# Write-Rationing Garbage Collection for Hybrid Memories

Shoaib Akram
Ghent University
Belgium

Jennifer B. Sartor
Vrije Universiteit Brussel and Ghent University
Belgium

Kathryn S. McKinley
Google
United States of America

Lieven Eeckhout
Ghent University
Belgium

## Abstract

Emerging Non-Volatile Memory (NVM) technologies offer high capacity and energy efficiency compared to DRAM, but suffer from limited write endurance and longer latencies. Prior work seeks the best of both technologies by combining DRAM and NVM in *hybrid memories* to attain low latency, high capacity, energy efficiency, and durability. Coarse-grained hardware and OS optimizations then spread writes out (*wear-leveling*) and place highly mutated pages in DRAM to extend NVM lifetimes. Unfortunately even with these coarse-grained methods, popular Java applications exact impractical NVM lifetimes of 4 years or less.

This paper shows how to make hybrid memories practical, without changing the programming model, by enhancing garbage collection in managed language runtimes. We find object write behaviors offer two opportunities: (1) 70% of writes occur to newly allocated objects, and (2) 2% of objects capture 81% of writes to mature objects. We introduce *write-rationing garbage collectors* that exploit these *fine-grained* behaviors. They extend NVM lifetimes by placing highly mutated objects in DRAM and read-mostly objects in NVM. We implement two such systems. (1) Kingsguard-nursery places new allocation in DRAM and survivors in NVM, reducing NVM writes by 5× versus NVM only with wear-leveling. (2) Kingsguard-writers (KG-W) places nursery objects in DRAM and survivors in a DRAM observer space. It monitors all mature object writes and moves unwritten mature objects from DRAM to NVM. Because most mature objects are unwritten, KG-W exploits NVM capacity while increasing NVM lifetimes by 11×. It reduces the energy-delay product by 32% over DRAM-only and 29% over NVM-only. This work opens up new avenues for making hybrid memories practical.

***CCS Concepts*** • **Information systems → Phase change memory**; • **Computer systems organization → Architectures**; • **Hardware → Non-volatile memory**; • **Software and its engineering → Garbage collection**;

***Keywords*** Managed runtimes, Garbage collection, Non-volatile memory (NVM), Hybrid DRAM-NVM memories

**ACM Reference Format:**
Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-Rationing Garbage Collection for Hybrid Memories. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192392

## 1 Introduction

Increasing hardware core counts and working-set sizes of applications places large demands on main memory capacity. At the same time, DRAM scaling has slowed [32, 34], motivating researchers to explore Non-Volatile Memory (NVM) technologies [26, 27, 42]. Our work focuses on Phase Change Memory (PCM) [24, 26, 27, 42], which offers five advantages: byte-addressability, high density, scalability (capacity), low standby power, and non-volatility, but four shortcomings: high access latency, write latencies exceed read latencies, high write energy, and low write endurance. Although improvements in PCM manufacturing technology are reducing latency [21, 36], it comes at the expense of write energy and endurance (lifetime). Endurance is the biggest challenge because each write changes the material form [12] and has thus far prevented NVM uptake.

Prototype PCM hardware has an endurance of 1 million (M) to 100 M writes [5, 26]. This wide range results from: (1) the tradeoff between write speed and endurance, and (2) the properties of PCM materials. Prior architecture, operating system (OS), and programming language optimizations redirect and eliminate writes to improve lifetimes [11, 17, 26, 27, 42, 43, 50]. In particular, hybrid memories combine DRAM and PCM technology, seeking the best of both approaches [26, 27, 42]: (1) DRAM hides the high access latency of PCM by buffering frequently accessed pages, and (2)

*PLDI'18, June 18–22, 2018, Philadelphia, PA, USA*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5698-5/18/06...$15.00
https://doi.org/10.1145/3192366.3192392

hardware and OS diffuse PCM writes with *wear-leveling* and reduce writes with page migration [11, 26, 27, 29, 42, 43, 50]. Wear-leveling moves pages and lines in pages to distribute writes uniformly. Page migration reactively places highly mutated pages in DRAM and read-mostly pages in PCM.

An open question for NVM memory systems is if modern applications can use NVM directly or if they require changes to runtimes and their programming models. Figure 1 explores this question with measurements of PCM lifetimes when executing Java applications. (Section 5 describes our experimental methodology.) These lifetimes motivate hybrid memories and some software support. The figure presents average lifetimes of a 32 GB PCM-only system for three different PCM endurance levels reported and used in prior work [12, 26, 29, 38, 38, 39, 42]. A 32-core PCM-only system with 32 GB of main memory and an endurance of 30 M writes per cell would wear out in 4 years, even with line write-back and wear-leveling [38, 39, 42]. Because lifetime is a linear function of writes, increasing endurance to 100 M per cell would improve lifetime to 13 years. However, running the Java application with the highest write rate would wear out a 32 GB PCM memory in less than 5 years. With current PCM endurance levels, a pure PCM memory system is thus impractical. For a 32 GB PCM-only system to last 15 years across a range of applications and endurance levels, write rates need to reduce by at least an order of magnitude.

This paper shows that specializing the Java runtime system results in a promising and practical approach for using hybrid memories. Limiting changes to the runtime system, rather than requiring changes to programming models and applications, will ease adoption of hybrid memories.

We introduce the design and implementation of a new class of *write-rationing* garbage collectors that reorganize objects to limit writes to PCM in hybrid main memories, while still utilizing PCM capacity. We exploit the managed runtime implementation for languages such as Java, C#, JavaScript, and Python. Prior work either manages coarse-grained pages or allocation sites in C++ programs [50, 55] or profiles allocation sites ahead-of-time in a Java managed runtime, optimizing performance, but not liftetime in hybrid memories [49]. In contrast, write-rationing collectors move and monitor individual objects in managed runtimes.

A detailed analysis of write behaviors in Java applications motivates our work. Figure 2 presents writes in an instrumented generational collector as a function of object age: young (nursery) versus mature space objects. 26% to 99% of writes occur to nursery objects, averaging 70%. The results are consistent with prior measurements [44, 56] and exhibit a wide range of behaviors. 2% of mature objects capture 81% mature object writes. 94% of all writes fall in one of these two categories. Our designs exploit the correlation between writes and object demographics (age) and detect the small fraction of mature objects that incurs most writes.

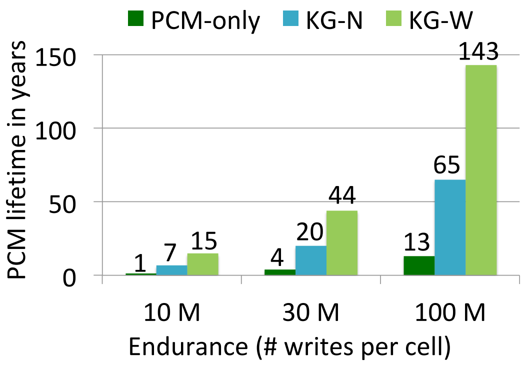


**Figure 1.** PCM-only is impractical. 32 GB lasts only 4 years on average with 30 M writes per cell and hardware line wear-leveling in simulation. The proposed KG-N and KG-W write-rationing garbage collectors manage DRAM and PCM, extending PCM's lifetime to practical levels.

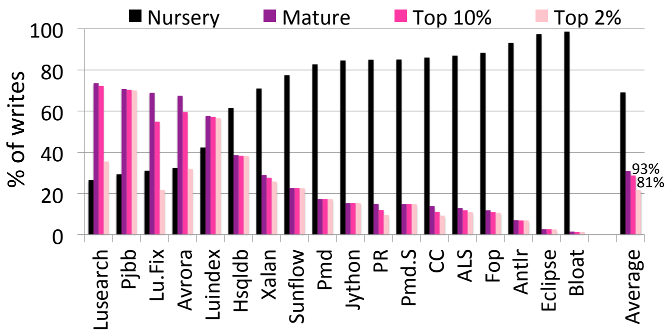


**Figure 2.** Nursery objects incur 70% of writes and mature objects incur 30% on average. The top 10% of written mature objects incur 93% of mature writes and the top 2% incur 81%.

We introduce two write-rationing collectors that *guard* (i.e., Kingsguard) PCM from writes. They organize heap memory in DRAM and PCM virtual memory, directing the OS explicitly. The *Kingsguard-nursery* (KG-N) collector allocates new objects in a DRAM nursery, since they incur between 25% and 98% of writes, and then promotes all nursery survivors to a PCM mature space.

*Kingsguard-writers* (KG-W) adds fine-grained monitoring and per-object placement of mature objects. It also uses a DRAM nursery, but promotes nursery survivors to a DRAM *observer* space. The Java Virtual Machine (JVM) tracks all mature object writes with a write-barrier [14, 16, 46, 52]. Observer space collections copy objects with zero writes from the observer space to the PCM mature space and copy any written objects to the DRAM mature space, using past writes to predict future writes. Kingsguard-writers promotes most observer space survivors (90%) to PCM memory, thus exploiting its capacity. When it detects written objects in PCM, it moves them back to DRAM. KG-W also includes optimizations for large objects.

Because hybrid memory systems are not available, we use cycle-level multicore simulation and hardware measurements for evaluation. We find KG-N and KG-W improve PCM lifetime by 5× and 11×, respectively on our simulated Java applications. KG-W needs 16 MB of DRAM to achieve these lifetimes. We compare to state-of-the-art OS write partitioning (WP) [55]. WP consumes about the same amount of DRAM, but has 3× more writes to PCM than KG-W. Even though memory accesses to PCM are slower, Kingsguard-nursery reduces the energy-delay product by 36% over a DRAM-only system and 33% over a PCM-only system. Kingsguard-writers adds 5% average overhead to monitor nursery survivors in the observer space and to copy objects between spaces. Kingsguard-writers reduces the energy-delay product by 32% on average over DRAM-only and by 29% over a PCM-only system. Compared to Kingsguard-nursery, Kingsguard-writers thus trades some overhead and DRAM to significantly improve PCM lifetime.

In summary, the contributions of this paper are:

- an empirical characterization of Java applications that motivates hybrid memories and fine-grained object placement;
- the design and implementation of write-rationing garbage collectors that manage hybrid memories, minimizing PCM writes while maximizing the use of their capacity;
- Kingsguard collectors that explicitly allocate and move objects into DRAM and PCM heap spaces based on their demographics and write behavior;
- execution and simulation results that show these collectors exploit PCM capacity while substantially improving PCM lifetimes and energy by reducing writes as compared to prior OS and hardware approaches; and
- a practical approach to exploiting hybrid memories that requires no new OS or hardware support.

## 2 Related Work

This section discusses related work for PCM memory hardware, OS, and memory management.

***Hardware and OS support for PCM.*** The two predominant approaches for improving PCM lifetime are making writes more uniform over the PCM capacity, called *wear-leveling*, and reducing the number of writes. Wear-leveling of pages allocates and moves pages to distribute writes uniformly through memory. Wear-leveling of lines within a page remaps lines to distribute writes uniformly on each page. Prior work proposes a number of wear-leveling approaches [40, 41, 45]. We use line wear-leveling from Qureshi et al. [42] as our baseline hardware.

Prior work proposes hardware and OS techniques for hybrid DRAM-PCM memories that monitor and move pages to reduce PCM writes [11, 17, 26, 27, 29, 42, 43, 50]. Their two main drawbacks are (1) they are reactive, and (2) they work at the page granularity. None of these systems considers reorganizing objects on pages to create pages of read-mostly objects and pages of mutated objects, as we do.

We implement OS Write Partition (WP) by Zhang et al. [29, 43, 55] and find that our write-rationing garbage collectors decrease writes by 3 × more than WP (see Section 6.1.3). WP treats DRAM as a partition for highly mutated pages, which it identifies using a ranking scheme. The ranking scheme is a variation of the widely used Multi Queue algorithm for managing OS buffer caches [57]. The OS places a new page in PCM first. The memory controller then counts writes to each physical page and tracks pages in queues ordered by power of 2 writes. When a page incurs a threshold number of writes, the memory controller promotes that page to a higher ranked queue: at $2^n$ writes, the OS promotes the page to the queue with rank $n$. The OS periodically migrates pages in the highest-ranked queues to DRAM. Subsequent work builds upon WP, adding performance optimizations, but does not change lifetime [29, 43]. Since write-rationing collectors optimize lifetime, we compare to WP and find our approach incurs substantially few PCM writes.

***Memory management and garbage collection.*** The closest related work also uses the managed runtime, but optimizes for performance in hybrid memories, as opposed to lifetime [49]. It performs an offline profiling phase to identify object allocation sites for cold (rarely read or written) and hot old objects. It places all nursery objects in DRAM. It promotes nursery survivors according to their tag, moving hot objects to DRAM and cold ones to PCM. Our work optimizes for a different goal — PCM lifetime. Our work requires no ahead of time profiling, which suffers when inputs do not match the profile. It dynamically monitors individual object writes in the observer space to manage writes to PCM.

A similar offline-profiling approach for C programs finds allocation sites that produce highly mutated memory and ones that produce read-mostly memory, modifying allocations sites to specify PCM or DRAM [50]. C semantics limit this approach because objects cannot move. Our work exploits managed language semantics to monitor and move objects, making fine-grained per-object decisions and correcting them if need be, regardless of allocation site.

To tolerate PCM line failures in a page in managed languages, Gao et al. introduce new hardware and OS approaches to mask defective lines, but when lines fail, they expose defective lines in page maps to the garbage collector [17]. The OS provides this map to the runtime and copies of data during a runtime failure, which prevents using or losing failed lines by the collector. They did not consider hybrid memories.

Prior work observes that nursery collections leave behind written cache lines full of dead objects and propose cache scrubbing instructions that mark these lines as dead after a nursery collection, preventing writes to DRAM [44]. Our approach is complementary. It exploits this observation to protect PCM from writes of highly mutated nursery objects.

Complementary approaches also include dividing the heap into hot and cold regions to better manage DRAM energy consumption [22], and data structure-aware heap partitioning to improve lifetimes, locality, and region-based memory management [35].

## 3 Background

This section provides background on the Immix mark-region generational garbage collector (GenImmix) [10] on which we build, and how large objects and metadata are managed.

***Immix: a generational mark-region collector.*** We modify GenImmix, the default best-performing collector in Jikes [10], to create Kingsguard collectors. GenImmix uses a copying nursery and a *mark-region* mature space. The mark-region mature space consists of a hierarchy of two regions: blocks and lines. Blocks are multiples of page sizes and consist of multiple lines. Lines are multiples of cache line sizes. Objects may cross lines, but not blocks. Bump pointer object allocation is contiguous in the nursery, in lines, and blocks. (Contiguous allocation is known to outperform free-list allocators due to its locality benefits [6, 10, 20].) Filling the nursery triggers a collection, which copies nursery survivors contiguously into free lines within blocks in the mature space. Filling the mature space triggers a full heap collection. Immix reclaims the mature space at a line and block granularity by marking lines and blocks live as it traces and marks live objects. Subsequent mature allocation bump-point allocates first into contiguous free lines in partially free blocks and then into completely free blocks. Allocation and reclamation use per-thread allocators and work queues to deliver concurrency and scalability. The per-thread allocators obtain blocks (partially and completely free) from a global allocator.

We use the default settings for Immix, including the maximum object size (8 KB), line size (256 bytes), and block size (32 KB). These settings match the Immix line size to the PCM line size. Immix tailors the heap representation to match the hardware memory system for performance, but it also matches the needs of PCM memory management for detecting and tolerating line failures, as Gao et al. [17] show.

***Large objects.*** Jikes RVM manages objects larger than the 8 KB threshold separately, allocating them directly into a non-copying large object space, and uses a *treadmill* to avoid copying them [19, 23]. A treadmill consists of two doubly-linked lists that store all references to large objects. During collection, live traced references are removed from one doubly-linked list and snapped to another. The collector then reclaims the large objects reachable from any unsnapped references. The cost of using a treadmill is high due to storing all the references to each large object which is only justified because it eliminates marking large objects.

***Object metadata.*** In addition to application writes, the JVM and collector also generate writes. In particular, they write object metadata during allocation and collection. Object metadata includes an object's type, layout, and liveness. The liveness information is often stored in a header word next to the object. Garbage collectors write to metadata when they mark objects live and they write to objects directly when they update their references after copying objects. When GenImmix marks an object live, it also writes block and line bits, stored separately from the object. Because marking live mature objects in PCM generates a lot of PCM writes when liveness is stored in the object header, the KG-W write-rationing collector includes an optimization that eliminates these writes by storing object liveness metadata in DRAM space separate from the objects.

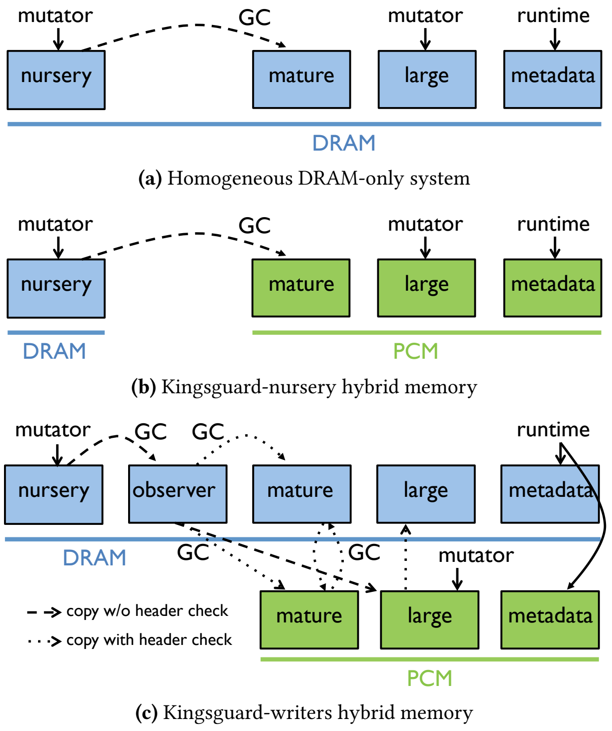


**Figure 3.** Main memory heap organizations (not to scale).

## 4 Write-Rationing Garbage Collection

This section presents the design of write-rationing garbage collectors that seek (1) all the performance advantages of high-performance garbage collection, (2) to maximize the use of PCM for its scalability properties, and (3) to limit writes to PCM to extend its lifetime. These collectors limit write traffic to PCM significantly compared to PCM-only memory by judiciously placing highly mutated objects in DRAM.

Our baseline memory systems contain DRAM-only or PCM-only memory with generational Immix (GenImmix) collector (see Section 3). Figure 3(a) illustrates this baseline generational heap organization with DRAM-only (and PCM-only) memory The mutator allocates objects into the nursery and large object space. The collector copies surviving nursery objects to the mark-region mature space. The JVM uses a metadata space. Mature and large object space collection is non-moving (not shown).

## 4.1 Kingsguard-nursery for Hybrid Memory

As motivated by the data in Figure 2, a promising strategy for limiting writes to PCM in hybrid memories is placing nursery objects in DRAM and all other objects in PCM. We call this collector Kingsguard-nursery (KG-N). It also puts nursery survivors (mature), large objects, and JVM metadata in PCM memory. Figure 3(b) illustrates how Kingsguard-nursery maps the baseline heap organization onto hybrid memory. The system requests DRAM memory from the OS for the nursery, where objects are freshly allocated, and requests PCM memory from the OS for everything else. Requests to the OS are at the page granularity (4 KB). This heap organization is appealing because it maximizes the number of objects that reside in PCM and it requires minimal changes to the VM and garbage collector.

Compared to a PCM-only system, Kingsguard-nursery eliminates the nursery writes to PCM (70% of all writes), saves energy, and increases PCM's lifetime. Because applications also write to mature objects, many applications will still wear out PCM with this approach.

## 4.2 Kingsguard-writers for Hybrid Memory

Kingsguard-writers (KG-W) adds new heap regions and mechanisms to further limit writes to PCM by monitoring and placing individual objects in DRAM or PCM. Kingsguard-writers also allocates all new objects in a DRAM nursery. In addition, it creates a new DRAM *observer* region for nursery survivor objects where it monitors their writes. It then chooses on an individual object basis to put mutated objects in a mature DRAM space and unwritten objects in a mature PCM memory space. Of course these objects can have subsequent writes, so we call them *read-mostly* objects. KG-W has two large object spaces: one in DRAM and one in PCM. KG-W initially places large objects in the nursery if space is available, since a surprising number die quickly, or otherwise allocates them directly to PCM memory. KG-W monitors small and large objects in PCM, and when it detects written objects, it moves them to their corresponding space in DRAM during the next collection. Figure 3(c) illustrates Kingsguard-writers' heap organization.

### 4.2.1 The Observer Space

Rather than monitoring all objects for writes, we restrict object monitoring to mature objects that survive at least one nursery collection. Because nursery objects are rapidly mutated, monitoring them would incur high overhead. Even when many survive, zeroing, initialization, and data structure creation produce a flurry of writes. Table 4 reports nursery survival rates of 17 % on average, as low as 0.001 %, and as high as 66%.

KG-W adds a new DRAM generation for all nursery survivors, called the *observer* space. While objects reside in the observer space, KG-W monitors all writes and marks a bit when objects are written. The observer space is a contiguous region and uses bump-pointer allocation. We make the observer space twice as large as the nursery and trigger an observer space collection when it is full. An observer collection thus results in pause times longer than nursery collections, but shorter than full heap collections. An observer collection moves live unwritten objects to the PCM mature space and live written objects to the DRAM mature space.

The observer space achieves two goals. (1) It gives objects more time to die, so fewer objects are even candidates for PCM memory. (2) If the system detects a write to an object while it resides in the observer space, the collector never moves it to PCM, because it uses writes to objects in the observer space as a predictor of future writes.

KG-W copies all surviving nursery objects to the observer space instead of the mature space and collects the observer space more frequently than the mature space in the baseline system. KG-W reserves a small amount of room in the observer space into which it copies surviving nursery objects during an observer space collection. It sizes this room using recent nursery survival rates. Some surviving objects will therefore die soon afterward in the observer space. Kingsguard-nursery would copy these objects to the PCM mature space, which creates dirty dead cache lines that are likely to be written back once the mutator resumes execution. Kingsguard-writers avoids these useless writes to PCM by using the observer space to avoid tenuring garbage.

In Hotspot [14, 37], a survivor space in the young generation serves a similar purpose of giving objects more time to die, but objects reside in this space only until the next nursery collection. Other collection strategies also seek to avoid tenured garbage [47, 48, 51]. All these approaches are orthogonal to our work.

### 4.2.2 Monitoring and Write Barriers

An *observer collection* includes the nursery and observer space, in isolation of other spaces, to make timely promotions from the observer space to PCM. To ease the recording of references into the nursery and observer space required to collect them independently of the mature space, we colocate the nursery and observer space on the same side of a virtual address space boundary. We thus can and do use the same fast boundary write-barrier as used by a standard generational collector. Figure 4 shows our modified reference write-barrier in Jikes RVM.

In generational collections, pointers from outside the nursery into the nursery are added to the root set for nursery collections. Similarly in KG-W, pointers from outside the nursery and observer spaces into those spaces are added to the root set for observer collections. Lines 7 to 12 in Figure 4 shows the code KG-W executes for remembering the pointers from outside the nursery and observer spaces. This part of the write-barrier is the same as a standard generational write-barrier. The later part of the barrier monitors reference and primitive writes to objects.

The main benefit of the observer space is to monitor writes to objects and use their behavior to determine on a fine-grained object level whether to put an object in the DRAM or in the PCM mature space. Our analysis shows that 81% of writes to non-nursery objects happen to 2% of objects, as shown in Figure 2. Objects written a number of times are, therefore, likely to be written again.

```
1  @Inline
2  public final void objectReferenceWrite(
3    ObjectReference src,
4    Address slot,
5    ObjectReference tgt)
6  {
7    if(!inNursery(slot) && inNursery(tgt)) {
8      remset.insert(slot);
9    }
10   if(!inNurseryOrObservers(slot) &&
         inNurseryOrObservers(tgt)) {
11     remset_observers.insert(slot);
12   }
13   if(!inNursery(src)) {
14     Object o = ObjectReference.fromObject(src);
15     Address a = o.toAddress();
16     a.store(Word.one(), EXTRA_WORD_OFFSET);
17   }
18   Magic.setObjectAtOffset(src, slot, target);
19 }
```

**Figure 4.** Our modified reference write-barrier. Lines 10 to 17 show the extra code KG-W executes on each reference write.

To monitor observer space writes, we use the write-barriers on (1) references and (2) primitives that are provided by MMTk [6]. All of our systems must monitor references (pointers) to collect the nursery and observer spaces independently, so additionally monitoring writes to references in the observer space incurs little additional overhead. On the other hand, monitoring primitives, writes to all other values, has higher overhead because primitive writes are more common than reference writes and are not necessary for collecting independent regions correctly. Reference writes, often, but not always, predict primitive writes. We show the performance-accuracy tradeoff of using the two types of barriers in Section 6.2.

We modify the generational write-barrier to monitor writes to references and primitive fields outside the nursery space. To remember written objects, we add a *write word* in each object header. When the program writes an object in the observer space, the write-barrier sets a bit in this header word as shown in lines 13 to 17 in Figure 4. We add a header word because GenImmix uses all the existing object header bits [10]. A careful re-design or disabling some Immix features, such as pinning, could steal a bit instead. Since we have an entire word, the barrier could record the number of writes. We leave reducing this extra header space and counting writes for future work. We use one of the remaining extra header bits for the metadata optimization presented below.

### 4.2.3 Mature DRAM and Mature PCM Spaces

During an observer collection, Kingsguard-writers checks the write bit and then copies all written objects from the observer space into the *mature DRAM* space, and the remaining objects to the *mature PCM* space. Figure 3(c) shows this selective copying with dotted lines.

KG-W monitors all writes in the mature DRAM and mature PCM spaces. During whole-heap collections, KG-W moves objects in the mature DRAM space whose write bit is zero, and we therefore predict will not be written, to the mature PCM space. This step adds copying work to more fully exploit the capacity of PCM. Similarly, when KG-W detects a written object in mature PCM, it copies the object to the mature DRAM space, and resets its write bit to zero. This step adds copying work to limit future writes (predicted by the past writes) to PCM by this object.

We trigger a full heap collection when the combined mature DRAM and mature PCM spaces run out of space, based on the heap size used. Kingsguard-writers does not limit the amount of DRAM space. However, because of the high mortality rates in both the nursery and observer spaces, very few objects are put into the mature DRAM space, which is between 26 MB and 40 MB for our applications. By design, the PCM portion of the mature generation contains objects that are likely long-lived and infrequently written. For applications that mostly create small objects, mature PCM is the largest portion of the heap. We discuss the amount of DRAM and PCM used per benchmark in Section 6.

### 4.2.4 Large Object Optimization (LOO)

Most collectors manage large objects separately because they are costly to copy. In hybrid memories, if they are frequently written, they are also costly in writes, degrading PCM lifetime. We achieve the best of both worlds by creating a large object space in DRAM and in PCM. To exploit the capacity of PCM, KG-W initially puts large objects directly in the large PCM space. If the mutator writes to large objects when they are in the large PCM space, we move them to the large DRAM space during the next major collection. We modify the non-moving treadmill data structure used for large objects to handle moving objects. When copying from large PCM to large DRAM, objects are unsnapped from the former treadmill's linked list and snapped on to the latter space's treadmill. Because copying large objects incurs a high overhead, once a large object is copied to DRAM, we never move it back to PCM.

We find empirically that large objects often follow the weak-generational hypothesis, i.e., they die quickly. Therefore we perform a dynamic optimization (LOO) to place some large objects in the nursery first to give them a chance to die and to avoid allocating large objects that die quickly in PCM. If a large object survives a nursery collection, we copy it to the observer space. If it survives an observer collection, KG-W copies it directly to the large PCM space, without consulting the write bit, to leverage the capacity of PCM.

Allocating large objects in the nursery should be done with caution, to ensure space for small objects. KG-W dynamically monitors the allocation rate to choose whether to devote part of the nursery to large objects or not. If at the end of a nursery collection, the allocation rate in the large PCM space is faster than the nursery's allocation rate, then we enable this optimization. The allocator allocates large objects less than half of the remaining nursery size in the nursery, and otherwise allocates them in the large PCM space. This technique gives large objects time to die and for arrays of references, gives any referent objects time to die as well. Section 6 shows this optimization saves a lot of allocation and writes to PCM, thus improving its lifetime. This optimization trades some copying overhead to limit writes to the large PCM space.

#### 4.2.5 Metadata Optimization (MDO)

As mentioned in Section 3, garbage collectors require metadata to track object liveness. GenImmix stores the mark state of objects in their headers, which would result in writes to all live mature objects in PCM memory when collecting the whole heap. Updating a single byte in the header of each live object in PCM would result in writing back one cache line for every live object in PCM on every major collection. KG-W thus performs an optimization (MDO) to decouple the mark state metadata from the PCM object.

KG-W stores the mark states of objects in mature PCM in a separate metadata region in DRAM (shown in Figure 3(c)). Using the Immix allocator, the PCM mature space reserves new space 4 MB at a time. When this happens, KG-W allocates a table in DRAM for the mark state of objects in that 4 MB region. The table size depends on the number of objects that fit in 4 MB. Object sizes vary from 4 bytes (a header with no payload) to 8 KB (the biggest small object). Accounting for the smallest object would incur a 25% DRAM overhead. We empirically find that most objects are larger than 16 bytes. We therefore reserve 262 KB for the mark state table for each 4 MB region of PCM, incurring a 6.25% overhead for storing the mark states separately. We use this table for all objects over 16 bytes. For objects 16 bytes and smaller, we mark them small in the write word in the object's header and use the normal mark bit in the header instead of the mark state table. For fast access, we store the address of the mark state table at the beginning of each 4 MB PCM region. To calculate the mark state address of a PCM object, we add the object offset in the 4 MB region to the starting address of the table. When the collector frees a 4 MB region in PCM, it also frees the DRAM space reserved for the mark state table. Section 6 shows that this optimization reduces the collector's writes to PCM for highly allocating applications.

**Table 1.** Collector configurations. Large Object nursery allocation (LOO) and PCM metadata in DRAM (MDO) are dropped in two configurations.

| Configurations | monitor writes | metadata in DRAM | LOO in nursery |
|---|---|---|---|
| KG-N: Kingsguard-nursery | ✗ | ✗ | ✗ |
| KG-W: Kingsguard-writers | ✓ | ✓ | ✓ |
| KG-W–LOO | ✓ | ✓ | ✗ |
| KG-W–LOO–MDO | ✓ | ✗ | ✗ |

## 5 Experimental Methodology

This section describes our experimental methodology, including our JVM, applications, collector configurations, hardware, architectural simulator, and memory models.

### 5.1 Software

***Java Virtual Machine.*** We implement our collectors in Jikes RVM 3.1.2 [2, 3] which combines good performance and software engineering that make it easy to modify [2, 3, 7, 16]. Jikes RVM is a Java-in-Java VM with a baseline compiler (no interpreter), just-in-time optimizing compiler of hot code, and a large number of state-of-the art garbage collectors [6, 10, 46]. It offers a wide range of easy-to-change reference-barriers [52], and a clean interface between the compiler and collector [16], which defines object layout, references, interior references, and object metadata in a few places.

We follow best practices for Java performance evaluation [18, 20]. We use replay compilation to eliminate nondeterminism introduced by just-in-time compilation. During profiling runs, the VM records a plan with the optimization level of each method for runs with the shortest execution time. We then run each benchmark for two iterations. In the first unmeasured iteration, the JIT compiler applies the pre-recorded optimization plan to each method. We then measure the second iteration, which excludes compile time and represents application steady-state behavior. We report the averages of 8 runs for our real hardware experiments and 4 for our simulation experiments. We compute detailed statistics in separate experiments, different from the performance runs.

***Garbage collectors and configurations.*** We compare to Jikes RVM's default stop-the-world generational Immix collector [10], with DRAM-only and PCM-only heaps. We use the default settings for maximum object size (8 KB), line size

**Table 2.** Simulated system parameters.

| Component | Parameters |
|---|---|
| Processor | 1 socket, 4 cores |
| Core | 4-way issue, 4.0 GHz, 128-entry ROB |
| Branch predictor | hybrid local/global predictor |
| Max. outstanding | 48 loads, 32 stores, 10 L1-D misses |
| L1-I | 32 KB, 4 way, 4 cycle access time |
| L1-D | 32 KB, 8 way, 4 cycle access time |
| L2 cache | 256 KB per core, 8 way, 8 cycle |
| L3 cache | shared 4 MB, 16 way, 30 cycle |
| Coherence protocol | MESI |
| Memory controller | FR-FCFS scheduling, line-interleaved mapping, closed-page policy |
| Memory bandwidth | 12 GB/s |
| Memory systems | 32 GB DRAM-only |
| | 32 GB PCM-only |
| | Hybrid 1 GB DRAM + 32 GB PCM |
| Organization | 8 1 Gb chips per rank |
| | 1-8 ranks per DIMM, 1-4 DIMMs |
| DRAM parameters | 45 ns read/write |
| | 0.678 Watts read, 0.825 Watts write |
| PCM parameters | 180 ns read, 450 ns write |
| | 0.617 Watts read, 3.0 Watts write |
| | 30.0 million writes per cell |
| | Fine-grained wear-leveling [42] |
| DRAM device | Micron DDR3 [33] |

(256 bytes), and block size (32 KB). We explore four write-rationing garbage collectors shown in Table 1: Kingsguard-nursery, Kingsguard-writers, and two variants that exclude the Large Object Optimization (LOO) and the Metadata Optimization (MDO) to tease apart their impact.

Nursery size impacts performance, pause time, and space efficiency [4, 6, 48, 56]. Our default configurations use a 4 MB nursery and a fixed-size maximum heap size of 2× minimum live size for each application, following prior work [1, 10, 35, 44, 56]. Fixing the heap size fairly controls the space-time tradeoffs that different collectors make. We set the default observer space size at 8 MB. We empirically find that sizing the observer space to be twice that of the nursery is the best compromise between tenured garbage and pause time. We explore other configurations, including larger nursery sizes (Section 6.2.1). Large nurseries reduce writes to mature objects, but are not sufficient to manage hybrid memories.

***Java applications.*** We use 16 Java applications: 12 DaCapo [8], pseudojbb2005 (pjbb) [9], and 3 graphchi [25]. The graphchi applications are disk-based graph processing of (1) page rank (PR), (2) connected components (CC), and (3) ALS matrix factorization (ALS). For PR and CC, we use the LiveJournal online social network [30] as the input dataset. For ALS, we use the training set of the Netflix Challenge. We process 1 M edges using PR and CC, and 1 M ratings using ALS. We use the default datasets for DaCapo and pjbb2005. In addition to the original versions of lusearch and pmd in DaCapo, we use an updated version of lusearch, called lu.Fix (described in [53]), that eliminates useless allocation, and an updated version of pmd, called pmd.S (described in [15]) that eliminates a scaling bottleneck due to a large input file.

## 5.2 Hardware and Simulation

Our evaluation uses both simulation of hybrid memories and execution on real hardware with DRAM memory because we lack access to systems with hybrid memories. Simulation has the advantage that it models PCM properties accurately and captures many details better than execution on DRAM. It has the disadvantage that multicore simulation is extremely time consuming, which limits the number of hardware configurations such as the number of cores we can practically explore. Due to limitations in the simulator, it unfortunately executes only a subset of the benchmarks. Fortunately, these benchmarks cover the extremes and thus a wide range of write of behaviors. All our applications execute on real hardware. We furthermore configure the real hardware in various ways to match and validate many of the simulation results.

### 5.2.1 Hardware Platform

We use the Intel Nehalem-based IBM x3650 M2 with two Intel Xeon X5570 processors for hardware execution time measurements. Each Xeon processor has 4 cores. Although there are two sockets, we use one to limit non-determinism and to match the multicore simulator, which is only practical to run with at most 4 cores. Each core has a private L1 with 32 KB for data and 32 KB for instructions. The unified 256 KB L2 cache is private, and the 8 MB L3 cache is shared across all four cores on each socket. The machine has a main memory capacity of 14 GB.

### 5.2.2 Simulator

Because PCM is not commercially available, we modify a simulator to model hybrid memories. We use Sniper [13] v6.0, an x86 simulator because it is cycle-level, parallel, high-speed and models multicore systems. We use its most detailed cycle-level and hardware-validated core model. Prior work extended Sniper for managed language runtimes, including dynamic compilation, and emulation of frequently used system calls [44]. Because Sniper is a user-level simulator, we are only able to execute ten of the Java applications. We eliminate fop, luindex, and avrora from simulation results because of their low allocation rates (see Table 4). These limitations motivate the use of additional results on actual hardware.

***Memory system and processor architectures.*** We compare three main memory systems in the simulator: (1) a 32 GB DRAM-only system, (2) a 32 GB PCM-only system, and (3) a hybrid system with 1 GB DRAM plus 32 GB PCM. We model PCM with a base read latency of 4× the DRAM latency, and a write latency of 12× the DRAM latency [21, 26, 36]. Table 2 presents other key architecture, DRAM, PCM, and cache memory parameters, which we hold constant. We model a quad-core processor configuration similar to the Intel Haswell processor i7-4770K. Each core is a superscalar out-of-order core with private L1 and L2, and shared L3 caches.

***Power and energy estimation.*** We use McPAT v1.0 [31] to model processor power consumption. We model DRAM power according to Micron's DDR3 device specifications [33]. PCM uses a 1 KB row buffer similar to DRAM. The remaining peripheral circuitry is also similar with one important distinction. When writing data from a row buffer to a PCM array, only the modified line is written back. PCM read operations do not require pre-charging due to their non-destructive nature and consume less energy than DRAM. The static power of PCM prototypes are negligible compared to DRAM [27]. Using latency and energy estimations of PCM prototypes from Lee et al. [26], we compute average power to write a cache line to a PCM array as 3 Watts. When estimating PCM latency and power consumption, we assume the same technology node for DRAM and PCM, and the scaling model from Lee et al. [26].

***PCM lifetime modeling.*** We estimate PCM lifetime using an optimistic analytical model from the literature [21, 36, 42]. Prior work demonstrates wear-leveling mechanisms for future non-volatile memories [11, 17, 26, 27, 42, 43, 50]. Therefore, we assume writes can be made uniform over the entire capacity of PCM. PCM memory lifetime in terms of years before failing is estimated as follows:

$$Y = \frac{S \times E}{B \times 2^{25}} \quad (1)$$

The size (S) of PCM main memory is 32 GB. We consider the PCM endurance (E) level used in prior work [38, 39]: 30 M writes per PCM cell. Finally, B is the write rate of an application during execution. Next, we describe our methodology for estimating the write rates of our applications on a 32-core machine.

***Write rate estimation.*** Due to limitations in simulator scalability, we are unfortunately only able to simulate a 4-core system. To extend our simulation results to write rates for a 32-core machine, we first obtain write rates for the 4-core system in Table 3. We then measure on a real hardware platform how write rates scale as we increase the number of cores from 4 to 32. We multiply the observed scaling behavior by our simulated write rates to estimate the write rates on a 32-core system. Our 32-core system has two Intel E5-2650L processors. Each processor has 8 cores and each core is 2-way SMT. Each processor has a 20 MB last level cache. The machine has 132 GB of main memory.

To measure write rates on real hardware, we use the Processor Counter Monitor from Intel. To fully utilize the 32 cores on our system, we run 32 instances of the same single-threaded benchmark, and 8 instances of the multithreaded benchmarks. Table 3 shows the scaling factor and write rates with multiple instances of each benchmark normalized to running a single instance of the benchmark. A few benchmarks scale linearly with the increase in core counts, but for others, such as antlr and bloat, the write rates increase by more than an order of magnitude. These applications experience increased contention in the last level cache. Table 3 shows estimated write rates vary from 3.1 GB/s to 24 GB/s.

**Table 3.** Measured scaling of and estimated write rates.

| Benchmark | Normalized scaling factor (measured) | Write rate in GB/s (estimated) |
|---|---|---|
| Xalan | 7.3× | 8.5 |
| Pmd | 7.7× | 3.1 |
| Pmd.Scale | 10.0× | 7.0 |
| Lusearch | 5.0× | 9.3 |
| Lu.Fix | 5.2× | 7.0 |
| Antlr | 52.0× | 19.0 |
| Bloat | 63.0× | 24.0 |

## 6 Results

Section 6.1 presents our simulation results which evaluate PCM lifetimes, write behavior, energy, and overhead. We compare DRAM-only, PCM-only, and hybrid systems with Kingsguard collectors. Our cycle-level simulator faithfully models the cache hierarchy found in real systems and wear-leveling hardware. Modeling caches is important because they absorb writes, and are thus the first line of defense in protecting PCM from writes. Modeling wear-leveling is important because by spreading writes to lines and pages evenly, it makes write rate the only necessary target for optimization.

Section 6.2 presents performance results on real hardware of all Kingsguard configurations. It includes statistics on how Kingsguard collectors organize the heap to influence PCM write traffic for 16 Java applications and write traffic to PCM measured in an architecture-independent manner.

Both sets of results show significant improvements in PCM lifetimes in hybrid memories using our write-rationing garbage collectors. The hardware results confirm the simulation results and explore overheads and optimizations in more detail.

### 6.1 Simulation Results

#### 6.1.1 Lifetime

Figure 5 presents PCM lifetime improvements normalized to PCM-only using lifetime estimates from the models in Section 5. On a 32 GB PCM-only system with an endurance of 30 M writes, line-level write back and wear-leveling, application lifetimes average 4 years, but are sometimes as low as 15 months, e.g., lusearch. Kingsguard collectors executing on a hybrid memory system deliver substantial lifetime improvements over PCM-only systems. KG-N improves lifetime on average 5×. Individual benchmarks improve by 1.9× for xalan and up to 11× for lu.Fix. KG-W improves lifetime even more: 11× longer than a PCM-only system on average. Individual benchmarks improve by 6× for lusearch and up

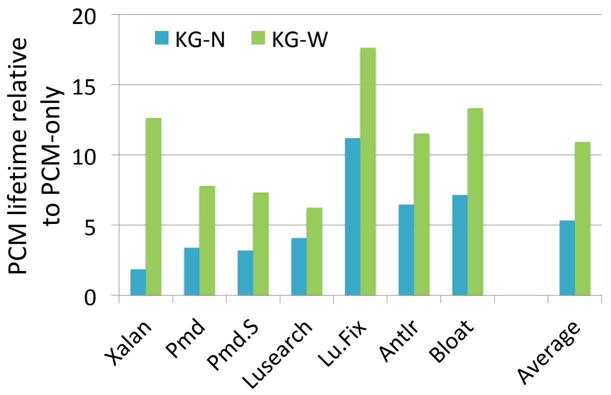


**Figure 5.** Kingsguard-nursery (KG-N) and Kingsguard-writers (KG-W) increase PCM lifetime.

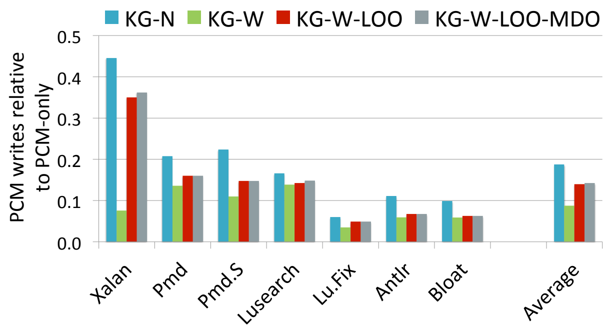


**Figure 6.** All Kingsguard configurations substantially reduce writes compared to the PCM-only baseline (1.0).

to 17× for lu.Fix. KG-W achieves these long lifetimes by minimizing writes to PCM memory, while using an average of only 16 MB of DRAM and at most 24 MB of DRAM for these applications (see Table 4 and Section 6.2.3).

### 6.1.2 Write Analysis

Figure 6 plots writes to PCM using the Kingsguard configurations from Table 1 normalized to PCM-only. While KG-N reduces writes to PCM by 81% on average, KG-W reduces writes by 91% compared to a PCM-only system. Leaving out the large object optimization (KG-W–LOO) and metadata optimization (KG-W–LOO–MDO) has a small overall impact, except for xalan. In xalan, the large object optimization reduces writes to large objects and, more surprisingly, writes to small objects to which the large objects point. While the effect on total writes is small with MDO, it does eliminate a lot of writes to PCM during major collections: 50% and 12% respectively for xalan and lusearch.

### 6.1.3 Comparison to Write Partitioning (WP)

We now compare Write Partitioning (WP), the state-of-the-art OS technique for reducing writes to PCM [55, 57]. Our implementation, as described in Section 2, uses the recommended eight queues, OS mapping time quantum of 10 ms, migrates pages in the four highest ranked queues to DRAM, and demotes all pages in DRAM to a lower ranked queue

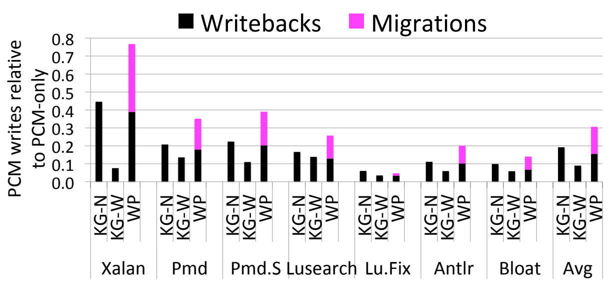


**Figure 7.** OS-managed write partitioning (WP) results in more writes to PCM than KG-N and KG-W.

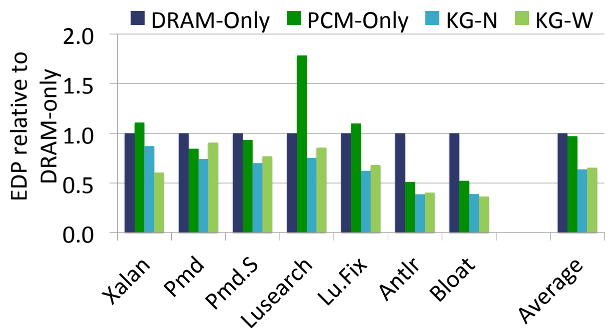


**Figure 8.** Kingsguard reduces the energy-delay product (EDP) compared to DRAM-only and PCM-only.

every 50 ms to optimize for phase behavior. We explore other configurations, but these parameters perform best for our workloads. Figure 7 plots PCM write reductions by KG-N, KG-W, and WP, normalized to PCM-only. *Writebacks* include writes by the application and collector. *Migrations* show writes due to WP migrating pages from DRAM to PCM. WP's reactive policy does eventually detect nursery pages as highly written, but this detection takes time. WP is effective at reducing application writes to PCM, but its migration policy moves pages from PCM to DRAM and back to PCM. For example, WP observes lots of writes when the default collector copies nursery objects to the mature space, which triggers WP to migrate pages from PCM to DRAM. Many of these pages incur few subsequent writes, so WP migrates them back to PCM. WP reduces writes to PCM by 69%, whereas KG-N and KG-W reduce writes by 81% and 91%, respectively. By using object demographics and fine-grained per-object monitoring, KG-W has over 3× fewer writes than WP.

### 6.1.4 Energy

To quantify the energy efficiency of KG-W, KG-N, and a PCM-only system, Figure 8 shows the energy-delay product (EDP) normalized to a DRAM-only system. EDP is energy multiplied by execution time, so it takes the higher latencies of PCM into account. The EDP is sometimes worse on a PCM-only system compared to DRAM-only, particularly for lusearch. Using KG-N reduces the average energy-delay

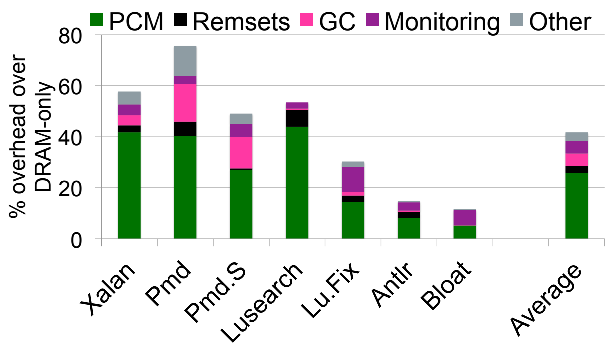


**Figure 9.** PCM access time overheads dominate collector and monitor overheads in KG-W.

product by 36% over the DRAM-only system, and also significantly improves over a PCM-only system. Because of KG-W's additional overhead, its energy-delay product is slightly higher, saving 32% over a DRAM-only system. In addition to reducing the EDP, both KG-N and KGW reduce the total energy consumption by 47% on average.

### 6.1.5 Breakdown of Overheads

A PCM-only system adds 70% to the execution time of a DRAM-only system on average. Our simulator results show that KG-N reduces overheads by over 50% compared to PCM-only, but still adds 31% to execution time on average over DRAM-only. KG-W adds overhead (40% on average) because it monitors individual objects and copies long-lived objects at least one more time than KG-N, since it first copies them to the observer space and then to a mature space.

We break down KG-W overheads into: (a) Remsets — write-barriers to remember pointers to the observer space; (b) GC — KG-W adds collections of the observer space, which often adds overhead; however, collection time sometimes reduces because objects die in the observer space, reducing full heap collections; (c) Monitoring — KG-W's write-barrier records more information about all writes to non-nursery objects; and (d) PCM — PCM has longer read and write latencies than DRAM. Other effects, such as cache locality, are unmeasured in this experiment and we report them in (e) Other. We configure the simulator and the VM in a variety of different ways to measure all these overheads.

Figure 9 presents this breakdown relative to DRAM-only. The largest overhead is the longer PCM access latencies (PCM), which add 25% to total time on average. The overhead of collecting KG-W's extra spaces (GC), and of monitoring writes to non-nursery spaces (Monitoring) are each a little under 5% on average. Keeping track of more remembered sets to collect the observer space in isolation (Remsets) adds around 3% overhead. Other overhead (Other) accounts for another 3% of extra execution time. Pmd has a high Other overhead because it has a high nursery survival rate (see Table 4) which triggers more observer collections and has cache effects. Our real system performance results in Section 6.2.2 confirm that the Kingsguard mechanisms themselves add little to total time. Using a hybrid DRAM-PCM memory system for its scalability properties inevitably adds latency to execution times. KG-W mitigates these latencies by redirecting some reads and writes to a small amount of DRAM.

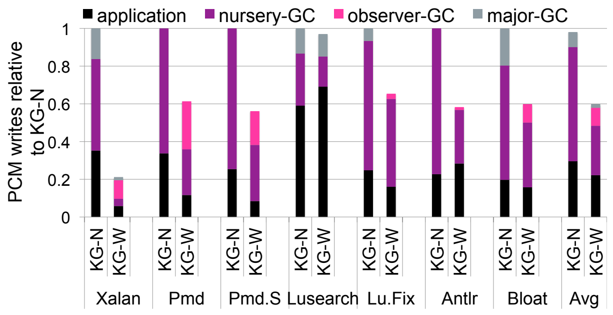


**Figure 10.** Where writes to PCM originate. KG-W reduces application and collector writes to PCM.

### 6.1.6 The Origin of Writes

Figure 10 classifies where writes to PCM originate: the application, nursery collection, observer collection, or major collection. We modify the simulator to track which phase last wrote each cache line, since LRU policies evict lines to PCM or DRAM well after their last access. KG-W reduces PCM application writes for most benchmarks compared to KG-N. This reduction corresponds to an increase in DRAM writes (not shown), as designed. The average increase in writes to both DRAM and PCM together (not shown) with KG-W over KG-N is 12% and the worst case is pmd at 25%. This increase stems from additional collection work. Figure 10 shows only the writes to PCM.

The collector induces writes when it initially places an object in PCM and when it updates PCM references to other objects. When analyzing the writes performed by the collector, we note that: (1) KG-N incurs writes to PCM during a nursery collection both due to copying survivors into the PCM mature space *and* due to updating the references in PCM that point to them; (2) KG-W eliminates major collections for lu.fix and bloat by reclaiming objects in the observer space; (3) writes to PCM during a nursery collection with KG-W are solely due to updating references in PCM spaces that point to surviving nursery objects copied to the observer space. These results suggest further PCM write reductions are possible by avoiding pointer updates in PCM and deploying better predictors of application writes.

## 6.2 Real Hardware Results

We now evaluate Kingsguard on real hardware. We analyze write behavior, performance, and memory characteristics. Lacking PCM hardware, all latencies are to DRAM.

### 6.2.1 Write Analysis

Figure 11 presents writes to the PCM heap using KG-N and KG-W as reported by write-barriers. These results are thus architecture-independent since they do not consider cache

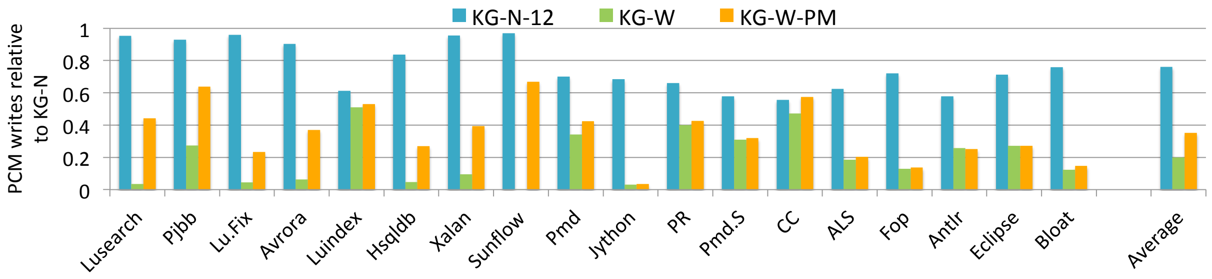


**Figure 11.** Application writes to PCM. Giving KG-N a larger nursery (KG-N-12) saves a small portion of PCM writes. KG-W saves 80% of PCM writes compared to KG-N. Excluding primitive monitoring of writes (KG-W–PM) increases writes to PCM.

effects that filter out some writes to both DRAM and PCM. We normalize to KG-N with a 4 MB nursery and compare to KG-N with a larger 12 MB nursery, and to KG-W with a 4 MB nursery with and without primitive write-barriers. Using a larger nursery reduces the writes to PCM by 24% on average compared to KG-N. A larger nursery is not effective at reducing PCM writes for four out of the five applications with more writes in the mature space than the nursery (the five left-most applications in this figure and in Figure 2). KG-W is much more effective than simply using a larger nursery, reducing writes to PCM by 80% on average.

For sunflow, KG-W eliminates 99.7% of writes to PCM by copying the written objects to mature DRAM during observer collections. For pjbb and hsqldb, we similarly observe many writes to the few mature DRAM objects. On the other hand, KG-W eliminates 97% of writes to PCM for lusearch by moving primitive arrays from PCM to DRAM during mature collections. KG-W reduces writes for all the GraphChi benchmarks, which all need very large heaps, by over 50% as compared to KG-N. For luindex and CC, large objects in PCM incur a lot of writes, and are only moved to DRAM during a mature collection. Interestingly, luindex with KG-W requires no mature space collections because so many objects die in the observer space. For CC, writes happen before a mature collection is triggered. These behaviors motivate additional policies for mature collection to be triggered by writes to PCM. We leave this exploration to future work.

***Primitive versus reference monitoring.*** We observe that excluding primitive monitoring (KG-W–PM) in Figure 11 significantly reduces writes compared to KG-N for several applications. For instance, in the case of pmd, eclipse, and bloat, reference writes capture most of the highly mutated objects. On the other hand, for seven applications, PCM writes increase quite a bit over KG-W. On average, KG-W-PM eliminates 65% of PCM writes compared to 80% for KG-W.

### 6.2.2 Performance

Figure 12 presents the performance of KG-W configurations normalized to KG-N. The results in Figure 12 understate the performance advantage of KG-W over KG-N. On a system with PCM, the large reduction in writes to PCM reduces execution time due to latency savings. With respect to KG-W, the

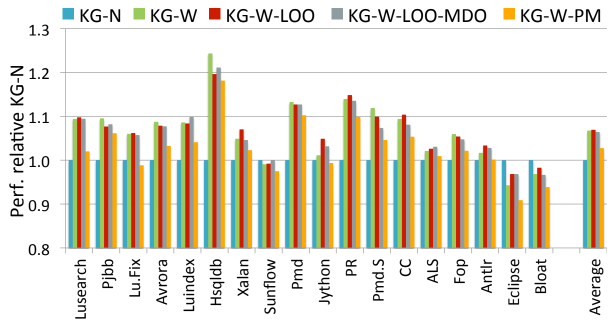


**Figure 12.** KG-N performs best. KG-W adds 7% overhead to execution time on average.

large object (KG-W–LOO) and metadata space optimizations (KG-W–LOO–MDO) are performance-neutral on average. KG-W increases the execution time on average by 7%, and hsqldb by 25% over KG-N. This overhead is mostly due to the additional observer collections. During each observer collection, some highly mutated objects are placed in mature DRAM, which results in a large reduction in writes to PCM, as shown in Figure 11. KG-W reduces execution time for a few benchmarks: sunflow, eclipse, and bloat, due to fewer full-heap collections. This result is a feature of KG-W. Observer collections are cheaper than full-heap collections because (1) they operate over smaller regions and (2) when they reclaim objects, they prevent mature DRAM and PCM from filling up. Finally, we observe that eliminating primitive monitoring (KG-W–PM) has the highest impact on lusearch: a 7% reduction in execution time.

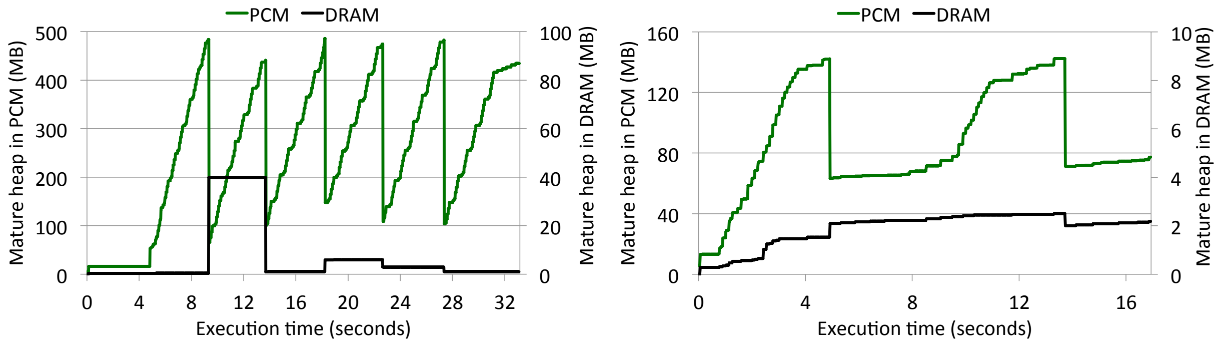
### 6.2.3 Memory and Demographic Analysis

Table 4 shows allocation and survival rates, and heap space occupancy per benchmark for our collectors. Applications allocate frequently, between 56 MB and 14 GB of memory (column 1), especially the GraphChi benchmarks. We gray out those applications with less than 100 MB of allocation and exclude them from the averages. Our applications have an average nursery survival rate of 17% and a maximum of 66% (columns 3 and 4).

A 4 MB DRAM nursery maximizes the use of PCM for 97% of the KG-N heap, averaging between 21 and 280 MB, and up to 502 MB of PCM, for the GraphChi benchmarks (columns 5 and 6). KG-W uses more DRAM: 6 to 86 MB of

**Table 4.** Object demographics. KG-N maximizes the use of PCM, whereas KG-W more selectively uses PCM since (1) many objects die before promotion and (2) KG-W retains a small fraction of objects in DRAM. Columns 12 and 13 show the DRAM consumed by WP for the simulated benchmarks.

| | allocation | Heap | % nursery | survival | KG-N | PCM MB | KG-W | PCM MB | KG-W | DRAM MB | WP | DRAM MB | KG-W % mature | KG-W | metadata MB | KG-W % observer | KG-W % held | in DRAM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | MB | MB | KG-N | KG-W | avg | max | avg | max | avg | max | avg | max | in DRAM | avg | max | survival | MB | Obj |
| | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) | (11) | (12) | (13) | (14) | (15) | (16) | (17) | (18) |
| Lusearch | 4294 | 68 | 4% | 4% | 49 | 64 | 41 | 63 | 6 | 7 | 31 | 38 | 0% | 1.8 | 2.8 | 29% | 6% | 9% |
| Pjbb | 2314 | 400 | 20% | 20% | 280 | 386 | 252 | 310 | 40 | 52 | | | 5% | 16 | 20 | 84% | 7% | 6% |
| Lu.Fix | 848 | 68 | 2% | 2% | 22 | 24 | 16 | 16 | 15 | 15 | 7 | 8 | 8% | 1.6 | 1.9 | 25% | 3% | 4% |
| Avrora | 64 | 98 | 15% | 15% | 24 | 25 | 20 | 20 | 15 | 16 | | | 0% | 3 | 4 | 0% | 0% | 0% |
| Luindex | 37 | 44 | 22% | 22% | 21 | 22 | 21 | 21 | 13 | 13 | | | 0% | 1 | 1 | 0% | 0% | 0% |
| Hsqldb | 165 | 254 | 66% | 60% | 85 | 137 | 70 | 120 | 19 | 21 | | | 1% | 6 | 8 | 88% | 0.2% | 0.02% |
| Xalan | 980 | 108 | 17% | 14% | 73 | 104 | 52 | 92 | 18 | 18 | 51 | 60 | 7% | 1.8 | 1.9 | 9% | 8% | 1% |
| Sunflow | 1920 | 108 | 2% | 2% | 31 | 45 | 21 | 22 | 18 | 18 | | | 15% | 2 | 2 | 13% | 35% | 30% |
| Pmd | 364 | 98 | 23% | 23% | 73 | 94 | 94 | 47 | 20 | 24 | 13 | 25 | 8% | 4.3 | 7.9 | 68% | 5% | 7% |
| Jython | 1150 | 80 | 0.001% | 0.2% | 73 | 80 | 68 | 64 | 21 | 26 | | | 6% | 5 | 10 | 12% | 30% | 25% |
| PR | 6946 | 512 | 36% | 36% | 254 | 502 | 263 | 490 | 32 | 60 | | | 3% | 13 | 6 | 99% | 0.3% | 0.1% |
| Pmd.S | 202 | 98 | 27% | 27% | 55 | 77 | 33 | 40 | 19 | 20 | 17 | 24 | 11% | 2.6 | 3.5 | 47% | 6% | 9% |
| CC | 5507 | 512 | 24% | 24% | 242 | 502 | 240 | 484 | 35 | 75 | | | 5% | 10 | 22 | 97% | 2% | 1% |
| ALS | 14245 | 512 | 9% | 10% | 254 | 502 | 215 | 430 | 86 | 224 | | | 25% | 4 | 22 | 63% | 50% | 41% |
| Fop | 56 | 80 | 20% | 20% | 26 | 28 | 24 | 24 | 17 | 17 | | | 8% | 3 | 3 | 82% | 14% | 7% |
| Antlr | 246 | 48 | 15% | 15% | 27 | 36 | 19 | 20 | 15 | 15 | 7 | 9 | 10% | 1.1 | 1.1 | 0.16% | 17% | 5% |
| Bloat | 1246 | 66 | 4% | 4% | 34 | 46 | 27 | 27 | 16 | 16 | 7 | 27 | 7% | 1.9 | 2.1 | 19% | 13% | 12% |
| Eclipse | 3082 | 160 | 15% | 14% | 114 | 155 | 110 | 145 | 23 | 25 | | | 5% | 6 | 7 | 37% | 1% | 1% |
| Avg (All) | 2900 | 206 | 17% | 17% | 111 | 184 | 98 | 140 | 26 | 40 | | | 8% | 5 | 8 | 46% | 12% | 10% |
| Heap % | | | | | 97% | 98% | 80% | 80% | 20% | 20% | | | 8% | 2% | 4% | | | |
| **Avg (Sim)** | **1169** | **79** | **13%** | **13%** | **48** | **64** | **34** | **47** | **16** | **16** | **19** | **27** | **4%** | **2.2** | **3.0** | **28%** | **8%** | **7%** |
| **Heap %** | | | | | **92%** | **94%** | **68%** | **88%** | **32%** | **22%** | **39%** | **36%** | **4%** | **4%** | **5%** | | | |



**(a)** Page Rank **(b)** Eclipse

**Figure 13.** MB in PCM (left-hand y-axis label) versus MB in DRAM (right-hand y-axis label) as a function of time for Page Rank and Eclipse. KG-W uses large amounts of PCM and small amounts of DRAM.

DRAM on average and up to 224 MB (columns 9 and 10), and 16 MB to 263 MB of PCM and up to 484 MB for CC (columns 7 and 8). KG-W trades higher utilization of DRAM (10%) for disproportionate increases in PCM liftetimes.

Columns 11 and 12 show the DRAM consumed by WP. The average DRAM consumption is 7% more than KG-W. Individual benchmarks behave differently. For instance, for lusearch and xalan, WP consumes 3× and 5× more DRAM on average. Both these benchmarks allocate many large objects directly in PCM. WP's reactive algorithm keeps these pages in DRAM. For the remaining benchmarks, WP consumes less or similar DRAM memory compared to KG-W.

Column 13 reports the percent of the total heap (column 1) occupied by KG-W's mature DRAM space, which ranges from 1.3 MB to 186 MB, only 8% of the heap on average. Columns 14 and 15 show that the DRAM metadata space consumes a small fraction of the KG-W heap. Overall, KG-W stores 80% of the heap in PCM versus KG-N's 98%.

Column 16 shows that the observer-space survival rate ranges from 0.2% to 99%. Benchmarks with large observer survival rates, such as hsqldb, PR, pmd, pmd.S, and CC have correspondingly higher overheads in Figure 12 due to having to copy objects twice before they reach the mature space. In contrast, the benchmarks with low observer space survival rates have lower overheads. The last columns shows KG-W copies most objects to PCM: it retains only between 0.2% and 41% of surviving observer objects in DRAM. KG-W uses less PCM memory than KG-N for two reasons: objects die in the observer space and KG-W keeps a few written objects in mature DRAM. Furthermore, even though these applications exhibit a wide range of diverse behaviors with respect to object demographics and writes, KG-W is extremely effective at limiting writes to PCM by managing object placement and migration in hybrid DRAM-PCM memories.

#### 6.2.4 Heap Composition

This section explores the way KG-W uses DRAM and PCM using heap composition graphs. Figure 13 plots the usage of PCM versus DRAM in MB over time for PR and eclipse. For both applications, full heap collections cause the amount of PCM memory used to decrease drastically, mostly because many objects die, but also because some are copied to DRAM.

For PR, while the amount of PCM used grows to close to 500 MB, the maximum amount of DRAM used (right axis) is around 40 MB. Visually, DRAM occupancy increases at the same time as PCM shrinks due to a full heap collection that moves objects from PCM to DRAM. For instance at around 10 seconds into execution, DRAM occupancy increases due to a full heap collection that copies written mature objects out of PCM. As Table 4 shows, Page Rank has a high observer survival rate, yet KG-W promotes very few of these surviving observer objects to mature DRAM. The graph shows how these observer collections quickly populate PCM and consequently trigger full heap collections.

We observe that eclipse uses up to 145 MB of PCM, but the DRAM usage maxes out around 2.5 MB. Highly mutated objects in DRAM also have long lifetimes for eclipse. During mature collections (around the 5 second and 14 second marks), whereas the PCM usage drops by almost half, much of mature DRAM stays alive. Table 4 shows that observer collections copy only 1% of surviving objects on average to mature DRAM. Fortunately, these DRAM objects are highly written, which protects PCM from writes.

### 6.3 Discussion

Researchers are still developing non-volatile memory technologies, so endurance levels, access latency, and energy characteristics may improve. Regardless, PCM cell endurance is very unlikely to reach DRAM levels because of material properties. Other technologies, such as resistive random-access memory prototypes, have higher, but still finite endurance, e.g., one trillion writes per cell [28].

Faster write operations require higher temperatures in many non-volatile memory technologies, i.e., the switching speed depends on the temperature. Recent studies trade off the speed of write operations for improved lifetimes [54]. Slower writes stall the processor pipeline, thus hurting performance. On the system side, as the number of cores on a processor increases, write rates to memory will also increase. These trends argue for software approaches, such as write-rationing garbage collection, to reduce memory writes and optimize energy consumption.

Our KG-W collector copies objects at least twice, from the nursery to the observer space and then from the observer space to either DRAM or PCM. This copying induces PCM writes for PCM object references to copied objects. The original Immix paper describes non-moving variants of Immix, which would eliminate these writes [10]. These collectors trade the locality of contiguous allocation for increased heap occupancy and fragmentation. This tradeoff is good for PCM lifetimes and merits but may increase DRAM requirements.

The default Immix collector defragments blocks based on a threshold that indicates fragmentation is preventing the collector from using some fraction of the memory in partially filled blocks in the mature space. Immix defragmentation combines marking with copying based on occupancy statistics of the partially filled blocks, seeking to move the fewest numbers of objects and to create the maximum number of completely free blocks. It thus trades increased numbers of writes to reduce memory consumption. PCM memory prefers exactly the opposite tradeoff – PCM memory is write-limited coupled with plentiful capacity. For the heap sizes this paper explores, Immix defragmentation was never triggered. However in the limit, the collector should monitor and limit extreme fragmentation. We leave the exploration of non-moving collectors and defragmentation for future work.

## 7 Conclusions

This paper introduces write-rationing garbage collectors, which seek to maximize the use of PCM while improving its lifetime in hybrid memory systems. Our Kingsguard collectors exploit object demographics and individual object write behaviors in Java applications. Kingsguard-nursery (KG-N) places nursery objects in DRAM and all other objects in PCM. Kingsguard-writers (KG-W) adds monitoring of mature object writes and moves objects between DRAM and PCM based on their individual write history. KG-N places 92% of heap objects on average in PCM, but still removes over 80% of writes to PCM compared to PCM-only with hardware wear-leveling, leading to a 5× improvement in PCM lifetime. KG-W places 68% of the heap in PCM to remove over 90% of all writes, thus greatly extending PCM lifetime by 11×. Both KG-N and KG-W improve over WP, the state-of-the-art OS approach [43, 55]; WP writes to PCM 3× more than KG-W. This work demonstrates that managed runtimes have a significant advantage over hardware and OS-only approaches because they can exploit, observe, and react to coarse-grained object demographics and to fine-grained object behaviors, opening up a new and promising direction for managing hybrid DRAM-PCM memory systems.

## Acknowledgements

This research is supported through FWO grants no. G.0434.16N and G.0144.17N, European Research Council (ERC) Advanced Grant agreement no. 741097, and VUB's VLAIO Smile-IT project.

## References

[1] Shoaib Akram, Jennifer B. Sartor, and Lieven Eeckhout. DEP+BURST: Online DVFS Performance Prediction for Energy-Efficient Managed Language Execution. *IEEE Trans. Comput.* 66, 4 (April 2017).

[2] Bowen Alpern, C. Richard Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño virtual machine. *IBM Systems Journal* 39, 1 (2000).

[3] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. The Jikes RVM Project: Building an Open Source Research Community. *IBM System Journal* 44, 2 (2005).

[4] Andrew W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Softw. Pract. Exper.* 19, 2 (Feb. 1989).

[5] Aravinthan Athmanathan, Milos Stanisavljevic, Nikolaos Papandreou, Haralampos Pozidis, and Evangelos Eleftheriou. Multilevel-Cell Phase-Change Memory: A Viable Technology. *IEEE J. Emerg. Sel. Topics Circuits Syst.* 6, 1 (2016).

[6] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and Realities: The Performance Impact of Garbage Collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS).*

[7] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *International Conference on Software Engineering (ICSE).*

[8] 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).*

[9] Stephen M Blackburn, Martin Hirzel, Robin Garner, and Darko Stefanović. 2010. pjbb2005: The pseudojbb benchmark, 2005. http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005

[10] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[11] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mossé. 2014. Concurrent Page Migration for Mobile Systems with OS-managed Hybrid Memory. In *Proceedings of the 11th ACM Conference on Computing Frontiers (CF).*

[12] Geoffrey W. Burr, Matthew J. Breitwisch, Michele Franceschini, Davide Garetto, Kailash Gopalakrishnan, Bryan Jackson, BÃŒlent Kurdi, Chung Lam, Luis A. Lastras, Alvaro Padilla, Bipin Rajendran, Simone Raoux, and Rohit S. Shenoy. Phase change memory technology. *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena* 28, 2 (2010).

[13] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 3 (2014).

[14] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM).*

[15] Kristof Du Bois, Jennifer B. Sartor, Stijn Eyerman, and Lieven Eeckhout. 2013. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA).*

[16] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. 2009. Demystifying Magic: High-level Low-level Programming. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE).*

[17] Tiejun Gao, Karin Strauss, Stephen M. Blackburn, Kathryn S. McKinley, Doug Burger, and James Larus. 2013. Using Managed Runtime Systems to Tolerate Holes in Wearable Memories. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[18] Jungwoo Ha, Magnus Gustafsson, Stephen M. Blackburn, and Kathryn S. McKinley. 2008. Microarchitectural Characterization of Production JVMs and Java Workloads. In *IBM CAS Workshop.*

[19] Michael Hicks, Luke Hornof, Jonathan T. Moore, and Scott M. Nettles. A Study of Large Object Spaces. *SIGPLAN Not.* 34, 3 (Oct. 1998).

[20] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The Garbage Collection Advantage: Improving Mutator Locality.. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA).*

[21] ITRS. 2015. Internatial Technology Roadmap for Semiconductors 2.0: Executive Report.

[22] Michael R. Jantz, Forrest J. Robinson, Prasad A. Kulkarni, and Kshitij A. Doshi. 2015. Cross-layer Memory Management for Managed Language Applications. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

[23] Richard Jones and Rafael Lins. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.*

[24] Mark H. Kryder and Chang Soo Kim. After Hard Drives — What Comes Next? *IEEE Transactions on Magnetics* 45, 10 (2009).

[25] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI).*

[26] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA).*

[27] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (Jan. 2010).

[28] Myoung-Jae Lee, Chang Bum Lee, Dongsoo Lee, Seung Ryul Lee, Man Chang, Ji Hyun Hur, Young-Bae Kim, Chang-Jung Kim, David H. Seo, Sunae Seo, U-In Chung, In-Kyeong Yoo, and Kinam Kim. A fast, high-endurance and scalable non-volatile memory device made from asymmetric Ta2O5-x/TaO2-x bilayer structures. *Nature Materials* 10, 3 (2011).

[29] Soyoon Lee, Hyokyung Bahn, and Sam H. Noh. 2011. Characterizing Memory Write References for Efficient Management of Hybrid PCM and DRAM Memory. In *IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS).*

[30] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[31] Sheng Li, Jung Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on*

*Microarchitecture (MICRO).*

[32] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA).*

[33] Micron. 2007. TN-41-01: Calculating memory system power for DDR3.

[34] Onur Mutlu and Lavanya Subramanian. Research Problems and Opportunities in Memory Systems. *Supercomputing Frontiers and Innovations* 1, 3 (Oct 2014).

[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).*

[36] Numonym. 2008. Phase Change Memory. http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf

[37] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java Hotspot server compiler. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM).*

[38] Moinuddin K. Qureshi. 2011. Pay-As-You-Go: Low-overhead Hard-error Correction for Phase Change Memories. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[39] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[40] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing lifetime and security of PCM-based Main Memory with Start-Gap Wear Leveling. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[41] Moinuddin K. Qureshi, Andre Seznec, Luis A. Lastras, and Michele M. Franceschini. 2011. Practical and secure PCM systems by online detection of malicious write streams. In *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA).*

[42] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA).*

[43] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS).*

[44] Jennifer B. Sartor, Wim Heirman, Stephen M. Blackburn, Lieven Eeckhout, and Kathryn S. McKinley. 2014. Cooperative Cache Scrubbing. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT).*

[45] Andre Seznec. A Phase Change Memory as a Secure Main Memory. *IEEE Computer Architecture Letters* 9, 1 (Jan 2010).

[46] Rifat Shahriyar, Stephen M. Blackburn, Xi Yang, and Kathryn S. McKinley. 2013. Taking Off the Gloves with Reference Counting Immix. In *ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA).*

[47] Darko Stefanovic, Kathryn S. McKinley, and J. Eliot B. Moss. 1999. Age-based Garbage Collection. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA).*

[48] David Ungar and Frank Jackson. An Adaptive Tenuring Policy for Generation Scavengers. *ACM Trans. Program. Lang. Syst.* 14, 1 (Jan. 1992).

[49] Chenxi Wang, Ting Coa, John Zigman, Fang Lv, Yunquan Zhang, and Xiaobing Feng. 2016. Efficient Management for Hybrid Memory in Managed Language Runtime. In *IFIP International Conference on Network and Parallel Computing (NPC).*

[50] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. 2015. Exploiting Program Semantics to Place Data in Hybrid Memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT).*

[51] Paul R. Wilson. A Simple Bucket-brigade Advancement Mechanism for Generation-bases Garbage Collection. *SIGPLAN Not.* 24, 5 (May 1989).

[52] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. 2012. Barriers Reconsidered, Friendlier Still!. In *Proceedings of the Eleventh ACM SIGPLAN International Symposium on Memory Management (ISMM).*

[53] Xi Yang, Stephen M Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S McKinley. 2011. Why Nothing Matters: The Impact of Zeroing. In *Proceedings of the ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*

[54] Lunkai Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T. Chong. 2016. Mellow Writes: Extending Lifetime in Resistive Memories Through Selective Slow Write Backs. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA).*

[55] Wangyuan Zhang and Tao Li. 2009. Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT).*

[56] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. 2009. Allocation Wall: A Limiting Factor of Java Applications on Emerging Multi-core Platforms. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA).*

[57] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference (ATC).*