only on the high-low threshold mechanism.

Placing objects with the same label in the same H2 region allows *TeraHeap* to reclaim them en masse. However, the transitive closure might include specialized objects that can have a longer lifetime and delay the region from being freed. For this reason, *TeraHeap* excludes from the transitive closure: (1) JVM metadata, such as *class objects* [49] and the class loader, and (2) objects that inherit the *java.lang.ref.Reference* class [50].

The JVM automatically creates class objects upon class loading. Each class has one class object containing the static fields of the class and multiple instance objects. As all instances share the static fields, class objects are not reclaimed until the last instance object of the class dies, which triggers the JVM to unload the class. As a result, class objects may be updated frequently due to static fields and exhibit long lifetimes.

Objects that inherit from the java.lang.ref.Reference class (such as soft, weak, final, and phantom references) require special treatment by the garbage collector. The garbage collector maintains four lists, one for each kind of reference. During the object graph traversal, the garbage collector adds the discovered live objects with these reference types to the respective lists. Then, it applies a different collection policy for each reference type. For example, soft references keep the objects alive as long as memory is available on the heap but discard them before an OOM error. Thus, objects with soft, weak, final, and phantom references have different lifetimes than those in the closure, extending the region's lifetime and preventing free operations.

TeraHeap moves marked objects from H1 to H2 during major GC. To reduce this cost, *TeraHeap* uses explicit asynchronous I/O. We avoid multiple system calls for small-sized objects (<1 MB), using a promotion 2 MB buffer per region in H2 that writes objects to the device in batches.

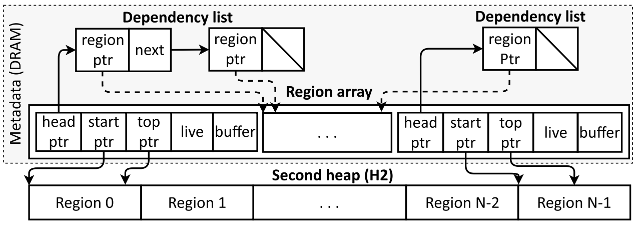


Fig. 3. H2 allocator metadata in DRAM.

3.3 Reclaiming Dead Regions

Figure 3 shows the region-based organization of H2 in virtual memory and each region metadata in DRAM. Unlike Broom [21], we do not impose any restrictions on regions, allowing objects in any region to refer to each other. *TeraHeap* ensures that while reclaiming a region, none of the objects in the region are referenced from live H1 or H2 objects in other regions. To find such regions, *TeraHeap* tracks cross-region and forward references without scanning H2 objects, which would generate excessive I/O.

Cross-Region References in H2. To allow internal H2 references *across* regions, *TeraHeap* tracks the direction of cross-region references. As shown in Figure 3, *TeraHeap* keeps a dependency list in per-region metadata in DRAM. Each node of the dependency list points to a (different) region referenced by objects of the current region. When we move objects to H2 we check if they have references in existing H2 regions. Then, the H2 allocator adds a new node (if it does not exist) to the dependency list of the region where objects will be moved. Another case we must consider is when an object in an existing H2 region (e.g., region A) references a candidate object in H1 that will be moved to a different region (e.g., region B) in H2. In this case, we update the dependency list of region A while updating the backward references from objects in H2 to objects in H1. The size of dependency lists is small, on average 10 nodes per region in our evaluation.

We also explore a simpler, *Union-Find* approach, using the notion of region groups that avoid tracking the direction of cross-region references. We track cross-region references by logically merging the source and destination regions in a single region group. Region groups grow over time to include all regions with cross-region references. If there is any reference from H1 to any object in the group's regions, then we consider the group alive. This approach does not consider the direction of region references, missing opportunities to reclaim dead regions with no incoming references. For instance, if there is a reference from region X to Y and a reference from region Y to Z, all three regions belong to the same region group and can be reclaimed when the whole group dies. We find that the direction of references matters for more efficient space reclamation. In the previous example, if only region Z is referenced by H1, then regions X and Y can still be reclaimed.

Forward References (H1 to H2). TeraHeap avoids scanning H2 objects by fencing the garbage collector from crossing into H2 from H1. This requires identifying all references from H1 to H2 and marking the referenced H2 objects as alive. *TeraHeap* uses a `live` bit in the per-region metadata (Figure 3) that signifies the objects in the region are reachable from H1. The garbage collector clears `live` bits at the beginning of the major GC. Upon encountering a reference from an object in H1 to an object in H2, the collector sets the corresponding region bit. If the dependency list of the current region is not empty, then we traverse the dependency lists of each dependent region recursively, setting their `live` bits, as well.

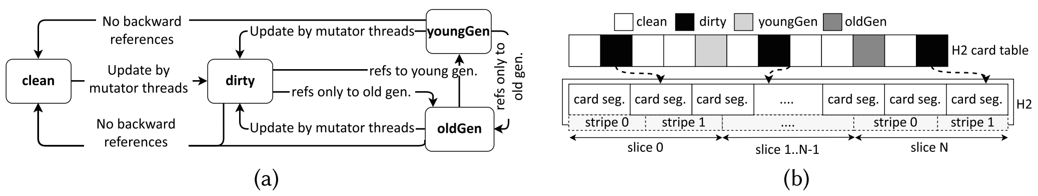


Fig. 4. (a) State transitions for H2 card table entries. (b) Organization of H2 as stripes and slices.

Freeing Dead Regions. At the end of major GC, any H2 region not marked as `live` is not reachable from any H1 object nor any H2 regions. To free these dead regions, we set their allocation pointer to zero and delete their dependency list (Figure 3). Upon JVM shutdown, we free all H2 metadata in DRAM.

3.4 Tracking Backward References (H2 to H1)

We use an extended card table for H2 to track backward references, optimized for use with storage devices. The H2 card table is a byte array (in DRAM) with one byte per fixed-size H2 segment (similar to vanilla JVM). Unlike H1, H2 is expected to be considerably larger, leading us to increase the size of H2 card segments to minimize card scanning overhead during GCs. Our evaluation suggests that a segment size of 8 KB is effective, in contrast to the 512-byte card segments used by the JVM's old generation. However, considering alternatives, such as a *remembered set* for more precise information about backward references, comes with trade-offs. While the 'remembered set' enhances precision, it concurrently amplifies memory consumption, especially as H2 scales with storage device capacity. Additionally, its implementation introduces a more intricate and expensive post-write barrier [18].

Setting H2 Card States. Using larger card segments requires scanning more objects, introducing device I/O in case they are dirty. To reduce the number of objects scanned during minor GC, we avoid scanning H2 objects that only reference objects in the old generation of H1, as the garbage collector does not move or reclaim objects in the old generation. Thus, we design an H2 card table where each card entry is in one of four states: (1) *clean*, when there are no backward references, (2) *dirty*, indicating object update by mutator threads, (3) *youngGen*, indicating references only to the young generation, or (4) *oldGen*, indicating references only to the old generation.

When an application thread updates an H2 object, *TeraHeap* marks the corresponding H2 card as dirty in the post-write barrier. As shown in Figure 4(a), during GC, we change the card value from dirty to oldGen if objects in the dirty card segment only reference objects in the old generation. Otherwise, we change the card value to youngGen. We set the card value as clean only if there are no backward references in the card segment. In minor GC, we only scan the objects in the card segments whose cards are marked as dirty or youngGen. In major GC we also scan oldGen objects. We adjust all backward references in both minor and major GC to refer to the new H1 object locations.

Scanning H2 Card Table. GC is multithreaded, and therefore, the H2 card table must support concurrent accesses from multiple threads without synchronization. Similar to H1, we divide H2 in slices and stripes to avoid contention between GC threads. As shown in Figure 4(b), each slice contains a number of fixed-size stripes equal to the number of GC threads. Each GC thread operates on the stripes with the same id in all H2 slices, avoiding thread contention.

In the native JVM, objects may span card segments and stripe boundaries. Given that a separate GC thread processes each stripe, two threads may need to access each boundary (first and last) card in a stripe. For this reason, the garbage collector avoids cleaning the boundary cards in H1. If H1 boundary cards become dirty, they remain dirty throughout execution. This phenomenon results in the garbage collector scanning in every GC the corresponding card segments for objects with backward references.

This extra scanning would be significant drawback for H2 because (1) we use large card segments to reduce H2 card table size and (2) the card segments are mapped to a storage device, resulting in high I/O traffic when scanning objects. *TeraHeap* resolves the issue of the dirty boundary cards by aligning objects to stripes and guaranteeing that no two threads will need to access the same card. *TeraHeap* uses stripe size equal to the H2 region size because *TeraHeap* guarantees that objects do not span H2 regions.

4 *TeraHeap* for PS GC

We implement *TeraHeap* in OpenJDK8 (jdk8u345) and OpenJDK17 (jdk17u067), both long-term support versions, by extending the PS [32] garbage collector and exporting *TeraHeap*'s interface through the *Unsafe* class to frameworks. We select PS over the Garbage-First garbage collector (G1GC) [73] because PS does not result in OOM errors when running with small heap sizes relative to datasets that contain large-sized (humongous) objects. Our evaluation (Figure 20) shows that G1GC is prone to OOM errors due to region fragmentation caused by humongous objects.

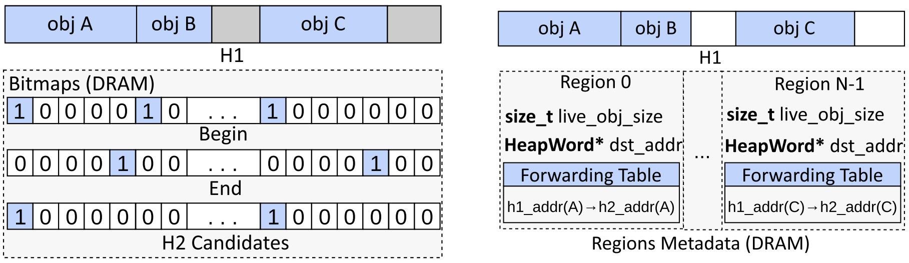
Post-Write Barriers. PS uses a post-write barrier and a card table to track updates in old generation objects that generate references to young objects. Such updates may originate from interpreted or JIT compiled methods with the C1 and C2 JVM compilers. When a mutator thread updates an object in the old generation, it also performs post-write barrier that updates the corresponding entry in the H1 card table.

To examine if the mutator thread updates an object that belongs to H1 or H2, we use an additional range check for references in the post-write barrier. This reference range check selects the appropriate (H1 or H2) card table, which we then mark with the existing post-write barrier code. We extend post-write barriers by augmenting the template-based interpreter and the JIT compilers to generate assembly code for the necessary checks. We evaluate the overhead of our modifications to post-write barriers using the DaCapo benchmark suite [7]. The overhead is small and within 3% of total execution time on average across all benchmarks. The additional overhead is zero for applications that do not enable `EnableTeraHeap`.

Minor GC. In minor GC, we perform two key tasks during liveness analysis. Firstly, we introduce a reference range check in the liveness analysis to fence PS from scanning objects in H2. Secondly, we prevent the reclamation of H1 objects referred from H2 objects (backward references) by scanning the H2 card table. Also, we update the backward references and H2 card values.

Major GC in Java8. The single-threaded version of major GC in PS, also known as Serial old, consists of four main phases, which we extend to support *TeraHeap*. In the first phase (marking), PS recursively scans both generations starting from roots (e.g., thread stacks) and marks live objects. PS assigns a new memory location to each live object in the second phase of major GC (pre-compaction). In the third phase (pointer-adjustment), PS adjusts the references of each object to point to the new location of the objects as determined in pre-compaction phase. In the final phase (compaction), PS moves objects to their new locations.

We extend the marking phase to perform five extra tasks. At the beginning of the marking phase, we reset all the `live` bits of the H2 regions. We mark all objects in H1 that are referenced by H2 as live. We add a reference range check (similar to minor GC) that detects forward references (H1 to H2) to fence PS from scanning objects in H2, and we set the `live` bit of the corresponding region.



(a) Marking Phase (b) Precompact Phase

Fig. 5. Extensions in PS garbage collector to work with *TeraHeap*.

We identify the root objects tagged with a label through *TeraHeap* interface and calculate their transitive closure. Finally, we free all dead regions in H2.

To determine which objects found in the marking phase should be moved to H2, we extend the pre-compaction phase. We assign these objects an address from H2 using their label.

We prolong the pointer adjustment phase to perform three additional operations: (1) we adjust all backward references to the new object locations in H1, (2) we identify the newly cross-region references, and (3) we track the newly created backward references. For this purpose, when we adjust the references of H1 objects that are candidates for H2 transfer, we check if they reference an existing H2 region or point to an H1 object. If a candidate object references an existing H2 region, we update the dependency list of the H2 region in which the candidate object is transferred. Also, in case the candidate object has a reference to an H1 object, we mark the appropriate entry in the H2 card table as dirty.

Finally, in the compaction phase, PS moves objects to H2.

Major GC in Java17. The major GC in PS in OpenJKD17 consists of three main phases, which we extend to support *TeraHeap*. In the first phase (marking), parallel GC threads scan both generations starting from roots (e.g., stacks from mutator threads, global variables, JIT methods) and mark live objects. In the second phase (summary), PS divides the heap into 512 KB regions and calculates sequentially for each region a destination address. In the third phase (compaction), firstly, the GC threads adjust in parallel the roots to point to the new location of the objects as determined in the summary phase. Next, the GC threads process the regions in parallel and copy the live objects to the beginning of each space (old generation and from-space in young generation).

PS uses two marking bitmaps to map the entire heap during the marking phase, as shown in Figure 5(a). Each bit in the bitmap corresponds to 64 bits of the heap because of the 8 byte alignment of fields in a JVM object. The first bitmap marks the start address of each live object, and the second bitmap marks the end address. To calculate the object's size, we multiply the bits between the enabled bits in the start and end bitmaps by eight. The collector divides the heap space into regions of 512 KB and maintains metadata for each object in DRAM. It updates the metadata of the region the object belongs to when it marks an object as live.

In our implementation, we extend PS to maintain a third bitmap during the marking phase, as shown in Figure 5(a). This bitmap tracks the start address of the objects that should be moved to the second heap (H2 candidates). In addition to the five extra tasks described in the major GC of Java8, we update the third bitmap for each object in the transitive closure. The third bitmap is essential in the H2 migration process and for updating H1 objects that reference H2 candidate objects, as it enables us to identify H2 candidate objects efficiently.

To make the space of H2 candidate objects available to the garbage collector to compact live H1 objects, we extend the metadata of each region to maintain a forwarding table for H2 candidate objects. In the case of an H1 object referencing an H2 candidate object, when we adjust its references, we perform a lookup in the forwarding table to find the new address of the candidate object in H2. Each entry of the forwarding tables, as shown in Figure 5(b) is 16 bytes because it stores for each candidate the old address in H1 (8 bytes) and their new address in H2 (8 bytes). We use our own allocator to optimize the memory allocation for the forwarding table. This approach avoids the overhead of multiple calls to the system `malloc()` and `free()` functions and can significantly improve performance. We note that forwarding tables might increase the metadata of JVM when we have hundreds of millions of objects to transfer to H2 in one GC cycle. Thus, we limit the size of the forwarding tables to 1 GB by transferring to H2 up to 67,108,864 objects per major GC cycle.

Avoiding forwarding tables requires storing inside the header of H2 candidate objects their new address in H2 (forwarding information). Although this approach avoids using extra DRAM space for metadata, it delays the time when the space of H2 candidates can be reclaimed. The GC must first process all objects that may reference the H2 candidate objects and then reclaim their space. For this purpose we need to split the compaction phase of PS into two sub-phases: a phase to adjust the references of the objects and a phase to compact H1 objects. We leave this investigation for future work.

At the start of the summary phase, we determine which objects found in the marking phase should be moved to H2. We scan the bitmap for H2 candidates, and for each candidate, we get an address from the H2 allocator. We assign this address to the forwarding table of the region. Next, we remove the candidate objects from the marking bitmaps (start and end) and decrease the total size of the live objects in the corresponding region.

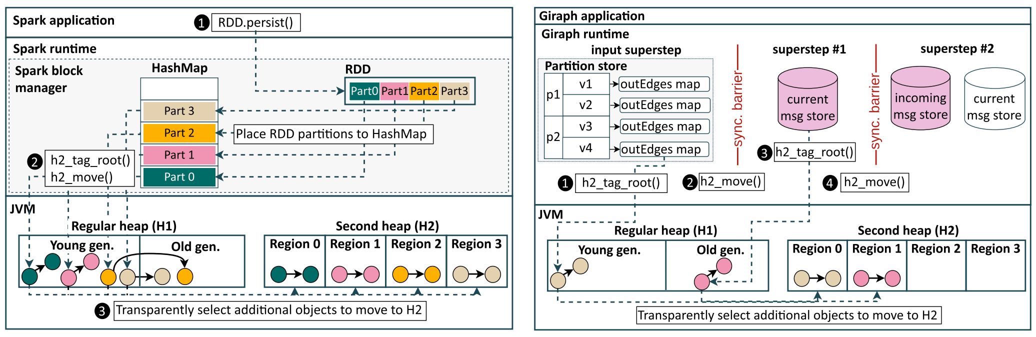
Before we compact H1 objects, we add one extra phase that adjusts the backward references and moves candidate objects to H2. In addition to the movement of the objects in H2, in this phase, we perform the same operations as described in the pointer adjustment phase of the major GC in Java8. Next, the PS continues to compact and adjust H1 candidate objects. At the end of the major GC, we reclaim all the allocated memory for the forwarding tables and clear marking bitmaps.

4.1 Applicability to Other Garbage Collectors

TeraHeap can be ported to work with other garbage collectors, such as G1GC. It is implemented as a new class within OpenJDK, providing an API for garbage collectors. For example, garbage collector developers can use this API to check whether an object belongs to the H2 or to mark an H2 region as live when it contains live objects. Developers must insert these API calls into the source code of each garbage collector, specifically at the points where liveness analysis is performed, new object addresses are assigned, object pointers are adjusted, and objects are moved to new locations. Furthermore, JVM developers need to modify the post-write barriers in both the interpreter and JIT compilers to update the H2 card table. In future work, we plan to port *TeraHeap* to G1GC and evaluate its performance in latency-sensitive applications, such as Cassandra [14] and Lucene [6].

5 Applications of *TeraHeap*

In this section, we describe how we integrate *TeraHeap* with two widely-used frameworks, Spark [72] and Giraph [55], which differ significantly in their use of off-heap memory. Spark operates with immutable objects, which presents a simpler scenario for *TeraHeap*, as immutability avoids the complexities of managing object mutations. In contrast, Giraph involves more complex mutation processes, requiring expensive read-modify-write operations to the storage device when moving updated objects off-heap. In the context of *TeraHeap*, these updates increase backward references from H2 to H1. This increase in cross-heap references stresses the GC process, as the GC must track



(a) (b)

Fig. 6. Use of TeraHeap in (a) Spark and (b) Giraph.

and manage these references. Additionally, we explore the application of *TeraHeap* in Neo4j's-GDS library [28], which cannot move objects in off-heap memory. Our approach enables GDS to handle problem sizes that exceed the limitations of DRAM.

5.1 Analytics Frameworks with Off-Heap Support

Spark uses off-heap memory to cache intermediate results, avoiding expensive recomputation. Cached objects are immutable at allocation time. Unlike Spark, Giraph offloads mutable objects, i.e., vertices, edges, and messages, to off-heap memory to ensure adequate DRAM is available for each superstep. Giraph updates vertex values throughout the computation, whereas edges and messages become immutable after graph loading (edges) or at the end of a superstep (messages).

Spark users explicitly annotate objects that need to be moved off-heap with the `persist()` call. Giraph transparently selects and moves objects to the storage device without application interaction. It maintains an *out-of-core* scheduler that monitors memory pressure in the managed heap and decides which vertices, edges, and messages to move off-heap. The out-of-core scheduler selects based on a **least recently used (LRU)** policy which objects to move off-heap.

Spark maintains deserialized objects in memory. This incurs significant S/D overhead during the off-heap movement. Giraph tries to reduce memory consumption on the managed heap and serializes vertices, edges, and messages into byte arrays, at allocation time. Therefore, Giraph does not require S/D when moving these byte arrays off-heap on the storage device. Next, we discuss how we extend Spark and Giraph to use *TeraHeap*.

Spark. We examine the use of *TeraHeap* as an on-heap compute cache in Spark. Spark requires only slight modifications to use *TeraHeap*. We note that Spark abstracts intermediate results as immutable collections using three sets of APIs [25]: resilient distributed datasets (RDDs) [71], DataFrames, and Datasets. To avoid time-consuming [65] recomputation of commonly used intermediate results across different jobs, Spark offers the flexibility of caching RDDs, Dataframes, or Datasets via its *persist()* API [71]. Users can persist RDDs, Dataframes, or Datasets in different storage levels: memory (on-heap), disk (off-heap), or both. When an RDD, Dataframe, or Dataset partition does not fit in storage memory, Spark serializes (e.g., using Kryo [56]) and moves another partition to a storage device, using an LRU policy. To use *TeraHeap*, we simply mark all cached partitions of RDDs, DataFrames, or Datasets, as root objects for moving to H2.

Figure 6(a) shows the flow of Spark caching operations using *TeraHeap*: ❶ The application code invokes `persist()` without any modifications. ❷ The Spark block manager places the selected data in the compute cache, a hashmap that contains all cached partitions. The block manager caches each partition independently, maintaining per-partition entries in the hashmap. When the block manager stores a new partition in the hashmap, we mark the partition descriptor as a root key-object with the `h2_tag_root()`, providing as label the RDD, dataset, or dataframe id. At the same time, we advise JVM to move marked partitions to H2, using `h2_move()`. *TeraHeap* allows Spark to cache all RDDs on-heap without replacing the Spark block manager. ❸ *TeraHeap* transparently marks additional objects and moves them to H2 during the major GC.

Giraph. Giraph computes in supersteps, with a synchronization barrier between supersteps. It loads and partitions the graph during the input superstep. A graph partition organizes its vertices in a hashmap, with each vertex belonging to a single partition. Each vertex maintains a map containing its outgoing edges. In each superstep, each vertex consumes all of its incoming messages from the previous superstep and updates its value. Then, it sends its updated value to its outgoing edges in a new message (per vertex). Messages produced in the current superstep are only consumed in the next superstep. Messages become immutable at the end of each superstep after the coordination phase guarantees they have been received and saved completely. In each superstep, Giraph has two message stores: the *incoming message store* with messages from the previous superstep (immutable) and the *current message store* with messages of the current superstep (mutable).

To use *TeraHeap*, Giraph requires small modifications, as well. We extend Giraph by marking edges and incoming messages as root objects. We do not mark vertices because they have frequent updates, and they will increase device (write) traffic. Note that edges and messages constitute a large portion of the heap [55]. Figure 6(b) shows the flow of Giraph execution using *TeraHeap*: ❶ When Giraph loads a new vertex at the input superstep, it marks the vertex's map that contains the outgoing edges with `h2_tag_root()`, providing the superstep id as label. ❷ At the end of the input superstep, Giraph advises *TeraHeap* to move marked edges to H2, using `h2_move()` in the next major GC. ❸ In each superstep, Giraph marks the generated messages of the current message store with `h2_tag_root()`, providing as label the superstep id. ❹ At the beginning of each (next) superstep, Giraph advises *TeraHeap* to move to H2 all marked messages from the previous superstep in the next major GC, using `h2_move()`.

5.2 Analytics Frameworks without Off-Heap Support

We aim to demonstrate the effectiveness of *TeraHeap* in analytics frameworks that lack support for offloading objects off-heap. GDS is a library for the Neo4j graph database, which provides many graph algorithms that can be used to analyze and extract insights from the graph stored in the database. GDS loads the graph data into memory by representing the graph as a **compressed sparse row (CSR)** format to reduce its size. Although this technique can reduce memory demands, it becomes less space efficient as the graph size increases [16].

Enabling the GDS to use *TeraHeap* requires inserting the hints in the GDS source code, which involves modification to two lines of code. *TeraHeap* does not change the CSR format of the graph in memory and allows GDS to operate in the usual manner. Importantly, it imposes no limitations on the operations that GDS perform over the graph. Figure 7 shows the flow of GDS using *TeraHeap*. ❶ Using a cypher query, GDS loads the graph from the Neo4j database. GDS uses `h2_tag_root()` to mark vertices and edges by providing as a label the thread ID. ❷ After creating the projected graph representation in memory, GDS uses `h2_move()` to advise *TeraHeap* to move all vertices and edges to the H2. After that, it continues the normal execution path.

5.3 Applicability to Other Frameworks

Given that *TeraHeap* is a JVM-level solution, higher layers can use *TeraHeap* transparently by only providing simple hints. The API of *TeraHeap* can be adopted by any big data framework that

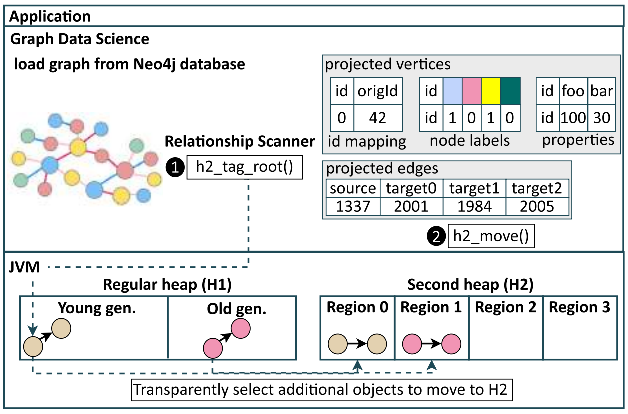


Fig. 7. Use of TeraHeap in GDS of Neo4j.

creates objects that span multiple computation steps and require a large amount of memory. Such frameworks include Hadoop [43], Flink [13], search engines like Lucene [6], or database systems such as Cassandra [14]. For example, Lucene [6] allocates its query cache on the heap, where frequently executed queries are stored to avoid recomputation. However, managing the query cache size can significantly impact GC. To mitigate this issue, Lucene can utilize the hint-based interface of *TeraHeap*, enabling the allocation of its query cache to H2, thus optimizing performance. Similarly, Cassandra [14], which traditionally stores cached rows off-heap, could further enhance its efficiency by leveraging *TeraHeap* to transfer cached rows to H2, incorporating specific API calls directly into its codebase for seamless integration.

6 Experimental Methodology

We answer the following questions in our evaluation:

(1) How does *TeraHeap* perform compared to native JVM and state-of-the-art Panthera with NVMe SSDs and NVM?
(2) What are the space requirements of H2 in the storage device and for its metadata in DRAM?
(3) What is the overhead of tracking references and moving objects to H2 during GC?
(4) How does *TeraHeap* scale with an increasing number of mutator threads and dataset size?
(5) How does the performance of *TeraHeap* vary based on (1) the distribution of DRAM between H1 and the page cache for H2, and (2) the type of storage device used?
(6) How does *TeraHeap* perform compared to newer garbage collectors?

Server Infrastructure. We evaluate *TeraHeap* both with block-addressable NVMe SSDs, SATA SSDs, and byte-addressable NVM as the backing device for H2. Table 1 shows the properties of each server. Our NVM server operates in two modes: (1) *App Direct mode* uses 192 GB DRAM as main memory and 2 TB NVM as persistent storage device. (2) *Mixed mode* partitions NVM to use 1 TB in memory mode and 2 TB in App Direct mode. DRAM (192 GB) acts as a cache for 1 TB NVM (memory mode) controlled by the CPU's memory controller. In App Direct mode, the system mounts NVM on an ext4-DAX file system to establish direct mappings to the device.

Baseline and TeraHeap Configurations. We use Spark v3.3.0 with the Kryo serializer [56], a state-of-the-art highly optimized S/D library for Java that Spark recommends. We run Spark with OpenJDK17. We run Giraph v1.2 with Hadoop v2.4 and OpenJDK8, as it does not support more

Table 1. NVMe and NVM Server Properties

ID	CPU	DRAM	Storage	Kernel
NVMe Server	Intel Xeon E5-2630 32 Cores @ 2.4 GHz	256 GB	2 TB Samsung PM983 PCI Express NVMe SSD	4.14
SATA Server	Intel Xeon E5-2630 32 Cores @ 2.4 GHz	256 GB	2× 256 GB Samsung SATA SSD 850	4.14
NVM Server	Intel Xeon Platinum 24 cores @ 2.4 GHz	192 GB	3 TB Intel Optane DC Persistent Memory	3.10

Table 2. Summary of Baselines

Baseline	DRAM	NVMe SSD	NVM (App Direct mode)	NVM (Memory mode)
Spark-SD	Heap	Off-heap	-	-
Spark-SD	Heap	-	Off-heap	-
Spark-MO	-	-	-	Heap
Giraph-OOC	Heap	Off-Heap	-	-
GDS-Heap	-	Heap	-	-
GDS-OldGen	Young gen.	Old gen.	-	-

recent versions of OpenJDK. We use an executor with eight mutator threads for both Spark and Giraph. Also, we run GDS using Neo4j v.5.15 and GDS v.2.7 with OpenJDK17. For GDS, we use four mutator threads, the maximum number of threads that the GDS community edition supports [28]. Our experiments use two garbage collectors in different configurations: PS in OpenJDK8 and OpenJDK17 and Garbage First (G1) [73] in OpenJDK17. For PS, we use 16 GC threads. G1 uses two parameters: (1) the number of parallel GC threads, which we set to eight (max value) and (2) the ratio of concurrent to mutator threads, which we set to two as the recommended configuration, is one-fourth of the parallel GC threads [5]. In our experiments, we utilize eight cores while disabling the remaining cores of the servers. For the performance scaling experiments (Section 7.8), we use 4, 8, and 16 cores to match the number of mutator threads.

Table 2 summarizes the Spark, Giraph, and GDS configurations we use as baselines. The two Spark-SD configurations place executor memory (heap) in DRAM and cache RDDs in the on-heap cache, up to 50% of the total heap size. Any remaining RDDs are serialized in the off-heap cache, over either NVMe SSD (first line of Table 2) or NVM in App Direct mode (second line of Table 2). Spark-MO places executor memory (heap) over NVM in memory mode, caching all RDDs on-heap. Giraph-OOC places the heap in DRAM and offloads vertices, edges, and messages off-heap to the NVMe SSD. GDS-Heap allocates the managed heap on an NVMe SSD using memory mapped I/O by setting the *-XX:AllocateHeapAt* runtime flag. GDS-OldGen allocates only the old generation of the managed heap on an NVMe SSD using memory mapping (setting the *-XX:AllocateOldGenAt* runtime flag) and the young generation in DRAM.

We configure *TeraHeap* to allocate H1 on DRAM and H2 over a file in NVMe SSD or NVM via memory-mapped I/O (mmio). The file in both NVMe and NVM servers is mapped to the JVM virtual address space where the application can access the data with regular load/store instructions [52]. Our experiments show that machine learning (ML) workloads in Spark access the individual elements of cached RDD partitions sequentially. For this reason, we configure *TeraHeap* for Spark ML workloads to use huge pages (2 MB) in H2 to reduce the frequency of page faults. Instead of

Table 3. Configuration for Each Workload on NVMe and NVM Servers for Spark-SD, Spark-MO, and *TeraHeap*

	GB	NVMe Server DRAM	Dataset Size	Spark-SD Heap	Spark-MO Heap	TeraHeap H1
GraphX	PageRank (PR)	80	32	64	1,024	64
	Connected Components (CC)	84	32	68	1,024	68
	Shortest Path (SSSP)	58	32	42	650	42
	SVDPlusPlus (SVD)	40	2	24	500	24
	Triangle Counts (TR)	80	2	64	64	64
MLlib	Linear Regression (LR)	70	256	54	1,084	54
	Logistic Regression (LgR)	70	256	54	1,084	54
	Support Vector Machine (SVM)	48	256	32	620	32
	Naive Bayes Classifier (BC)	98	21	82	82	82
SQL	RDD-RL	63	16	47	96	47

Table 4. Giraph-OOC and *TeraHeap* Configurations for Each Workload on Our NVMe Server

GB	NVMe Server DRAM	Dataset Size	Giraph-OOC Heap	Giraph-OOC DR2	TeraHeap H1	TeraHeap DR2
PageRank (PR)	85	31	70	15	50	35
Community Detection Label Propagation (CDLP)	85	31	70	15	60	25
Weakly Connected Components (WCC)	85	31	70	15	60	25
Breadth-first Search (BFS)	65	31	48	17	35	30
Shortest Path (SSSP)	90	31	75	15	50	40

the native *mmap*, we use HugeMap [41] a custom, open source, mmio path that enables huge pages for file-backed mappings.

In Spark-SD, to capture the effect of large datasets and limited DRAM capacity [15], we use a small heap size that caches a limited number of RDDs on-heap and the rest off-heap (Spark-SD column in Table 3). In Spark-MO we find and use the minimum heap size that fits all the cached data on-heap (Spark-MO column in Table 3). In Giraph-OOC, we experimentally find the minimum heap size for each workload (Giraph-OOC Heap column in Table 4). *TeraHeap* uses the same amount of DRAM as Spark-SD and Giraph-OOC but divides it between H1 (DR1) and system (DR2). The DRAM devoted to system use (DR2) includes the Spark and Giraph drivers and the kernel page cache for I/O.

In GDS-OldGen, by default, the JVM sets the maximum value of the young generation to be 80% of the available DRAM to balance the tradeoff between GC efficiency and heap size. However, setting the maximum size of the young generation to 80% can lead to longer pause times during GC. Thus, we investigate different values for the maximum size of the young generation and provide the best configuration for our experiments. Specifically, for PR and WCC, the young generation is 30% of DRAM, while for CDLP is 20% of the DRAM.

For the division of DRAM, we explore H1 sizes between 50% and 90% of DRAM capacity, and we report results with a configuration hand-tuned for each workload. Tables 3 and 4 show the H1 size of *TeraHeap* in each workload, in Spark and Giraph, respectively. DR2 is always 16 GB for Spark, whereas Table 4 shows the DR2 size for Giraph-OOC and *TeraHeap* in each Giraph workload. We limit the available DRAM capacity in our experiments with the NVMe server using *cgroups*. Tables 3 and 4 show the total DRAM capacity in the NVMe server for each workload.

Table 5. Parameters We Use with Each GDS-Neo4j Configuration in Our Experiments

GB	NVMe Server DRAM	Dataset Size	GDS-Heap DR2	GDS-OldGen YoungGen	GDS-OldGen DR2	TeraHeap H1	TeraHeap DR2
PageRank (PR)	14	70	14	4	10	11	3
Community Detection Label Propagation (CDLP)	14	70	14	4	10	11	3
Weakly Connected Components (WCC)	30	70	30	6	24	22	8

Workloads and Datasets. We use ten memory-intensive workloads from the Spark-Bench suite [36]. For Giraph and GDS, we use workloads from the LDBC Graphalytics suite [29]. SparkBench provides a data generator which we use to generate input datasets for our Spark workloads. For Giraph and GDS workloads, we use the datagen-9_0-fb dataset [29] and datagen-9_4-fb dataset [29], respectively. Tables 3–5 show the dataset size for each workload.

Execution Time Breakdowns and S/D Overhead. Our experiments are long-running, and to navigate the design space in a practical timeframe, we repeat each experiment five times, reporting the average end-to-end execution time. We observe low run-to-run variation (up to 2%), and five iterations suffice. We break execution time into four components: *other* time, S/D + I/O time, minor GC time, and major GC time. Other time includes mutator threads time. In *TeraHeap*, the other time potentially includes I/O wait due to page faults when accessing the H2 backing device. In Spark-SD (see Table 2), S/D time includes S/D time both for shuffle and caching. In *TeraHeap* and Spark-MO (see Table 2), all S/D time is due to shuffling. The JVM reports the time spent in minor and major GC.

To estimate S/D overhead, which occurs in mutator threads, we use a sampling profiler [51] to collect execution samples from the mutator threads. The samples include the stack trace, similar to the flame graph [23] approach. Then we group the samples for all the paths that originate from the top-level `writeObject()` and `readObject()` methods of the KryoSerializationStream and KryoDeserializationStream classes. These samples include both S/D for the compute cache and the shuffle network path of Spark. We then use the ratio of S/D samples to the total application thread samples as an estimate of the time spent in S/D, and we plot this separately in our execution time breakdowns. We run the profiler with a 10 ms sampling interval, verifying that this does not create significant overhead (less than 2% of total execution time).

7 Evaluation

7.1 Performance under Fixed DRAM Size

First, we investigate the performance benefits of *TeraHeap* under a fixed DRAM size. Figure 8 shows the performance of *TeraHeap* compared to Spark-SD and Giraph-OOC for the NVMe SSD setup, using OpenJDK17 for Spark and OpenJDK8 for Giraph. We normalize execution time to the first bar in each figure. Missing bars indicate OOM errors.

Using the same DRAM size, *TeraHeap* reduces execution time in Spark between 18% (SSSP) and 73% (BC) compared to Spark-SD. In Giraph, *TeraHeap* reduces execution time between 21% (CDLP) and 28% (PR). In both cases, the performance improvement results from reducing the GC overhead, by up to 96% and 54% in Spark and Giraph, respectively. This overhead occurs mainly because cached objects in Spark and messages and edges in Giraph occupy almost half of the heap, triggering GC more frequently. *TeraHeap* transfers objects to H2, stressing H1 less.

In addition, *TeraHeap* reduces S/D cost in Spark-SD, between 2% (BC) and 93% (LR), as it provides direct access to deserialized objects in H2. Note that S/D cost in TR and BC for *TeraHeap* is similar

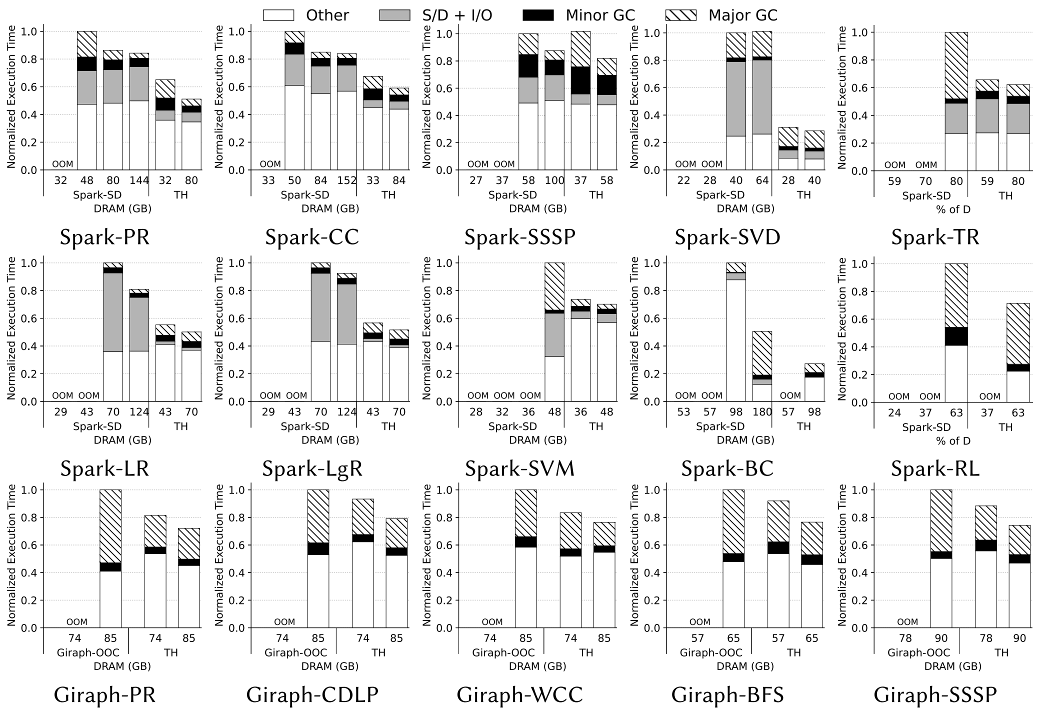


Fig. 8. Overall performance of TeraHeap (TH) compared to Spark-SD and Giraph-OOC on the NVMe server.

to Spark-SD because the cached data fits in the on-heap cache. In Giraph, the impact of *TeraHeap* on S/D overhead (part of other) is minimal because Giraph serializes objects in the managed heap as well, and not only as part of moving objects off-heap. Also, in LR, LgR, and SVM other time with *TeraHeap* increases by up to 43% compared to Spark-SD. These workloads perform streaming access on elements of cached RDDs in each iteration of the ML training phase, which is the largest part of the execution (100 iterations). Thus, they do not exhibit locality in the I/O page cache, fetching data from the storage device during the computation. The average read throughput in these workloads is 2.9 GB/s, which is the peak device read throughput. Using more NVMe SSDs can reduce other time for LR, LgR, and SVM.

To examine pressure on the managed heap, Figure 9 shows GC behavior for PR with Spark-SD and *TeraHeap* with a 64 GB heap. We examine the execution time for each minor and major GC cycle and monitor the percentage of the old generation consumed by cached objects. We note that Spark-SD suffers from frequent major GC cycles. There are 171 major GC cycles, each requiring, on average, 3.7 s. Each cycle in Spark-SD reclaims 10% of the old generation objects (0-3000 s in Figure 9), as the remaining objects are live cached objects. However, *TeraHeap* performs only 13 major GC cycles. Each cycle in *TeraHeap* takes, on average, 16 s. More than 70% is due to I/O during the compaction phase of major GC. Finally, moving objects directly from the young generation to H2 reduces total minor GC time by 38% compared to Spark-SD. This reduction is because *TeraHeap* scans fewer cards that track old-to-young references than Spark-SD.

To illustrate the potential for eliminating S/D without increasing GC overhead, Figure 10 shows the RDD access patterns for PageRank (PR) and Connected Components (CC). In our experiments, each RDD has 256 partitions. The horizontal axis shows time (seconds), and the vertical axis shows the IDs of accessed RDD partitions (from 1 to maximum partition ID for each RDD). For each ID,

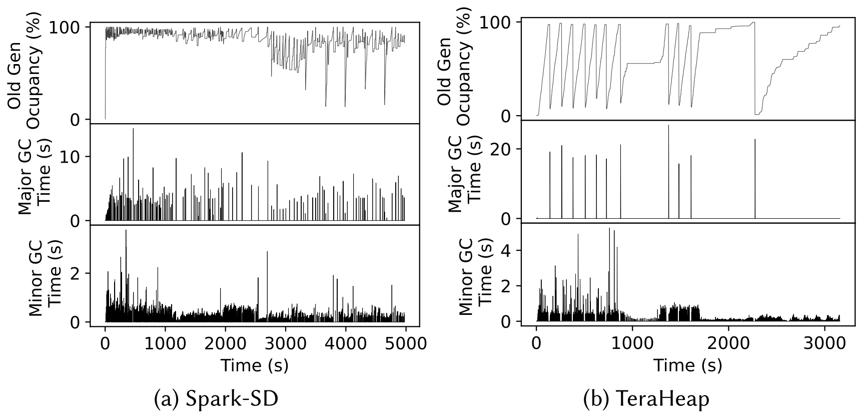


Fig. 9. GC time and old generation occupancy in PR for (a) Spark-SD and (b) TeraHeap. The heap size is 64 GB.

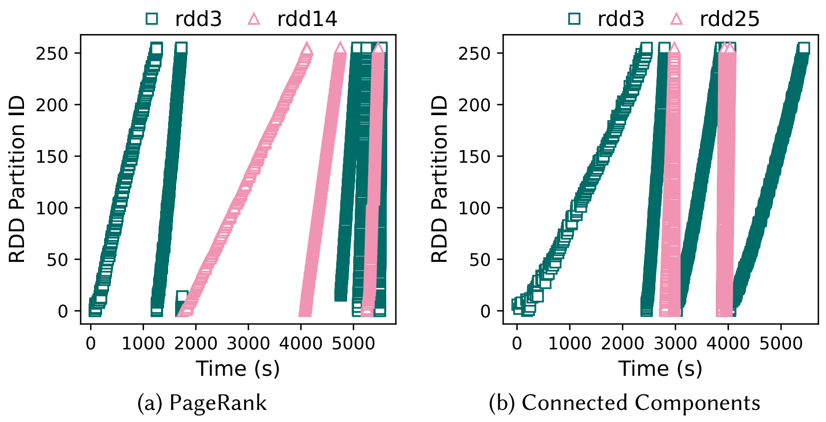


Fig. 10. Analysis of cached RDD accesses in PageRank (PR) and Connected Components (CC).

the first dot marks the time when that partition is cached. Each subsequent dot represents access to this partition by the application.

Once Spark needs an RDD for computation, it tends to access all partitions of an RDD sequentially. For this reason, in Figure 10, RDD accesses form "lines" and large intervals exist between accesses to RDD partitions. For example, in PR and CC, Spark accesses the first partition of RDD3 on its creation around $T = 0$ s but then again only after at least 1,500 s (PR) and 2,500 s (CC), respectively. We observe similar temporal gaps between accesses to RDD14 (PR) and RDD25 (CC) partitions. For certain RDDs, each partition is potentially accessed multiple times. For instance, for RDD25, each partition is accessed twice around $T = 3{,}000$ s. However, there is still a significant period of inactivity (about 1,000 s) until the next set of accesses around $T = 4{,}000$ s. Occasionally, successive accesses to the same partition exhibit temporal locality because two jobs take turns to cache and materialize partitions. For example, at $T = 1{,}100$ s, Job0 creates and caches the last partition of RDD3, and then Job1 materializes this partition at $T = 1{,}200$ s. Overall, the time interval between accesses to each partition varies between 100 and 1,500 s in both applications. This behavior allows us to place cached RDDs (unlike temporary, short-lived RDDs) on storage devices (off-heap) that offer high capacity.

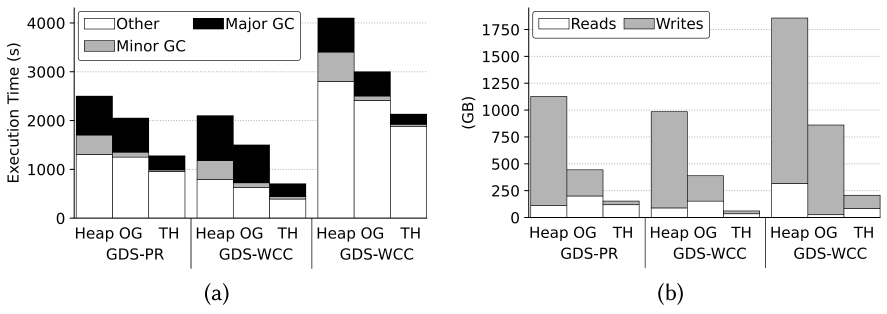


Fig. 11. (a) Performance and (b) device I/O traffic of GDS with three configurations: TeraHeap (TH), native with the full heap mapped to a device (Heap), and native with the old generation mapped to a device (OG).

Furthermore, data objects in each partition have similar lifetime. When applications *unpersist* an RDD, Spark drops all RDD partitions from its cache. At this point, in most cases, no references exist to the corresponding JVM objects. This observation reveals an opportunity to reclaim cached objects en masse by organizing the compute cache in groups of data objects with similar lifetimes.

7.2 Reducing DRAM Capacity Demands

We examine the potential benefit of *TeraHeap* in reducing DRAM capacity demands in Figure 8. Using between 1.3× and 4.6× less DRAM, *TeraHeap* outperforms by up to 65% (SVD) compared to Spark-SD. In Giraph, *TeraHeap* with 1.2× less DRAM improves performance between 7% (CDLP) and 18% (PR). For example, using *TeraHeap* in Giraph-PR, the heap usage in the first phase of the application (0–330 s) is between 70% and 100%. Then, at the end of the fifth major GC, *TeraHeap* reduces heap usage to 13% because it moves 17 GB of objects to H2. By reducing memory pressure in H1, *TeraHeap* with less DRAM can provide similar or higher performance than Spark-SD and Giraph-OOC.

7.3 Effectiveness of *TeraHeap* for In-Memory Frameworks

This section evaluates *TeraHeap* with GDS library in Neo4j using OpenJDK17. We aim to show that *TeraHeap* can be effective with analytics frameworks written for in-memory operations only. Unlike Spark and Giraph, GDS requires keeping all the graph data in memory and not offloading part of the graph off-heap. For this purpose, we compare *TeraHeap* with two possible approaches that GDS can use today by enabling the OpenJDK to: (1) allocate the managed heap on an NVMe SSD (GDS-Heap) and (2) allocate only the old generation over NVMe SSD (GDS-OldGen). Figure 11(a) shows our results.

Our experiments show that executing applications using GDS-Heap approach increases execution time between 43% and 63% compared to using *TeraHeap.* Although DRAM is used as a large page cache in this approach, the JVM cannot control where each object is placed (page cache or storage), and therefore it does not avoid expensive GC scans. Instead, the OS decides which pages to write to the device and which pages to keep in the page cache based on an LRU policy. Placing arbitrary objects in the storage device increases minor GC time by up to 91% compared to *TeraHeap.* This increase in time is because the application reads and updates young objects over the device. Using *TeraHeap*'s approach of using two separate heaps to avoid GC in the storage device is essential for maintaining performance.

On the other hand, *TeraHeap* still outperforms between 21% and 26% compared to GDS-OldGen. Minor GC time decreases when allocating the entire heap over the storage device. However,

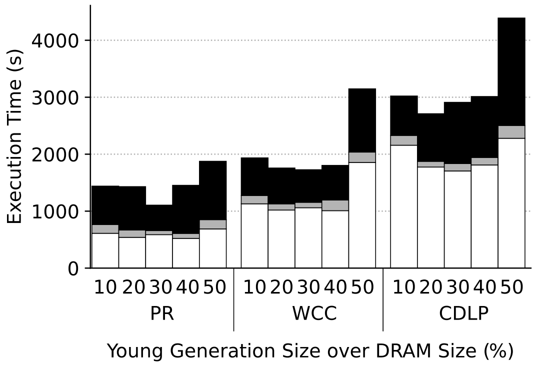


Fig. 12. Sensitivity analysis for the size of the young generation over the total DRAM size. As the young generation's size increases, the old generation's page cache, which is memory-mapped over NVMe SSD, decreases.

increasing the young generation's size in DRAM reduces the page cache size for the objects in the old generation. As we see, there is no sweet spot for the division of DRAM between the young generation and the page cache for the old generation.

Figure 11(b) depicts the read and write traffic to the storage device of *TeraHeap* compared to mapping the full heap over the storage device and mapping only the old generation over the storage device. Overall, *TeraHeap* reduces I/O traffic by up to 93% (WCC) compared to both approaches. When the young generation is in DRAM reduces the I/O traffic compared to allocating the entire heap over the storage device because it absorbs the read and writes to young objects in DRAM. Notably, in configurations where the young generation constitutes 80% of the total DRAM (Figure 12), there is an increase in read traffic due to the limited size of the page cache (3 GB for PR and WCC, and 6 GB for SSSP, respectively). The size of the page cache directly impacts accesses to objects in the old generations by mutator threads, the major GC time, and the minor GC time involved in updating references from objects in the old generation to those in the young generation.

7.4 Effects of Transfer Hint and Low Threshold

This section examines the performance effect of using the transfer hint `h2_move()`. Figure 13(a) shows the performance of *TeraHeap* with and without using `h2_move()`. Frameworks use `h2_move()` to advise *TeraHeap* when to move objects with a specified label to H2. Note that in case of high memory pressure, *TeraHeap* moves to H2 all marked objects without waiting for `h2_move()` hints. Given that *TeraHeap* can use only the high threshold mechanism to decide when to move objects to H2, we explore eliminating `h2_move()`. This results in objects staying longer in H1. With a high threshold of 85%, we see (Figure 13(a)) that using `h2_move()` improves *TeraHeap* performance between 29% (SSSP) and 55% (WCC) compared to not using the hint. In WCC, using `h2_move()`, we move objects to H2 on average every 215 s, reducing GC cost by 39% compared to not using the transfer hint, which transfers objects on average every 485 s. Moving objects with frequent updates to H2 increases other time by up to 59% (WCC) due to the large cost of read-modify-write operations on an I/O device. This increases device traffic by up to 98% (writes) due to page-based access to the device. Thus, using the transfer hint is necessary to delay moving objects with frequent updates to H2 until they become immutable.

Next, we study how effective is the *TeraHeap* low threshold mechanism. Figure 13(b) shows *TeraHeap* performance using `h2_move()` with and without a low threshold. We use a low threshold of 50% (and we leave the high threshold to 85%). *TeraHeap* will move objects until it reduces H1

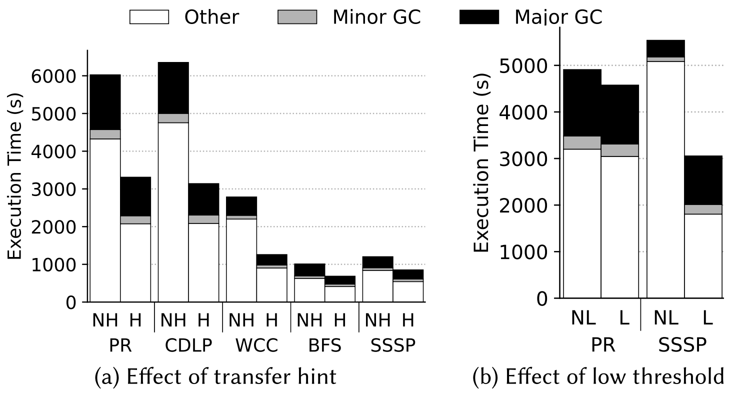


Fig. 13. TeraHeap performance for Giraph (a) with (H) and without (NH) transfer hint, and (b) with (L) and without (NL) low transfer threshold.

Table 6. Metadata Size per TB of H2 Space

Region Size (MB)	1	2	4	8	16	32	64	128	256
MetaData Size (MB)	417	209	104	52	26	13	7	3	2

usage to 50%. We use PR and SSSP with a large dataset (91 GB) in Giraph. These two workloads trigger the high threshold mechanism. We use 170 GB DRAM and 200 GB DRAM in PR and SSSP, respectively. The percentage of DR1 over total DRAM is similar to the corresponding workload in Figure 13(a).

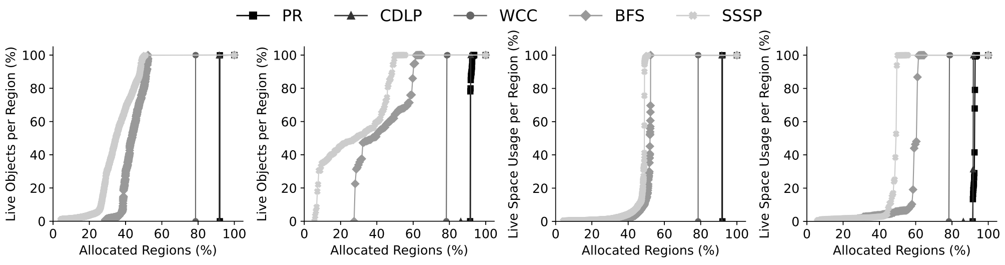
Using a low transfer threshold improves *TeraHeap* performance by up to 44% (SSSP), compared to using only the transfer hint with the high threshold. For example, in SSSP, during graph loading, we detect high memory pressure in the fourth major GC. After the fourth major GC, most objects in H1 are related to marked edges. Then, in the fifth major GC, we move 44 GB of marked objects to H2, reducing H1 usage to 50%. Therefore, the low transfer threshold reduces read-modify-write operations on the device by up to 95%, decreasing the other time by up to 65%. Although there may be benefits in setting the low and high thresholds dynamically, we leave this for future work.

7.5 Storage Capacity Consumption

This section investigates the storage requirements of H2. *TeraHeap* organizes object groups with a similar lifetime in fixed-size regions in H2 and reclaims them in bulk. This approach may result in waste of space for two reasons. First, when the number of objects in a group is small, unused space in the corresponding region can be large. Second, one live object can keep the entire region alive, preventing *TeraHeap* from reclaiming it. Generally, a smaller region size reduces both of these factors at the cost of increasing metadata in DRAM.

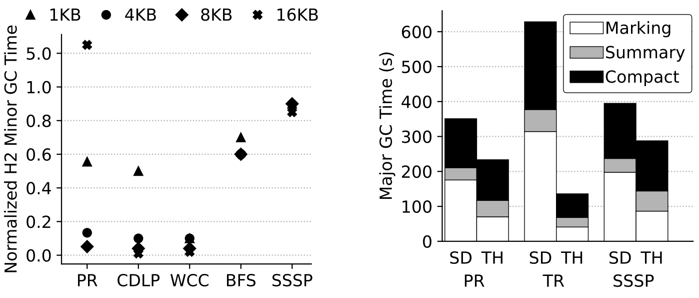
We first show how the region size affects the metadata size for H2. Table 6 shows the metadata size in DRAM per TB of H2, for region sizes between 1 MB and 256 MB. As we increase the region size from 1 MB to 256 MB, the total metadata in DRAM decreases from 417 MB to 2 MB.

Figure 14(a) and (b) shows the CDF of the percentage of live objects per region for *all allocated* regions with 16 MB and 256 MB size, respectively. Figure 14(c) and (d) shows the CDF of the percentage of space occupied by live objects for 16 MB and 256 MB regions. The number of allocated regions is equal to the sum of reclaimed regions during execution and the active regions before JVM shutdown. Although not shown in these figures, we observe in our measurements that unused



(a) Region size = 16 MB (b) Region size = 256 MB (c) Region size = 16 MB (d) Region size = 256 MB

Fig. 14. CDF of the percentage of live objects (a and b) and space occupied by live objects (c and d) during execution.



(a) (b)

Fig. 15. (a) Minor GC time in H2 for different card segment sizes using a 256 MB stripe size in Giraph. (b) Major GC time using Spark-SD (SD) and TeraHeap (TH).

space is between 1% and 3% for all workloads in both region sizes. Essentially, *TeraHeap* is able to use the space in each region it allocates with its append-only placement.

In Figure 14 (a) and (b) all regions with 0% live objects are reclaimed during execution. We see that in PR, CDLP, and WCC, *TeraHeap* reclaims most allocated regions and around 90% in CDLP and PR for both region sizes. In BFS and SSSP, *TeraHeap* reclaims 28% and 6% of the total allocated regions, respectively. In BFS and SSSP, although most of the objects in a region are live, most of the space is occupied by large dead arrays. For example, in SSSP with 256 MB regions, in 90% of the regions at least 20% of the objects are live. However, in 45% of the regions, the live objects occupy less than 10% of the allocated region space (Figure 14(d)). In these cases, using 16 MB regions is more appropriate because they reduce by 10% (BFS) the space waste compared to 256 MB regions. We believe that future work can investigate object placement policies for H2 that takes into account object size to further improve space efficiency on storage devices.

7.6 GC Overhead

The garbage collector in *TeraHeap* performs additional work during minor GC that involves scanning H2 cards and updating backward references. We evaluate this overhead for different card segment sizes in Giraph. Figure 15(a) shows minor GC time in H2 for 1 KB, 4 KB, 8 KB, and 16 KB card segments, normalized to 512 B card segments. We see that increasing the size of card segments from 512 B to 16 KB reduces minor GC time on average by 64%. Larger card segments result in a smaller card table, and less time is required to scan the respective cards. However, increasing card segment size increases the cost of scanning each card segment if the respective card is marked as dirty. For example, increasing the card segment size in PR from 512 B to 16 KB leads to an increase

Table 7. Categorizing H2 Objects into Size Ranges, Such as B, KB, and MB, Showing the Total Number of Objects for Each Category as a Percentage over the Total Allocated Objects in H2

	Spark										Giraph				
	PR	**CC**	**SSSP**	**SVD**	**TR**	**LR**	**LgR**	**SVM**	**BC**	**RL**	**PR**	**CDLP**	**WCC**	**BFS**	**SSSP**
B (%)	87.51	50.36	99.33	100.00	80.33	60.20	93.80	60.61	100.00	100.00	83.44	81.41	82.92	82.33	73.32
KB (%)	10.95	48.40	0.66	0.00	19.54	0.00	0.00	0.00	0.00	0.00	16.56	18.59	17.08	17.67	26.68
MB (%)	1.54	1.24	0.01	0.00	0.13	39.80	6.20	39.39	0.00	0.00	0.00	0.00	0.00	0.00	0.00

in minor GC time for scanning and updating H2 objects with backward references (H2 to H1) by 5×. In Spark, updates to H2 objects are infrequent compared to Giraph, as RDDs are immutable.

Next, we examine the overheads introduced by *TeraHeap* during major GC for H1 by moving objects to H2, which involves device I/O when using SSDs as the backing device. Figure 15(b) shows the three phases of major GC time using Spark-SD and *TeraHeap*. Overall, *TeraHeap* improves marking and compaction phases of major GC by up to 87% (PR) compared to Spark-SD because we avoid scanning H2 objects. For example, in PR, the collector avoids following in each GC, on average, 95 million forward references from H1 to H2 objects. The summary phase takes up to 20% of the major GC time as it needs to scan the bitmap, which indicates H2 candidates and assigns them an H2 address to move during compaction. In contrast, in native PS, the summary phase is much faster, as it simply assigns each region of H1 a destination address, eliminating the need for scanning individual objects. We note that the compaction phase takes 50% of the major GC time in *TeraHeap* due to the device I/O. We should note that the overhead of reclaiming H2 dead regions accounts up to 2% of the total major GC cost because we zero only the metadata of each region in DRAM without introducing any I/O overhead.

Size Distribution of H2 Objects. Next, we examine the size of the objects in H2 to understand how object size affects device I/O traffic when moving objects from H1 to H2. Table 7 categorizes H2 objects into size ranges, such as B, KB, and MB, showing the total number of objects for each category as a percentage of the total allocated objects in H2. In all the workloads, both Spark and Giraph, the size of the majority of the objects (up to 100%) are in the order of bytes. Although Giraph avoids using objects in the order of MBs, 39% of the objects in LR and SVM are in the order of MB. LR and SVM create large arrays of objects used in each algorithm iteration. Based on the observation that most objects are in the order of bytes, we use a promotion buffer per region in H2 that writes objects to the device in batches, amortizing the system call overhead.

7.7 *TeraHeap* Performance with NVM

Figure 16 shows the performance of Spark-SD, Spark-MO, and *TeraHeap* on our NVM-based setup. Our goal is to examine the benefits of *TeraHeap* when using NVM to increase the heap size, which can eliminate S/D at increased GC cost for native. Figure 16(a) shows that *TeraHeap* improves performance by up to 79% and on average by 56%, compared to Spark-SD. Unlike the off-heap cache in Spark-SD, *TeraHeap* allows Spark to directly access cached objects in H2 via load/store operations to NVM, without the need to perform S/D. *TeraHeap* significantly reduces S/D and GC time compared to Spark-SD by up to 97% and 93%, respectively.

Figure 16(b) shows that *TeraHeap* improves performance by up to 86% and on average by 48%, compared to Spark-MO. The main improvement of *TeraHeap* results from the reduction of minor GC and major GC time by up to 88% (on average by 52%) and 96% (on average by 46%) compared to Spark-MO, respectively. In Spark-MO, running the garbage collector on top of NVM (using DRAM as a cache) incurs high overhead due to the latency of NVM [68] and the agnostic placement of objects. For instance, minor GC time in Spark-MO increases on average by 36% compared to

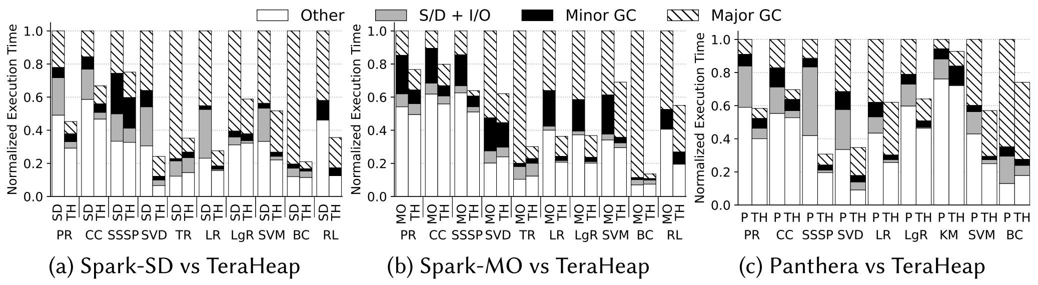


Fig. 16. TeraHeap (TH) performance compared to (a) Spark-SD, (b) Spark-MO, and (c) Panthera (P) over NVM server.

Spark-SD (Figure 16(b)) because objects of the young generation are placed in NVM, resulting in higher access latency for the garbage collector. Unlike *TeraHeap* that controls object placement in NVM (H2), Spark-MO relies on the memory controller to move objects between DRAM and NVM. We measure that Spark-MO incurs on average 5.3× and 11.8× more read and write operations to NVM compared to *TeraHeap*, resulting in higher overhead. Therefore, the ability to maintain separate heaps allows *TeraHeap* to both limit GC cost and reduce the adverse impact of the increased NVM access latency on GC time.

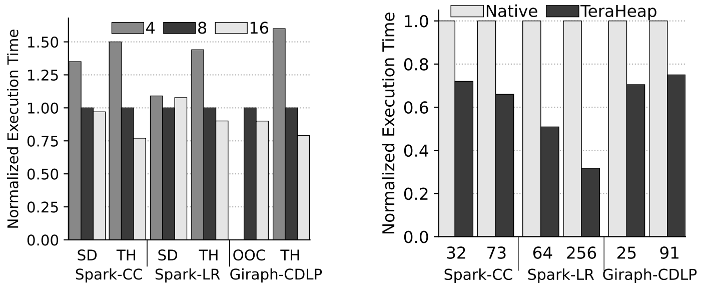
We also compare *TeraHeap* with Panthera [60],[1] a system designed to use NVM as a heap in Spark. Panthera extends the managed heap over DRAM and NVM, placing the young generation in DRAM and splitting the old generation into DRAM and NVM components. We configure Panthera similar to Wang et al. [60] with 64 GB heap, 25% on DRAM (16 GB), and 75% on NVM. We set the size of the young generation to $\frac{1}{6}$ (10 GB) of the total heap size and place it entirely on DRAM. We set the size of the old generation to the rest of the heap size (54 GB) and place 6 GB on DRAM and the rest (48 GB) on NVM. We configure *TeraHeap* to use an H1 of 16 GB and map H2 to NVM. Thus, both systems use the same DRAM and NVM capacity.

Figure 16(c) shows that *TeraHeap* improves performance between 7% and 69% compared to Panthera across all workloads. Panthera bypasses the allocation of some objects in the young generation, allocating them directly to the old generation. However, each major GC still scans all objects in the old generation, which increases overhead as the heap address space grows. Instead, *TeraHeap* reduces the address space that needs to be scanned by the garbage collector. Note that Panthera incurs more accesses to NVM because it allocates mature long-lived objects that are highly read and updated by the mutator threads. Specifically, it increases other by up to 53% because it performs more NVM read (up to 54×) and NVM write (up to 51×) operations than *TeraHeap*.

7.8 Performance Scaling

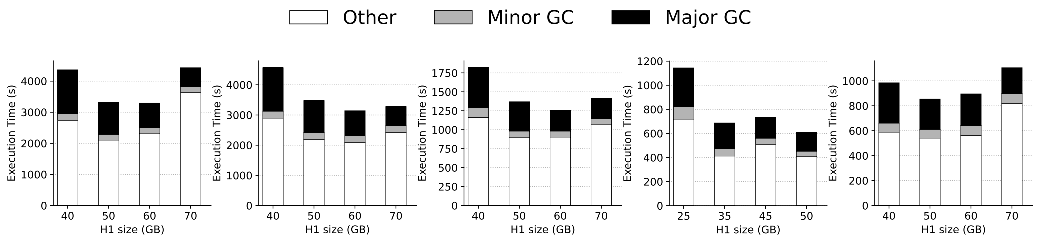
A benefit of *TeraHeap* is that it allows increasing the number of mutator threads in Spark and Giraph executors. In both Spark and Giraph, each mutator thread processes a separate partition. Thus, as the number of threads in the executor increases, the object allocation rate increases, leading to higher GC cost. Figure 17(a) shows the performance of CC, LR, and CDLP (other workloads show similar behavior) using Spark-SD, Giraph-OOC, and *TeraHeap* (TH) with 4, 8, and 16 threads, normalized to 8 threads per configuration. We note that Giraph-OOC with four threads results in an OOM error. *TeraHeap* allows applications to scale performance further to 23% with 2× more threads. However, Spark-SD does not scale beyond 8 threads in LR because GC cost increases (by 44%), eliminating any benefits from using more threads. We note that increasing the number

[1]As Panthera is not publicly available, we are thankful to the authors for providing us their code.



(a) Number of mutator threads (b) Dataset size (GB)

Fig. 17. Performance scaling with (a) number of mutator threads and (b) dataset size in the NVMe server.



(a) Giraph-PR (b) Giraph-CDLP (c) Giraph-WCC (d) Giraph-BFS (e) Giraph-SSSP

Fig. 18. Sensitivity of *TeraHeap* to the size of H1 (relative to the size of page cache for H2) with a constant amount of DRAM capacity in NVMe server.

of threads in Spark-SD reduces S/D cost by up to 55% (CC) because Spark parallelizes the S/D process. Although Giraph-OOC (native) improves performance by 10% using 16 executor threads, it still performs 1.4× more major GCs than eight executor threads. Finally, *TeraHeap* significantly alleviates memory pressure by moving a large portion of H1 objects to H2, leaving more room for mutator threads to work without the need for frequent GC.

We also investigate the performance benefits of *TeraHeap* for a larger dataset in Figure 17(b). We observe similar (CDLP) or higher improvements (CC, LR) compared to the smaller datasets. *TeraHeap* is robust to different dataset sizes and improves performance by up to 70% compared to Spark-SD and Giraph-OOC, while our expectation is that benefit will increase further as dataset size increase.

7.9 Sensitivity Analysis

Next, we examine the sensitivity of *TeraHeap* to dividing a fixed amount of DRAM between H1 and page cache for H2 in Giraph workloads. Figure 18 shows the performance of *TeraHeap* when H1 varies from 40 GB to 70 GB in PR, CDLP, WCC, and SSSP workloads. In BFS, H1 varies from 25 GB to 50 GB. Since the DRAM capacity remains constant in all configurations (75 GB in PR, CDLP, WCC, and SSSP, and 55 GB in BFS), the size of DR2 decreases accordingly. By increasing H1, both minor and major GC time decrease by up to 39% and 57%, respectively, because the garbage collector performs less GC cycles. However, the limited size of the page cache (when H1 occupies a large part of the DRAM) affects the accesses to objects in H2, increasing other time by up to 30% (SSSP). We note that in WCC, CDLP, and BFS, using small H1 affects the other time due to the high I/O traffic to the device. Small H1 size affects the transfer rate of the candidate objects from H1 to H2 because we bypass the transfer hint from the framework. This results in moving objects

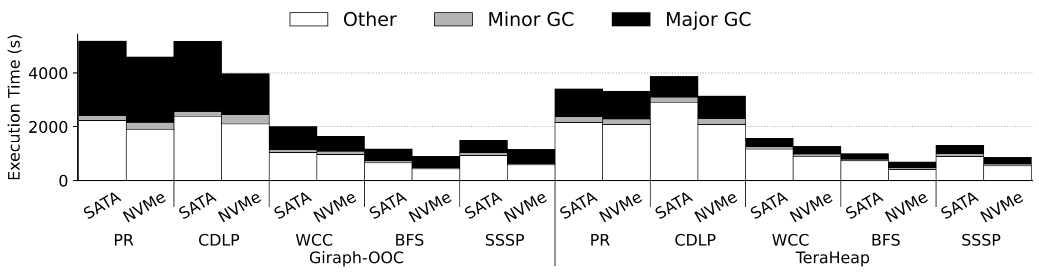


Fig. 19. Performance of *TeraHeap* compared to Giraph-OOC in terms of storage technology.

to H2 before they become immutable. As a future work, *TeraHeap* could investigate the dynamic division of DRAM between H1 and page cache for H2 in a manner that minimizes the sum of the two dominating metrics: GC time for H1 and I/O time for H2.

Next, we examine the sensitivity of *TeraHeap* compared to storage devices heterogeneity, allocating H2 over SATA SSD. In our evaluation, we utilize a RAID 0 configuration of two SATA SSDs. SATA SSDs differ from NVMe SSDs in terms of interface and speed. NVMe SSDs use the PCIe interface, which provides faster data transfer speed than SATA SSDs that use the older SATA interface. Also, the read and write throughput of our NVMe SSD is 6× and 4× higher compared to SATA SSD, respectively.

Figure 19 shows the performance of *TeraHeap* compared to Giraph-OOC with OpenJDK8 using NVMe SSD and SATA SSD with the same amount of DRAM. Despite the use of a slower storage device with SATA SSD, *TeraHeap* still outperforms Giraph-OOC, improving performance between 12% (SSSP) and 34% (PR). This improvement is mainly due to reducing major GC overhead by up to 71% (CDLP). Notably, the slower storage device impacts the performance of *TeraHeap* in terms of object access by mutator threads and object transfer to H2 during major GC.

7.10 Comparison with Newer Garbage Collectors

We next present the performance of *TeraHeap* implemented in OpenJDK17 compared to PS and G1 on OpenJDK17. G1 is a generational, region-based garbage collector which uses concurrent and parallel phases to reduce pause time and to maintain high GC throughput. When G1 determines that a GC is necessary, it collects the regions with the least live data first (garbage first). Figure 20 shows the performance of Spark with PS, G1, and *TeraHeap*, for the same amount of DRAM.

TeraHeap improves performance compared to PS between 5% (RL) and 56% (SVD). The main performance improvement of *TeraHeap* is from the reduction of S/D overhead because *TeraHeap* provides direct access to H2 objects for computation. We note that in RL, the *TeraHeap* major GC time is 36% higher than that of Spark-SD. This difference is because a significant portion of the major GC time is spent on performing forwarding table lookups. Before moving objects from H1 to H2 in RL, we must adjust a subset of H1 objects with 5 million combined forwarding references to the H2 candidate objects. As discussed in Section 4, future work should examine alternative approaches beyond forwarding tables.

On the one hand, G1 outperforms PS in three out of ten workloads between 5% (PR) and 25% (TC) because it reduces GC time by up to 95%. On the other hand, G1 underperforms PS in WCC, SVD, LR, and LGR between 2% (WCC) and 12% (LgR). The source of the slowdown is the expensive GC post-write barriers that G1 uses. The expensive post-write barriers increase mutators' threads time (other time). Also, G1 cannot eliminate the high S/D (up to 44%) caused by the limited DRAM size and the amount of cached data. Unlike G1, *TeraHeap* eliminates S/D overhead, providing direct

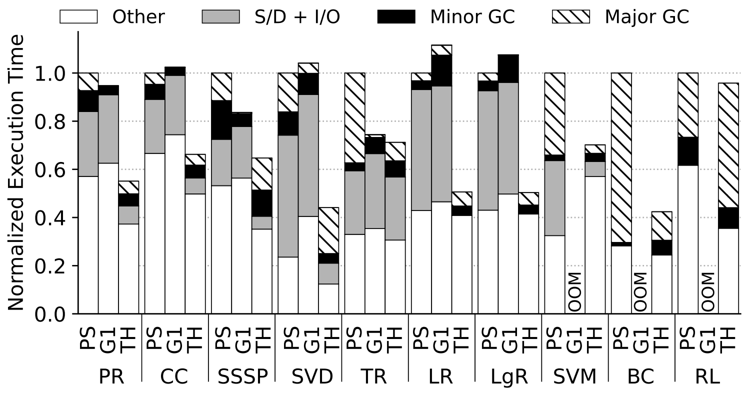


Fig. 20. Performance of TeraHeap (TH) compared to native execution of Spark-SD on the NVMe server with the PS and Garbage First (G1) garbage collectors on OpenJDK17.

access to the storage resident objects. Thus, *TeraHeap* improves performance between 4% (TC) and 57% (SVD) compared to G1.

Note that G1 cannot run SVM, BC, and RL due to fragmentation problems caused by *humongous objects*. Humongous objects in G1 are ones that are bigger than half of the G1 region size. Such objects are allocated separately in contiguous regions (humongous regions). A humongous region can accommodate only one humongous object. The space between the end of the humongous object and the end of the humongous region, which in the worst case can be close to half the region size, is unused. Therefore, when many long-lived humongous objects exist, G1 exhibits significant fragmentation, resulting in OOM errors. PS resolves fragmentation, performing object compaction when the heap becomes full.

We note that *TeraHeap* can also be used with G1 to eliminate S/D cost and reduce the amount of data subject to GC, by moving long-lived, humongous objects to H2.

8 Related Work

TeraHeap combines techniques from several areas, including memory management and storage. Thus, we group the related work in the following categories: (1) scaling managed heaps beyond local DRAM, (2) region-based memory management, and (3) mitigating S/D overhead.

8.1 Taxonomy of Scaling Managed Heaps Beyond DRAM

Recent efforts target local block-addressable storage devices, such as HDD and NVMe SSDs, and local byte-addressable NVM, or remote memory (DRAM) for storing managed heaps beyond DRAM.

Scaling Managed Heaps with Local Block-Addressable Storage Devices. One line of recent work focuses on using block-addressable storage devices. Melt [10] and Leaksurvivor [57] use a primary heap over DRAM and a second heap over an HDD for moving cold objects that programs do not use. Unlike *TeraHeap*, these works do not use a memory-mapped I/O but maintain a specialized lookup mechanism in existing JVM code to track the offset of each object in the device. Although this technique targets the high latency of the hard disk drives, it requires multiple systems calls and incurs the high hit overhead of user space lookups in every load/store operation. Also, LeakSurvivor [57] leverages the capacity of the HDD to perform lazy object reclamation. However, it reclaims objects in the HDD only when the device becomes full, paying high GC cost for long-running applications. Performing GC scans over a storage device introduces high I/O traffic, increasing GC pause time. TMO [63] monitors application DRAM usage and transparently offloads cold data to NVMe SSD. Unlike these works, *TeraHeap* controls which objects to move to the second heap and eliminates slow GC traversals over objects on NVMe SSD. Finally, prior effort [34] discusses a preliminary prototype of *TeraHeap* [33], while this article presents the design and implementation of *TeraHeap* in OpenJDK17.

Scaling Managed Heaps with Byte-Addressable Devices. Recent research efforts focus on extending the managed heaps with local byte-addressable NVM or remote DRAM. Akram et al. [2, 3], GCMove [37], and ThinGC [67] focus on improving NVM write endurance. They select which objects to move from DRAM to NVM based on their hotness. Existing work uses two techniques to track object hotness: (1) through load reference barriers used by concurrent copying collectors to identify the most accessed objects, and (2) through post-write reference barriers to identify the most written/updated objects. Both techniques increase the cost of the barriers and can work with specific garbage collectors. NVM-friendly-GC [66] report high GC overhead with NVM-backed volatile heaps and optimize the G1 GC for Intel Optane persistent memory. It allocates its heap entirely on NVM and uses DRAM as a user-space cache for GC operations. However, all the accesses to objects by application (mutator) threads are served by NVM. Panthera [60] extends the managed heap over hybrid DRAM and NVM to scale on-heap caching in Spark. Panthera increases GC overhead as scanning and compacting objects on the managed NVM heap costs more than collecting the DRAM heap. Also, JDPHeap [70] uses a managed heap in DRAM and a second managed heap over NVM. It introduces an extensive programming model that enables users to select which objects to move from DRAM to NVM and vice versa. However, unlike *TeraHeap*, JPDHeap cannot avoid expensive GC scans over objects in NVM.

Also, there is an amount of effort [39, 61, 62] for extending the managed heaps using remote memory. On the one hand, Memliner [62] is designed for software-disaggregated datacenters. MemLiner reorganizes the access order of the objects by the GC threads to follow a similar (not identical) memory-access path with the mutator threads to enhance the speed of far-memory accesses during GC. On the other hand, Semeru [61] and Mako [39] target hardware-designed datacenters. Semeru supports running managed big data frameworks on disaggregated hardware by dividing the conventional JVM into two instances: (1) the CPU-JVM instance that executes the program on the CPU server and (2) the memory-JVM instance that performs GC on the memory server. To maximize GC performance, Semeru proposes a distributed garbage collector that offloads the liveness analysis for Java objects to memory servers concurrently with mutator threads. Unlike Semeru, Mako offloads to memory servers both object scanning for liveness analysis and evacuation, reducing GC pause time. Although DRAM is byte-addressable, these approaches rely on the OS-swap system to move objects between local and remote memory at page granularity. *TeraHeap* is orthogonal to these approaches because it exploits fast local storage devices for scaling the capacity of the managed heap.

Table 8 illustrates the taxonomy of existing approaches that scale the managed heaps beyond local DRAM capacity based on the following characteristics: (1) development effort, (2) object reclamation on the device or remote memory and (3) device type. *TeraHeap* with low development effort uses the framework's knowledge regarding which objects are suitable for moving to the storage device. Also, *TeraHeap* avoids GC scans over the storage device because it leverages the high capacity of the devices to perform lazy object reclamation. Finally, it can still improve application performance even if H2 is in a local block-addressable storage device or a byte-addressable device.

Table 8. Comparison of TeraHeap with Existing State-of-the-Art Work

System	**Developer Effort**	**GC Scans on the Device**		**Device Type**	
		Lazy	**Eager**	**Block-addressable**	**Byte-addressable**
LeakSurvivor [57]	High	(✓)	-	✓	-
Melt [10]	None	(✓)	-	✓	-
Write Rationing [3]	None	-	✓	-	✓
Panthera [60]	Low	-	✓	-	✓
NVM-friendly-GC [69]	None	-	✓	-	✓
JPD Heap [70]	High	-	✓	-	✓
GCMove [37]	None	-	✓	-	✓
ThinGC [67]	None	-	✓	-	✓
TMO [63]	None	-	✓	✓	-
Semeru [61]	None	-	✓	-	✓
MemLiner [62]	None	-	✓	-	✓
Mako [39]	None	-	✓	-	✓
TeraHeap (our work)	Low	✓	-	✓	✓

8.2 Region-Based Memory Management

Managed big data frameworks have started to use region-based memory management for large heaps. Systems, such as Gerenuk [44] and Facade [47], provide a compilation framework that transforms programmer-specified classes for off-heap allocation at compile time, thereby reducing the serialization overhead. However, these systems increase programmer effort by requiring the developer to specify when to free objects from native memory and modify legacy applications. Broom [21] uses region annotations but requires refactoring of applications' source code. Yak [46] requires programmers to annotate epochs in applications. Yak allocates all objects in an epoch on a second region-based heap to reduce GC time. The epoch abstraction is appropriate for the map-reduce programming pattern. However, it cannot handle objects computed lazily or accessed from arbitrary program locations. Deca [38] proposes lifetime-based memory management for Spark. However, their work only applies to Spark and cannot be used for other frameworks. Unlike prior work, *TeraHeap* requires adding hints only in the framework layer. Then, *TeraHeap* dynamically selects all appropriate objects in the transitive closure of root objects. NG2C [11] uses runtime profiling to identify long-lived objects. They incur online profiling overhead. Other work uses offline allocation site profiling to manage objects [8, 9]. Lifetime profiling can complement *TeraHeap* and further improve efficiency.

8.3 Mitigating S/D Overhead

Several libraries [20, 22, 56] improve the efficiency of S/D, but they cannot reduce high GC cost in big data frameworks. Skyway [45] reduces the S/D cost by directly transferring objects through the network in distributed managed heaps, but it does not cope with DRAM limitations and GC overheads. SSDStreamer [4] is a userspace SSD-based caching system that uses DRAM as a stream buffer for SSD devices. Although SSDStreamer reduces S/D cost by providing a lightweight serializer, it cannot reduce GC cost and the memory pressure in the managed heap. Recent work [31, 53] proposes optimizations for S/D with custom hardware extensions and modifications to the programming model. Other work [54, 58, 64] focuses on reducing S/D cost by reducing the number of object copies across buffers. This body of work does not mitigate directly GC overhead. *TeraHeap* is the first work that eliminates S/D cost without increasing GC overhead for a large portion of objects in big data analytics frameworks.

9 Conclusions

Managed big data analytics frameworks demand increasing the heap size as datasets grow. In managed language environments, such as JVM, H2 incur excessive GC overhead. Thus, frameworks avoid using large heaps and resort to expensive off-heap S/D when managing large datasets. This work proposes and evaluates *TeraHeap*, which extends the JVM to use a transparent, H2 over a fast storage device alongside the H1, reducing memory pressure. *TeraHeap* reduces GC overhead and eliminates S/D cost by fencing the collector from scanning the second heap and providing direct access to objects on the second heap. We find that *TeraHeap* improves Spark and Giraph performance by up to 73% and 28%, respectively. In addition to frameworks that explicitly transfer objects off heap, *TeraHeap* can also benefit frameworks that operate solely in memory. Consequently, we showcase how utilizing *TeraHeap* can improve the performance of the Neo4j-GDS library by up to 26%. Overall, our proposed approach of managing large memory in the JVM as customized, separate heaps is a promising direction for incorporating large address spaces in managed environments and reducing memory pressure without incurring high GC overhead.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and help in preparing this article's final version.

References

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, 971–985. DOI: https://doi.org/10.1145/3297858.3304061

[2] Shoaib Akram, Jennifer Sartor, Kathryn McKinley, and Lieven Eeckhout. 2019. Crystal Gazer: Profile-Driven Write-Rationing Garbage Collection for Hybrid Memories. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 1 (March 2019), Article 9, 27 pages. DOI: https://doi.org/10.1145/3322205.3311080

[3] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-Rationing Garbage Collection for Hybrid Memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, New York, NY, 62–77. DOI: https://doi.org/10.1145/3192366.3192392

[4] Jonghyun Bae, Hakbeom Jang, Jeonghun Gong, Wenjing Jin, Shine Kim, Jaeyoung Jang, Tae Jun Ham, Jinkyu Jeong, and Jae W. Lee. 2019. SSDStreamer: Specializing I/O Stack for Large-Scale Machine Learning. *IEEE Micro* 39, 5 (July 2019), 73–81. DOI: https://doi.org/10.1109/MM.2019.2930497

[5] Monica Beckwith. 2013. Garbage First Garbage Collector Tuning. Retrieved from https://www.oracle.com/technical-resources/articles/java/g1gc.html#::text=During%20mixed%20collections%2C%20the%20G1,overall%20acceptable%20heap%20waste%20percentage

[6] Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. 2012. Apache lucene 4. In *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*, 17.

[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, et al. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, 169–190. DOI: https://doi.org/10.1145/1167473.1167488

[8] Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss, and Ting Yang. 2007. Profile-Based Pretenuring. *ACM Trans. Program. Lang. Syst.* 29, 1 (January 2007), 2–es. DOI: https://doi.org/10.1145/1180475.1180477

[9] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, and J. Eliot B. Moss. 2001. Pretenuring for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, 342–352. DOI: https://doi.org/10.1145/504282.504307

[10] Michael D. Bond and Kathryn S. McKinley. 2008. Tolerating Memory Leaks. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA '08)*. ACM, New York, NY, 109–126. DOI: https://doi.org/10.1145/1449764.1449774

[11] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. 2017. NG2C: Pretenuring Garbage Collection with Dynamic Generations for HotSpot Big Data Applications. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM '17)*. ACM, New York, NY, 2–13. DOI: https://doi.org/10.1145/3092255.3092272

[12] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. 2019. Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications. In *Proceedings of the 14th EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, Article 28, 16 pages. DOI: https://doi.org/10.1145/3302424.3303988

[13] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *Bull. Tech. Committ. Data Eng.* 38, 4 (2015), 28–38.

[14] Apache Cassandra. 2014. Apache Cassandra. Retrieved from https://planetcassandra.org/what-is-apache-cassandra

[15] Zhiguang Chen, Yutong Lu, Nong Xiao, and Fang Liu. 2014. A Hybrid Memory Built by SSD and DRAM to Support In-Memory Big Data Analytics. *Knowl. Inf. Syst.* 41, 2 (November 2014), 335–354. DOI: https://doi.org/10.1007/s10115-013-0727-6

[16] Neo4j Community. 2024. How to Leverage Flash Memory to cache Neo4j data - Neo4j Graph Platform - Neo4j Online Community. Retrieved from https://community.neo4j.com/t/how-to-leverage-flash-memory-to-cache-neo4j-data/43243

[17] Databricks. 2015. Project Tungsten: Bringing Apache Spark closer to bare metal. Retrieved from https://www.databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

[18] David Detlefs, Ross Knippel, William D. Clinger, and Matthias Jacob. 2002. Concurrent Remembered Set Refinement in Generational Garbage Collection. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (Java VM '02)*. USENIX Association, 13–26.

[19] Apache Software Foundation. 2016. Apache Arrow: A Cross-Language Development Platform for In-Memory Data. Retrieved from https://arrow.apache.org/

[20] Apache Software Foundation. 2018. Apache Thrift. Retrieved from https://thrift.apache.org/

[21] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingan, Derek Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping out Garbage Collection from Big Data Systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS '15)*. USENIX Association, Article 2, 2 pages.

[22] Google. 2001. Protocol Buffers. Retrieved from https://developers.google.com/protocol-buffers/docs/javatutorial

[23] Brendan Gregg. 2017. Visualizing Performance with Flame Graphs. USENIX Association.

[24] Konrad Grochowski, Michał Breiter, and Robert Nowak. 2019. Serialization in Object-Oriented Programming Languages. In *Introduction to Data Science and Machine Learning*. IntechOpen, Rijeka, Chapter 12. DOI: https://doi.org/10.5772/intechopen.86917

[25] Scott Haines. 2022. Bridging Spark SQL with JDBC. In *Modern Data Engineering with Apache Spark*. Springer, 117–151.

[26] Elliotte Rusty Harold. 2006. *Java I/O: Tips and Techniques for Putting I/O to Work*. O'Reilly Media, Inc.

[27] Barry Hayes. 1991. Using Key Object Opportunism to Collect Old Objects. In *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*. ACM, New York, NY, 33–46. DOI: https://doi.org/10.1145/117954.117957

[28] Amy E. Hodler and Mark Needham. 2022. Graph Data Science Using Neo4j. In *Massive Graph Analytics*. Chapman and Hall/CRC, 433–457.

[29] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (September 2016), 1317–1328. DOI: https://doi.org/10.14778/3007263.3007270

[30] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, et al. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714. Retrieved from http://arxiv.org/abs/1903.05714

[31] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. 2020. A Specialized Architecture for Object Serialization with Applications to Big Data Analytics. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA '20)*. IEEE Press, New York, NY, 322–334. DOI: https://doi.org/10.1109/ISCA45697.2020.00036

[32] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (1st ed.). Chapman & Hall/CRC.

[33] 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 (ASPLOS '23)*. ACM, New York, NY, 694–709. DOI: https://doi.org/10.1145/3582016.3582045

[34] 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 the 12th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '20)*. USENIX Association, USA, Article 4, 4 pages.

[35] Dongyang Li, Fei Wu, Yang Weng, Qing Yang, and Changsheng Xie. 2018. HODS: Hardware Object Deserialization Inside SSD Storage. In *Proceedings of the IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '18)*. IEEE, 157–164. DOI: https://doi.org/10.1109/FCCM.2018.00033

[36] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2017. SparkBench: A Spark Benchmarking Suite Characterizing Large-Scale in-Memory Data Analytics. *Cluster Comput.* 20, 3 (September 2017), 2575–2589. DOI: https://doi.org/10.1007/s10586-016-0723-1

[37] Zhe Li and Mingyu Wu. 2022. Transparent and Lightweight Object Placement for Managed Workloads atop Hybrid Memories. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '22)*. ACM, New York, NY, 72–80. DOI: https://doi.org/10.1145/3516807.3516822

[38] 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. *Proc. VLDB Endow.* 9, 12 (August 2016), 936–947. DOI: https://doi.org/10.14778/2994509.2994513

[39] 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 the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*. ACM, New York, NY, 92–107. DOI: https://doi.org/10.1145/3519939.3523441

[40] Mohammad Sultan Mahmud, Joshua Zhexue Huang, Salman Salloum, Tamer Z. Emara, and Kuanishbay Sadatdiynov. 2020. A Survey of Data Partitioning and Sampling Methods to Support Big Data Analysis. *Big Data Mining and Analytics* 3, 2 (February 2020), 85–101. DOI: https://doi.org/10.26599/BDMA.2019.9020015

[41] Ioannis Malliotakis, Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2021. HugeMap: Optimizing Memory-Mapped I/O with Huge Pages for Fast Storage. In *Proceedings of the International Conference on Parallel Processing Workshops (Euro-Par '20)*. Springer International Publishing, Cham, 344–355. DOI: https://doi.org/10.1007/978-3-030-71593-9_27

[42] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2010. Four Trends Leading to Java Runtime Bloat. *IEEE Softw.* 27, 1 (January 2010), 56–63. DOI: https://doi.org/10.1109/MS.2010.7

[43] Jyoti Nandimath, Ekata Banerjee, Ankur Patil, Pratima Kakade, Saumitra Vaidya, and Divyansh Chaturvedi. 2013. Big Data Analysis Using Apache Hadoop. In *Proceedings of the IEEE 14th International Conference on Information Reuse & Integration (IRI)*, 700–703. DOI: https://doi.org/10.1109/IRI.2013.6642536

[44] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, New York, NY, 538–553. DOI: https://doi.org/10.1145/3341301.3359643

[45] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting Managed Heaps in Distributed Big Data Systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, 56–69. DOI: https://doi.org/10.1145/3173162.3173200

[46] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, 349–365.

[47] 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 *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, 675–690. DOI: https://doi.org/10.1145/2694344.2694345

[48] Patrick Niemeyer and Daniel Leuck. 2013. *Learning Java*. O'Reilly Media, Inc.

[49] Oracle. 2014. Class (Java Platform SE 8). Retrieved from https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html

[50] Oracle. 2021. Reference Class (Java SE 17 & JDK 17). Retrieved from https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ref/Reference.html

[51] Andrei Pangin. 2018. Async-profiler. Retrieved from https://github.com/jvm-profiling-tools/async-profiler

[52] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-Mapped I/O for Fast Storage Devices. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '20)*. USENIX Association, Article 56, 15 pages.

[53] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. 2020. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the 25th*

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20). ACM, New York, NY, 1203–1216. DOI: https://doi.org/10.1145/3373376.3378501

[54] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. 2021. Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. ACM, New York, NY, 199–205. DOI: https://doi.org/10.1145/3458336.3465287

[55] Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. 2017. *Large-Scale Graph Processing Using Apache Giraph* (1st ed.). Springer Publishing Company, Incorporated.

[56] Esoteric Software. 2013. Kryo. Retrieved from https://github.com/EsotericSoftware/kryo

[57] Yan Tang, Qi Gao, and Feng Qin. 2008. {LeakSurvivor}: Towards Safely Tolerating Memory Leaks for {Garbage-Collected} Languages. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 08)*. USENIX Association, 307–320.

[58] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. 2021. Naos: Serialization-free RDMA networking in Java. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '21)*. USENIX Association, 1–14.

[59] David Ungar. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. ACM, New York, NY, 157–167. DOI: https://doi.org/10.1145/800020.808261

[60] 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 the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. ACM, New York, NY, 347–362. DOI: https://doi.org/10.1145/3314221.3314650

[61] 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 the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 261–280. Retrieved from https://www.usenix.org/conference/osdi20/presentation/wang

[62] 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 the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. USENIX Association, 35–53. Retrieved from https://www.usenix.org/conference/osdi22/presentation/wang

[63] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. ACM, New York, NY, 609–621. DOI: https://doi.org/10.1145/3503222.3507731

[64] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. 2021. Zerializer: Towards Zero-Copy Serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. ACM, New York, NY, 206–212. DOI: https://doi.org/10.1145/3458336.3465283

[65] Erci Xu, Mohit Saxena, and Lawrence Chiu. 2016. Neutrino: Revisiting Memory Caching for Iterative Data Analytics. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '16)*. USENIX Association, 16–20.

[66] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. 2019. An Experimental Evaluation of Garbage Collectors on Big Data Applications. *Proc. VLDB Endow.* 12, 5 (January 2019), 570–583. DOI: https://doi.org/10.14778/3303753.3303762

[67] Albert Mingkun Yang, Erik Österlund, Jesper Wilhelmsson, Hanna Nyblom, and Tobias Wrigstad. 2020. ThinGC: Complete Isolation with Marginal Overhead. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management (ISMM '20)*. ACM, New York, NY, 74–86. DOI: https://doi.org/10.1145/3381898.3397213

[68] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*. USENIX Association, 169–182.

[69] Yanfei Yang, Mingyu Wu, Haibo Chen, and Binyu Zang. 2021. Bridging the Performance Gap for Copy-Based Garbage Collectors atop Non-Volatile Memory. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys '21)*. ACM, New York, NY, 343–358. DOI: https://doi.org/10.1145/3447786.3456246

[70] Litong You, Tianxiao Gu, Shengan Zheng, Jianmei Guo, Sanhong Li, Yuting Chen, and Linpeng Huang. 2021. JPDHeap: A JVM Heap Design for PM-DRAM Memories. In *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC '21)*. IEEE, 31–36. DOI: https://doi.org/10.1109/DAC18074.2021.9586279

[71] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*. USENIX Association, 2 pages.

[72] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '10)*. USENIX Association, Article 10, 10 pages.
[73] Wenyu Zhao and Stephen M. Blackburn. 2020. Deconstructing the Garbage-First Collector. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*. ACM, New York, NY, 15–29. DOI: https://doi.org/10.1145/3381052.3381320

Received 25 September 2023; revised 14 September 2024; accepted 30 September 2024