Crystal Gazer: Profile-Driven 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

Non-volatile memories (NVM) offer greater capacity than DRAM but suffer from high latency and low write endurance. Hybrid memories combine DRAM and NVM to form scalable memory systems with the promise of high capacity, low energy consumption, and high endurance. Automatically managing hybrid NVM-DRAM memories to achieve their promise without changing user applications or their programming models remains an open question. This paper uses garbage collection in managed languages to exploit NVM capacity while preventing NVM wear out in hybrid memories with no changes to the programming model.

We introduce profile-driven write-rationing garbage collection. Allocation sites that produce frequently written objects are predicted based on previous program executions. Objects are initially allocated in a DRAM *nursery* space. The collector copies surviving nursery objects from highly written sites to a *mature* DRAM space and read-mostly objects to a mature NVM space. Write-intensity prediction for 15 Java benchmarks accurately places objects in the correct space, eliminating expensive object monitoring from prior write-rationing garbage collectors. Furthermore, our technique exposes a Pareto tradeoff between DRAM usage and NVM lifetime, unlike prior work. Experimental results on NUMA hardware that emulates hybrid NVM-DRAM memory demonstrates that profile-driven write-rationing garbage collection reduces the number of writes to NVM compared to prior work to extend its lifetime, maximizes the use of NVM for its capacity, and achieves good performance.

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

Additional Key Words and Phrases: Non-volatile memory (NVM), endurance, profiling, write-intensity prediction, garbage collection

ACM Reference Format:
Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2019. Crystal Gazer: Profile-Driven Write-Rationing Garbage Collection for Hybrid Memories. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 1, Article 9 (March 2019), 27 pages. https://doi.org/10.1145/3311080

1 INTRODUCTION

DRAM manufacturing complexity is increasing main memory cost. Recent semiconductor analyses show that DRAM price per gigabit increased by 50% between 2017 and 2018, whereas the year-over-year bit volume growth continued to decline [27, 33]. Main memory trends are especially

Authors' addresses: 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
2476-1249/2019/3-ART9 $15.00
https://doi.org/10.1145/3311080

worrisome as emerging applications have an insatiable desire for memory. Expecting further DRAM supply shortages, Facebook is already experimenting with hybrid DRAM and non-volatile memory (NVM) systems [23].

Production NVM uses phase-change memory (PCM). PCM offers high capacity, low idle power, and non-volatility, while being byte addressable, unlike some NVM. It however suffers two major drawbacks over DRAM: long latency and limited write endurance [16, 37, 38]. New materials may eventually bridge the latency gap [30, 42]. However, endurance will remain a problem because PCM writes change the material form [16]. Although prior hardware and software approaches improve PCM lifetime, an endurance gap remains [3, 37, 38, 48].

Memory system analysts and others report that Intel's Optane SSD, a commercially available NVM with 375 GB of capacity, can sustain a write rate of up to 10 TB/day, i.e., 140 MB/s without wearing out within the 3 year warranty period [23, 26]. Our experimental results (using an optimistic PCM latency) show that 13 out of our 15 modern Java applications exceed this write rate when using a PCM-only main memory. In practice, memory systems will therefore need hybrid DRAM and PCM memories. Similar products with a smaller capacity (up to 32 GB) only support write rates of up to 1.5 MB/s [26]. All of our Java applications exceed this write rate even using state-of-the-art hybrid memory management.

Including PCM into the main memory system requires solutions to tolerate the limited write endurance. Prior work uses hardware wear-leveling at the coarse-grain PCM write-granularity (256 B) to spread writes out across the entire PCM capacity to mitigate wear-out [45, 46, 48]. OS approaches migrate frequently written pages (4 KB) from PCM to DRAM to increase PCM lifetime [15, 49, 61]. Unfortunately, these approaches result in impractical lifetimes (one or two years in some cases) for managed language Java workloads [3].

In this paper, we aim to manage hybrid memories for highly allocating managed applications, without changes to the application, programming language or programming model, by exploiting PCM's capacity while improving its lifetime. Managed programming languages, such as Java, C#, and Python, use garbage collection to automatically organize and reclaim objects. We harness the garbage collector to re-organize objects efficiently into the DRAM and PCM memories based on profiling the objects' write behavior.

Prior work introduces *write-rationing garbage collection* for hybrid DRAM-PCM memory to improve endurance by placing frequently written objects in DRAM spaces and read-mostly objects in a PCM mature space [3]. Write-rationing garbage collectors, like other well-performing garbage collectors, are generational, meaning that new objects are first allocated into a space called a *nursery*. When the nursery fills, the collector identifies the live objects and *promotes*, or copies, them into a *mature* space, thus reclaiming the nursery space for new allocation. Generational collectors are efficient because for managed languages, most objects die young. Two write-rationing collectors exist in prior literature: Kingsguard-Nursery (KG-N) and Kingsguard-Writers (KG-W) [3]. Both back the nursery space by DRAM memory, because the nursery is highly written. For nursery survivors, the KG-W collector introduces a DRAM *observer space*. The observer space *dynamically* monitors object write behavior, and promotes read-mostly objects to a mature PCM space and highly written objects to a mature DRAM space. Unfortunately, it suffers from three main drawbacks. (1) It incurs high overhead from monitoring objects to dynamically discover frequently written objects. (2) It is reactive; it monitors object writes in a limited time window and must wait until the next collection to act on the information, leading to mispredictions and allocation of frequently written objects in PCM, particularly large objects. (3) It consumes excessive DRAM capacity.

This paper introduces profile-driven write-rationing garbage collection for hybrid memories, called Crystal Gazer (CGZ). We leverage the fact that modern mobile and server workloads execute frequently, which makes profiling practical. Prior work shows that allocation site, or the code

location where the object is allocated, is a good predictor of object lifetimes [7, 11, 14, 19, 32], and we find it is also a good predictor of write-intensity. We first profile individual object writes and their allocation sites, classifying a site as producing read-mostly or highly written objects in an offline run. We show the predictor is highly accurate with *true advice* (different inputs for training than classification) for 15 Java benchmarks from three suites (DaCapo, Pjbb and GraphChi).

CGZ uses the profile at runtime to guide object placement in mature DRAM-backed and PCM-backed memory spaces. CGZ initially allocates all objects in the DRAM nursery. It uses the advice to label objects at allocation time as coming from read-mostly or highly written allocation sites. When it promotes a nursery survivor, it copies the object to the DRAM or PCM mature space according to the predicted write-intensity label. If there is no advice at all, a production system should fall back on KG-W for dynamic monitoring. Unprofiled allocation sites may default to PCM or DRAM. CGZ promotes frequently written objects to DRAM, protecting PCM from writes. It promotes read-mostly objects to PCM, exploiting PCM capacity. It places the majority of mature objects in PCM, because only a small fraction of objects are frequently written.

Leveraging profile information overcomes the three major drawbacks of existing write-rationing garbage collectors. (1) Profile-driven promotion to mature DRAM and PCM spaces eliminates the overhead of dynamically monitoring objects. (2) CGZ is proactive in placing objects in DRAM or PCM, which combined with the high accuracy of ahead-of-time profiling, reduces mispredictions, particularly of large objects, and the need to copy objects between spaces, further extending PCM's lifetime. (3) Because writes are concentrated in a small fraction of objects and well predicted by allocation site, it reduces the amount of DRAM capacity needed, leveraging PCM's large capacity.

A key feature of CGZ is its ability to trade off PCM lifetime for DRAM capacity by using different heuristics and thresholds to classify allocation sites, using only one profiling run. Our experimental evaluation shows that CGZ provides a Pareto-optimal tradeoff between PCM lifetime and DRAM capacity. In contrast, KG-W provides a single sub-optimal operating point. Our experimental evaluation with 15 Java workloads uses an emulated hybrid memory system on multi-socket NUMA hardware to explore CGZ's effectiveness. We bind the application to one socket and emulate DRAM as the local NUMA node memory and PCM as the remote NUMA node memory. Compared to KG-W, the state-of-the-art in terms of improving PCM lifetime for hybrid memories, CGZ reduces the execution time overhead by 8% on average and up to 30%. CGZ also eliminates 30% more PCM writes on average than KG-W, when optimized for extending PCM's lifetime. It consumes 68% less DRAM capacity, when optimized for the smallest DRAM capacity.

Counter-intuitively perhaps, the *static* profile-driven CGZ solution outperforms KG-W's *dynamic* approach. The reason is twofold. (1) Allocation site is a good predictor for write-intensity, i.e., most objects allocated from a single site are either frequently written or read-mostly — we refer to this property as allocation site write homogeneity. (2) A small number of allocation sites captures the bulk of writes to a small fraction of the entire mature space heap volume. The high prediction accuracy for write-intensive objects allocated from a limited number of allocation sites makes CGZ outperform KG-W, which needs to dynamically learn object write-intensity and may place highly written objects in PCM, which it cannot move to DRAM until the next full-heap collection.

Overall, this paper makes the following contributions:

- the design and implementation of profile-driven write-rationing garbage collection for hybrid memories;
- offline profiling, advice generation, and a compilation framework that gathers, generates, and uses allocation advice to trade off PCM lifetime and DRAM capacity;
- emulation results demonstrating reduced execution time overhead compared to state-of-the-art write-rationing garbage collection while at the same time extending PCM lifetime and reducing DRAM capacity needs.

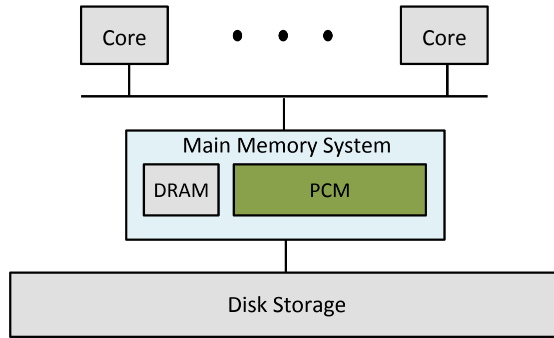


Fig. 1. Our assumed memory and storage hierarchy.

2 BACKGROUND

This section describes background on our assumed memory and storage hierarchy, Java Virtual Machine (JVM), garbage collection, the specific Immix garbage collector on which we build, and prior write-rationing collectors to which we compare.

Storage Hierarchy. PCM can be integrated in the storage hierarchy in different ways owing to its byte-addressable and persistent features. In this paper, we assume PCM is part of the main memory system and place it next to DRAM. The data in DRAM and PCM is backed by disk which acts as the secondary storage. Figure 1 illustrates our assumed memory and storage hierarchy. Using PCM as main memory increases the capacity of the main memory system in the face of DRAM scaling challenges. In this work, we do not exploit the persistent nature of PCM. Others have explored persistent heaps for the Java programming language assuming a storage hierarchy similar to ours [57].

Java Virtual Machine. We use the open-source Jikes Research VM (RVM) as our platform because it combines good performance with software engineering advances that make it easy to modify [4, 24]. 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 [8, 8, 9, 13, 52]. It also offers a wide range of easy-to-change barriers. In particular, we use write barriers, which call a specially-defined method on all writes to do bookkeeping. Reference barriers are widely used in garbage collectors to track pointer references between independently collected regions [58]. We modify them to profile object writes. A clean interface between the compiler and collector [24] defines object layout, references, interior references, and object metadata in a few places. In contrast, changing barriers, object layout, or metadata in the widely-deployed Hotspot system [43, 44] requires numerous wide-ranging changes in the compiler and garbage collection code.

Generational Garbage Collection. High-performance collectors exploit the generational hypothesis that many objects die young by dividing the heap into a *nursery* for newly allocated objects and a mature space for objects that survive a nursery collection [53]. The application (*mutator*) allocates new objects contiguously into a nursery. The JVM keeps track of all pointers from the mature space to nursery objects using a write barrier; Jikes RVM records the source object in a sequential store buffer [58]. When allocation exhausts the nursery memory, a *minor* collection first identifies live *roots* that point into the nursery, e.g., from global variables, the stack, registers, and the recorded mature space objects. It identifies *reachable* objects by tracing references from these roots. It *promotes* these objects to the mature space, copying reachable objects when it first encounters

them to a mature space. It leaves a forwarding pointer in place of the object and updates all the references to previously forwarded objects as it traces. When there are no reachable objects left, it reclaims all nursery memory for subsequent new allocation.

Immix. We build on the best-performing collector in Jikes and in the literature, generational Immix (GenImmix) [13, 52]. During nursery collection, it copies live objects to a *mark-region* mature space and reclaims the entire nursery en-masse. The mark-region space is organized into a hierarchy of coarse-grained blocks, which are multiples of the page size, and further partitioned into fine-grained lines, which are multiples of the cache line size. Objects that survive nursery collection are copied contiguously into free lines, first in partially occupied blocks, then in completely free blocks. Full-heap collections mark live objects, lines, and blocks, and reclaim completely free blocks and completely free lines. Immix uses unsynchronized per-thread allocators that obtain partially full and completely empty blocks from a synchronized global allocator. It uses multiple garbage collection threads with a global work queue. We use Immix' default settings: lines are 256 Bytes, blocks are 32 KB, and large objects (≥ 8 KB) are allocated in a separate non-moving Large Object Space (LOS). Managed heaps treat large objects specially to avoid high copying costs.

Kingsguard collectors. Our previous work proposed two write-rationing garbage collectors: Kingsguard-nursery (KG-N) and Kingsguard-writers (KG-W) [3]. Both collectors allocate nursery objects in DRAM because nursery objects are highly mutated, or written, for two reasons: (1) Java requires zero-initialization [59], (2) applications allocate at high rates [62]. KG-N copies all objects that survive a nursery collection to a PCM mature space to exploit PCM capacity. KG-N drastically reduces the number of PCM writes compared to a system with both the nursery and mature space in PCM. The heap organization in Figure 6 (a) (which is described more elaborately in Section 4.5) is similar to the one used by KG-N.

KG-W reduces PCM writes further by dynamically monitoring writes to objects that survive a nursery collection. It promotes nursery survivors into a DRAM *observer* space. KG-W uses a write barrier that labels written observer space objects. During an observer space collection, KG-W copies written objects to the DRAM mature space and the others to the PCM mature space. KG-W continues to monitor objects in the mature spaces to correct any mispredictions at the next full-heap collection. In addition to observing write behavior, the observer space also gives objects more time to die before promotion to the DRAM or PCM mature space, thus limiting tenured garbage. The heap organization in Figure 6 (b) is similar to the one used by KG-W.

KG-W includes two optimizations for large objects and metadata. The large object optimization (LOO) is triggered if the allocation rate of large objects is higher than the nursery allocation rate. LOO allocates large objects that take up less than half of the remaining nursery size in the nursery and the others directly into LOS in PCM. However, if the LOS objects incur writes, the collector promotes them to a DRAM LOS during the next full-heap collection. The metadata optimization (MDO) stores metadata for objects stored in PCM memory in DRAM to eliminate PCM writes to mark bits during full-heap collections.

KG-W improves PCM lifetime by 50% on average compared to KG-N, making it the most effective write-rationing collector. However, KG-W suffers from some limitations. (1) KG-W uses a write-barrier to continuously monitor object write behavior to determine whether to copy an object into a DRAM or PCM space, which incurs non-negligible execution time overhead (up to 35% for some applications). (2) KG-W is reactive and needs to dynamically learn an object's write behavior in a short time period (while it resides in the observer space) and then promotes objects during the next collection. It occasionally makes mistakes, for example, because it directly allocates most large objects in PCM. Correcting mistakes by copying objects, particularly large objects, between the mature DRAM and PCM spaces is costly and occurs only during infrequent full heap collections.

(3) KG-W uses a large amount of DRAM to protect PCM from writes. Its DRAM usage is partly due to mispredictions for infrequently written objects that end up in DRAM. Unlike CGZ, KG-W has no way to change the amount of DRAM it consumes, as it offers one fixed point in the DRAM capacity versus PCM lifetime spectrum.

3 ALLOCATION SITE AS A WRITE PREDICTOR

This section examines how well allocation site predicts writes to objects and the distribution of writes in heap memory. We use 15 Java workloads, including modern transaction and graph processing workloads with huge memory footprints. Prior work establishes allocation site as an accurate predictor of object lifetime [7, 11, 14, 19, 32]. Our prior work shows that nursery objects incur many writes and thus, to avoid writes to NVM, should be put in DRAM [3]. We show here that allocation site is also a good predictor of the write-intensity of old objects and that these writes are concentrated in a small volume of objects.

Contributions over preliminary work. A preliminary workshop paper studied allocation site's prediction of writes on a subset of DaCapo benchmarks with similar results [1]. The workshop paper also reported preliminary accuracy results, but for self advice (same inputs for training and classification) on a subset of the DaCapo benchmarks. The paper explored the potential of predicting write-intensity based on object size and type, but found that allocation site was the most accurate and input-neutral. We distinguish the contributions of this paper from the workshop paper: (1) We build and report on a novel real-world system that manages hybrid memories, including a garbage collector that automatically acts upon allocation site advice stored on disk. (2) We evaluate Crystal Gazer on a hardware emulation platform using commodity NUMA servers. We report PCM writes and write rates, execution times, DRAM capacity, and insights about which allocation sites get a lot of writes, all new results in this work. (3) The preliminary work did not include real system effects such as CPU caches and an OS, did not include Java virtual machine modifications, emerging graph analytics workloads, a bounded generational heap, allocation site advice, and most importantly, a garbage collector running on top of hybrid memory hardware that places write-intensive objects in DRAM.

Allocation site homogeneity. To assess the predictive power of allocation site for object write-intensity, we measure the homogeneity of writes on the basis of allocation site. We use the information-theoretic notion of entropy to capture write homogeneity. An entropy of 0 means perfect homogeneity, i.e., 100% of objects are highly written or read-mostly. A homogeneity of 1 means no homogeneity, i.e., 50% of objects are highly written and 50% are read-mostly. To compute entropy, we classify objects as highly written if they are written once after allocation in the mature space, and read-mostly otherwise. Figure 2 shows average homogeneity curves for the DaCapo, Pjbb and GraphChi benchmark suites. The percentage of heap volume is reported as a function of allocation site homogeneity. The bottom and top horizontal axes report entropy and the fraction of objects that are homogeneous in write-intensity, respectively. The homogeneity of allocation sites is very high: 40% to 60% of the heap volume is perfectly homogeneous, and 80% to 90% is from sites with very high (at least 90%) homogeneity.

The observation that allocation site is a good predictor of write-intensity motivates a profile-driven approach, i.e., classifying allocation sites through profiling provides a prediction for object write-intensity. However, allocation site write-homogeneity is not enough for a well-performing write-rationing garbage collector. They also are most efficient if the heap volume of write-intensive objects is small, so that we can allocate the least possible volume in DRAM to leverage PCM's capacity to the fullest.

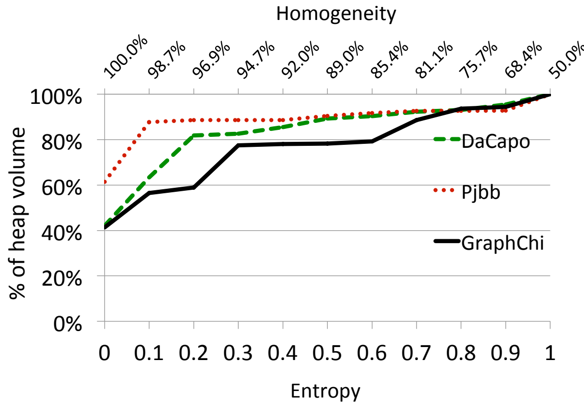


Fig. 2. Write homogeneity for allocation site. *40-60% of the heap volume is allocated from a perfectly homogeneous allocation site; 80-90% of the heap volume is allocated from allocation sites that are at least 90% homogeneous.*

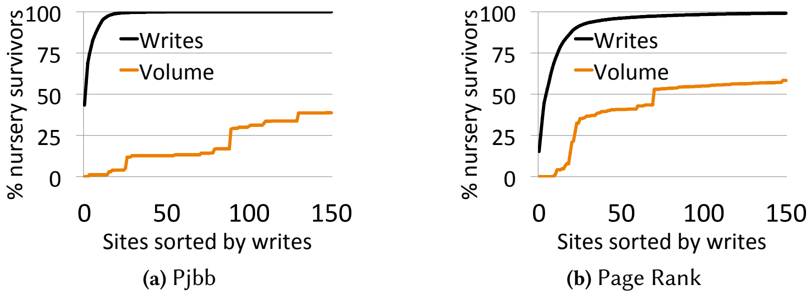


Fig. 3. Distribution of mature writes and heap volume by allocation site for Pjbb and Page Rank. *A few sites capture a majority of mature object writes and occupy a small fraction of the heap.*

Write distribution. Figure 3 shows the cumulative distribution of writes to objects in the mature space and its heap volume as a percentage of the total mature allocation on a per allocation site basis for two representative benchmarks: Pjbb (most homogeneous) and GraphChi's Page Rank (least). We observe that a couple dozen sites out of a couple thousand capture the vast majority of mature writes and constitute only a small fraction of the total heap volume. Similar results hold for all other benchmarks. These two key observations reveal the opportunity that we exploit in Crystal Gazer: profiling accurately identifies sites that allocate a small volume of highly written objects.

4 CRYSTAL GAZER

This section describes Crystal Gazer, profile-driven write-rationing garbage collection for hybrid memories. CGZ uses offline profiling to analyze object write-intensity, from which we generate advice to be used in a subsequent production run. CGZ eliminates the high cost of dynamic monitoring and unnecessary copying compared to KG-W, the previous best write-rationing garbage collector. CGZ achieves a combination of better performance, reduced DRAM usage, and fewer writes to PCM than KG-W.

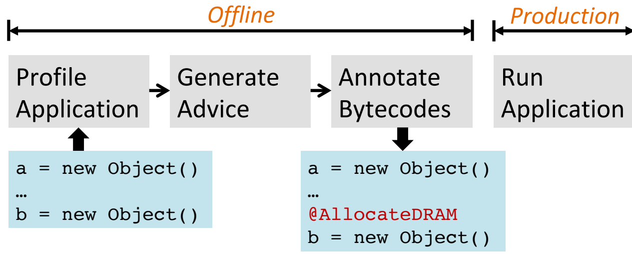


Fig. 4. Overview of Crystal Gazer. Offline analysis identifies allocation sites of highly written objects (e.g., object b) which are annotated in the bytecode. During production, the collector will allocate an object in DRAM if predicted highly written versus PCM if predicted read-mostly, upon a nursery collection.

Object	Writes	Bytes	Method:Idx
O1	0	4	A():10
O2	0	4	A():10
O3	1	4	A():10
O4	1	4	A():10
O5	16	16	A():10
O6	1024	4096	B():4

(a) Example write intensity trace

Heuristic	θ_h	θ_f	θ_d	DRAM Sites
FREQ	5%	1	X	A & B
FREQ	5%	10	X	A & B
FREQ	5%	100	X	B
DENS	5%	X	0.1	A & B
DENS	5%	X	1	A
DENS	5%	X	10	None

(b) Allocation site prediction

Fig. 5. Example of a write-intensity trace with allocation sites in the last column (a) and prediction of allocation sites using the FREQ and DENS heuristics (b).

4.1 Overview

Figure 4 shows the Crystal Gazer work flow. We first profile the application to collect a trace of writes to each object and their allocation sites. We group objects by allocation site and use various heuristics to label allocation sites as *DRAM* (i.e., objects allocated from this site are predicted frequently written) or *PCM* (i.e., objects allocated from this site are predicted read-mostly). Advice files record allocation sites labeled *DRAM*. All other sites are implicitly labeled as *PCM*. Thus, an unprofiled site may be labeled either DRAM or PCM; we default to PCM in this work. For expediency, we use bytecode rewriting to insert a new_dram bytecode based on the profile. The standard portable mechanism is to annotate bytecodes [35], since Java compilers simply ignore unsupported annotations. During a production run, our modified compiler generates a special allocation sequence to process the new_dram bytecode that, in addition to reserving the space, labels objects. Crystal Gazer then promotes objects from DRAM-labeled sites to a mature DRAM space and other objects (expected to be read-mostly) to a mature PCM space.

4.2 Profiling

Our prior work shows that nursery objects plus a small fraction (2%) of all mature objects capture 90% of all application writes [3]. The KG-W write-rationing garbage collector incurs significant overhead to discover the highly written 2% of mature objects. Our offline profiling eliminates this overhead by identifying allocation sites that produce highly written objects in previous executions. Profiling produces a write-intensity trace that records for each object: (1) a unique identifier, (2) the number of writes, (3) its size in bytes, and (4) the allocation site. See Figure 5(a) for an example. The last column lists the object's allocation site as a <method-name:bytecode-index> pair. Because most

objects die young, we only profile mature objects which also reduces the size of the write-intensity trace.

To identify objects, we use mature space addresses. We configure the heap size to use the entire 32-bit virtual address space in Jikes RVM. This setting eliminates full-heap collections for DaCapo benchmarks and SPECjbb. As a result, each object has a unique address in the trace. The GraphChi benchmarks allocate more memory and thus require full-heap collections. In this case, we compute the write-intensity trace per full-heap collection. We process individual traces and combine them to gather allocation advice for the entire application. An alternative option would be to consolidate object statistics on the fly upon full-heap collections.

We use the same nursery size during profiling as we use during a production run. If the production nursery size is unknown, a small (thus conservative) nursery will capture a large fraction of mature objects and their allocation sites in the write-intensity trace. Using small nurseries during profiling produces write-intensity characteristics for more objects, but increases the size of the write-intensity trace.

We use write barriers to count the number of writes to each object. Reference write barriers are required for all generational collectors to collect the nursery independently, to record old to young pointers. We also enable write barriers to primitives during profiling. Profiling ignores zero-initializing writes. Write barriers capture all writes regardless of whether the object or any of its fields are physically in a processor cache or main memory. Our profiling is thus architecture-independent, as is the allocation advice we produce for Crystal Gazer.

During profiling, we label objects with their allocation site at allocation time. We associate each allocation site with a unique identifier which the compiler creates when it first encounters each new bytecode during profiling, following prior work [28]. The compiler generates an allocation sequence that stores this identifier in the header of each object. At the end of program execution, we record the allocation site along with each object's address and other attributes in the write-intensity trace. To correlate allocation sites between executions of an application, the trace records the class, method, and bytecode index of the allocation site.

Collecting a write-intensity trace incurs a 2.4× slowdown over native execution on average according to our measurements. The trace's size ranges between 200 KB and 120 MB after compression for our benchmarks. We did not optimize this overhead further since it is incurred infrequently in non-production runs.

4.3 Allocation Site Classification

Next, we analyze the write-intensity trace to generate allocation advice, classifying allocation sites as *DRAM* versus *PCM*. We use two criteria for classification [1]. (1) The fraction of total objects allocated from a site that are write-intensive. (2) Thresholds that define write-intensive objects. For the first criterion, we use a *write homogeneity* threshold. If the fraction of write-intensive objects allocated from a site is above the write homogeneity threshold (θ_h), we classify the site as *DRAM*. Otherwise, we classify the site as *PCM*. A small homogeneity threshold works best to limit the number of writes to PCM but puts more pressure on DRAM capacity. A high homogeneity threshold reduces DRAM capacity usage at the expense of more PCM writes. For the second criterion that classifies an object as write-intensive, we consider two heuristics.

Write-Frequency (FREQ) uses the frequency of writes to identify write-intensive objects. If an object gets more than a write-frequency threshold θ_f of writes, the object is considered write-intensive.

Write-Density (DENS) uses the ratio of writes to object size (in bytes) to identify write-intensive objects. Objects with a write-density above a write-density threshold θ_d are considered write-intensive. DENS gives higher weight to small objects that collect a relatively large number of writes. DENS prioritizes small objects for DRAM allocation and large objects for PCM allocation, thereby better exploiting PCM's capacity compared to FREQ.

Example. Figure 5(a) shows an example write-intensity trace consisting of 6 objects from two allocation sites: from method A and B. We analyze the trace using the FREQ and DENS heuristics, and identify which of the two sites are classified as *DRAM* in Figure 5(b). We assume a homogeneity threshold of 5%. We increase θ_f from 1 to 100, and θ_d from 0.1 to 10 to observe their impact on site classification. Setting θ_f to 1 classifies both A and B as *DRAM*. Raising θ_f to 100 excludes A from *DRAM* classification because it does not have 5% of objects with more than 100 writes. However, B will still be labeled as *DRAM*. If we want to preserve DRAM capacity by excluding B, we need even larger values for θ_f. On the other hand, if we consider DENS, which uses less DRAM capacity, both A and B are classified as *DRAM* with a θ_d of 0.1. When we increase θ_d to one, only A is classified as *DRAM*. With this setting, objects allocated from B do not get sufficient writes per byte to be classified as *DRAM*. Finally, when θ_d is set to 10, both A and B are excluded from *DRAM* labeling. This example illustrates that large write-density thresholds favor exploiting PCM capacity.

4.4 Bytecode Generation

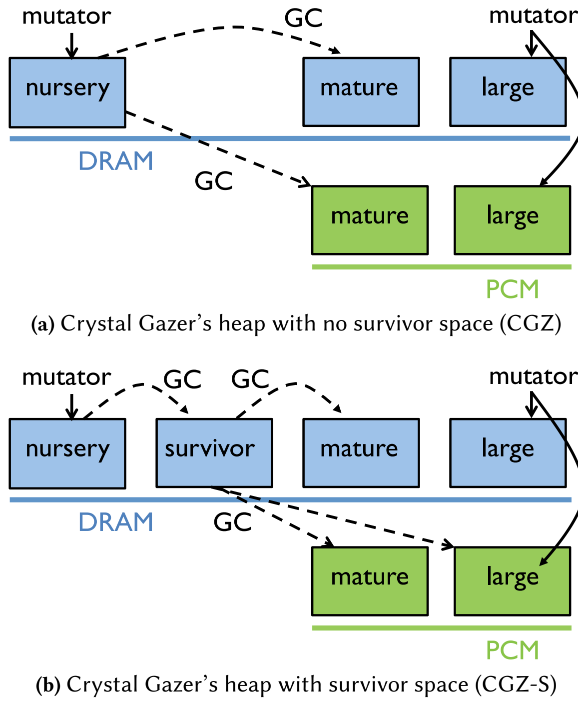
The previous step generates allocation site advice as a file of <site-string, advice> pairs. The advice file only includes the allocation sites labeled *DRAM*. Unlabeled allocation sites default to *PCM*. Future systems could consider labeling unprofiled sites as *DRAM* or dynamically profiling just these objects. Since a minority of allocation sites are labeled *DRAM*, the size of the advice file is minimized.

We use bytecode rewriting to communicate allocation site labels to the managed runtime. The bytecode rewriter first identifies the allocation site, and then queries the advice file to check whether the site is present. If it is not, the rewriter leaves the new bytecode unchanged. If it is, the rewriter overwrites the new bytecode with the newly introduced new_dram bytecode. The runtime, when interpreting or compiling the new bytecode, uses the default allocator, called ALLOC_DEFAULT. The runtime will then copy all objects allocated by such sites to PCM if they survive a nursery collection. For the new_dram bytecode, the runtime uses the newly added ALLOC_DRAM allocator. This allocator sets a bit in the object header which notifies the garbage collector to copy these objects to DRAM if they survive a nursery collection.

4.5 Heap Organization

This section describes our heap organizations and how Crystal Gazer copies and allocates highly written objects in DRAM and read-mostly objects in PCM. We consider two heap organizations, see Figure 6. They are patterned after Kingsguard heap configurations (for KG-N and KG-W, respectively) to compare apples-to-apples with them. Crystal Gazer collectors follow Kingsguard by always placing new objects in a DRAM nursery, because nursery objects are highly mutated. Some large objects, discussed below, are allocated directly in the mature space. We partition the mature and large object spaces into DRAM and PCM regions. We first describe our system using the heap organization in Figure 6(a), and then motivate and describe the heap organization in Figure 6(b).

Fresh allocation is a two-step process: (1) reserving space and (2) initializing the object header, called post-allocation. Objects less than 8 KB are always allocated in the nursery. For nursery objects, post-allocation sets a bit in the object's header if its allocation site is labeled DRAM, as shown in Figure 6. We steal a bit not in use from the object header in Jikes RVM and call it the DRAM_BIT.



(a) Crystal Gazer's heap with no survivor space (CGZ)

(b) Crystal Gazer's heap with survivor space (CGZ-S)

Fig. 6. Heap organizations without and with a survivor space. Objects are allocated in a DRAM nursery and survivor spaces before being promoted to the mature spaces in DRAM and PCM depending on the object's predicted write behavior.

Objects with the DRAM_BIT set are predicted to be highly written. During a nursery collection, the garbage collector checks the DRAM_BIT of each object. If the bit is set, it promotes the object to the mature space in DRAM. Otherwise, it promotes the object, predicted to be read-mostly, to the PCM mature space.

```
1  @Inline
2  public Address postAlloc(ObjectReference ref, int allocator) {
3    if (allocator == Gen.ALLOC_DRAM) {
4      byte old = readHeaderByte(ref);
5      writeHeaderByte(ref, (byte) (old | DRAM_BIT));
6    }
7  }
```

Figure 6. Our post allocation sequence sets a special bit in the header of objects that are predicted highly written.

In the default GenImmix, objects larger than 8 KB are allocated directly into a Large Object Space (LOS). For these objects, Crystal Gazer's (1) ALLOC_DEFAULT allocates the object directly in the LOS PCM space, and (2) ALLOC_DRAM places the object directly in the LOS DRAM space, as depicted in Figure 5(a). Crystal Gazer by default uses both KG-W's metadata optimization and large object optimization (LOO) (see Section 2). With LOO, the allocator places large objects that are

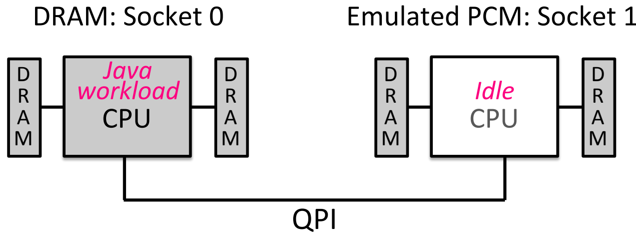


Fig. 7. Our platform for hybrid memory emulation. The workload runs on socket 0; local memory serves as DRAM whereas remote memory on socket 1 emulates PCM.

less than half of the remaining nursery size in the nursery to give them a chance to die, because surprisingly some do die quickly. In this case, the object's DRAM_BIT is set based on the advice, and then consulted during the next minor garbage collection to promote the object to the large object space (LOS) in DRAM or PCM.

Copying nursery survivors directly to the mature space results in tenured garbage because some die quickly in the mature space. Figure 6(b) shows an alternative heap organization with an intermediate space called the *survivor space* between the nursery and the mature spaces. The HotSpot generational collectors use an *eden* space for nursery survivors that serves a similar purpose [20]. The KG-W collector differs because it uses its observer space to monitor writes to these objects as well. In Crystal Gazer, the survivor space's only purpose is to give objects longer time to die and thus limit the amount of tenured garbage. In Crystal Gazer configurations with the survivor space, which we call CGZ-S, nursery survivors are first copied to the survivor space and objects predicted to be highly written carry their DRAM_BIT with them to this space. Next, upon a survivor space collection, the garbage collector checks the DRAM_BIT and copies objects to the mature spaces in DRAM and PCM, accordingly.

5 EXPERIMENTAL METHODOLOGY

This section discusses experimental methodology including the experimental platform, workloads, and the different write-rationing garbage collector configurations that we evaluate.

Java Virtual Machine. We use Jikes RVM v3.1.2, a Java-in-Java VM with a baseline and a just-in-time optimizing compiler (no interpreter) as discussed in Section 2. We use its memory management tool kit (MMTk) to create new collectors by combining and extending existing modules, changing the calls to the C and OS allocators, and adding to its selection of write barriers [58]. We modify Jikes to map DRAM and PCM virtual memory address ranges to the local and remote node in our NUMA emulation platform, as we describe below.

Measurement Methodology. We use best practices from prior work for evaluating Java applications [25, 29], including replay compilation to eliminate non-determinism due to the optimizing compiler. Replay compilation requires two iterations of a Java application in a single experiment. During the first iteration, the VM compiles each method to a pre-determined optimization level recorded in a prior profiling run. We also generate the appropriate allocation sequence (ALLOC_DEFAULT versus ALLOC_DRAM) depending on the site's classification. The second iteration does not recompile methods leading to steady-state behavior. We take our measurements during the second iteration. This run is deterministic and primarily measures the application performance, instead of the adaptive compiler or JVM start-up behavior. We perform each experiment four times and report the arithmetic mean.

Emulation platform. There exists no commercially available platform with a hybrid DRAM-PCM memory system. Although simulation enables evaluating future systems, it is tedious, extremely time-consuming, which limits the software configurations that can be run, and precludes our biggest workloads. Instead, we use emulation which has the advantage of being fast, accurate, and sufficiently flexible. Figure 7 illustrates our multi-socket NUMA emulation platform. We run the workload (application plus JVM threads) on socket 0 and disable the CPUs on socket 1. For hybrid memory emulation, memory allocated from the local socket 0 is local DRAM and memory allocated from the remote socket 1 emulates PCM. We modify Jikes' MMTk to split the virtual heap into DRAM (local socket 0) and PCM (remote socket 1) regions. Each heap region is managed separately. We further modify the interface between Jikes and the OS to specify physical memory allocation from DRAM (local) or PCM (remote). We use the standard NUMA library in Linux to specify local or remote memory.

We use a two socket Intel Sandy Bridge E5-2650L processor. Each socket has 8 physical cores and two hyperthreads per core. The platform features 132 GB of main memory, evenly distributed between the two sockets. We use all DRAM channels on both sockets. All cores share the 20 MB LLC on each processor. The maximum bandwidth to memory is 51.2 GB/s, more than the maximum bandwidth consumed by any of our workloads. The two sockets are connected via a QPI link that supports up to 8 GT/s. We use Ubuntu 12.04.2 with a 3.16.0 kernel. We use Intel's pcm-memory utility from the Performance Counter Monitor framework for measuring write rates.

This emulation platform does not accurately represent end-to-end performance of a hybrid memory system because the access latency to remote memory is much less than to PCM. It does, however, accurately captures the execution time overhead of the Crystal Gazer modifications to the Java managed runtime on real hardware. Most importantly for optimizing lifetime, it accurately represents the number of PCM writes as measured by accesses to remote memory. Furthermore, it accurately represents the use of DRAM capacity with local memory usage. Accessing remote memory incurs a slight performance degradation compared to local memory, which we find to affect performance by approximately 1% on average and up to 6% for Pjbb.[1]

Java Applications. We use 15 Java applications from three diverse sources: 11 from DaCapo [10], pseudojbb2005 (Pjbb) [12], and 3 applications from the GraphChi framework for processing graphs [36]. The GraphChi applications include page rank (PR), connected components (CC) and ALS matrix factorization (ALS). Compared to prior work [3], we drop jython as it does not execute stably with our Jikes configuration. We use an updated version of lusearch, called lu.Fix, that eliminates useless allocation [59]. To match our hardware platform, we run the multithreaded DaCapo applications, Pjbb and GraphChi applications with four application threads. We use the default data sets for profiling with the DaCapo benchmarks; we use 8 warehouses and 10 K transactions for Pjbb; for GraphChi's PR and CC, we process 1 M edges using the LiveJournal online social network [40], and for ALS, we process 1 M ratings from the training set of the Netflix Challenge. Our datasets for production runs differ from profiling runs. For production runs, we use the large data set for DaCapo; 4 warehouses and 50 K transactions for Pjbb; and a different set of randomly chosen 1 M edges and ratings for GraphChi.

Workload Formation. Multiprogrammed workloads better reflect real-world server workloads because: (1) a single application does not always scale with more cores, and (2) servers typically execute multiple programs to amortize costs. Our multiprogrammed workloads consist of four instances of the same application. To avoid non-determinism due to sharing in the OS caches in

[1] This overhead was measured by comparing the performance of Crystal Gazer with the entire heap in remote memory versus local memory on our emulation platform.

multiprogrammed workloads, we use independent copies of the same dataset for the different instances. All four application instances in our multiprogrammed workloads synchronize at a barrier and start the second iteration at the same time. We take the execution time of the longest running instance as the total execution time to run the workload.

Nursery and Heap Sizes. Nursery size affects performance, response time, and space efficiency [5, 8, 54, 62]. We use a nursery of 4 MB for DaCapo and Pjbb. Because a 32 MB nursery improves performance over 4 MB for the GraphChi applications, we use a 32 MB nursery for them. We use a modest heap size that is twice the minimum heap size, reflecting typical production heap sizes and prior work [2, 13, 41, 50, 62]. For all the benchmarks, Table 1 lists the heap sizes, the total allocation in MB, nursery and survivor space survival rates, and other statistics (discussed later).

Garbage Collectors and Configurations. We compare Crystal Gazer against the state-of-the-art Kingsguard KG-N and KG-W collectors. Because KG-N is the most basic design and straightforward to implement for a hybrid memory system, we normalize to KG-N as our baseline. Prior work demonstrated KG-N's efficiency and huge write reductions compared to a PCM-only system [3]. Similar to prior work, we place the stack, and two smaller heap spaces, boot and meta-data, in DRAM. We set the observer space in KG-W, and the survivor space in CGZ-S to twice the size of the nursery. We use two garbage collection threads which is best for Immix [21]. We use a homogeneity threshold of 1% as the default, which we find to be a good compromise between PCM lifetime and DRAM capacity. We present four CGZ configurations: CGZ-F1, CGZ-S-F1, CGZ-D1, and CGZ-S-D1 that vary the optimization goal and include/exclude the survivor space. The CGZ configurations with 'S' include the survivor space, the others do not. We use $\theta_f = 1$ for the FREQ heuristic (denoted 'F1') which minimizes the number of PCM writes, and $\theta_d = 1$ for the DENS heuristic (denoted 'D1'), which minimizes the amount of DRAM capacity.

6 RESULTS

We compare Crystal Gazer configurations to the state-of-the-art Kingsguard collectors along three primary metrics: (1) writes to PCM, (2) DRAM capacity, and (3) performance.

6.1 PCM Writes

Figure 8 reports the number of PCM writes normalized to KG-N. This baseline reduces writes compared to a PCM-only main memory system by 75% (not shown), which results in improved, but still impractical PCM write rates (which are shown in Figure 15). KG-W reduces the number of PCM writes compared to KG-N by 45% on average. Compared to previous work [3], this paper evaluates KG-W for GraphChi applications for the first time. The GraphChi applications, on average, write to PCM more often than other application groups, even with KG-W. This is because they allocate more large objects directly in PCM than other application groups.

CGZ-S-F1 is the most effective configuration in reducing PCM writes. It eliminates 65% and 30% more PCM writes compared to KG-N and KG-W, respectively. CGZ-D1 reduces the number of PCM writes compared to KG-N for the DaCapo benchmarks (by 23%) and Pjbb (by 31%), but not the GraphChi workloads because CGZ-D1 prioritizes small objects, while putting more large objects in PCM. CGZ-F1 does slightly worse than KG-W on average, reducing the number of PCM writes by 35%. CGZ-F1 leads to an increase in PCM writes over KG-W for many DaCapo applications as a result of the lack of a survivor space in front of the mature space as in KG-W. However, CGZ-F1 is more effective than KG-W at eliminating PCM writes for the GraphChi applications because it leverages ahead-of-time information about the write behavior of large objects. Adding a survivor space greatly reduces the number of PCM writes. CGZ-S-D1 reduces the number of PCM writes by 54%. Finally, CGZ-S-F1 eliminates the largest number of PCM writes. Although both CGZ-S-F1

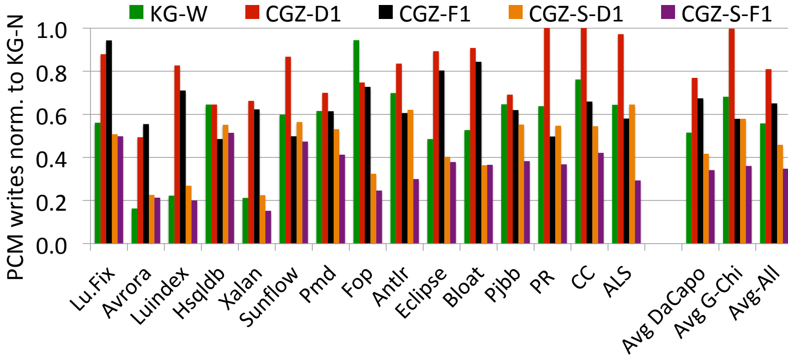


Fig. 8. Number of PCM writes normalized to KG-N. *CGZ reduces the number of PCM writes, especially with a survivor space.*

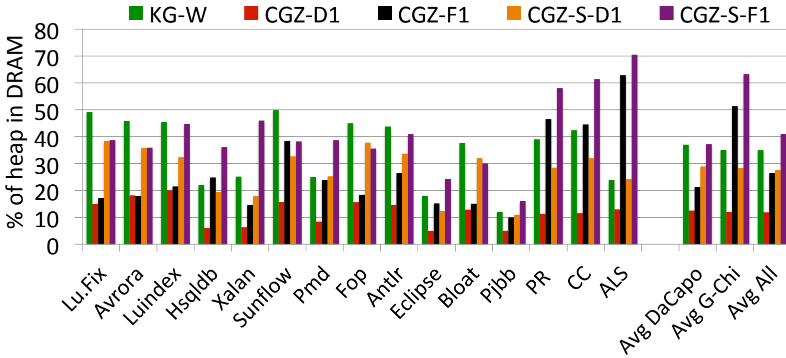


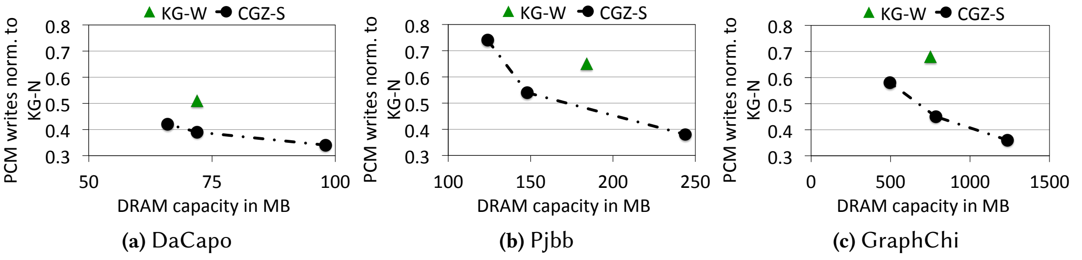
Fig. 9. Fraction of heap allocated in DRAM. *CGZ reduces DRAM capacity needs compared to KG-W, especially without a survivor space.*

and CGZ-S-D1 reduces PCM writes significantly compared to KG-N, CGZ-S-D1 saves more DRAM capacity than CGZ-S-F1 at the expense of PCM writes.

We conclude that Crystal Gazer eliminates a large number of PCM writes compared to KG-N, on par with or significantly surpassing KG-W. These large reductions are a result of ahead-of-time profiling of application for write-intensive allocation sites.

6.2 DRAM Capacity

DRAM will likely be a scarce resource in hybrid DRAM-PCM systems, which motivates minimizing the use of DRAM. Because KG-N stores only newly allocated nursery objects in DRAM, it consumes the least amount of DRAM among the collectors: only 4 MB for DaCapo and Pjbb, and 32 MB for GraphChi. All the other collectors trade reduced writes to PCM for DRAM capacity. Figure 9 shows the percentage of the heap that the CGZ and KG-W collectors allocate in DRAM. We take a snapshot of the heap at every collection cycle and compute the average heap volume (in MB) in DRAM and PCM. KG-W and CGZ-S-F1 allocate the largest fraction of the heap in DRAM: 35% and 41%, respectively. CGZ-S-F1 pays this price to reduce the number of PCM writes the most, as discussed in the previous section. CGZ-S-D1 reduces the use of DRAM, but incurs the space cost of the DRAM survivor space, placing 28% of the heap in DRAM. Finally, CGZ-D1 consumes the least



(a) DaCapo **(b)** Pjbb **(c)** GraphChi

Fig. 10. Pareto-optimal configurations for CGZ-S compared to KG-W in terms of PCM writes versus DRAM capacity. *CGZ provides the flexibility of trading off PCM writes for DRAM capacity and vice versa.*

amount of DRAM, 12% on average (a reduction of 23% compared to KG-W), but incurs more PCM writes. In terms of MB of DRAM consumed, CGZ-D1 only requires 40 MB compared to 132 MB for KG-W on average for our workloads, a reduction of 68%.

6.3 Trading Off PCM Writes and DRAM Capacity

Combining the results in Figure 8 and 9 illustrates how the different CGZ collector configurations trade off fewer PCM writes (Figure 8, left to right decreases) for increases in DRAM usage (Figure 9, left to right increases). The DENS heuristic reduces allocation in DRAM at the cost of an increased number of PCM writes. The FREQ heuristic on the other hand, increases DRAM allocation while reducing the number of PCM writes. Figure 10 visualizes this tradeoff by reporting normalized PCM writes (to KG-N) versus DRAM capacity for a number of Pareto-optimal configurations for CGZ-S by setting different thresholds for the different heuristics on a per-application basis. The key take-away point from these results is that Crystal Gazer offers a set of Pareto-optimal tradeoffs between PCM writes and DRAM capacity, and that KG-W is sub-optimal compared to CGZ-S, i.e., KG-W incurs more PCM writes for the same DRAM capacity and/or requires more DRAM capacity for the same number of PCM writes. Furthermore, KG-W offers no tradeoffs in DRAM capacity versus PCM writes, only offering a single operating point.

We can further configure CGZ collector thresholds to control the PCM write and DRAM capacity tradeoff. To minimize writes to PCM (the right-most points in Figure 10), we set θ_h to 1% and θ_f to 1 for FREQ. For minimum DRAM usage (the left-most points in Figure 10), we set θ_h to 1% for DaCapo and 25% for Pjbb and GraphChi, and use DENS with θ_d set to 1 for all applications. We observe that setting θ_f between 5K and 50K also results in Pareto-optimal configurations. In general, increasing θ_h and θ_f minimizes DRAM usage but increases PCM writes. The best θ_d ranges between 0.1 and 1.

6.4 Allocation Site Analysis

To better understand write-intensive objects, we classify them with the F1 heuristic into four categories based on the location of the allocation site: (1) gnu libraries, (2) Java class libraries, (3) Jikes RVM class files, and (4) application-specific code. Figure 11 shows that application code allocates 58% of write-intensive objects; the Java class libraries allocate 28% of them; Jikes class files 12%, and gnu libraries only 2%. We observe a similar breakdown for D1. Since application-specific code allocates most write-intensive objects, this further motivates profiling applications for write-intensive objects.

Next, we show the reduction in PCM writes by selectively labeling certain allocation sites as DRAM for Pjbb. The goal is to understand which allocation sites result in the largest reductions

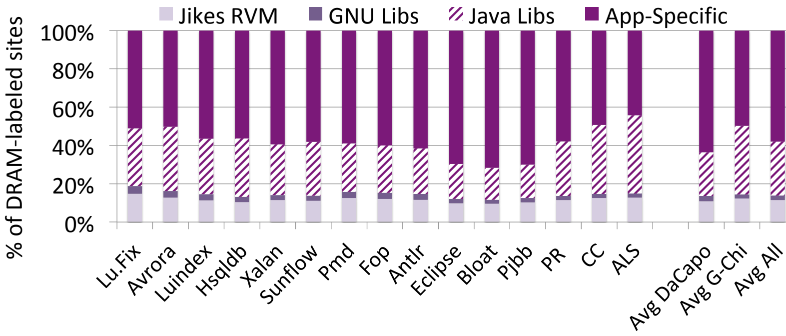


Fig. 11. Breaking down the *DRAM*-labeled allocation sites into four categories: gnu libraries, Java libraries, Jikes RVM class files, and application code. *The application-specific class files allocate the majority of write-intensive objects.*

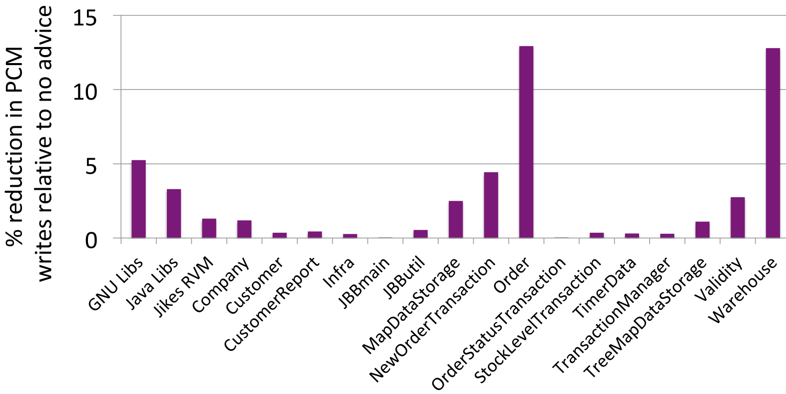


Fig. 12. Understanding the reduction in PCM writes on a per allocation site basis for Pjbb. *Keeping the objects allocated from sites in the application-specific class files,* `Order` *and* `Warehouse`*, leads to the greatest reduction in PCM writes.*

in PCM writes. Figure 12 breaks down the contributions of various allocation sites to PCM write reductions. We report the reduction in PCM writes with CGZ-S-F1 compared to a baseline that uses no advice, i.e., no allocation site is labeled as DRAM. We observe that allocation sites in the application-specific class files are the greatest source of PCM writes, and labeling them as DRAM saves the most writes to PCM. Specifically, gnu libraries, Java class libraries, and Jikes class files, together reduce writes to PCM by up to a maximum of 5%. On the other hand, the two application-specific class files, `Order` and `Warehouse`, each reduce 13% of writes to PCM relative to using no advice at all with CGZ-S-F1.

The results for Pjbb in this section, showing that application-specific class files are the greatest source of PCM writes, are also representative of other benchmarks we evaluate in this work.

6.5 Performance

All hybrid memory systems will incur overhead due to the higher latencies to PCM versus DRAM, which our emulation infrastructure does not accurately capture, but was explored in simulation

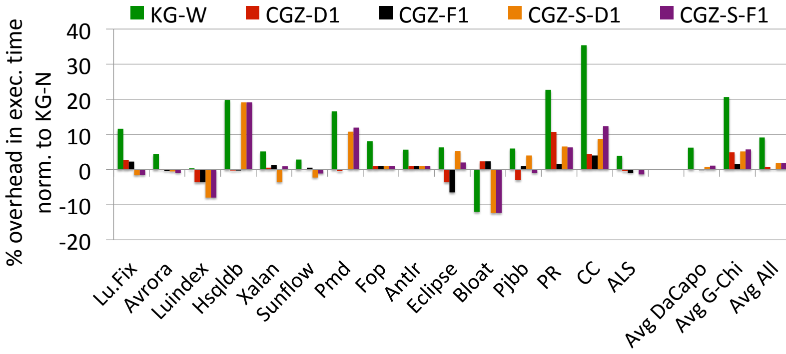


Fig. 13. Execution time normalized to KG-N. *CGZ incurs negligible execution time overhead compared to KG-N.*

previously [3]. This section quantifies the performance overheads in Crystal Gazer compared to previously proposed Kingsguard write-rationing garbage collectors.

Figure 13 reports execution time normalized to KG-N. KG-W incurs an average execution time overhead of 9% over KG-N (and up to 35%). The main reason for this overhead is the extra code KG-W executes in the write barrier to monitor object writes, whereas KG-N has no monitoring. KG-N simply promotes all objects to PCM.

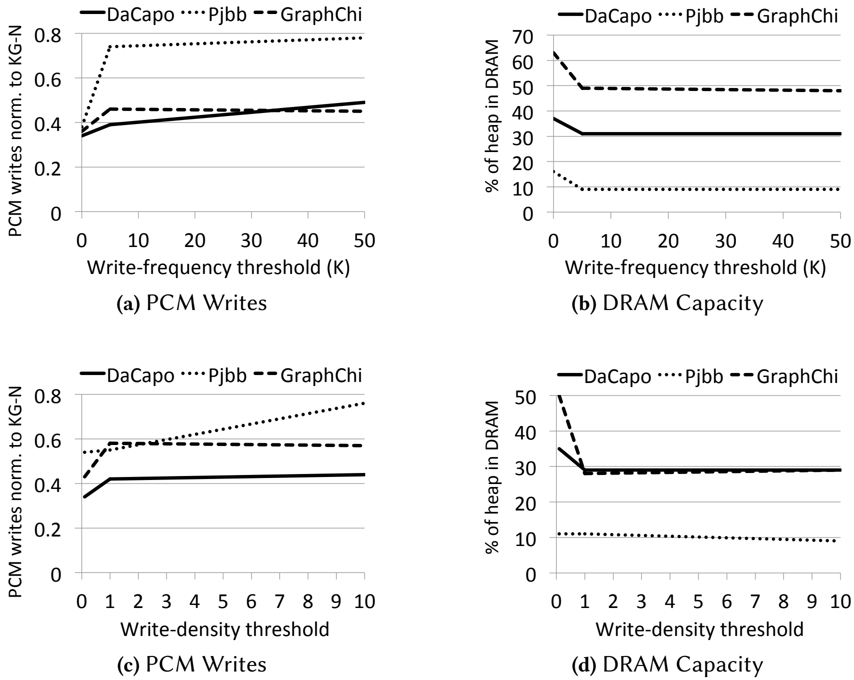
CGZ eliminates all of the overhead of KG-W for most applications. The best CGZ configurations (CGZ-F1 and CGZ-D1) improve performance over KG-W by 8% on average. hsqldb is notably better than KG-W with 20% reduction in execution time for CGZ-F1. CGZ also reduces the execution time of eclipse compared to KG-N by 3%. However, some applications do incur a slight performance degradation as shown in Figure 13. The reasons include: (1) setting the DRAM_BIT for objects predicted as highly written, and (2) checking whether the DRAM_BIT is set during nursery collection. This degradation is particularly prominent for two of the GraphChi applications: PR and CC. We observe only slight performance differences between the F1 and D1 configurations which are the result of a different number of objects being placed in DRAM versus PCM.

CGZ-S also eliminates the monitoring overheads in KG-W, but has higher overhead for some applications (e.g., hsqldb) than CGZ because of the additional survivor space collections. On the other hand, CGZ-S places more objects in DRAM than CGZ. The overhead is limited to less than 2% on average compared to CGZ, putting CGZ-S on par with KG-N. One benchmark, namely bloat, performs better with a survivor space compared to CGZ. Survivors space collections when running bloat preclude full-heap collections by giving objects more time to die in the survivor space. This reduces the total garbage collection time in bloat which translates to an overall performance improvement. Overall, profile-driven garbage collection brings down the high execution time overhead of KG-W to a level similar to KG-N.

6.6 Sensitivity Analyses

Figures 14 (a) and (b) show PCM writes and DRAM capacity as we vary θ_f from 1 to 50K. PCM writes normalized to KG-N increase as we increase θ_f. We observe an increase in PCM writes up to a θ_f of 5K for all three application groups. From 5K to 50K, PCM writes stabilize. Conversely, the percentage of heap in DRAM shows a decrease up to 5K. Beyond that, increasing θ_f does not impact DRAM capacity much.

Figures 14 (c) and (d) show PCM writes and DRAM capacity as we vary θ_d from 0.1 to 10. Increasing θ_d beyond 1 has limited impact on PCM writes normalized to KG-N on average for



(a) PCM Writes (b) DRAM Capacity

(c) PCM Writes (d) DRAM Capacity

Fig. 14. Showing the impact on PCM writes and DRAM capacity from changing the write-frequency threshold (a and b) and write-density threshold (c and d). *Increasing the write-frequency and write-density thresholds increases PCM writes but reduces DRAM space usage.*

DaCapo and GraphChi applications. Individual applications from the two suites sometimes show different trends. Conversely for Pjbb, PCM writes increase linearly as we increase θ_d from 0.1 to 10. The impact on the percentage of heap in DRAM is less prominent for Pjbb. This is because a small percentage of objects are responsible for most writes to the mature heap.

6.7 Memory and Demographic Analysis

Table 1 reports object demographics and shows the average and maximum DRAM usage in MB for KG-W and different CGZ-S configurations for all of the benchmarks considered in this study. The total allocation of our applications varies, between 56 MB and 14 GB of memory (column 1). The GraphChi applications in particular allocate more than DaCapo and Pjbb. The average nursery survival rate of our applications is 17% and a maximum of 66% (column 3). We show the survival rate of objects in the survivor space in CGZ-S in column 4. A number of applications, such as xalan and bloat, benefit greatly from a survivor space, as it gives many objects a chance to die in DRAM. For instance, only 8% of objects in xalan are promoted to the mature space; which means an even smaller percentage of objects are written to PCM.

The remaining columns show the average and maximum DRAM space occupancy in MB for KG-W (column 5 and 6) and six CGZ-S configurations (column 7 through 18). Specifically, we show the DRAM space occupancy for different θ_f and θ_d. We also show the percentage of heap in DRAM across the three benchmark suites and overall for all applications. To calculate the percentage heap in DRAM, we first measure the average DRAM and PCM space occupancy in MB. The sum of the

Table 1. Total allocation, the heap sizes of our applications, the survival rates of nursery and survivor spaces in CGZ-S, and the average and maximum DRAM usage in MB for KG-W and six CGZ-S configurations. The homogeneity threshold is fixed to 1%. Three write-frequency thresholds of 1, 5 K, and 50 K, and three density-thresholds of 0.1, 1, and 10, are used to show the DRAM space occupancy of CGZ-S.

	allocation MB	Heap MB	CGZ-S nursery survival %	CGZ-S survivor survival %	DRAM KG-W		DRAM $\theta_f = 1$ $\theta_h = 1\%$		DRAM $\theta_f = 5K$ $\theta_h = 1\%$		DRAM $\theta_f = 50K$ $\theta_h = 1\%$		DRAM $\theta_d = 0.1$ $\theta_h = 1\%$		DRAM $\theta_d = 1$ $\theta_h = 1\%$		DRAM $\theta_d = 10$ $\theta_h = 1\%$	
	(1)	(2)	(3)	(4)	avg (5)	max (6)	avg (7)	max (8)	avg (9)	max (10)	avg (11)	max (12)	avg (13)	max (14)	avg (15)	max (16)	avg (17)	max (18)
Lusearch	4294	68	4%	29%	8	11	8	10	7	10	7	10	7	10	7	10	8	10
Lu.Fix	848	68	2%	25%	11	15	9	14	9	14	9	14	9	14	9	14	9	14
Avrora	64	98	15%	0%	12	16	10	14	10	14	10	14	10	14	10	14	10	14
Luindex	37	44	22%	0%	12	14	12	14	11	13	11	13	12	14	9	11	9	12
Hsqldb	165	254	60%	88%	16	22	27	37	15	21	15	21	27	37	14	21	14	20
Xalan	980	108	14%	9%	14	19	29	58	16	27	11	17	20	37	11	17	11	17
Sunflow	1920	108	2%	13%	12	16	10	17	9	14	9	15	10	17	9	14	9	14
Pmd	364	98	23%	68%	19	27	22	32	15	24	16	24	16	23	15	22	16	25
Pmd.S	202	98	27%	47%	14	19	16	25	13	20	13	21	13	21	12	19	13	21
Fop	56	80	20%	82%	12	16	11	15	12	15	12	16	11	15	12	15	11	15
Antlr	246	48	15%	0.16%	9	14	11	16	11	16	12	17	12	17	9	14	9	14
Eclipse	3082	160	14%	37%	19	25	29	51	18	26	17	24	22	34	15	22	15	21
Bloat	1246	66	4%	19%	12	16	11	16	11	16	11	16	11	16	12	17	11	16
Avg DaCapo	1040	100	17%	32%	13	18	16	25	12	18	12	79	14	21	11	16	11	16
Heap %					30%		35%		30%		30%		33%		28%		28%	
Pjbb	2314	400	20%	84%	32	46	40	61	23	31	23	52	28	37	28	37	21	31
Heap %					12%		16%		9%		9%		11%		11%		9%	
PR	6946	512	36%	99%	97	225	140	321	88	174	90	177	96	202	73	136	72	138
CC	5507	512	24%	97%	99	225	134	347	93	180	92	169	91	185	72	135	72	129
ALS	14245	512	10%	63%	66	113	191	260	182	241	180	242	187	249	65	108	65	110
Avg GraphChi	9000	512	23%	86%	87	188	155	309	121	198	121	196	125	212	70	126	70	126
Heap %					35%		63%		49%		48%		50%		28%		29%	
Avg All	2900	206	17%	46%	27	49	42	77	32	50	32	37	34	55	22	37	22	37
Heap %					30%		46%		34%		34%		37%		24%		24%	

two spaces (MB) is the total heap used by the application. The percentage of total heap in DRAM is shown in Table 1 as Heap %.

We observe that CGZ-S-D10 maximizes the use of PCM for many applications. Conversely, CGZ-S-F1 maximizes the use of DRAM to eliminate the largest number of writes to PCM. In particular for GraphChi applications, CGZ-S-F1 places more than 200 MB per application instance in DRAM. Thus, by prioritizing small objects, the density heuristic minimizes DRAM usage for modern graph analytic workloads, with large heaps and high allocation rates.

6.8 PCM Lifetime and Write Rates

This section analyzes the PCM write rate results and their implications for PCM lifetime. Eliminating PCM writes improves PCM lifetime. PCM lifetime in years depends on its write rate and cell endurance. Prototype PCM has a cell endurance between 10 M and 100 M writes per cell [6, 37]. All of our results assume hardware wear-leveling is enabled. As writes to PCM reduce, so does the write rate, but the write rate is also inversely proportional to execution time. Optimizing only for write rate would thus lead to incorrect conclusions. As an example, turning off compiler optimizations to make the program execute slower would decrease write rate. Because execution time depends on a number of factors including on-chip cache sizes, number of threads, cores, and garbage collection algorithm, it is not meaningful to directly compare normalized write rates.

Figure 15 shows the absolute PCM write rates in MB/s that we observe on our emulation platform. Since PCM hardware is still evolving, the write rates on future PCM hardware may differ from our measurements. Recent work uses a real PCM prototype to evaluate hybrid memory at Facebook [23].

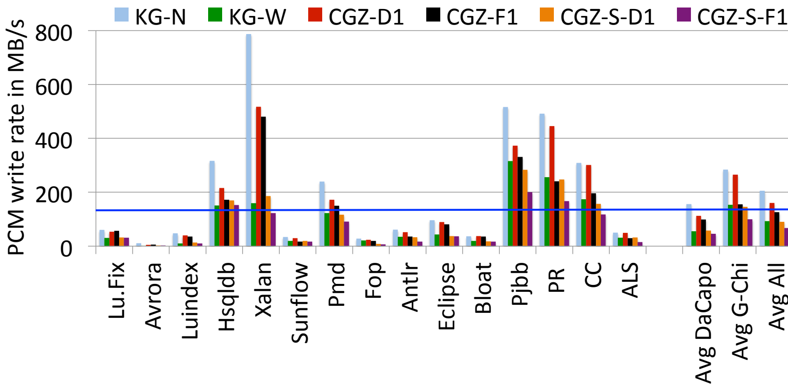


Fig. 15. PCM write rates in MB/s for all of our benchmarks using various write-rationing garbage collectors. *Profile-driven write-rationing garbage collection makes PCM a practical DRAM replacement by significantly reducing its write rates.*

Their work shows that hardware vendors limit the number of times the entire PCM memory (or drive) can be written per day. The specific metric is called drive writes per day (DWPD). The most recently reported DWPD for a 375 GB NVM drive is 30 [23, 26]. This DWPD results in a recommended write rate of 140 MB/s (blue horizontal line in Figure 15). We observe in Figure 15 that all write-rationing collectors significantly reduce the write rates and many are brought below the recommended rate to make PCM practical as main memory. In particular, CGZ-S-F1 limits the PCM write rates of all but 3 workloads to below the recommended rate, while improving the performance over KG-W at the same time.

We observe that many workloads write at a rate that is still not practical for PCM. For instance, two of the graph applications have write rates above 140 MB/s, even with CGZ-S-F1. Furthermore, future servers with more cores will likely run many more applications in parallel, which will result in even higher write rates. This necessitates more research in software approaches to bring write rates down even further. Some reduction will come from innovations at the device and architecture level, or from using hybrid memories where some writes could be guided to DRAM. Due to the nature of PCM material, software has a greater role to play in making PCM practical as (part of) main memory.

6.9 Threats to Validity and Future Work

PCM technology trends. Non-volatile memory technologies are still evolving, and therefore, their write endurance levels may improve. Nevertheless, there is considerable evidence from material physics and industrial products that PCM cell endurance will not reach DRAM levels. However, other promising memories in advanced stages of production, such as various resistive memories, have a higher write endurance. In particular, memristor-based resistive memories can endure up to a trillion writes per cell [39].

The work on write-rationing garbage collection is also relevant for resistive and other promising memory technologies. Write operations have drawbacks other than limiting memory lifetime: they are slower and consume a lot of energy. There also exists a tradeoff between write latency and energy [60]. Faster write operations consume more power because the switching speed depends on temperature. Slower writes stall the processor pipeline and hurt the performance of applications. Therefore, software approaches to minimize writes to non-volatile memories such as Crystal Gazer are interesting avenues of optimization for upcoming non-volatile memory technologies.

PCM latency and emulation. This work targets PCM write endurance. Another disadvantage of the PCM technology is its high access latency. In this work, we are concerned with the reduction in PCM writes and overheads of Crystal Gazer, and also improving upon Kingsguard. Therefore, PCM access latency does not affect the conclusions of this work. Regardless, we made an effort to accurately model PCM access latency on our emulation platform. Our solution to introduce interference in the remote socket to slow down remote (PCM) memory accesses lead to non-determinism in the execution time results, and thus we removed it for the final experiments. Instead of using a simulator to model PCM latency, we believe emulation is a more valuable way to do the evaluation of hybrid memories. Emulation includes real system effects (advanced caching, prefetching, memory bandwidth resource contention, etc.) that no simulator can accurately model and lets us explore many more software configurations and bigger workloads in the same resource budget.

Crystal Gazer could be extended to better tolerate PCM's high access latencies. One future work could be to not only guide highly written objects away from PCM, but also to find highly accessed objects and place them in DRAM to reduce their access latency. In this way, we could use garbage collection to fight both of the drawbacks of PCM, resulting in a high-performance hybrid memory system with a long lifetime.

Profiling weaknesses. Our profiling in this work is architecture independent, i.e., in the profiling step we do not track whether the write operation hits or misses in the cache. Therefore, our classification of allocation sites as DRAM is conservative. A DRAM-labeled site that allocates objects which have high cache locality needlessly wastes DRAM space as writes to these objects hit in the cache. Unfortunately, current hardware precludes correlating allocation sites to cache misses, making it difficult to incorporate cache effects in the profiling step. Nevertheless, our PCM writes and execution time results with Crystal Gazer increase our confidence in the accuracy of our current profiling approach.

In our evaluation, we use different input datasets for training and production experiments. The benchmarks from the DaCapo suite have default and large input datasets. For Pjbb and GraphChi, we experiment with a range of newly created datasets for production runs. We show results with one dataset. Nevertheless, the results depend on the specific dataset and application, and whether profiling correctly predicts the allocation sites' write-intensity.

Our Crystal Gazer collectors by default promote objects that outlive a nursery or a survivor collection to PCM. If advice is not available, or highly written objects are promoted to PCM due to misclassification of allocation sites, our Crystal Gazer collectors lack a dynamic approach to move objects away from PCM. When advice is not available, KG-W is the best approach. Future work should also consider dynamic call site write-prediction, similar to dynamic call site lifetime-prediction [32]. Production systems will require some dynamic monitoring of PCM objects or memory, perhaps by the OS, to recover from changes in behavior, bad predictions, and malicious write attacks.

Optimizing CGZ-S. Our CGZ-S collector performs two copies of long-lived objects, from the nursery to the survivor space, and from the survivor space to either DRAM or PCM. This extra copying has two disadvantages: (1) it hurts performance, and (2) copying necessitates updating PCM object references to copied objects leading to PCM writes. Prior work proposes allocation site based prediction of object lifetimes for Java workloads [11, 14]. Object lifetime prediction used with pretenuring can eliminate the extra copying of long-lived objects. Specifically, the mutator can directly allocate objects from allocation sites that are classified as long-lived in the mature DRAM or PCM. The resulting system is likely to use the DRAM and PCM spaces more efficiently. Another

promising avenue to eliminate the extra copying in CGZ-S is to explore non-moving variants of Immix [13].

7 RELATED WORK

This section discusses related work on managing hybrid memories and profile-based optimizations for Java workloads.

Hardware and OS support for hybrid memory. On the hardware and OS side, the two main approaches to mitigate PCM wear-out are hardware wear-leveling [46, 47, 51] and OS write limiting [49, 61]. Wear-leveling spreads writes out across the entire PCM capacity to mitigate wear-out at the granularity of writes (256 B) within a page and pages (4 KB). Unfortunately, recent work reports that Java applications will wear out a PCM-only system in less than 5 years [3]. OS techniques face several limitations: (1) they react to writes, thus delaying page migrations after PCM writes; (2) page migrations cause TLB shootdowns which degrade performance; and (3) they operate at a coarse-grain granularity, whereas write-intensity varies at the byte and object granularity.

Hardware and software both have a role to play in mitigating PCM wear-out. Hardware wear-leveling is effective in making writes uniform and its implementation cost is low [46]. For native C and C++ applications, the OS can help migrate highly written pages to DRAM. Write-rationing garbage collectors complement hardware and OS techniques by significantly reducing wear-out for managed workloads [3].

Hybrid memory management and garbage collection. Write-rationing garbage collectors have a large advantage over OS and hardware only approaches, because they operate on individual objects[3]. The best write-rationing garbage collector (KG-W) requires dynamic monitoring of object writes. This approach faces three major drawbacks: (1) continuously monitoring the write behavior of objects using write barriers incurs significant execution time overhead; (2) its reactivity, monitoring objects for a short time and expecting that to predict the future, leads to mispredictions and allocation of highly written objects in PCM, limiting PCM lifetime; and (3) it consumes excessive DRAM capacity. To overcome these shortcomings, CGZ collectors statically classify allocation sites as *DRAM* versus *PCM*. The garbage collector then allocates and promotes objects to DRAM or PCM based on the object's allocation site. Crystal Gazer has low overhead, is proactive, and can offer tradeoffs between DRAM capacity and PCM lifetime based on different heuristics.

Wang et al. [55] tackle increased PCM read latency but neglect wear-out. They use offline profiling to classify hot and cold methods by their execution frequency. Their approach allocates objects from hot methods in DRAM to improve performance, and allocates the remaining objects in PCM. In contrast, CGZ collectors improve PCM lifetimes.

Wei et al. [56] explore read and write predictability for allocation sites in native C applications. C semantics limit the applicability of their approach, and their applications allocate a majority of the heap from a small number of (up to 4) sites. Our work considers popular Java applications with tens of thousands of allocation sites. This prior work migrates coarse-grained pages between DRAM and NVM which is inefficient and degrades performance.

Profile-driven DRAM memory management. Prior works use allocation sites to optimize the performance and energy of DRAM-based memory systems. Jantz et al. [31] use offline profiling to divide allocation sites into hot and cold sites. The heap is partitioned into hot and cold regions, and the OS maps virtual to physical pages with the goal to reduce DRAM energy consumption without hurting performance. In contrast, we focus on memory lifetimes in hybrid DRAM-PCM memories. Our work is the first to show allocation site homogeneity of writes, and a garbage collector that acts upon allocation site advice to place highly written objects in DRAM. Recent

work also exploits allocation sites to place program data in upcoming memory systems with high-bandwidth DRAM placed next to traditional DRAM [22]. Their work is limited by C++ semantics. In contrast to our work, they do not use a real-world hardware prototype to evaluate their data placement strategies and hence, among other limitations, are unable to report the total execution time of their applications.

Profile-driven pretenuring. Prior work uses allocation sites to predict object lifetimes and allocate long-lived objects directly in a mature space (pretenuring), eliminating nursery promotion costs. We follow the same approach of ahead-of-time profiling and then applying advice in production runs as in Blackburn et al. [11, 14], but based on writes instead of lifetime. Dynamic lifetime profiling has the advantages that it does not require a profile and can react to program phases, but the disadvantage of dynamic monitoring costs and warm up time [32]. In our work, advice predicts write-intensity and the collector allocates objects in DRAM or PCM, both at allocation (large objects) and promotion (small objects) time. We find allocation site better predicts write-intensity than lifetime and believe combining both predictions, and some dynamic monitoring, are interesting avenues for future work.

Other profile-driven optimizations. Krintz et al. [35] profile Java applications offline to discover compiler optimizations that speed up a method's execution. They annotate the bytecode to communicate these optimizations to the compiler which reduces compilation overhead during run-time. In subsequent work, Krintz [34] combines offline and online profiling to reduce compilation overhead even further. Profiling has been used to improve memory management in Java workloads. Buytaert et al. [17] collect information offline about when to trigger garbage collection to maximize collection yield. They also use offline analysis to decide, during run-time, between triggering nursery or full-heap collections. Chen et al. [18] leverage profile information to proactively reorganize the heap to improve data locality.

8 CONCLUSIONS

This paper demonstrates that profile-driven write-rationing garbage collection can improve PCM's lifetime in hybrid memories while limiting consumption of DRAM capacity. Crystal Gazer overcomes the shortcomings of the prior state-of-the-art write-rationing garbage collector, KG-W, which dynamically monitors writes to nursery survivors to decide whether to promote to mature DRAM or mature PCM. Crystal Gazer improves accuracy and reduces the cost of write-rationing garbage collection by predicting object write-intensity based on offline allocation site profiling. It copies nursery survivors to mature DRAM if predicted highly written or to mature PCM if predicted read-mostly. Using a survivor space in-between the nursery and mature space further reduces the number of writes to PCM at the cost of increasing DRAM capacity consumption. Because allocation site prediction of write-intensity is highly accurate, our static technique out-performs the dynamic techniques used by the Kingsguard KG-W collector. We demonstrate that Crystal Gazer provides Pareto-optimal operating points in terms of PCM lifetime and DRAM capacity by changing the heuristics and thresholds to classify allocation sites. Our experimental results use emulation on real hardware and show that Crystal Gazer significantly improves performance compared to the state-of-the-art, KG-W, while reducing the number of PCM writes when optimized for PCM lifetime, and requiring less DRAM capacity when optimized for the smallest DRAM capacity. Profile-driven write-rationing garbage collection makes PCM a practical DRAM replacement by aggressively reducing writes to it in a hybrid memory setting. It requires minimal OS support and enables the use of PCM for commodity applications without changes to the programming language or model.

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, Kathryn S. McKinley, Jennifer B. Sartor, and Lieven Eeckhout. 2018. Managing Hybrid Memories by Predicting Object Write Intensity. In *Proceedings of the Conference Companion of the International Conference on Art, Science, and Engineering of Programming (Programming'18 Companion)*.

[2] 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).

[3] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-rationing Garbage Collection for Hybrid Memories. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[4] 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).

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

[6] 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).

[7] David A. Barrett and Benjamin G. Zorn. 1993. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[8] 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)*.

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

[10] 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 Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[11] Stephen M. Blackburn, Matthew Hertz, Kathryn S. McKinley, J. Eliot B. Moss, and Ting Yang. 2007. Profile-based Pretenuring. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 1.

[12] 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

[13] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[14] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. 2001. Pretenuring for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[15] 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 ACM Conference on Computing Frontiers (CF)*.

[16] 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).

[17] Dries Buytaert, Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. GCH: Hints for Triggering Garbage Collections. *Trans. HiPEAC* 1 (2007), 74–94.

[18] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. 2006. Profile-guided Proactive Garbage Collection for Locality Optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[19] Perry Cheng, Robert Harper, and Peter Lee. 1998. Generational Stack Collection and Profile-driven Pretenuring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

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

[21] 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)*.

[22] T. Chad Effler, Adam P. Howard, Tong Zhou, Michael R. Jantz, Kshitij A. Doshi, and Prasad A. Kulkarni. 2018. On Automated Feedback-Driven Data Placement in Multi-tiered Memory. In *Architecture of Computing Systems (ARCS)*, Mladen Berekovic, Rainer Buchty, Heiko Hamann, Dirk Koch, and Thilo Pionteck (Eds.).

[23] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the European Conference on Computer Systems (EuroSys)*.

[24] 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)*.

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

[26] Jim Handy. 2017. Examining 3D XPoint's 1,000 Times Endurance Benefit. https://thememoryguy.com/examining-3d-xpoints-1000-times-endurance-benefit/

[27] Joel Hruska. 2018. Why RAM Prices Are Through the Roof. https://www.extremetech.com/computing/263031-ram-prices-roof-stuck-way

[28] Jipeng Huang and Michael D. Bond. 2013. Efficient Context Sensitivity for Dynamic Analyses via Calling Context Uptrees and Customized Memory Management. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*.

[29] 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)*.

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

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

[32] Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. 2004. Dynamic Object Sampling for Pretenuring. In *Proceedings of the International Symposium on Memory Management (ISMM)*.

[33] Patrick Kennedy. 2018. Why Server ASPs Are Rising the 2017-2018 DDR4 DRAM Shortage. https://www.servethehome.com/why-server-asps-are-rising-the-2017-2018-ddr4-dram-shortage/

[34] Chandra Krintz. 2003. Coupling On-line and Off-line Profile Information to Improve Program Performance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.

[35] Chandra Krintz and Brad Calder. 2001. Using Annotations to Reduce Dynamic Optimization Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

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

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

[38] 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).

[39] 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).

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

[41] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-performance Big-data-friendly Garbage Collector. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

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

[43] OpenJDK Group. 2019. Hotspot VM. http://openjdk.java.net/groups/hotspot/

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

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

[46] 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 IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[47] 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 *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[48] 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 International Symposium on Computer Architecture (ISCA).*

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

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

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

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

[53] David Ungar. 1984. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE).*

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

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

[56] 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).*

[57] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2018. Espresso: Brewing Java For More Non-Volatility with Non-volatile Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

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

[59] 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).*

[60] 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 International Symposium on Computer Architecture (ISCA).*

[61] 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 International Conference on Parallel Architectures and Compilation Techniques (PACT).*

[62] 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 ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA).*

Received December 2018; revised January 2019; accepted February 2019