Bridging the Performance Gap for Copy-based Garbage Collectors atop Non-Volatile Memory

Yanfei Yang, Mingyu Wu, Haibo Chen, Binyu Zang
Institute of Parallel and Distributed Systems
Shanghai Key Laboratory for Scalable Computing Systems
Shanghai Jiao Tong University

Abstract

Non-volatile memory (NVM) is expected to revolutionize the memory hierarchy with not only non-volatility but also large capacity and power efficiency. Memory-intensive applications, which are often written in managed languages like Java, would run atop NVM for better cost-efficiency. Unfortunately, such applications may suffer from performance slowdown due to the unmanaged performance gap between DRAM and NVM. This paper studies the performance of a series of Java applications atop NVM and uncovers that the copy-based garbage collection (GC), the mainstream GC algorithm, is an NVM-unfriendly component in JVM. GC becomes a severe performance bottleneck especially when memory resource is scarce. To this end, this paper analyzes the memory behavior of copy-based GC and uncovers that its inappropriate usage on NVM bandwidth is the main reason for its performance slowdown. This paper thus proposes two NVM-aware optimizations: *write cache* and *header map*, to effectively manage the limited NVM bandwidth. It further improves the GC performance with hardware instructions like non-temporal memory accesses and prefetching. We have implemented the optimizations on two mainstream copy-based garbage collectors in OpenJDK. Evaluation with various memory-intensive applications shows that our optimizations can improve the GC time, application execution time, application tail latency by up to 2.69×, 11.0%, and 5.09×, respectively.

CCS Concepts: • **Software and its engineering → Runtime environments**; • **Hardware → Memory and dense storage**.

Keywords: Non-Volatile Memory, Java Virtual Machine, Garbage Collection

EuroSys '21, April 26–28, 2021, Online, United Kingdom
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456246

ACM Reference Format:
Yanfei Yang, Mingyu Wu, Haibo Chen, Binyu Zang. 2021. Bridging the Performance Gap for Copy-based Garbage Collectors atop Non-Volatile Memory. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3447786.3456246

1 Introduction

Commercialized non-volatile memory (NVM) devices, such as Intel Optane DC PM [22], promise large memory capacity, energy efficiency, and low per-GB cost, making them a supplement to traditional DRAM devices. Thanks to NVM, data centers can embrace a hybrid memory architecture to benefit from both the access speed of DRAM and the large capacity of NVM. It is anticipated that applications will execute atop NVM.

Due to the portability and productivity of managed languages, many memory-intensive applications are written in those languages and thus running atop the corresponding language runtime. To meet the eager memory demands of applications, language runtimes have provided support for alternative memory devices. For example, the HotSpot Java Virtual Machine (JVM) has allowed creating a data heap from NVM. Afterward, applications can directly manipulate Java objects on the NVM device. Note that the NVM support merely increases the memory capacity and provides no persistence guarantee.

Unfortunately, prior work [24, 42] has shown that a significant performance gap stands between NVM and DRAM. First, the access latency of NVM, regardless of types (read/write) and patterns (random/sequential), is larger than DRAM. Second, the NVM bandwidth is much smaller than DRAM (regardless of access patterns) and much more sensitive to workloads. When the number of write operations increases, the overall bandwidth will dramatically decline. Although previous work [9–11, 18, 20, 24, 25, 27, 42] has analyzed and optimized the performance of various workloads atop NVM, the performance with managed runtimes is less known. Therefore, this work studies the performance of a series of Java memory-intensive applications with NVM. The observation is that the garbage collection (GC) performance in JVM dramatically drops. The mainstream GC in JVM is copy-based and has a large demand for memory bandwidth to conduct massive object traversal and copying, which cannot be satisfied due to the shortage and instability of NVM bandwidth. As a result, the accumulated GC time can be prolonged by up to 8.25× and thus contributes to nearly 20% of the execution time for memory-intensive applications.

To bridge the performance gap between DRAM and NVM for managed runtimes, this work mainly studies the performance of Garbage First Garbage Collector (G1GC), the default garbage collector of OpenJDK. We uncover two problems that hinder G1GC from gaining better performance atop NVM. First, the GC algorithm of G1GC mixes write operations with reads. In G1GC, GC threads copy objects (write operations) when they are traversing the heap (read operations), which results in a mixed workload and hurts the overall NVM bandwidth. Second, G1GC generates many random write operations during object copying. For each reference in a live object, it has to be updated twice during GC, while the second update is a random write. Furthermore, each object header will also be modified twice. Those write operations further affect the NVM bandwidth.

To solve the problems above, this work proposes to redesign the original copy-based algorithm. First, the algorithm should be split into multiple sub-phases to avoid mixing read operations with writes. Second, the number of write operations atop NVM should be minimized to sufficiently leverage its bandwidth. To achieve those two goals, this work introduces two optimizations: *write cache* and *header map*. The write cache temporarily stores live objects in DRAM and flushes them into NVM at the end of GC. With the write cache, G1 is split into a read-mostly sub-phase and a write-only subphase to separate NVM reads from writes. Meanwhile, the header map stores the updates to headers in a DRAM-based hashmap and thus eliminates unnecessary NVM writes.

This work implements the NVM-friendly mechanisms on G1GC in the HotSpot JVM of OpenJDK 12, which already provides preliminary NVM support for applications. To further optimize the performance of G1GC atop NVM, we use the off-the-shelf hardware features, such as non-temporal instructions and software prefetching. We show that non-temporal instructions are suitable for copy-based GC and can be leveraged to reduce the memory overhead of the write cache. Furthermore, software prefetching instructions are also helpful to improve the locality of GC traversal. We also implement our optimizations on another copy-based garbage collector (Parallel Scavenge, PS) to show that they are not limited to a specific GC algorithm.

We evaluate our NVM-friendly GC with memory-intensive workloads, including applications in Spark [44], Cassandra [19], and the Renaissance [32] benchmark suite. The results show that our optimized G1 scales better compared with the baseline and improve the GC pause time by up to 2.69×. Thanks to shorter GC pauses, our optimizations can improve the application execution time by up to 11.0% and reduce the tail latency by as much as 5.09×.

To summarize, the contributions of this work include:

- Comprehensive analysis of copy-based GC atop real NVM devices, including performance evaluation on pause time, bandwidth, and scalability, and step-by-step analysis of the GC algorithm.
- NVM-friendly optimizations, including *write cache* and *header map*, to increase the available NVM bandwidth during GC and eliminate unnecessary write operations.
- An NVM-friendly implementation on the default G1 garbage collector on the HotSpot JVM, which is accelerated with off-the-shelf hardware features like non-temporal instructions and software prefetching.
- A performance evaluation with various memory-intensive workloads to showcase the improvement in GC pause time and application performance.

The rest of this paper is organized as follows. Section 2 introduces the background of G1GC and analyzes its performance atop NVM. Section 3 uncovers the NVM-unfriendly problems in G1 and proposes corresponding optimizations. Section 4 discusses details on implementing an NVM-aware garbage collector and shows how to leverage hardware features to further optimize its performance. Section 5 evaluates the performance of our NVM-aware GC, Section 6 discusses related work, while Section 7 concludes.

2 GC Performance Analysis on NVM

Copy-based garbage collectors sort out the data heap by copying live objects together. The resulting heap consists of a contiguous free space for subsequent memory allocation. In this way, copy-based collectors get rid of complicated issues like fragmentations and free lists, and a heap allocation would be ultra-fast by manipulating a pointer which indicates how much free memory is left (known as *bump pointer*). Due to its neat and efficient design, copy-based GC is popular in managed runtimes. Most collectors in the HotSpot JVM of OpenJDK adopt a copy-based algorithm, such as Parallel Scavenge (PS), Garbage-First (G1) [12], and Shenandoah [13]. This work will take G1 as an example to introduce the workflow of copy-based collectors.

2.1 Garbage-First Garbage Collection

Garbage-First Garbage Collection (G1) is the default collector in the HotSpot JVM starting from OpenJDK 9 [30]. G1 is a partially-concurrent copy-based collector. It embraces a generational design [38] and consists of two spaces: *young space* and *old space*. The young space serves the allocation requests from application threads (known as *mutators*), while the old space stores objects which have survived many collections. The collection algorithm in G1 is three-fold. *Young GC* collects the young space in a stop-the-world (STW) fashion; *mixed GC* identifies live objects in a concurrent marking phase and copies a part of them for memory reclamation with an STW pause; *full GC* pauses mutators to collect the whole heap. Note that the full GC algorithm only stands as a bottom-line for G1. Only when the other two algorithms prove inefficient to reclaim memory should the full GC be triggered. In our evaluation, no full GC is observed for various memory-intensive workloads. Even the mixed GC happens much more rarely than the young GC, and its copy phase is quite similar to that in the young GC. Therefore, this work will mainly focus on the young GC algorithm in G1.

The basic memory management unit in G1 is called a *region*. Each region contains a *remembered set*, recording references that lay in the old space and point to objects within this region. During young GC, objects referred to by the remembered set will be treated as live. Those objects will therefore be copied to other regions and traversed to find more live objects. The copy-and-traverse phase is the most time-consuming part of G1.

In the copy-and-traverse phase, GC threads should process all regions in the young space and move live objects out (also known as *evacuation*). Each GC thread is also assigned with a region (known as *survivor region*) to which live objects will be copied.[1] When the region is filled up, the GC thread will request a new survivor region. To process a region in the young space, GC threads will scan all references in the corresponding remembered set, as objects pointed by those references are treated alive. For each reference, a GC thread locates its referent (the referred object), copies the referent to the survivor region, and updates the reference with the referent's new address. Afterward, the GC thread needs to traverse all references stored in the referent since objects referred to by them are also alive. Those references are pushed into a per-thread working stack for further processing.

During region processing, it is possible that multiple references point to the same object. To avoid repeated copying, G1 adopts a technique named *forwarding pointers*. When an object has been copied, the GC thread will install its new address to the header of its old copy. If other references also point to this object, the corresponding GC thread will find the forwarding pointer in the header and directly modify the reference with the stored address. The forwarding pointer is installed with an atomic instruction to ensure that only one GC thread succeeds in copying the object.

2.2 Performance analysis atop NVM

We have conducted a series of experiments to understand how memory-intensive applications and copy-based GC will behave on NVM. The OpenJDK version is 12.0.1, which has supported allocating Java heap from alternative memory devices [31]. During the evaluation, we leverage the option *-XX:AllocateHeapAt* to allocate the heap onto the Intel Optane DC PM devices and compare it with the baseline, which allocates its heap on DRAM. In this work, we only consider scenarios where NVM is used for capacity considerations, so we neglect the runtime overhead for persistence guarantee (e.g., the cost of cache flushing instructions).

As for workload, we first choose two applications in Spark: *page-rank* and *kmeans*. Those two have been used by prior work to study the runtime behavior in a heterogeneous memory environment [39]. We also extract four memory-intensive applications from the recently released Renaissance benchmark [32], which contains various workloads like machine learning, graph processing, and software transactional memory. The detailed experiment setup is shown in Section 5.1. We have two important findings from the evaluation.

NVM has a larger impact on the performance of GC compared with applications. Figure 1 shows the GC performance for all six applications when replacing DRAM with NVM. One may expect that the slowdown should be similar to the access latency gap between DRAM and NVM (2-3×). However, the result shows the GC pause time actually increases by 2.02×-8.25× with NVM (averaging 6.53×). In contrast, the application execution time without GC is less affected by NVM. For the above six applications, when all the Java heap has been put onto NVM, the execution time without GC is increased by 2.68× on average, and some applications such as *movie-lens* have similar execution time to those running on DRAM.

GC becomes a more severe bottleneck when running atop NVM. The duration of GC pauses is vital especially for memory-intensive applications. When running with NVM, since the GC performance is significantly affected, GC pauses will consume a higher proportion of the overall time. For the above applications, GC pause time represents 3.0% of the execution time on DRAM. However, when it is migrated on NVM, the ratio grows to 6.3%. For the page-rank application in Spark, 17.6% of execution time is occupied by GC. In a memory-hungry configuration where the heap size is smaller, optimizing GC for NVM may become even more important.

There are two reasons to explain why the copy-based GC shows unsatisfying performance. First, the copy-and-traverse phase in GC often exhibits worse locality than normal execution. Since references in the remembered set come from different regions throughout the heap, manipulating them will introduce random read and write operations. Furthermore, when traversing the references inside a live object, since their referents can reside anywhere in a heap, the traversal also results in random memory accesses. When the locality is poor, the possibility of cache-miss increases and the JVM has to suffer from a higher penalty when fetching cache lines from the NVM devices. Second, the copy-based GC involves excessive memory operations and easily saturates the memory bandwidth of NVM. As prior work suggests [24, 42], the NVM bandwidth is smaller than DRAM. Worse still, its bandwidth is not stable and sensitive to the workloads. When the number of write operations increases, the overall NVM

[1] Some live objects will be directly copied to regions in the old space, but it rarely happens.

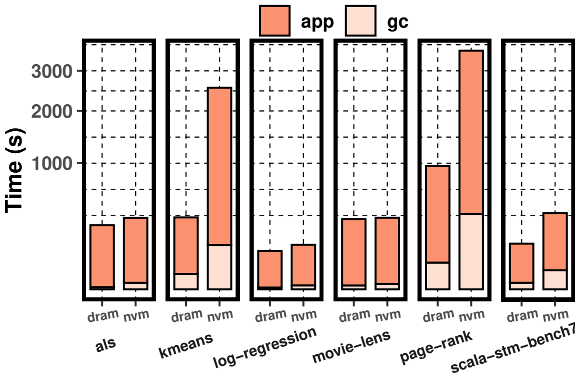


Figure 1. The application and GC time when replacing DRAM with NVM

bandwidth dramatically declines. Since the locality problem is the inherent limitation of copy-based GC, this work will mainly focus on the bandwidth issue (Section 4.3 will further discuss locality).

2.3 Detailed bandwidth analysis

We have studied the NVM bandwidth when running with different numbers of GC threads. The application is *page-rank* in Spark, and the bandwidth statistics are collected with the Intel PCM tool. We also run those applications atop DRAM and collect the bandwidth data for comparison.

Figure 2 shows the consumed bandwidth of the page-rank benchmark for DRAM and NVM respectively. The solid vertical lines demarcate the time intervals when copy-based GC is active. According to Figure 2a, the write bandwidth for DRAM during GC significantly increases due to massive object copying, and so does the overall bandwidth. However, the results are quite different in the NVM case (Figure 2b). Although GC threads have a large appetite for the bandwidth, the overall bandwidth dramatically drops. This phenomenon can be explained by three reasons. The primary one is that NVM bandwidth is quite sensitive to workloads. When the number of write operations increases, the overall bandwidth will be strongly affected and decline. This problem is possibly caused by NVM's asymmetric bandwidth: its peak read bandwidth is much larger than the peak write bandwidth [20, 24]. Other NVM technologies like phase-change memory (PCM) also have similar problems [34]. Furthermore, the random memory access pattern and contentions between GC threads are also harmful to the overall bandwidth.

To study the relationship between the shortage of NVM bandwidth and the performance of copy-based GC, we further conduct a scalability test by varying the number of GC threads for the page-rank applications. Adding GC threads can theoretically increase the collection parallelism and reduce the GC time. However, as shown in Figure 2c, when the number of GC threads exceeds 8, the NVM bandwidth barely changes. Consequently, the copy-based GC is non-scalable with eight or more threads. On the contrary, GC can scale with more threads when running on DRAM as the consumed bandwidth can continuously increase (Figure 2d). The result confirms the importance of NVM bandwidth to the GC performance.

The NVM bandwidth shortage problem can also be used to explain the performance variance among different applications. Figure 3 shows the consumed NVM bandwidth of *als* during GC is larger than that during application execution, which is similar to the curve in the DRAM setting. This suggests that the NVM bandwidth is not saturated when GC is not active, so the application time is not affected much compared with page-rank (see Figure 1).

2.4 Similarities in other copy-based collectors

Although copy-based collectors have different design goals, they share similarities in design. For example, Shenandoah [13] is designed as a mostly-concurrent garbage collector to reach low GC pauses, but it also divides its heap into equal-sized regions as G1 does. When GC is triggered, GC threads in Shenandoah also allocate a thread-local buffer to evacuate live objects. Thread-local buffers are also used in the copy-based young GC algorithm of PSGC to store newly-copied objects. Furthermore, those collectors also install forwarding pointers in the object header. Due to the similarities among copy-based collectors, NVM-friendly optimizations on G1 may also be beneficial to other collectors.

3 Design

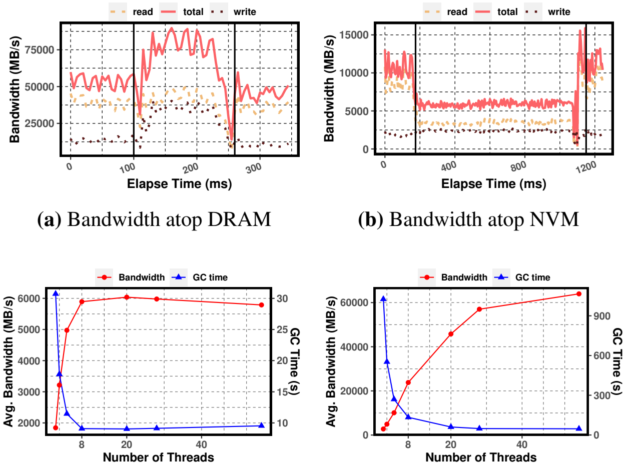
Since the shortage and instability of NVM bandwidth is the culprit of inefficient garbage collections, this work analyzes the memory behavior of the time-consuming copy-and-traverse phase. It then proposes two optimization techniques to maximize the available NVM bandwidth and improve the GC performance.

3.1 Memory behavior analysis

As discussed in Section 2.1, each GC thread works with its thread-local stack and repeats the following four steps in the copy-and-traverse phase:

1. Fetch a reference from its thread-local working stack and find its referent object (random read).
2. Copy the object to the survivor region (sequential read/write).
3. Update the forwarding pointer in the old copy (random write).
4. Update the reference with the new address of its referent (random write), and push references in the referent to the working stack (sequential read).

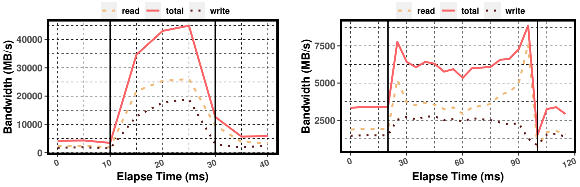
This algorithm has two major problems when running on NVM. First, it mixes read operations with writes. When GC threads are in the copy-and-traverse phase, they will issue both read and write operations to NVM. The write operations



(a) Bandwidth atop DRAM **(b)** Bandwidth atop NVM

(c) Bandwidth vs. scalability (NVM) **(d)** Bandwidth vs. scalability (DRAM)

Figure 2. Bandwidth statistics for the page-rank application



(a) Bandwidth atop DRAM **(b)** Bandwidth atop NVM

Figure 3. Bandwidth statistics for the als application

will interfere with read operations and significantly reduce the overall bandwidth. Second, it generates excessive NVM writes for each object. As for a reference in the object, it needs to be updated twice: the first time is object copying while the second time is reference updating. The header should also be updated twice to install the forwarding pointer. Those two problems result in NVM bandwidth shortage and instability during copy-based GC, so even a small number of GC threads would saturate the bandwidth and become no longer scalable even with more computing resources.

To this end, we propose that the original copy-based GC algorithm should be reconstructed to unleash the NVM bandwidth. First, the copy-and-traverse phase should be divided into sub-phases to reduce the number of NVM writes overlapped with reads. Second, the write operations on NVM should be minimized to leverage the NVM bandwidth to its fullest. Keeping those two rules in mind, we provide two optimization techniques: write cache and header map.

3.2 Write cache

We first introduce the write cache, which temporarily stores survivor regions in DRAM. When a GC thread copies a live object to the survivor region (step 2), it will instead copy the object into a cache region on DRAM. The cache region will absorb all NVM writes to survivor regions and only be written back to NVM before GC ends.

The write cache divides the copy-and-traverse phase into two sub-phases: *read-mostly sub-phase* and *write-only sub-phase*. In the first read-mostly sub-phase, all write operations on newly-copied objects will be absorbed by the write cache, so the number of NVM writes is significantly reduced. In the second write-only sub-phase, GC threads have finished evacuating live objects, so they only need to simultaneously write back the cache regions into NVM. In this way, NVM writes and reads are separated into different sub-phases, and the available NVM bandwidth will become more abundant. Furthermore, by caching survivor regions in DRAM, random reference updates (step 4) on newly-copied objects will not occur on NVM. Instead, all NVM writes on the survivor regions now become sequential write-back operations, which can leverage the NVM bandwidth to its fullest.

Since the write cache reduces the number of NVM writes for each GC cycle, it also introduces other benefits. First, the reference updates on newly-copied objects become faster as they are operated on DRAM. Second, the reduction on NVM writes can also prolong the lifetime of NVM [2].

An issue introduced by the write cache is the address remapping. When an object is copied in the copy-and-traverse phase, the reference should be updated to its new address. However, since the object is temporarily stored in the write cache, the new address is a DRAM address, which becomes invalid after GC. To resolve this problem, the new address for an object on NVM should be determined even though it resides in the write cache. Therefore, we introduce a region mapping, where each cached DRAM region is related to an NVM region in the Java heap. When a cached region is filled up, the GC thread should allocate a new region in DRAM together with a new survivor region in NVM. Afterward, the GC thread will maintain a mapping between those two regions. Once an object is copied into the write cache, the GC thread will use the mapping to calculate its corresponding NVM address and update the reference.

For some applications, the size of live objects is numerous, so caching all of them would induce a considerable DRAM footprint. Therefore, we provide an option to set the upper bound of the write cache size. When the write cache is full, the GC thread stops allocating new cache regions and directly copies objects into NVM.

3.3 Header map

The write cache optimization has separated most NVM write operations with NVM reads. However, GC threads still need to update headers in the old copy (step 3) and references in non-moved objects (step 4), which introduce random NVM writes and affect the overall bandwidth. Although updating the non-moved objects in the NVM is a must, updating the headers is not necessary. Since the installation of forwarding pointers is only useful during GC, we propose a *header map* in DRAM to reduce write operations on NVM.

The header map is organized as a global lock-free hashmap to support fast lookup and modification. Each entry in the map contains two addresses: an object's old address (key) and new address (value). An alternative design would be maintaining per-thread header maps. However, this design is not suitable for state-of-the-art GC algorithms where all threads are allowed to collect objects in any location of the heap. To check if an object has been copied, a GC thread may need to check all others' thread-local hash tables, which introduces even more overhead.

Since the DRAM consumption of our approach should be moderate (otherwise allocating the whole heap on DRAM might be a better choice), the header map embraces closed-hashing so as to bound its DRAM footprint. Like the write cache, we also provide an option to adjust the maximum memory consumption of the header map. A larger header map means that more forwarding pointers will be installed in DRAM instead of NVM, so the NVM read bandwidth can be further increased with fewer NVM writes.

Algorithm 1 shows the pseudo-code for the pointer installation. When a forwarding pointer should be installed, a GC thread will first try putting it into the header map (Line 2). The put operation starts by transforming the old address of an object into an index with a hash function (Line 7). Afterward, it scans the header map from the computed index. The scanning phase is bounded: if the number of scanned entries exceeds a preset threshold, the header map cannot find a free entry to store the forwarding pointer, so the put operation will return with a null value (Line 11-13). In this case, the GC thread should install the forwarding pointer into the corresponding header on NVM (Line 3). If the header map manages to find a free entry (Line 16-19), the put operation will occupy it with an atomic compare-and-swap (CAS) instruction (Line 20-21). If the instruction fails, the free entry must be occupied by other GC threads. Therefore, the GC thread should check if others are processing the same object (Line 22). If so, the GC thread waits until others finish installing the forwarding pointer in the header map and then returns the value of the pointer (Line 23-27). If other GC threads occupy the entry for other objects, the put operation simply skips this entry to find another free one (Line 28-30). If the CAS instruction succeeds, the GC thread will install the forwarding pointer by storing the new address of an object to the *value* field of the entry (Line 31-32). During scanning, if the GC thread finds that the forwarding pointer has been installed by others, it will directly return with the value of the pointer (Line 35-39).

If a GC thread wants to check if an object has ever installed a forwarding pointer, it needs to query the header map by invoking the *get* API, which has a similar workflow to *put*. After gaining an index with the hash function, the GC thread conducts a bounded scanning to find if the forwarding pointer is installed into some entry. The scanning threshold is the same as that in *put* to ensure that every possible entry will be searched. If the pointer has been installed, its value will be returned, which stands for the new address of the queried object. Otherwise, the get operation returns with a null value, and the GC thread should check the object header on NVM to ensure that the forwarding pointer has not been installed yet.

The main benefit introduced by the header map is to reduce unnecessary write operations. Since the maximum read bandwidth interferes with the write bandwidth, reducing writes means that the available read bandwidth is enlarged to support more concurrent GC threads. When the number of GC threads is small, the read bandwidth is not saturated, so the header map brings less improvement. Worse still, since the *get* operations may access both headers and the header map, the latency can be larger than directly accessing NVM. Therefore, the header map is only enabled when the number of GC threads exceeds a threshold (8 by default).

Since the header map contents are only meaningful during GC, they should be removed when GC ends. As the header map size is relatively large, we leverage all GC threads to empty it simultaneously, and the clean-up time is trivial compared with the GC pauses.

4 Optimizations

To prove the effects of the aforementioned optimization techniques, we have implemented them in G1 of the HotSpot JVM. We also leverage off-the-shelf hardware features like non-temporal instructions and software prefetching to further improve the performance of G1GC.

4.1 Embracing non-temporal instructions

Architectures like x86 introduce non-temporal instructions to mark memory accesses as lacking temporal locality. For example, a memory write instruction may be issued by a logger thread, which logs its current state and will be reused only when the program runs into an unexpected error. Since the memory is write-once and unused afterward, it is unnecessary to load it into the cache. In this case, developers can leverage non-temporal write instructions (e.g., *MOVNTDQ* in x86) to write the value to the corresponding memory address while bypassing the cache hierarchy. Since non-temporal write instructions can avoid cache pollution and reduce the memory traffic, it is efficient to implement NVM writes. Nevertheless, non-temporal instructions still have two disadvantages. First, its performance is not satisfying for small and random writes. Second, since non-temporal instructions bypass the cache hierarchy, they are not managed by the cache coherence protocol, and users should carefully add fence instructions to ensure correctness. Since copy-based GC contains many small and random write operations, it cannot directly use non-temporal instructions.

Algorithm 1 The pseudo code for installing a forwarding pointer with the header map

```
function INSTALLPOINTER($oldaddr, newaddr$)
    if $map.PUT(oldaddr, newaddr) == NULL$ then
        UpdateHeader($oldaddr, newaddr$)
    end if
end function
function PUT($oldaddr, newaddr$)
    $idx \leftarrow hash(oldaddr) \& hash\_mark$
    $cnt \leftarrow 0$
    while $TRUE$ do
        $cnt \leftarrow cnt + 1$
        if $cnt > SEARCH\_BOUND$ then
            return $NULL$
        end if
        $idx \leftarrow (idx + 1) \& hash\_mark$
        $probed\_key \leftarrow map[idx].key$
        if $probed\_key$ != $oldaddr$ then
            if $probed\_key$ != $0$ then
                continue
            end if
            $cmp\_key \leftarrow cmpxchg(oldaddr,$
                        $\&map[idx].key, probed\_key)$
            if $cmp\_key == oldaddr$ then
                while $TRUE$ do
                    if $map[idx].value$ != $0$ then
                        return $map[idx].value$
                    end if
                end while
            else if $cmp\_key$ != $0$ then
                continue
            else
                $map[idx].value \leftarrow newaddr$
                return $newaddr$
            end if
        else
            while $TRUE$ do
                if $map[idx].value$ != $0$ then
                    return $map[idx].value$
                end if
            end while
        end if
    end while
end function
```

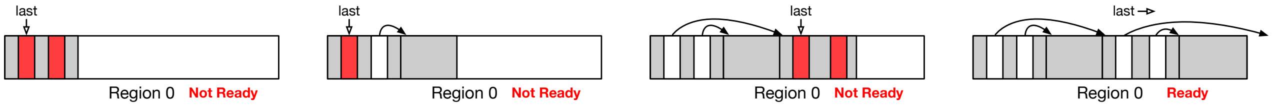
However, we find that non-temporal instructions perfectly fit the write cache optimization. Since the write cache has transformed small and random NVM writes into large sequential write-back operations, the non-temporal writes can be used to improve the performance. As for the cache-coherence issue, since non-temporal instructions are only issued in the write-only sub-phase, a GC thread will not read objects written back by others. Therefore, we only need to insert one fence before GC ends for the correctness guarantee.

4.2 Asynchronous region flushing

As discussed in Section 3.2, since objects in the write cache are flushed into NVM only before GC ends, the DRAM footprint can be quite large, especially when the number of live objects is numerous. A straightforward method to reduce the DRAM footprint is to asynchronously flush cached regions into NVM and reclaim them in advance. However, prior work [20] has reported negative results on this method. Although asynchronous flushing can reduce DRAM consumption, it introduces NVM write operations, which may lead to less NVM bandwidth and worse GC performance.

Fortunately, embracing non-temporal instructions has paved the way for asynchronous flushing. As shown in prior work [24], using non-temporal writes can enlarge the available NVM bandwidth compared with normal ones in a mixed workload. Therefore, when a region is ready for flushing, a GC thread can write the contents therein to NVM with sequential non-temporal instructions.

Another challenge of asynchronous flushing is to decide when it is appropriate to flush a region into NVM. A strawman design would be flushing a region once it is filled up, but this approach fails to absorb subsequent reference updates to this region. To determine if a region is ready for flushing, we need to check if all references therein have been updated. Instead of tracking all references in a region, we propose an efficient tracking mechanism according to the processing order of references (illustrated in Figure 4). Since copy-based GC algorithms usually use a stack-based depth-first-search (DFS) algorithm to traverse the heap, reference processing will follow a last-in-first-out (LIFO) order. Therefore, when the first reference of a region is pushed into the working stack, the GC thread will memorize it as potentially the last reference to be processed. As Figure 4a shows, suppose an object with two references has been copied into a newly-allocated region, the references should be pushed into the working stack in left-to-right order (colored bars). In this case, the leftmost reference will be memorized by the *last* field of the region. When the GC thread pops and processes the rightmost reference (changed to a white bar in Figure 4b), the memorized reference remains unchanged. Afterward, when the leftmost reference is popped (Figure 4c), the GC thread finds that the region is not filled up, so it is open to accept more live objects. Therefore, the GC thread updates the *last* field to the leftmost reference of the object pointed by the previously memorized reference. Finally, when the reference stored in *last* is popped in Figure 4c, since the region is full, all references have been processed. At this moment, the GC thread can mark the region as *ready* so that it can be asynchronously flushed into NVM. It is possible to track references and flush objects in a finer granularity (e.g., 4KB



(a) Memorizing the first pushed reference

(b) Popping others does not affect the memorized one

(c) The region is not filled up: memorize another reference

(d) The region is filled up: mark it as *ready* for flushing

Figure 4. Our reference tracking mechanism for asynchronous flushing

pages), but it requires tracking more units and induces larger maintenance overhead.

The last issue is that modern collectors provide a work-stealing mechanism to achieve load-balance among multiple GC threads. In G1, a GC thread can steal a reference from others' working stacks. This mechanism can break the LIFO order of reference processing. Instead of proposing a more complicated tracking algorithm to consider work-stealing, we simply avoid asynchronous flushing on regions where references have been stolen. This refinement still allows most regions to be flushed asynchronously, as the work-stealing phase has a short duration and affects only a small number of regions.

4.3 Software prefetching

As analyzed in Section 2.2, the copy-based GC exhibits poor locality, which will be exacerbated atop NVM due to a larger miss penalty. To further confirm the poor locality of copy-based GC, we have evaluated applications in Section 2 by assigning them only 1/16 of the last level cache (LLC) with the Intel Cache Allocation Technology (CAT). However, the evaluation results show that GC time barely changes, suggesting that copy-based GC cannot fully leverage cache resources. Therefore, we propose reducing cache misses by adding prefetching instructions.

Prefetching instructions (e.g., *PREFETCH* in x86) help improve the program locality by fetching memory contents to the cache in advance. With prefetching, subsequent memory accesses can hit in the cache, and the cache miss rate is reduced. Prior work [3–5, 36, 45] has studied how to use software prefetching to improve the program performance, but none mentions the effect of prefetching on NVM. To this end, this work first explores the benefit of prefetching with a micro-benchmark. The micro-benchmark creates a large array on DRAM or NVM and generates a series of indices to access the array randomly. For each iteration, the benchmark reads the content in the randomly-generated index and updates it. Since the indices are pre-generated, we add prefetching instructions to load the contents to the cache in advance. The following table shows the evaluation results for both DRAM and NVM with 40 million accesses. Although both DRAM and NVM can benefit from prefetching instructions, the improvement for NVM (3.05×) is significantly larger than that in DRAM (1.58×). The result suggests that software prefetching can boost application performance, especially for NVM-based systems.

Configuration	Results (s)
DRAM-noprefetch	1.513
DRAM-prefetch	0.958
NVM-noprefetch	4.171
NVM-prefetch	1.369

In copy-based GC, consider the steps of copy-and-traverse mentioned in Section 3.1. When a reference is popped from the working stack, the GC thread should fetch its referent, which generates random read operations. The vanilla G1 collector is aware of the locality problem and proposes to reduce the miss penalty with software prefetching. When a reference is being pushed to the working stack, GC threads will prefetch the data in the referent to the cache. Unfortunately, since copy-based GC algorithms usually use DFS to traverse the heap, the benefit of software prefetching is not stable. For a reference that already resides in the stack, it has to wait until all preceding references have been transitively traversed, so the waiting time is quite non-deterministic. A possible approach to stabilize the waiting time is to adopt a queue-based breadth-first-search (BFS) algorithm. Compared with DFS, the order of reference processing in BFS is deterministic, so GC threads can prefetch those which will be processed in the near future. However, BFS is proved detrimental to the locality of applications, as it tends to put irrelevant objects together [28]. As a result, we reuse the software prefetching strategy in G1 as it approximately loads the needed cache lines with moderate overhead. Meanwhile, we also extend the original prefetching instructions to consider the random read operations on the header map.

4.4 Migrating to other collectors

As analyzed in Section 2.4, since copy-based collectors share similarities in design, they may also benefit from NVM-aware optimizations. Therefore, we have implemented our optimizations also in the Parallel Scavenge (PS) garbage collector.

PS is a stop-the-world (STW) generational collector in OpenJDK, and it was used by default before OpenJDK 9. PS also adopts copy-based young GC to collect its young space. The young GC contains a similar copy-and-traverse phase, but GC threads manage live objects in a smaller unit named local allocation buffers (LABs). Furthermore, PS also allows

GC threads to directly copy objects without using LABs. Therefore, we only cache regions which are contiguous in the address space. As for the header map, since the pointer installation logic in PS is similar to that in G1, we modify it so that the forwarding pointers will be first stored into DRAM.

Due to unawareness of NVM, PS does not introduce any software prefetching instructions during its young GC. To this end, we add prefetching instructions when a reference is pushed into the thread-local working stack. We also add prefetching for the header map to mitigate the effect of cache misses.

5 Evaluation

We mainly implement our optimizations on the G1 collector in the HotSpot JVM with approximately 4,000 LoCs. The version of JDK is 12.0.1.

5.1 Experiment setup

The evaluation is conducted on a machine with dual Intel Xeon Gold 6238R CPUs (28 physical cores each, simultaneous multithreading enabled). Each CPU has 6 Intel Optane DC PM devices, and each device contains 128GB NVM. Since cross-NUMA NVM accesses will induce prohibitive overhead, all experiments are bound to run on a single CPU with the *numactl* command. To leverage NVM, we mount all 6 NVM devices attached to a single CPU as a DAX file system and adopt the *-XX:AllocateHeapAt* option to allocate NVM from the file system. Frequency scaling is disabled to stabilize the application performance.

We mainly use the vanilla G1 as the baseline. The number of heap regions is set to 2048, which is the default setting in G1. Since our optimizations consume more DRAM resources, we also evaluate against an optimized version of G1 where the extra DRAM regions are used to serve allocation requests from application threads (Section 5.2).

We use applications from the recently-released Renaissance benchmark [32], Spark [44], and Cassandra [19].

Renaissance. Renaissance (the version is 0.10) contains diverse applications to test various modules in JVM. We exclude three applications from Renaissance due to duplicated experiments in Spark (page-rank and scala-kmeans) and incompatibility with JDK 12 (db-shootout). For evaluated applications, the maximum heap size is set to 16GB while the young space is 4GB. The maximum size of the header map and the write cache is set to 512MB.

Spark. Spark (the version is 2.2.0) is a unified data processing engine. Spark applications are memory-intensive as they will generate quantities of immutable temporary datasets during runtime [43]. We pick four different applications for evaluation: page-rank, kmeans, connected-components (cc), single-source-shortest-path (sssp). As for workload, we leverage the same datasets in Panthera [39], which also studies the runtime behavior of Spark on NVM. Since Spark applications target large datasets, the maximum heap is set to 256GB, and the young space is 64GB. We also set the maximum size of the header map and the write cache to 2GB and 8GB respectively.

Cassandra. Cassandra (the version is 4.0-beta2) is a NoSQL database that allows low-latency access to persistent data. We leverage its built-in testing tool named *cassandra-stress* to evaluate its tail latency atop NVM. The workload contains two phases: a write-only phase and a read-only phase. The latency statistics are collected from the two phases respectively. The heap configuration for Cassandra is the same as those in Renaissance.

All presented results are averages of five runs with standard deviations as error bars (except for bandwidth and tail latency). We do not add extra warm-up phases since evaluated applications reinitialize themselves every time they are executed.

5.2 GC time reduction

Figure 5 shows the reduction in GC time with our optimizations. 23 of 26 applications can benefit from our optimizations, and the GC time can be reduced by 1.69× on average for all applications (up to 2.69×). The remaining three applications contain infrequent and short GC pauses, so enlarging the NVM bandwidth is not useful for GC time reduction. The write cache alone can improve the GC pause time by 1.17× on average (up to 2.08×), mainly thanks to its reduction on NVM writes and bandwidth-friendly NVM access patterns.

Figure 5 also provides the GC pause time for those applications running atop DRAM. The result shows that the average performance gap for GC has shrunk from 4.21× to 2.28×, which is close to the inherent latency gap between DRAM and NVM (2-3×). It suggests that although the NVM devices suffer from shortage and instability in bandwidth, the problems can be resolved with NVM-aware designs.

We also compare our approach with the mechanism where DRAM is used as the young generation for allocation. As illustrated in Figure 5, it outperforms our optimizations for most applications. Using DRAM as allocation regions are beneficial mainly because it introduces more DRAM accesses during GC and mitigates the bandwidth shortage problem for NVM. It is possible to merge this mechanism with our optimizations by using DRAM for both allocation and GC, and we leave this as our future work.

5.3 Bandwidth improvement

We have also evaluated the consumed NVM bandwidth during GC. The number of GC threads is set to 56 threads to saturate the bandwidth. As shown in Figure 6, our NVM-aware optimizations can improve the NVM bandwidth by 55.0% on average. Applications in Spark show relatively larger improvement (69.3%) compared with those in Renaissance since their GC traversal phase lasts longer and involves massive random read and write operations on small objects in Spark RDDs.

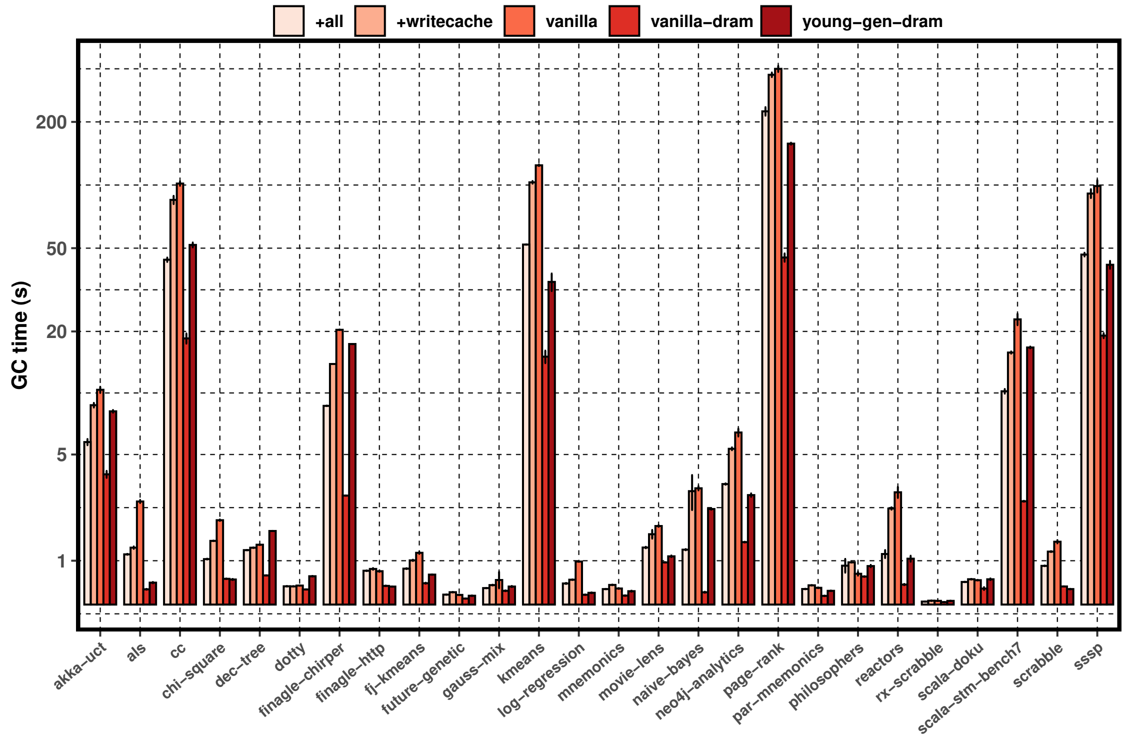


Figure 5. GC time for various applications. The lower the better

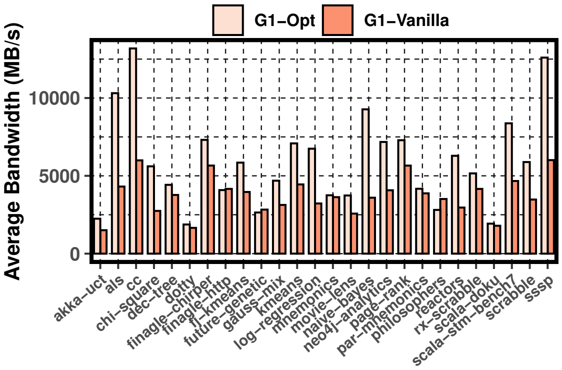


Figure 6. NVM bandwidth during GC

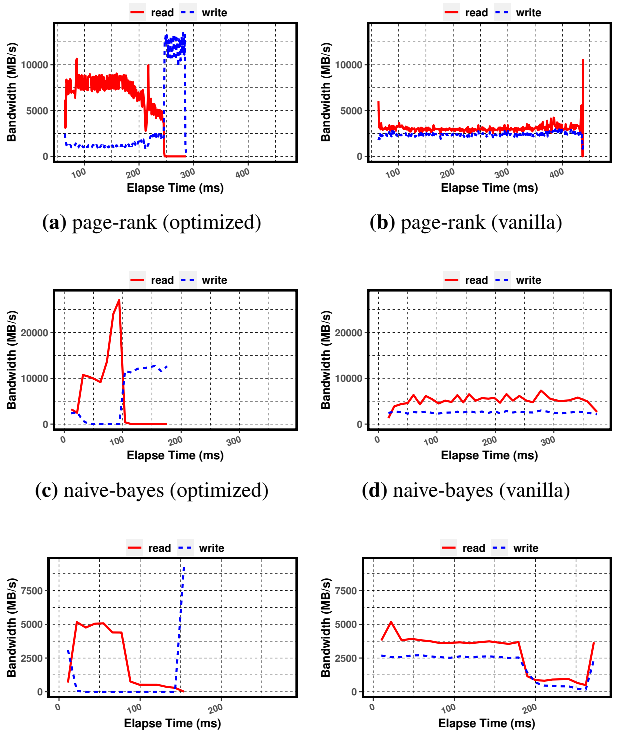
The evaluation confirms that our optimizations enlarge the available NVM bandwidth and improve the GC performance.

We further show the NVM bandwidth of three applications during GC by splitting it into *read-bandwidth* and *write-bandwidth*. The chosen applications have different behaviors during GC and thus exhibit different results on bandwidth improvement. The left part of Figure 7 shows the bandwidth after optimizations while the right part shows the vanilla one.

Page-rank. The read bandwidth of the page-rank benchmark strongly correlates with the write bandwidth. When the write bandwidth decreases, the consumed read bandwidth mostly increases. As shown in Figure 7a, the optimized version reduces the write bandwidth and thus improves the read bandwidth. Furthermore, since the write cache will only be written back to NVM before GC ends, the write bandwidth will reach its peak, which is much larger than that in the vanilla version. As the write-back operations are conducted sequentially with bandwidth-friendly non-temporal instructions, the peak write bandwidth is close to the upper bound evaluated in prior work [24].

Naive-bayes. For the *naive-bayes* benchmark, the write bandwidth in the vanilla version is similar to that in page-rank, but the read bandwidth is larger (Figure 7d). It is because GC in naive-bayes contains many copy operations on primitive arrays, which induces sequential read operations on NVM and sufficiently leverages the available bandwidth. Similarly, the read bandwidth in the optimized version is also improved and can reach as large as 26.5 GB/s (Figure 7c). Furthermore, the write cache is also helpful by introducing sequential NVM writes before GC ends. Since naive-bayes contains many copy operations on primitive arrays, it is a write-intensive application, so the write-only sub-phase is relatively longer compared with page-rank, which is one of the main reasons why the consumed bandwidth is greatly enlarged.

Akka-uct. Compared with the other two applications, the average NVM bandwidth of *akka-uct* is still moderate with our optimizations. This can be explained by excessive random read operations and imbalance workload. As shown in



(a) page-rank (optimized) **(b)** page-rank (vanilla)

(c) naive-bayes (optimized) **(d)** naive-bayes (vanilla)

(e) akka-uct (optimized) **(f)** akka-uct (vanilla)

Figure 7. The split NVM bandwidth during GC for three different applications

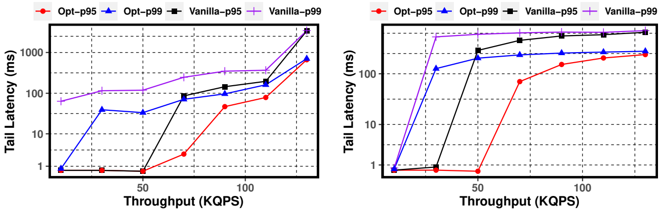
Figure 7e, although the write bandwidth in the first read-mostly phase is close to zero, the read bandwidth is still moderate. However, when the elapsed time exceeds 80 ms, the read bandwidth dramatically decreases. After analyzing the GC behavior of akka-uct, we find that it suffers from load-imbalance, and most GC threads remain idle. Lastly, since the number of live objects in akka-uct is limited, the write-only phase finishes quickly and has little impact on the average bandwidth during GC.

5.4 Application improvement

Figure 9 shows the improvement for applications in Renaissance and Spark. Since the execution time is fixed for Cassandra, we provide a throughput-latency curve for performance analysis.

Renaissance. Most applications in Renaissance show trivial changes in execution time with/without our optimizations since GC occupies an insignificant portion. However, as for several GC-intensive applications, their execution time can be reduced (e.g., scala-stm-bench7).

Spark. The completion time of all Spark applications is reduced thanks to shorter GC pauses. The improvement ranges from 3.2% (cc) to 6.9% (sssp), which mainly relies on the share consumed by GC in the overall execution time.



(a) Read operations **(b)** Write operations

Figure 8. The tail latency reduction for Cassandra

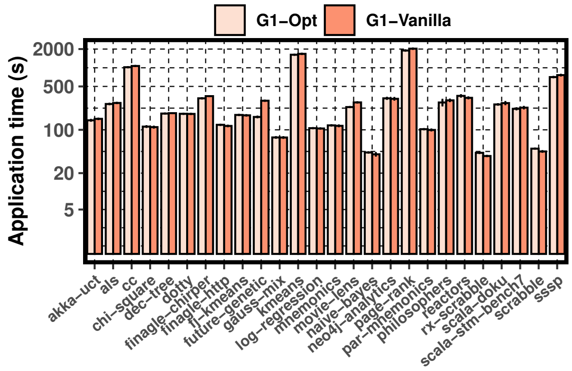


Figure 9. Application time reduction

Cassandra. To evaluate the latency of Cassandra, we run a client JVM in another socket on the same machine and send read and write requests to the server. We vary the throughput to draw throughput-latency curves for read and write operations in Figure 8. The results indicate our NVM-aware GC can improve the p95 and p99 latency for both operations under different throughput settings. In the largest throughput setting (130,000 requests per second), our NVM-aware GC can improve the p95 and p99 read latency by 5.09× and 4.88×. For write operations, the improvement is 2.74× and 2.54× respectively. Since our optimizations effectively reduce GC pauses, they can shorten the worst-case waiting time for requests and thus mitigate the long tail problem.

5.5 DRAM consumption vs. performance

Header maps. We have evaluated the GC performance when varying the maximum size of the header map. As Figure 10 shows, GC pauses can be reduced for most applications when increasing the header map size. With a larger header map, fewer forwarding pointers are installed in NVM, which reduces NVM writes and further enlarges the available read bandwidth for GC threads. However, when increasing the maximum size from 512MB to 2GB, the improvement for Renaissance applications is limited (3.3% on average). This suggests that 512MB is enough to cache forwarding pointers for applications with a 16GB heap. Some applications (such

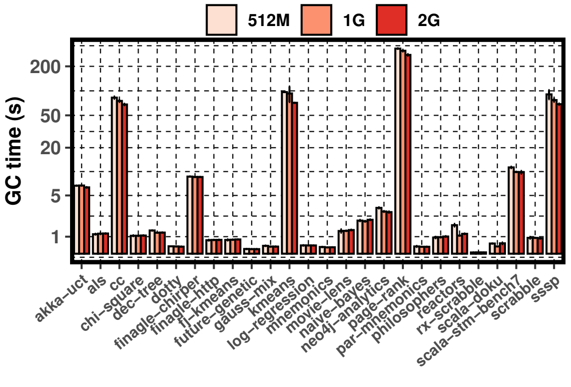


Figure 10. Results with different header map sizes

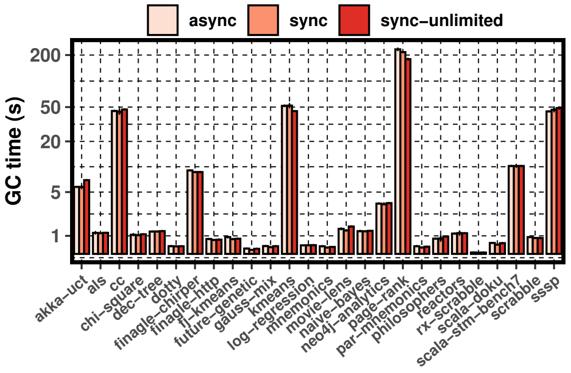


Figure 11. Results with different write cache settings

as naive-bayes) have a low occupancy on the header map even for the 512MB setting, so its DRAM footprint can be further reduced. As for Spark applications, the average improvement is larger (21.1%), and their occupancy on the 2G setting is close to 100%, which suggests that larger header maps are helpful for performance speedup.

Write caches. We evaluate the effect of write cache size by removing its upper bound. Figure 11 shows that most applications do not benefit from an unlimited write cache, which suggests that the default setting (1/32 of the heap) is enough for object caching. Exceptions are page-rank and kmeans in Spark, which copy many small objects during GC. With more DRAM resources, page-rank can be improved by 2.00× for GC time and 11.0% for application time compared with vanilla G1.

Figure 11 also shows the GC performance with asynchronous flushing enabled. Thanks to the non-temporal instructions, asynchronous flushing only induces a 6.9% slowdown on average while timely reclaiming DRAM resources.

Cost-efficiency analysis. We use *GC-improvement-per-dollar* to analyze the cost-efficiency of our optimizations. This metric describes the GC time reduction (in seconds) by adding one dollar to the cost budget. The baseline setting is using NVM to allocate the whole heap, and the per-GB price for evaluated DRAM and NVM devices is 7.81 and 3.01 dollars respectively (2.59×).

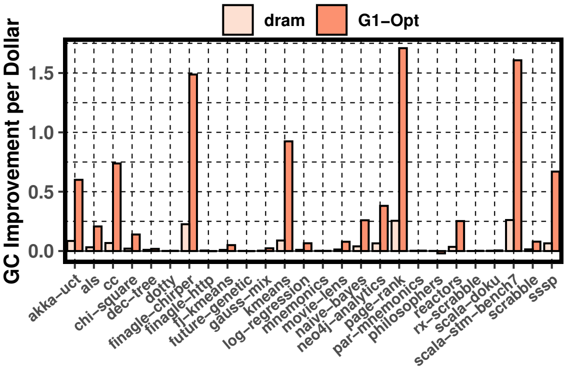


Figure 12. Cost-efficiency analysis

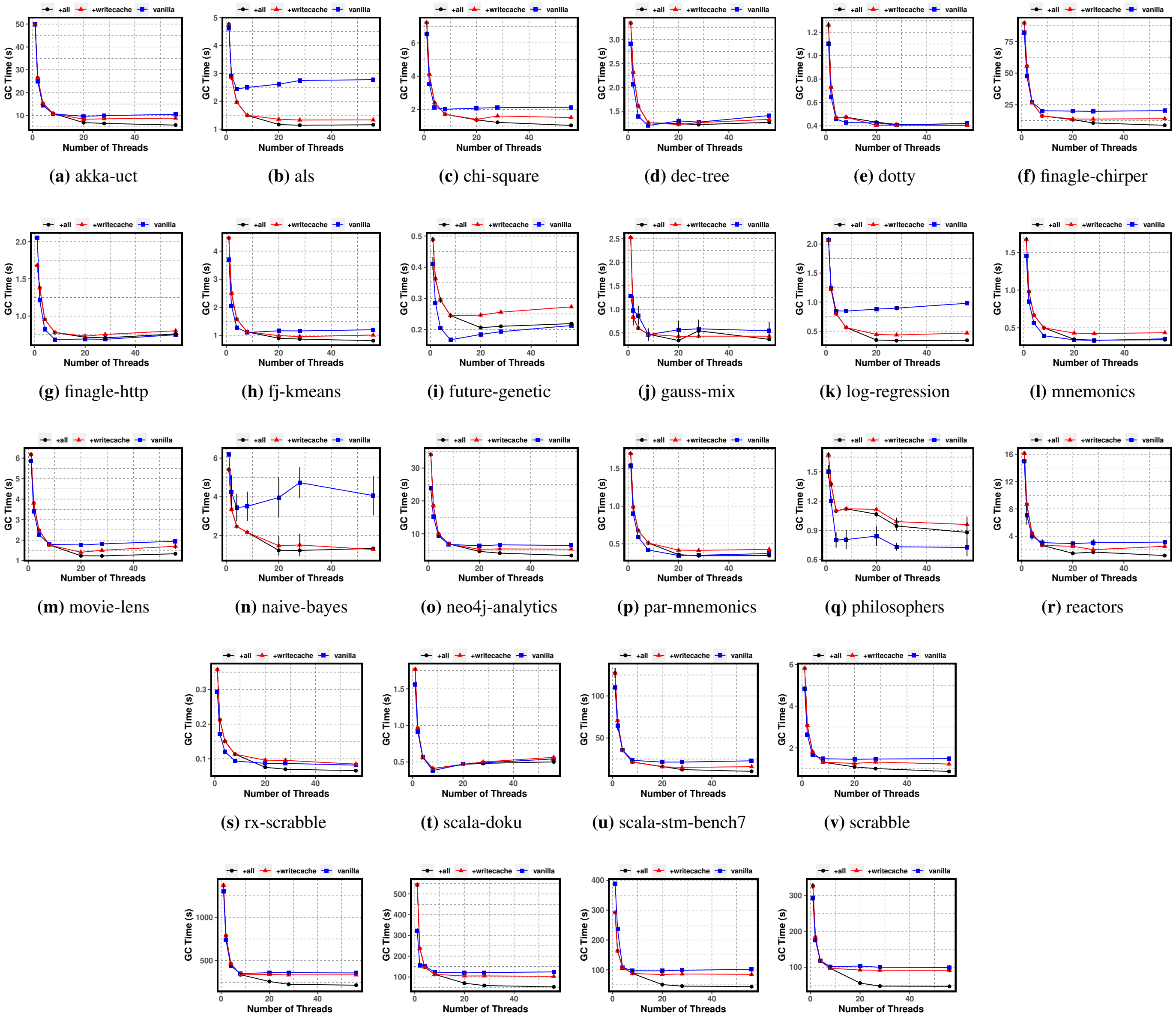
Figure 12 shows the results by comparing our optimizations with directly using DRAM. Although directly using DRAM can provide more performance improvement, our optimizations are more cost-effective for most applications as they improve the GC performance by only introducing a limited amount of expensive DRAM resource. For memory-intensive applications in Spark, the average GC-improvement-per-dollar for our optimizations is 9.58× compared with that when directly using DRAM. Even if considering the application execution improvement introduced by allocating objects from DRAM, our optimizations are still better than directly using DRAM for Spark (1.14× on average).

5.6 Scalability

Figure 13 showcases the accumulated GC time for applications in Renaissance and Spark, with different numbers of GC threads (1, 2, 4, 8, 20, 28, 56). When the number of GC threads is limited (less than 8), the NVM bandwidth is not saturated, and the vanilla G1 performs well. However, it fails to scale with more computing resources. When adding more cores, the accumulated GC time of G1 even increases for some workload due to contentions on NVM bandwidth. After adding the write cache optimization, the performance of G1 becomes better and more scalable as fewer writes occur on NVM. However, most applications can still only scale to 20 cores. With the header map optimization, random write operations are further reduced, and G1 can even scale to 56 logical cores for most applications.

5.7 Improvement on other collectors

We also evaluate the improvement in PS, another copy-based garbage collector. As shown in Figure 14, our optimizations can also improve the performance of PS for applications in Renaissance. The performance speedup ranges from 0.61× (movie-lens) to 2.26× (reactors). Compared with G1, the



(a) akka-uct **(b)** als **(c)** chi-square **(d)** dec-tree **(e)** dotty **(f)** finagle-chirper

(g) finagle-http **(h)** fj-kmeans **(i)** future-genetic **(j)** gauss-mix **(k)** log-regression **(l)** mnemonics

(m) movie-lens **(n)** naive-bayes **(o)** neo4j-analytics **(p)** par-mnemonics **(q)** philosophers **(r)** reactors

(s) rx-scrabble **(t)** scala-doku **(u)** scala-stm-bench7 **(v)** scrabble

(w) page-rank **(x)** kmeans **(y)** cc **(z)** sssp

Figure 13. GC scalability

improvement drops since the object copying in PS is more irregular, and the write cache can absorb fewer NVM write operations. We have also used PS to study the effect of prefetching, and Figure 14 shows that adding prefetch instructions can reduce the GC pause time by 4.8% on average for PS.

6 Related work

6.1 Performance analysis on NVM

NVM devices have drawn great attention once it becomes openly available. Prior work has studied its performance characteristics and guided how to leverage them to implement highly-efficient systems. Izraelevitz et al. [24] and Yang et al. [42] pioneer in measuring the basic performance of Intel Optane DC PM devices. Chen et al. [9] provide performance analysis for key-value stores atop NVM and focus on resolving the mismatch between small random updates and the persistence granularity in NVM. Their solution is batching small write operations into larger ones, which is somewhat similar to our write cache. Wu et al. [40] show that excessive accesses on NVM are harmful to application performance and propose to build snapshots on DRAM. Haria et al. [18] study the performance of the Intel *clwb* instruction, which

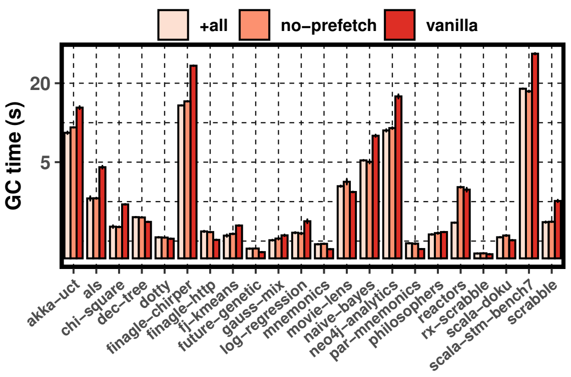


Figure 14. GC time for PS

writes back a cache line into memory subsystems. They reveal that current NVM devices can support no more than 16 concurrent cache line write-back operations. According to this finding, they build Minimally Ordered Data structure (MOD) to hide the flush latency. Hildebrand et al. [20] mainly focus on the bandwidth characteristics of NVM to estimate the data movement overhead between DRAM and NVM. Their work also reports that although asynchronous data movement can overlap data transfer with computation, its performance is discouraging. Our work also studies the performance characteristics of NVM, and some of our findings agree with prior work. Nevertheless, we also share some new findings, including the benefits brought by software prefetching instructions, and that non-temporal instructions are helpful to improve asynchronous data movement.

6.2 NVM support for Java

The thriving NVM technology stimulates studies on supporting NVM in Java. PCJ [21] allows Java programs to access data in NVM managed by native libraries like NVML [23]. Espresso [41] pioneers in directly managing NVM data as Java objects with crash-consistent memory management. AutoPersist [35] proposes a reachability-based persistency model in Java, where objects are automatically copied into NVM with crash-consistency guarantees. GCPersist [40] instead designs a GC-assisted lazy persistency model and integrates the NVM data persistency with copy-based garbage collection. Their work mainly focuses on accelerating Java persistent applications with NVM.

Another line of work uses NVM as a supplement to DRAM to benefits from its large capacity and power efficiency. OpenJDK supports allocating the whole Java heap from alternative devices since JDK 10 [31]. Akram et al. [1, 2] uncover that the NVM devices have limited write endurance and will wear out quickly when running popular Java applications. Therefore, they propose write-rationing GC to store read-mostly objects in NVM devices to extend their lifetime. Panthera [39] places infrequently-used datasets in big-data applications to NVM to improve power efficiency while inducing moderate runtime overhead. Our work also leverages NVM as an alternative memory device but mainly focuses on improving the GC performance according to the bandwidth characteristics of NVM.

6.3 GC optimizations

The performance of GC is crucial to managed runtime, so plenty of work has analyzed its performance and proposed corresponding optimizations. Gidra et al. [14–16] introduce NUMA-aware optimizations into GC to reduce expensive cross-node memory access and improve the GC performance. Suo et al. [37] uncover the mismatch between the JVM synchronization protocol and the OS scheduler. Qian et al. [33] refine the work-stealing algorithm to improve the success rate and load balance. Khanh et al. [29] and Gog et al. [17] propose to use region-based collection mechanisms according to the memory behavior of big-data applications, while Bruno et al. [6–8] embrace pre-tenuring to reduce unnecessary object copying. Lebeck et al. [26] embrace a co-design between GC and page swapping to achieve fast memory reclamation for mobile platforms. Our work is aware of the shortage and instability of NVM bandwidth and proposes techniques to leverage the bandwidth to its fullest.

7 Conclusion

Non-volatile memory (NVM) technology is a good fit for memory-intensive applications due to its byte-addressability and large capacity. This work analyzes the runtime behavior of various memory-intensive Java applications. It uncovers that due to the shortage and instability of NVM bandwidth, the garbage collector (GC) module becomes a serious performance bottleneck when running atop NVM. To this end, this work proposes NVM-aware designs for GC and further accelerate it with off-the-shelf hardware features. The evaluation on memory-intensive applications shows that an NVM-aware garbage collector can reduce the GC pause time by up to 2.69×. The source code is available at https://ipads.se.sjtu.edu.cn:1312/opensource/nvm-friendly-gc.

8 Acknowledgement

We sincerely thank our shepherd Luís Pina and the anonymous reviewers for their insightful suggestions. This work is supported in part by the National Natural Science Foundation of China (No. 61925206). Mingyu Wu is the corresponding author.

References

[1] Shoaib Akram, Jennifer Sartor, Kathryn McKinley, and Lieven Eeckhout. Crystal gazer: Profile-driven write-rationing garbage collection for hybrid memories. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(1):1–27, 2019.

[2] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 62–77. ACM, 2018.

[3] Hassan Al-Sukhni, Ian Bratt, and Daniel A. Connors. Compiler-directed content-aware prefetching for dynamic data structures. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT 2003), 27 September - 1 October 2003, New Orleans, LA, USA*, pages 91–100. IEEE Computer Society, 2003.

[4] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying memory access patterns for prefetching. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 513–526. ACM, 2020.

[5] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 462–473. ACM, 2019.

[6] Rodrigo Bruno and Paulo Ferreira. POLM2: Automatic profiling for object lifetime-aware memory management for hotspot big data applications. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Middleware 2017, Las Vegas, NV, USA, December 11-15, 2017*, pages 147–160. ACM, 2017.

[7] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. 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 2017, Barcelona, Spain, June 18, 2017*, pages 2–13. ACM, 2017.

[8] Rodrigo Bruno, Duarte Patrício, José Simão, Luís Veiga, and Paulo Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 28:1–28:16. ACM, 2019.

[9] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 1077–1091. ACM, 2020.

[10] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 799–812. USENIX Association, 2020.

[11] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 5:1–5:15. ACM, 2020.

[12] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, pages 37–48. ACM, 2004.

[13] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ 2016, Lugano, Switzerland, August 29 - September 2, 2016*, pages 13:1–13:9. ACM, 2016.

[14] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. Assessing the scalability of garbage collectors on many cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems, PLOS@SOSP 2011, Cascais, Portugal, October 23, 2011*, pages 7:1–7:5. ACM, 2011.

[15] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16-20, 2013*, pages 229–240. ACM, 2013.

[16] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. NumaGiC: A garbage collector for big data on big NUMA machines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 661–673. ACM, 2015.

[17] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek Gordon Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association, 2015.

[18] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: Minimally ordered durable datastructures for persistent memory. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 775–788. ACM, 2020.

[19] Eben Hewitt. *Cassandra: the definitive guide*. " O'Reilly Media, Inc.", 2010.

[20] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 875–890. ACM, 2020.

[21] INTEL. Persistent collections for java. https://github.com/pmem/pcj.

[22] INTEL. Intel® optane$^{\text{TM}}$ dc persistent memory. https://www.intel.com/content/www/us/en/architectureandtechnology/optane-dc-persistent-memory.html, 2020.

[23] INTEL. pmem.io: Persistent memory programming. http://pmem.io/, 2020.

[24] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[25] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 494–508. ACM, 2019.

[26] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 873–887. USENIX Association, 2020.

[27] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 462–477. ACM, 2019.

[28] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, August 5-8, 1984, Austin, Texas, USA*, pages 235–246. ACM, 1984.

[29] Khanh Nguyen, Lu Fang, Guoqing (Harry) Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *12th USENIX Symposium on*

Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016, pages 349–365. USENIX Association, 2016.

[30] OpenJDK. Jep 248: Make g1 the default garbage collector, 2020.

[31] OpenJDK. Jep 316: Heap allocation on alternative memory devices, 2020.

[32] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 31–47. ACM, 2019.

[33] Junjie Qian, Witawas Srisa-an, Du Li, Hong Jiang, Sharad C. Seth, and Yaodong Yang. Smartstealing: Analysis and optimization of work stealing in parallel garbage collection for java VM. In *Proceedings of the Principles and Practices of Programming on The Java Platform, PPPJ 2015, Melbourne, FL, USA, September 8-11, 2015*, pages 170–181. ACM, 2015.

[34] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 24–33. ACM, 2009.

[35] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: An easy-to-use java NVM framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 316–332. ACM, 2019.

[36] Seung Woo Son, Mahmut T. Kandemir, Mustafa Karaköy, and Dhruva R. Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 209–218. ACM, 2009.

[37] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 35:1–35:15. ACM, 2018.

[38] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*, pages 157–167. ACM, 1984.

[39] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 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 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 347–362. ACM, 2019.

[40] Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, and Haibing Guan. GCPersist: An efficient gc-assisted lazy persistency framework for resilient java applications on NVM. In *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, pages 1–14. ACM, 2020.

[41] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 70–83. ACM, 2018.

[42] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.

[43] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012.

[44] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, page 95. USENIX Association, 2010.

[45] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006), 26-29 March 2006, New York, New York, USA*, pages 50–64. IEEE Computer Society, 2006.