TeraHeap: Reducing Memory Pressure in Managed Big Data Frameworks

Iacovos G. Kolokasis*† FORTH-ICS, Greece kolokasis@ics.forth.gr
Giannos Evdorou*† FORTH-ICS, Greece evdorou@ics.forth.gr
Shoaib Akram‡ ANU, Australia shoaib.akram@anu.edu.au
Christos Kozanitis* FORTH-ICS, Greece kozanitis@ics.forth.gr
Anastasios Papagiannis Isovelent, Inc., USA anastasios@isovalent.com
Foivos S. Zakkak Red Hat, Inc., UK fzakkak@redhat.com
Polyvios Pratikakis*† FORTH-ICS, Greece polyvios@ics.forth.gr
Angelos Bilas*† FORTH-ICS, Greece bilas@ics.forth.gr

ABSTRACT

Big data analytics frameworks, such as Spark and Giraph, need to process and cache massive amounts of data that do not always fit on the managed heap. Therefore, frameworks temporarily move long-lived objects outside the managed 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 to the heap for processing.

In this paper, we propose *TeraHeap*, a system that eliminates S/D overhead and expensive GC scans for a large portion of the objects in big data frameworks. *TeraHeap* relies on three concepts. (1) It eliminates S/D cost 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 big data analytics frameworks to leverage knowledge about objects to populate H2. (3) It reduces GC cost by fencing the garbage collector from scanning H2 objects while maintaining the illusion of a single managed heap.

We implement *TeraHeap* in OpenJDK and evaluate it with 15 widely used applications in two real-world big data frameworks, Spark and Giraph. Our evaluation shows that for the same DRAM size, *TeraHeap* improves performance by up to 73% and 28% compared to native Spark and Giraph, respectively. Also, it provides better performance by consuming up to 8× and 1.2× less DRAM capacity than native Spark and Giraph, respectively. Finally, it outperforms Panthera, a state-of-the-art garbage collector for hybrid 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**.

KEYWORDS

Java Virtual Machine (JVM), large analytics datasets, 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. 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), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3582016.3582045

1 INTRODUCTION

Managed big data frameworks, such as Spark [56] and Giraph [44], 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 [51] and (2) objects in big data frameworks exhibit long lifetimes [10, 48, 52]. Although production garbage collectors efficiently manage short-lived objects, they do not perform well under high memory pressure introduced by long-lived objects [32].

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., 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 [14], they do not offer support for computation over arbitrary objects, resulting in applications specific solutions, such as Spark SQL [12].

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 [34, 43, 50]. Recent

*Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS), Greece
†Department of Computer Science, University of Crete, Greece
‡Australian National University, Australia

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582045

efforts [25, 27] 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 [49] 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 work, 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 [30]. *TeraHeap* provides a hint-based interface that uses *key-object opportunism* [22] and enables frameworks to mark objects and indicate when to move them to H2. During GC, *TeraHeap* starts from root key-objects and dynamically identifies the objects to move 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. 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 [16, 35] *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 concerning the objects they move off-heap. We investigate how Spark and Giraph, two widely used frameworks, resolve the trade-off between GC cost due to large heaps and the overhead of off-heap accesses. 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.

We implement *TeraHeap* and its mechanisms in OpenJDK, extending the Parallel Scavenge 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% and 28% compared to native Spark and Giraph, respectively. *TeraHeap* provides similar or better performance by consuming up to 8× and 1.2× less DRAM capacity than native Spark and Giraph, respectively. Also, it outperforms Panthera [48], 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 and (2) a high-capacity heap in Giraph to store messages and edges.

2 BACKGROUND

This section provides background related to JVM garbage collection and serialization/deserialization.

Garbage Collection: 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 [47]. 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 using memory-mapped I/O (e.g., Linux *mmap*). However, existing garbage collectors are designed for DRAM-backed heaps and incur significant overhead for storage-backed heaps [54]. 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, 41].

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 significant limitations and overheads during execution. 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 [19, 21, 37]. In addition, extracting and recreating the object state requires mechanisms that bypass constructors and ignore class and field accessibility. Performance-wise, traversing the object graph requires effort proportional to the volume of objects in the transitive closure of the root object. 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 [33, 34, 43, 46].

3 TERAHEAP DESIGN

3.1 Overview

The key idea of *TeraHeap*, as shown in Figure 1, 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 [41]. NVMe SSDs are particularly attractive as datasets grow because they provide high density (capacity) and lower cost per bit compared to DRAM and NVM [49].

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.

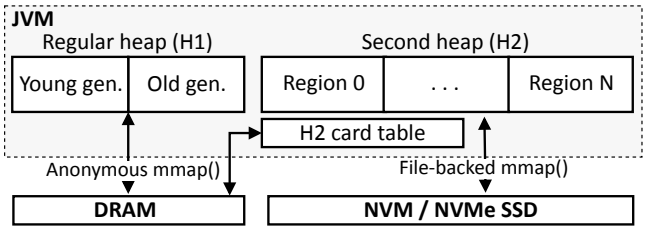


Figure 1: TeraHeap design overview.

TeraHeap exposes a novel hint-based interface that uses key-object opportunism [22] 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 interface works at the framework level and is entirely transparent to applications written on top of such frameworks (§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. Most objects that are reachable by root key-objects exhibit a similar lifetime [16]. *TeraHeap* leverages the high capacity of NVMe SSDs to resolve the space-performance trade-off differently than existing work [16, 35]. 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. To ensure memory safety while reclaiming dead H2 regions, *TeraHeap* must take into account forward (H1 to H2) and cross-region references (§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 (§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.

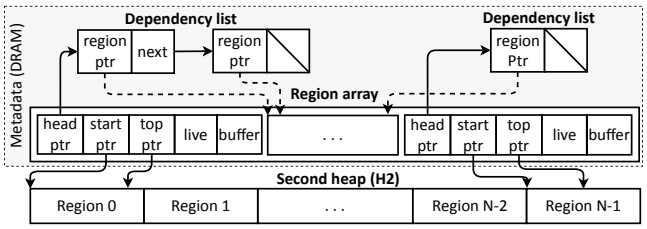


Figure 2: H2 allocator metadata in DRAM.

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 [24]. 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 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* [38] and the class loader, and (2) objects that inherit the *java.lang.ref.Reference* class [39].

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.

3.3 Reclaiming Dead Regions

Figure 2 shows the region-based organization of H2 in virtual memory and each region metadata in DRAM. Unlike Broom [16], 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 2, *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. 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 avoids 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 2) 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.

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 2). Upon JVM shutdown, we free all H2 metadata in DRAM.

3.4 Tracking Backward References (H2 to H1)

To track backward references, we use an extended card table for H2, 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). Although using a *remembered set* provides more precise information about backward references, it increases memory consumption for regions with many references, especially as H2 size grows with storage device capacity. It also requires a more elaborate and expensive post-write barrier [13].

Setting H2 card states. We expect H2 to be much larger than H1. Thus, we increase the size of H2 card segments to reduce the number of cards and the card scanning overhead during collections. However, larger card segments require scanning more objects, in case they are dirty, introducing device I/O. 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. 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 3, 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.

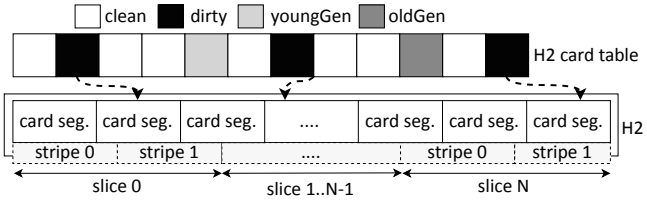


Figure 3: Organization of H2 in stripes and slices.

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 is a 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 PARALLEL SCAVENGE GC

We implement *TeraHeap* in OpenJDK8 (jdk8u345), which is a long-term support version, by extending the Parallel Scavenge (PS) garbage collector and exporting *TeraHeap*'s interface through the *Unsafe* class to frameworks. PS is a generational garbage collector which divides its heap into young and old generations. Next, we discuss our extensions in (1) the post-write barriers in interpreter and just-in-time (JIT) compilers, (2) minor GC, and (3) major GC.

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, this operation is followed by the 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 reference range check 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, guarded by the flag we introduce `EnableTeraHeap`. We evaluate the overhead of our modifications to post-write barriers using the DaCapo benchmark suite [6]. The overhead is small and within 3% over total execution time on average across all benchmarks. The additional overhead is zero for applications that do not set `EnableTeraHeap`.

Minor GC. In minor GC, we perform two key tasks during liveness analysis: (1) fence PS from scanning objects in H2 and (2) prevent reclamation of H1 objects referred from H2 objects (backward references). For the first task, we introduce a reference range check in the liveness analysis to fence PS from scanning references that cross from H1 to H2. For the second task, we scan the H2 card table to identify and update backward references and the H2 card state.

Major GC. The major GC in PS 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 `live` bits of the H2 regions metadata. 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 the PS from scanning objects in H2 and sets the `live` bit of the corresponding region. We identify the root objects tagged with a label through *TeraHeap* interface and calculate their transitive closure. At the end, 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.

5 APPLICATIONS OF TERAHEAP

In this section, we describe how we use *TeraHeap* in two popular frameworks, Spark [56] and Giraph [44], which differ significantly in how they use off-heap memory. 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.

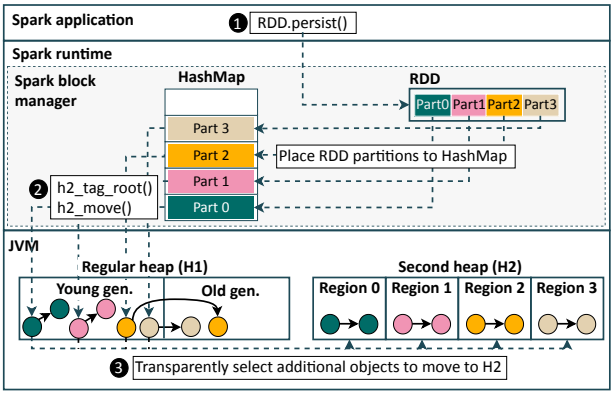


Figure 4: Use of TeraHeap in Spark.

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 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. Spark requires only slight modifications to use *TeraHeap*. We note that Spark abstracts intermediate results as immutable collections using three sets of APIs [20]: resilient distributed datasets (RDDs) [55], DataFrames, and Datasets. We mark all cached partitions of RDDs, DataFrames, or Datasets, as root objects for moving to H2. Figure 4 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()`. 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

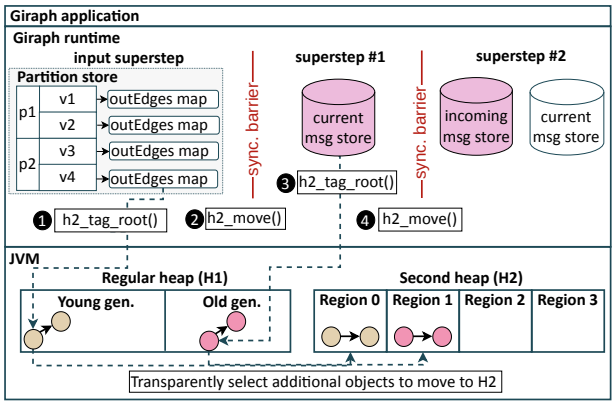


Figure 5: Use of TeraHeap in Giraph.

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

Table 1: NVMe and NVM server properties.

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 [44]. Figure 5 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()`.

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?

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	-	-

Table 2: Summary of baselines.

Server infrastructure. We evaluate *TeraHeap* both with block-addressable NVMe 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 [45], a state-of-the art highly optimized S/D library for Java that Spark recommends. We run Spark with OpenJDK8, OpenJDK11, and OpenJDK17. We run Giraph v1.2 with Hadoop v2.4 and OpenJDK8, as it does not support more recent versions of OpenJDK. We use two garbage collectors in different configurations: PS in OpenJDK8 and OpenJDK11, and Garbage First (G1) in OpenJDK17. We use an executor with eight mutator threads for both Spark and Giraph. For PS, we use 16 GC threads for minor GC and the default single-threaded old generation GC. 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].

Table 2 summarizes the Spark and Giraph 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.

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 [41]. 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 the native *mmap*, we use HugeMap [31] 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 [11], we use a small heap size that caches a limited number of RDDs on-heap and the rest off-heap (Spark-SD column

	GB	NVMe Server DRAM	Data-set Size	Spark-SD Heap	Spark-MO Heap	Tera Heap H1
GraphX	PageRank (PR)	80	32	64	1024	64
	Connected Components (CC)	84	32	68	1024	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	1084	54
	Logistic Regression (LgR)	70	256	54	1084	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 3: Configuration for each workload on NVMe and NVM servers for Spark-SD, Spark-MO, and *TeraHeap*.

	NVMe Server	Data-set	Giraph-OOC		TeraHeap	
GB	DRAM	Size	Heap	DR2	H1	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

Table 4: Giraph-OOC and *TeraHeap* configurations for each workload on our NVMe server.

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.

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. We omit exploration results due to space constraints. Table 3 and Table 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*. Table 3 and Table 4 show the total DRAM capacity in the NVMe server for each workload.

Workloads and datasets. We use ten memory-intensive workloads from the Spark-Bench suite [28] and five workloads from the LDBC Graphalytics suite [23] for Giraph. We synthesize datasets for Spark workloads with the SparkBench data generators. For Giraph workloads, we use the datagen-9_0-fb dataset [23]. Table 3 and Table 4 show the dataset size for each workload.

Execution time breakdowns and S/D overhead. We repeat each experiment five times and report the average end-to-end execution time. 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 [40] to collect execution samples from the mutator threads. The samples include the stack trace, similar to the flame graph [18] 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 6 shows the performance of *TeraHeap* compared to Spark-SD and Giraph-OOC for the NVMe SSD setup. We normalize execution time to the first bar in each figure. Missing bars indicate out-of-memory (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 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 cached RDDs elements 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.

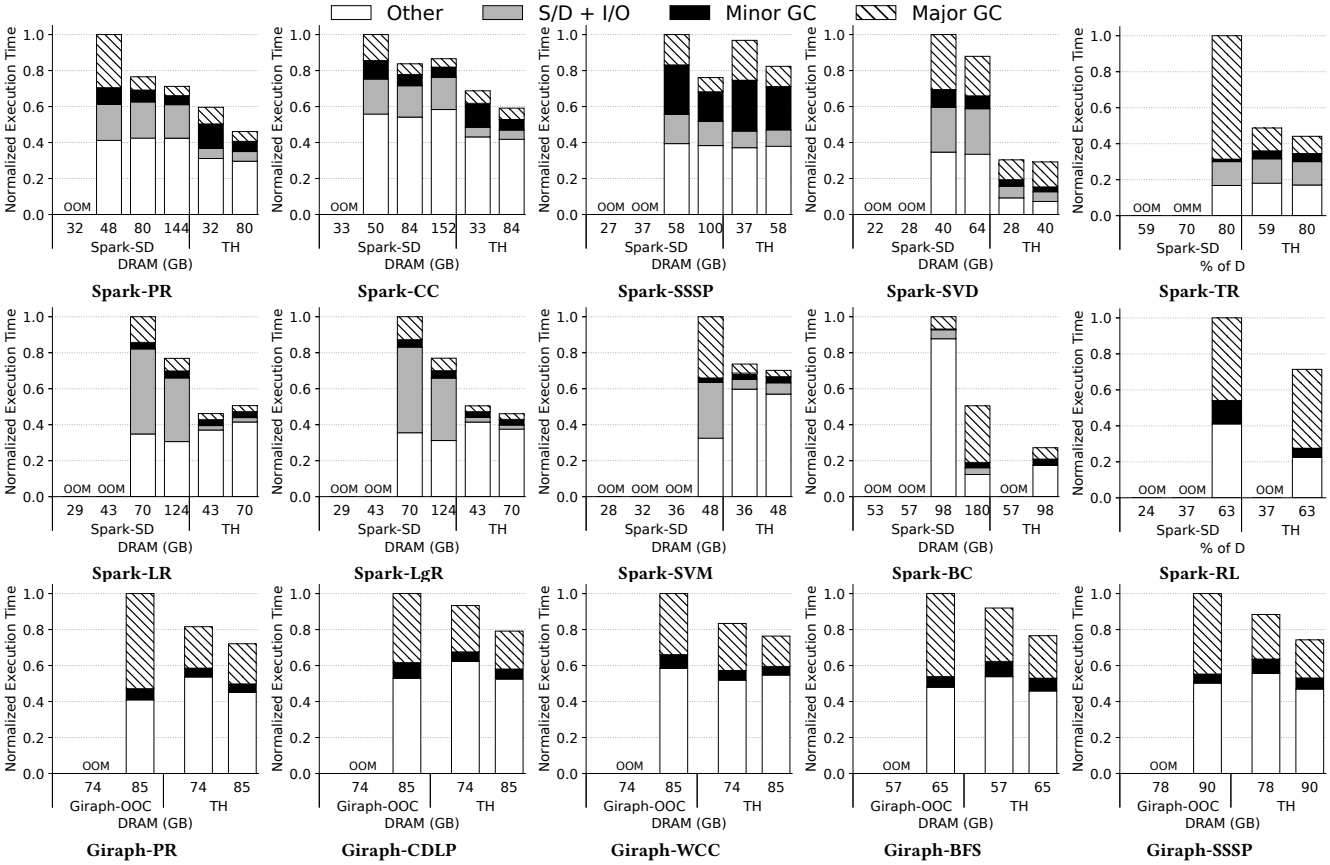
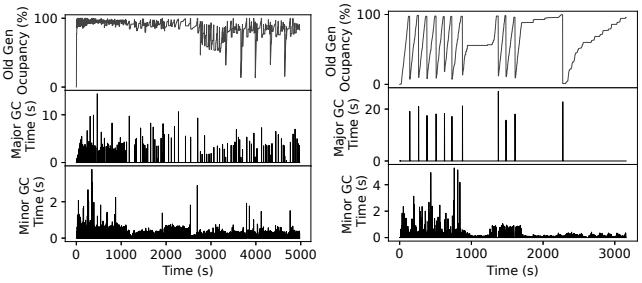


Figure 6: Overall performance of TeraHeap (TH) compared to Spark-SD and Giraph-OOC on the NVMe server.



(a) Spark-SD **(b) TeraHeap**

Figure 7: GC time and old generation occupancy in PR for (a) Spark-SD and (b) TeraHeap. The heap size is 64 GB.

To examine pressure on the managed heap, Figure 7 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 7), 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. We omit similar results for other workloads due to space constraints.

Reducing DRAM capacity demands. We examine the potential benefit of *TeraHeap* in reducing DRAM capacity demands in Figure 6. Using between 2× and 8× 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.

Comparison with newer garbage collectors. We next present the performance of *TeraHeap* compared to an optimized version of PS on OpenJDK11 and G1 on OpenJDK17. G1 is a generational, region-based garbage collector which uses concurrent and parallel

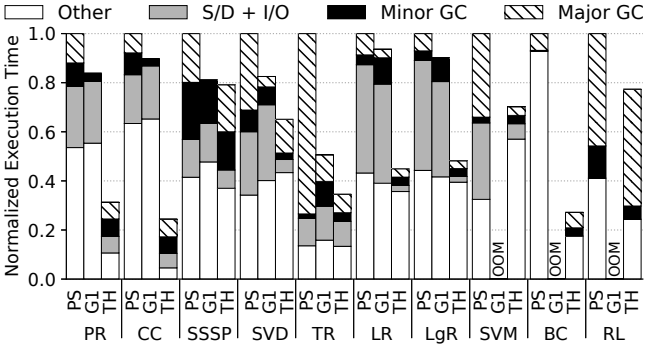


Figure 8: Performance of TeraHeap (TH) compared to Parallel Scavenge (PS) on OpenJDK11 and Garbage First (G1) on OpenJDK17 garbage collectors on the NVMe server.

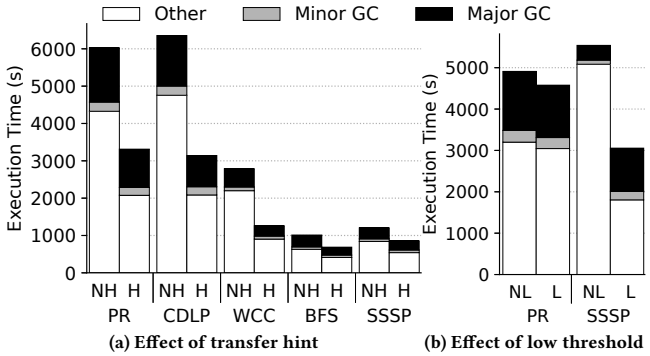


Figure 9: TeraHeap performance for Giraph (a) with (H) and without (NH) transfer hint, and (b) with (L) and without (NL) low transfer threshold.

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

Table 5: Metadata size per TB of H2 space.

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 8 shows the performance of Spark with PS, G1, and *TeraHeap*, for the same amount of DRAM.

G1 outperforms PS between 7% (LR) and 72% (TC) because it reduces GC time by up to 95%. However, 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 access to the storage resident objects. Thus, *TeraHeap* improves performance between 21% (CC) and 48% (LgR) 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 these 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.

7.2 Effects of Transfer Hint and Low Threshold

This section examines the performance effect of using the transfer hint `h2_move()`. Figure 9(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 9(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 accesses 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 9(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 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 9(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.

7.3 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

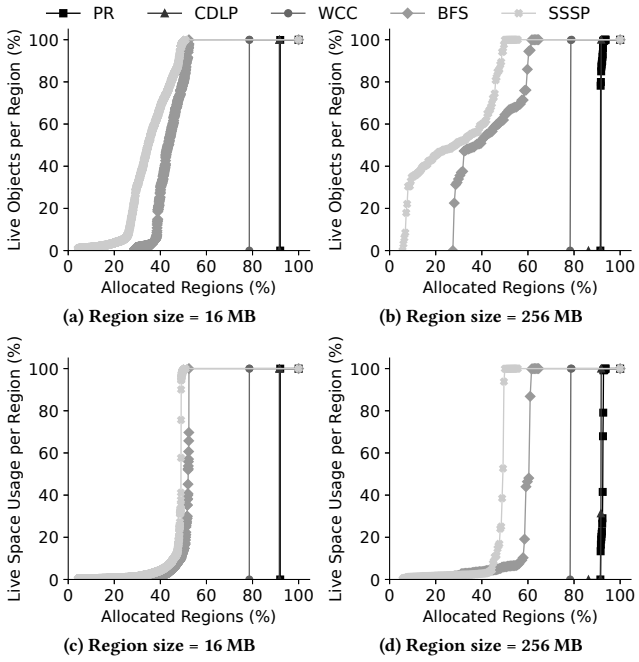


Figure 10: CDF of the percentage of live objects (top row) and space occupied by live objects (bottom row) during execution.

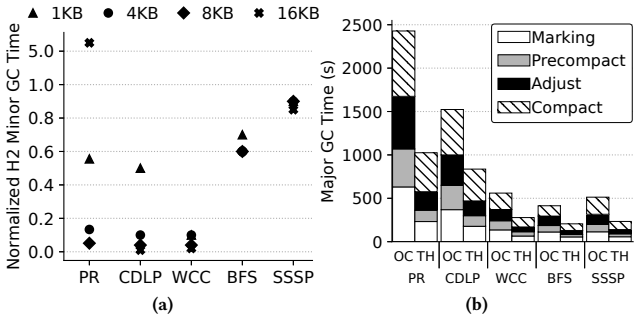


Figure 11: (a) Minor GC time in H2 for different card segment sizes using a 256 MB stripe size in Giraph. (b) Major GC time using Giraph-OOC (OC) and TeraHeap (TH).

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 5 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.

Figures 10(a) and 10(b) show the CDF of the percentage of live objects per region for *all allocated* regions with 16 MB and 256 MB size, respectively. Figures 10(c) and 10(d) show 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 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 Figures 10 (a) and 10(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 10(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.

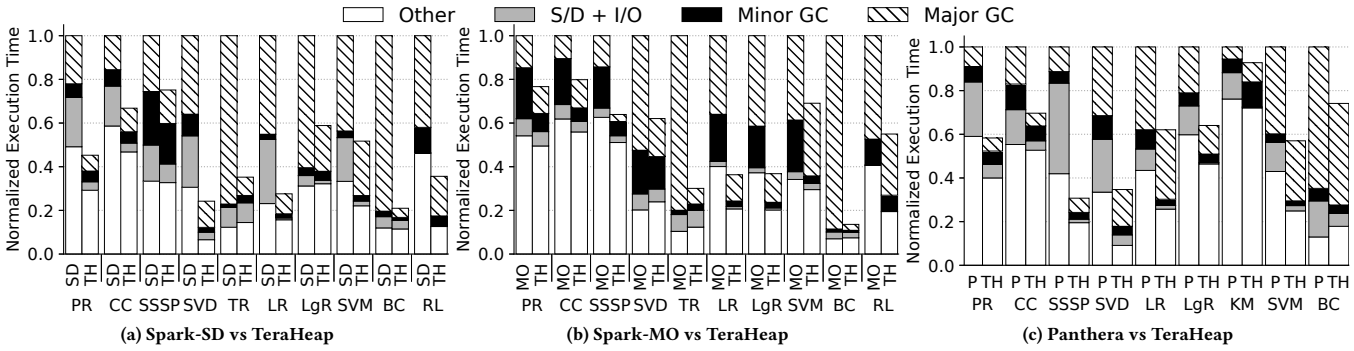
7.4 GC Overhead

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 11(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 in minor GC time for scanning and update 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 11(b) shows the four phases of major GC time using Giraph-OOC and *TeraHeap*. Overall, *TeraHeap* improves all phases of major GC by up to 75% (BFS) compared to Giraph-OOC because we avoid scanning H2 objects. For example, in PR, the collector avoids following in each GC, on average, 109 million forward references from H1 to H2 objects. We note that the compaction phase takes between 37% and 44% of the major GC time in *TeraHeap* due to the device I/O.

7.5 TeraHeap Performance with NVM

Figure 12 shows the performance of Spark-SD, Spark-MO, and *TeraHeap* on our NVM-based setup. We present only Spark workloads due to space constraints. Our goal is to examine the benefits of



(a) Spark-SD vs TeraHeap (b) Spark-MO vs TeraHeap (c) Panthera vs TeraHeap

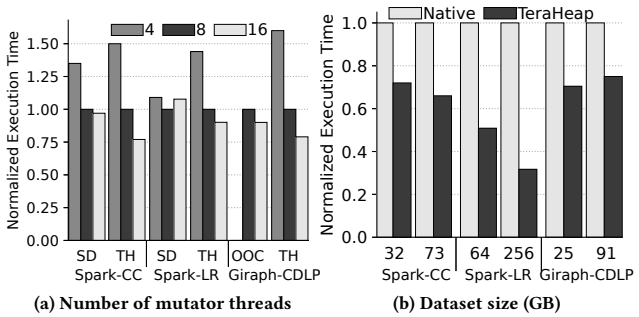
Figure 12: TeraHeap (TH) performance compared to (a) Spark-SD, (b) Spark-MO, and (c) Panthera (P) over NVM server.

TeraHeap when using NVM to increase the heap size, which can eliminate S/D at increased GC cost for native. Figure 12(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 12(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 [53] and the agnostic placement of objects. For instance, minor GC time in Spark-MO increases on average by 36% compared to Spark-SD (Figure 12b) 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 [48][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* [48] 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.

[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)

Figure 13: Performance scaling with (a) number of mutator threads and (b) dataset size in the NVMe server.

Figure 12(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*.

7.6 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 13(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 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 13(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.

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) region-based memory management, (2) scaling managed heaps beyond DRAM capacity, and (3) mitigating S/D overhead.

Region-based memory management. Managed big data frameworks have started to use region-based memory management for large heaps. Facade [36] provides a compilation framework that transforms programmer-specified classes for off-heap allocation. However, it increases the programmer's effort because they need to specify when to free objects from native memory. Broom [16] uses region annotations but requires refactoring of applications' source code. Yak [35] 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 [29] 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 [9] uses runtime profiling to identify long-lived objects. They incur online profiling overhead. Other work uses offline allocation site profiling to manage objects [7, 8]. Lifetime profiling can complement *TeraHeap* and further improve efficiency.

Scaling managed heaps beyond DRAM capacity. Recent efforts target NVM for storing managed heaps beyond DRAM. Akram et al. [2, 3] focus on improving NVM write endurance. Yang et al. [54] report high GC overhead with NVM-backed volatile heaps and optimize the G1 GC for Intel Optane persistent memory. Panthera [48] extends the managed heap over hybrid DRAM and non-volatile memory (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, TMO [49] monitors application DRAM usage and transparently offloads cold data to NVMe SSD. Unlike these works, *TeraHeap* control which objects to move to the second heap and eliminates slow GC traversals over objects on NVM or NVMe SSD. Finally, prior effort [26] discusses a preliminary prototype of *TeraHeap*, while this paper presents the full design and evaluation with production frameworks.

Mitigating S/D overhead. Several libraries [15, 17, 45] improve the efficiency of S/D, but they cannot reduce high GC cost in big data frameworks. Skyway [34] 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 [25, 42] reduces S/D overheads in analytics frameworks using custom hardware and modifications to the programming model. Other work [43, 46, 50] 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 both GC and S/D 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, high-capacity heaps 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, high-capacity heap over a fast storage device alongside the regular heap, 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 the Spark and Giraph performance by up to 73% and 28%, respectively. Overall, our proposed approach of managing large memory in the JVM as customized, separate heaps is a promising direction for incorporating huge 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 their help in preparing the final version of the paper. We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the projects EVOLVE (grant agreement No 825061). This research is also partly supported by project EUPEX, which has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101033975. The JU receives support from the European Union's Horizon 2020 research and innovation programme and France, Germany, Italy, Greece, United Kingdom, Czech Republic, Croatia. Iacovos G. Kolokasis is also supported by the Meta Research PhD Fellowship and the State Scholarship Foundation of Cyprus.

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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 971–985. 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, Article 9 (March 2019), 27 pages. 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)*. Association for Computing Machinery, New York, NY, USA, 62–77. 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. https://doi.org/10.1109/MM.2019.2930497

[5] Monica Beckwith. 2013. Garbage First Garbage Collector Tuning. Retrieved January 2023 from https://www.oracle.com/technical-resources/articles/java/g1gc.html#:~:text=During%20mixed%20collections%2C%20the%20G1,overall%20acceptable%20heap%20waste%20percentage

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

[7] 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 (Jan. 2007), 2–es. https://doi.org/10.1145/1180475.1180477

[8] 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)*. Association for Computing Machinery, New York, NY, USA, 342–352. https://doi.org/10.1145/504282.504307

[9] 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)*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/3092255.3092272

[10] 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 Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 28, 16 pages. https://doi.org/10.1145/3302424.3303988

[11] 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 (Nov. 2014), 335–354. https://doi.org/10.1007/s10115-013-0727-6

[12] Databricks. 2015. Project Tungsten: Bringing Apache Spark closer to bare metal. Retrieved January 2023 from https://www.databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

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

[14] Apache Software Foundation. 2016. Apache Arrow: A cross-language development platform for in-memory data. Retrieved January 2023 from https://arrow.apache.org/

[15] Apache Software Foundation. 2018. Apache Thrift. Retrieved January 2023 from https://thrift.apache.org/

[16] 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, USA, Article 2, 2 pages.

[17] Google. 2001. Protocol Buffers. Retrieved January 2023 from https://developers.google.com/protocol-buffers/docs/javatutorial

[18] Brendan Gregg. 2017. Visualizing Performance with Flame Graphs. USENIX Association, Santa Clara, CA.

[19] 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. https://doi.org/10.5772/intechopen.86917

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

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

[22] Barry Hayes. 1991. Using Key Object Opportunism to Collect Old Objects. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*. Association for Computing Machinery, New York, NY, USA, 33–46. https://doi.org/10.1145/117954.117957

[23] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[24] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[25] 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, 322–334. https://doi.org/10.1109/ISCA45697.2020.00036

[26] 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.

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

[28] 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 Computing* 20, 3 (Sept. 2017), 2575–2589. https://doi.org/10.1007/s10586-016-0723-1

[29] 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 (Aug. 2016), 936–947. https://doi.org/10.14778/2994509.2994513

[30] 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 (Feb. 2020), 85–101. https://doi.org/10.26599/BDMA.2019.9020015

[31] Ioannis Malliotakis, Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2021. HugeMap: Optimizing Memory-Mapped I/O with Huge Pages for Fast Storage. In *Euro-Par 2020: Parallel Processing Workshops (Euro-Par '20)*. Springer International Publishing, Cham, 344–355. https://doi.org/10.1007/978-3-030-71593-9_27

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

[33] 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)*. Association for Computing Machinery, New York, NY, USA, 538–553. https://doi.org/10.1145/3341301.3359643

[34] 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 Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 56–69. https://doi.org/10.1145/3173162.3173200

[35] 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, USA, 349–365.

[36] 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 Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS*

'15). Association for Computing Machinery, New York, NY, USA, 675–690. https://doi.org/10.1145/2694344.2694345
[37] Patrick Niemeyer and Daniel Leuck. 2013. *Learning Java.* O'Reilly Media, Inc.
[38] Oracle. 2014. Class (Java Platform SE 8). Retrieved January 2023 from https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html
[39] Oracle. 2021. Reference Class (Java SE 17 & JDK 17). Retrieved January 2023 from https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/ref/Reference.html
[40] Andrei Pangin. 2018. Async-profiler. Retrieved January 2023 from https://github.com/jvm-profiling-tools/async-profiler
[41] 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 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '20)*. USENIX Association, USA, Article 56, 15 pages.
[42] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1203–1216. https://doi.org/10.1145/3373376.3378501
[43] 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)*. Association for Computing Machinery, New York, NY, USA, 199–205. https://doi.org/10.1145/3458336.3465287
[44] Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. 2017. *Large-Scale Graph Processing Using Apache Giraph* (1st ed.). Springer Publishing Company, Incorporated.
[45] Esoteric Software. 2013. Kryo. Retrieved January 2023 from https://github.com/EsotericSoftware/kryo
[46] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. 2021. Naos: Serialization-free RDMA networking in Java. In *Proceedings of the 2021 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '21)*. USENIX Association, USA, 1–14.
[47] David Ungar. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. Association for Computing Machinery, New York, NY, USA, 157–167. https://doi.org/10.1145/800020.808261
[48] 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)*. Association for Computing Machinery, New York, NY, USA, 347–362. https://doi.org/10.1145/3314221.3314650
[49] 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)*. Association for Computing Machinery, New York, NY, USA, 609–621. https://doi.org/10.1145/3503222.3507731
[50] 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)*. Association for Computing Machinery, New York, NY, USA, 206–212. https://doi.org/10.1145/3458336.3465283
[51] 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, USA, 16–20.
[52] 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 (Jan. 2019), 570–583. https://doi.org/10.14778/3303753.3303762
[53] 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, USA, 169–182.
[54] 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 Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 343–358. https://doi.org/10.1145/3447786.3456246
[55] 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, USA, 2 pages.
[56] 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, USA, Article 10, 10 pages.

A ARTIFACT APPENDIX

A.1 Abstract

The main goal of this appendix is to allow readers to reproduce the paper's results. We provide information about the source code, hardware, and software dependencies and the build process. We also discuss how to install *TeraHeap* and run the main experiments of the paper.

A.2 Artifact Check-list (Meta-information)

- **Program:** Spark v.3.3.0, SparkBench, Giraph v.1.2, LDBC Graphalytics Benchmark Suite for Giraph
- **Compilation:** gcc v.8.3.1, g++ v.8.3.1, maven v.3.5.4, GNU Make v.4.2.1
- **Data set:** KDD12 (21 GB), datagen-9_0-fb (30 GB), datagen-sf3k-fb (91 GB), datagen-8_9-fb (25 GB)
- **Run-time environment:** OpenJDK8 (v.1.8.0_345), OpenJDK11 (v.11.0.17+6), OpenJDK17 (v.17.0.4.1+0), Centos 7, Linux Kernel v.4.14, Linux Kernel v.3.10
- **Hardware:** Two 1 TB Samsung PM983 PCI Express NVMe SSD, 3 TB Intel Optane DC Persistent Memory, 4 TB HDD (for datasets)
- **Execution:** single user, cgroups
- **Metrics:** Execution time
- **Output:** CSV files and graphs
- **Experiments:** Bash and Python scripts
- **How much disk space required (approximately)?:** 1.5 TB for the datasets and 1 TB for H2 allocation
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 12 days
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** The GNU General Public License (GPL)
- **Data licenses (if publicly available)?:** The Apache Licence v.2.0
- **Archived (provide DOI)?:** Yes. https://doi.org/10.5281/zenodo.7590151

A.3 Description

A.3.1 How to access. All scripts are available in the GitHub repository https://github.com/CARV-ICS-FORTH/asplos2023_ae. All sources, including JVM, frameworks, and benchmarks, are included as public git submodules. Also, the artifact is available at https://doi.org/10.5281/zenodo.7590151.

A.3.2 Hardware dependencies. We recommend a dual-socket server that is equipped with two Intel(R) Xeon(R) CPU E5-2630 v3 CPUs running at 2.4 GHz, each with eight physical cores and 16 hyper-threads for a total of 32 hyper-threads. The server should have at least 128 GB DRAM. We recommend using two 1 TB Samsung PM983 PCI Express NVMe SSDs and an HDD (larger than 1.5 TB) to allocate the datasets. For the evaluation with NVM, we consider using a dual-socket server with two Intel Xeon Platinum 8260M CPUs at 2.4 GHz, with 24 cores and (96 hyper-threads), and 192 GB of DDR4 DRAM. We use Intel Optane DC Persistent Memory with

a total capacity of 3 TB, of which 1 TB is in Memory mode and 2 TB are in AppDirect mode.

A.3.3 Software dependencies. The compilation environment and the provided scripts assume Centos 7, which uses Linux Kernel v.3.10 and v.4.14. Also, you need to install additional packages using the following script:

```
## Install packages
$ cd scripts
$ ./install_package.sh
```

A.3.4 Data sets. The required datasets for Spark workloads (except BC) are automatically generated using the SparkBench suite dataset generator. The dataset will be generated when executing the specific scripts to run Spark workloads. The datasets for Spark-BC and Giraph workloads are downloaded automatically before each workload execution.

A.4 Installation

For the installation, either download the complete artifact from Zenodo (https://doi.org/10.5281/zenodo.7590151) or clone the GitHub repository from https://github.com/CARV-ICS-FORTH/asplos2023_ae. The GitHub contains all the required scripts to run the artifact. The source code of *TeraHeap*, frameworks, benchmarks will be downloaded by running the following script:

```
$ cd scripts/build_jvm
$ ./build_jvm.sh
$ cd ../build_apps
$ ./build_spark.sh
$ ./build_giraph.sh
```

A.5 Experiment workflow

Once you build TeraHeap, Spark, SparkBench suite, Giraph, Graphalytics Benchmark suite, and the native JVM, you can execute the workloads as described below. We provide individual scripts for each figure in the paper.

A.5.1 Experiments with NVMe SSD. For the workloads that use NVMe SSD run the appropriate `./gen_figXX.sh` under each `./experiments/figureXX` directory. Generate first Figure 5 and then the rest of the figures because some of the produced results (e.g., results with TeraHeap) are reused in the other figures.

```
# To generate Figure 5
$ cd scripts/experiments/figure5
$ gen_fig5.sh
# To generate Figure 7
$ cd scripts/experiments/figure7
$ gen_fig7.sh
```

A.5.2 Experiments with NVM. For the workloads that use NVM run the following scripts on the server with PMEM:

```
$ cd scripts/experiments/figure11a
$ gen_fig11a.sh

$ cd scripts/experiments/figure11b
$ gen_fig11b.sh

$ cd scripts/experiments/figure11c
$ gen_fig11c.sh
```

A.6 Evaluation and expected results

When you have completed all or some of the experiments, you can compare the produced results with the results in the paper. The reference results used in the paper are located in the folder `results/reference/`.

Copy the directory `results/` to your local system and execute:

```
$ cd scripts
$ ./generate_plots.sh
```

Open `teraheap-artifacts-report.html` in your web browser to check the generated plots side-by-side with the results in the paper.

Received 2022-10-20; accepted 2023-01-19