Emulating and Evaluating Hybrid Memory for Managed Languages on NUMA Hardware

Shoaib Akram
Ghent University

Jennifer B. Sartor
Ghent University
Vrije Universiteit Brussel

Kathryn S. McKinley
Google

Lieven Eeckhout
Ghent University

***Abstract*—Non-volatile memory (NVM) has the potential to become a mainstream memory technology and challenge DRAM. Researchers evaluating the speed, endurance, and abstractions of hybrid memories with DRAM and NVM typically use simulation, making it easy to evaluate the impact of different hardware technologies and parameters. Simulation is, however, extremely slow, limiting the applications and datasets in the evaluation. Simulation also precludes critical workloads, especially those written in managed languages such as Java and C#. Good methodology embraces a variety of techniques for evaluating new ideas, expanding the experimental scope, and uncovering new insights.**

This paper introduces a platform to emulate hybrid memory for managed languages using commodity NUMA servers. Emulation complements simulation but offers richer software experimentation. We use a thread-local socket to emulate DRAM and a remote socket to emulate NVM. We use standard C library routines to allocate heap memory on the DRAM and NVM sockets for use with explicit memory management or garbage collection. We evaluate the emulator using various configurations of write-rationing garbage collectors that improve NVM lifetimes by limiting writes to NVM, using 15 applications and various datasets and workload configurations. We show emulation and simulation confirm each other's trends in terms of writes to NVM for different software configurations, increasing our confidence in predicting future system effects. Emulation brings novel insights, such as the non-linear effects of multi-programmed workloads on NVM writes, and that Java applications write significantly more than their C++ equivalents. We make our software infrastructure publicly available to advance the evaluation of novel memory management schemes on hybrid memories.

I. Introduction

Recent advances in memory technologies have the potential to disrupt the boundary between memory and storage. Emerging non-volatile memory (NVM) technologies have access speeds closer to DRAM and persistence similar to disk. On the main memory side, NVM promises abundant memory to address the scaling problems of DRAM [29], [33]. One of the most promising NVM technologies is phase-change memory (PCM). The disadvantages of PCM are that: (1) write endurance is limited, and (2) latency is high. Recent work combines DRAM and PCM to form hybrid main memories seeking benefits from both technologies [26], [40]. DRAM is fast and has high endurance whereas PCM is dense and has low energy consumption. Hardware mitigates PCM wear-out using wear-leveling and other approaches [26], [38], [39], [40], [45], while the OS keeps frequently accessed data in DRAM [28], [30], [36], [41], [53], [57]. Recent work also explores using managed runtimes to mitigate wear-out [2], [3], tolerate faults [20], and keep frequently read objects in DRAM [52]. Collectively, prior research illustrates opportunities to exploit PCM as main memory, tolerating its deficiencies at many levels of the system stack.

According to analysts and scientists, technology improvements may eventually bridge the DRAM versus NVM gap in access latency. More specifically, the ITRS 2.0 road map reports that the latency of *performance-focused* variants of emerging NVM is much closer to DRAM than earlier *cost-focused* NVM products [24]. Furthermore, scientists recently demonstrated PCM prototypes with a thousandfold lower latency than DRAM [56]. This big (theoretical) gap is unrealistic to attain because faster PCM accesses require higher temperatures. Nevertheless, latencies similar to DRAM seem realistic in the future. Endurance, on the other hand, is a much harder problem because writes change the material form of PCM cells causing them to wear out [13].

Most prior work evaluating PCM as main memory uses simulation [26], [28], [38], [39], [40], [41], [45], [53], [57]. The advantages of simulation are: (1) it eases modeling of new hardware features, and (2) it reveals sensitivity to architectural parameters. Its limitations include: (1) it is many orders of magnitude slower than real hardware, (2) it narrows the scope of application domains, datasets, and implementation languages, owing to limited time frames, and (3) frequent hardware changes, microarchitecture complexity, and hardware's proprietary nature make it difficult to faithfully model real hardware. Because of these limitations, researchers recently have complemented simulation with architecture-independent measurements [2], [3], [50]. Unfortunately, these measurements have limited value because they miss the important effect of CPU caching when evaluating hybrid memory.

To overcome the above challenges, prior work proposes emulation platforms for hybrid memories [31], [51]. Their limitations are: (1) they focus only on native applications written in C and C++, and (2) they focus only on emulating the latency of PCM in hybrid DRAM-PCM systems, neglecting write analysis and thus PCM wear-out issues. This paper proposes a new emulation platform for hybrid memories, significantly expanding the scope of applications that we can evaluate. More specifically, we target managed languages because they are popular among programmers today [47]. We focus on accurately measuring PCM writes in hybrid memories, because PCM writes dictate its lifetime (in years). Assuming perfect wear-leveling, PCM lifetime is inversely proportional to the number of writes per second. Reliable measurements of PCM writes evaluate its practicality as main memory.

We present the design, implementation, and evaluation of an emulation platform that uses widely available commodity NUMA server hardware to model hybrid DRAM-PCM systems. We use the local socket to emulate DRAM and the remote socket to emulate PCM. The applications and the runtime environment execute on the DRAM socket. Our new layout for managed heaps splits virtual memory into separate DRAM and PCM regions, which are exposed to the garbage collector that manages them using two free lists. The garbage collector tells the OS where in memory (on which NUMA node) to map heap regions.

Contrary to most prior work, our newly proposed emulation platform supports both applications written in native programming languages such as C and C++, which use manual memory management, and managed programming languages such as Java, C#, Python, and JavaScript, which use automatic memory management. Although we focus on the Java runtime environment in this paper, our work generalizes to other managed languages.

We use the emulation platform to evaluate a rich set of managed workloads written in Java running on top of hybrid memory. We form workloads from three diverse benchmark suites: (1) 11 DaCapo applications, (2) Pjbb, and (3) three graph processing applications from the GraphChi framework. Furthermore, we use two input datasets, seven garbage collector configurations; and multiprogrammed workloads consisting of one, two and four application instances executing simultaneously. We use our new emulation platform to evaluate recently proposed write-rationing garbage collectors for hybrid memories [2]. Write-rationing collectors keep highly mutated objects in DRAM to improve PCM lifetime, while putting read-mostly objects in PCM to exploit its capacity.

We focus primarily on observing PCM writes in hybrid DRAM-PCM systems because they determine PCM lifetime and thus its practicality as a viable DRAM replacement. We compare prior results on PCM writes from simulation to our emulation results, showing that they agree. Emulation enables us to generate a lot more results, faster, and to explore much richer software configurations and workloads. For example, we compare the PCM writes of equivalent C++ and Java applications, showing that Java applications allocate a lot more, leading to more writes to memory, which would quickly wear out a PCM-Only main memory system.

Our software infrastructure is publicly available to help researchers evaluate hybrid memory for managed language workloads. We summarize our key findings below:

- Simulation and emulation reveal similar trends in the reduction of PCM writes with write-rationing garbage collectors for hybrid memories. This finding increases our confidence in both simulation and emulation to evaluate hybrid memories.
- Java graph processing applications allocate far more than their equivalent C++ applications and write to memory up to $3\times$ more.
- Executing multiple instances of an application often super-linearly increases the number of writes to PCM due to LLC interference. Previously proposed write-rationing garbage collectors are shown to be especially effective at taming PCM writes in multiprogrammed environments.
- Emerging graph processing applications in Java use larger heaps, allocate more large objects, and thus write more to PCM, wearing it out faster, than traditional Java benchmarks. Future work should therefore include a diverse mix of workloads and production datasets when evaluating hybrid memories.

II. Background

We first briefly describe the characteristics of PCM hardware and the role of DRAM in hybrid DRAM-PCM systems. We then discuss write-rationing garbage collection [2] that protects PCM from writes and prolongs PCM lifetime.

A. PCM and Hybrid Memory

A promising non-volatile memory technology that is currently in production is phase change memory (PCM) [32]. PCM cells store information as the change in resistance of a chalcogenide material [13]. During a write operation, electric current heats PCM cells to high temperatures and the cells cool down into amorphous or crystalline states that have different resistances. The read operation detects the resistance of the cell. PCM cells wear out after 1 to 100 million writes because each write changes their physical structure [13], [26], [40]. Writes are also an order of magnitude slower and consume more energy than in DRAM. Reading the PCM array is up to $4\times$ slower than DRAM [26].

Hybrid memories combine DRAM and PCM to mitigate PCM wear-out and tolerate its higher latency. Frequently accessed data is kept in DRAM which results in better performance and longer lifetimes compared to a PCM-Only main memory system. The large PCM capacity reduces disk accesses which compensates for its slow speed.

B. Garbage Collection

Managed languages such as Java, C#, Python, and JavaScript use garbage collection to accelerate development and reduce memory errors. High-performance garbage collectors today exploit the generational hypothesis that most objects die young [48]. With generational collectors, applications (mutators) allocate new objects contiguously into a nursery. When allocation exhausts the nursery, a minor collection first identifies live roots that point into the nursery, e.g., from global variables, the stack, registers, and the mature space. It then identifies reachable objects by tracing references from these roots. It copies reachable objects to a mature space and reclaims all nursery memory for subsequent fresh allocation. When the mature space is full, a full-heap (mature) collection collects the entire heap.

Recent work introduces write-rationing garbage collectors to manage hybrid memories [2]. Write-rationing collectors keep frequently written objects in DRAM in hybrid memories and read-mostly objects in PCM to improve PCM lifetime [2]. They come in two main variants. (1) The *Kingsguard-nursery* (KG-N) collector allocates nursery objects in DRAM and promotes all nursery survivors to PCM. Applications mutate (write) nursery objects at a high rate. KG-N places the nursery in DRAM which reduces PCM write rates significantly compared to putting all data in PCM (PCM-Only). The reduced write rates lead to a longer PCM lifetime. (2) *Kingsguard-writers* (KG-W) monitors nursery survivors in a DRAM observer space. Observer space collections copy objects with zero writes to a PCM mature space, and copy written objects to a DRAM mature space. KG-W incurs a moderate performance overhead over KG-N due to monitoring and extra copying of nursery survivors. However because past writes are a good predictor of future writes, KG-W improves lifetimes significantly over KG-N. The Kingsguard collectors build on the best-performing collector in Jikes RVM, namely generational Immix (GenImmix) [11], which uses a copying nursery and a mark-region mature space.

KG-W includes two additional optimizations to protect PCM from writes. Traditional garbage collectors allocate large objects directly in a non-moving mature space to avoid copying them from the nursery to the mature space. KG-W's Large Object Optimization (LOO) allocates some large objects, chosen using a heuristic, in the nursery to give them time to die. The mutator allocates the remaining large objects directly in a PCM mature space. The collector copies highly written large objects from PCM to DRAM during a mature collection. Garbage collectors also write to object metadata to mark them live. Marking live objects generates writes to PCM during a mature collection. The MetaData Optimization (MDO) places PCM object metadata in DRAM to eliminate garbage collector writes to object metadata.

III. Emulation Platform for Managed Languages

We now describe the design and implementation of our hybrid memory emulator for managed languages. We first discuss our heap layout in hybrid memory, and how we allocate the heap regions in DRAM and PCM. We then provide details for the hardware configuration: platform requirements, mapping virtual to physical DRAM and PCM memory, and thread scheduling.

A. Heap Layout and Management

The widely used Java runtime environments today manage heap memory using a multi-level hierarchy of blocks and spaces. A space is a coarse-grained partition of the heap. Typically, objects that reside in the same space share a common property. For instance, in generational heaps, the nursery space is used to allocate all the newly created objects. During a (minor) garbage collection, all objects that survive a nursery collection are copied to the mature space. A space is further logically divided into blocks or chunks. The size of the block is a multiple of the page size and it is the minimum unit of virtual memory handed out to a space. A free-list records the location and size of free blocks, and the space to which each block is mapped. The heap manager is responsible for requesting that the operating system maps blocks to physical memory (pages).

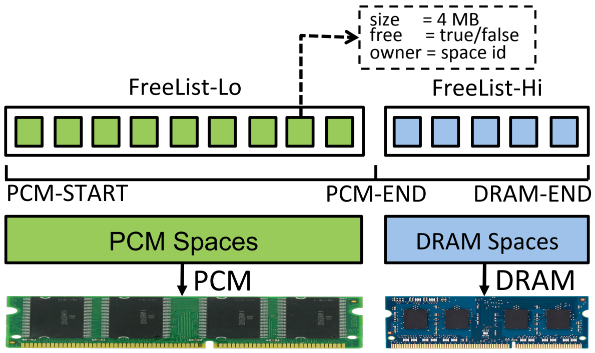


Fig. 1: The organization of our heap in hybrid memory. Memory composition is exposed to the language runtime. Two free lists keep track of available virtual pages in DRAM and PCM.

Figure 1 provides a high-level view of our heap layout in a hybrid DRAM-PCM system. We use the 32-bit Jikes RVM, but our approach generalizes to other JVMs. In a 32-bit environment, the Linux OS owns the upper 1 GB of the 4 GB virtual memory available to a process. In addition, system libraries use *some* amount of virtual memory for the *malloc* heap. We use the middle 2 GB for the managed heap. We divide virtual heap memory into two portions: (1) a DRAM-backed portion, and (2) a PCM-backed portion. We use a free-list to manage the blocks that belong to each portion: FreeList-Hi and FreeList-Lo. In our heap layout, PCM_START marks the beginning of the user heap, and PCM_END is the end of the PCM-backed portion of the heap, and the beginning of the DRAM-backed portion.

Each space requests that the allocator associated with FreeList-Lo or FreeList-Hi reserve virtual memory. Jikes RVM uses mmap() for reserving virtual memory if none is available as indicated by the free lists. The allocator finds a free chunk and returns the address to the requesting space. The space then makes sure the chunk is mapped in physical memory. In our approach, once a chunk is mapped in physical memory, we do not remove its mapping in the OS page tables even if the chunk is no longer in use by the requesting space. The chunk is recycled by the allocator when another space requests a free chunk. We modify the chunk allocator to map memory in DRAM or PCM.

We use the default size for each chunk in Jikes RVM, i.e., 4 MB. Each entry in the free-list contains meta-information about the chunk: (1) the size, (2) the status (free or in use), and (3) the current owner.

The runtime reserves the address range of the nursery space at boot-time. Similar to the baseline design, we place the nursery at one end of virtual memory. This configuration enables the standard fast boundary write-barrier for generational collection. Other contiguous spaces (such as the observer space in KG-W) are placed next to the nursery. Mature spaces use a request mechanism to acquire chunks at runtime. These spaces share the pool of available chunks with other spaces. A space is specified as DRAM or PCM using a flag in the constructor of each space.

We allocate memory using the Linux OS calls for specifying a memory allocation on the local or remote memory socket on a NUMA machine. We use the local socket as the DRAM socket and the remote socket as the PCM socket. To bind a virtual memory range to a particular socket, we call mbind() with the socket number after each call to mmap(). We use a NUMA-specific version of the C memory allocator to call these routines. We modify the Java Virtual Machine to call the C routines for DRAM and PCM allocation.

The alternative approach to manage DRAM and PCM spaces is to use a monolithic heap with a single free-list. The efficiency of such an approach is low because it requires unmapping freed chunks from physical memory. If not, a DRAM space could end up using a logical chunk that is physically mapped in PCM. The flexibility of leaving the free chunks mapped in physical memory is a result of our design with two free lists.

B. Emulation on NUMA Hardware

Hardware requirements: Our hardware requirement to emulate hybrid memory is a commodity NUMA platform with two sockets. We require both sockets to be populated with DRAM chips. Threads run on one socket, referred to as the local DRAM socket. No threads execute on the other remote PCM socket. Figure 2 shows an example NUMA hardware platform. Allocation on Socket #0 (S0) is local to the threads and we use it to allocate DRAM memory. Memory accesses on Socket #1 (S1) are remote and emulate PCM.

Space to Socket Mapping: Table I shows the space to socket mapping for three of the seven collectors we evaluate in this work on our emulation platform. KG-W and its variants use extra spaces in DRAM that are mapped to Socket #0 (S0). The observer space in KG-W is placed in DRAM and is used to monitor object writes. KG-W has a mature, large, and metadata space in both DRAM (S0) and PCM (S1). KG-W-MDO does not include the metadata optimization (see Section II). Therefore, it does not use an extra metadata space in DRAM.

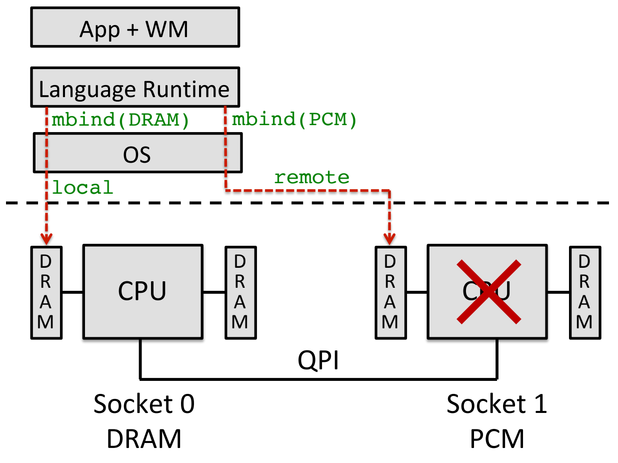


Fig. 2: Our platform for hybrid memory emulation. The application and write rate monitor (WM) run on Socket #0. The memory on Socket #0 is DRAM and Socket #1 is PCM.

	KG-N		KG-W		KG-W - MDO	
	S0	S1	S0	S1	S0	S1
Nursery	✔	✗	✔	✗	✔	✗
Observer	✗	✗	✔	✗	✔	✗
Mature	✗	✔	✔	✔	✔	✔
Large	✗	✔	✔	✔	✔	✔
Metadata	✗	✔	✔	✔	✗	✔

TABLE I: Spaces in Kingsguard collectors and their mapping to Socket S0 (DRAM) or Socket S1 (PCM). KG-N does not use an observer space. KG-W uses a mature, large, and metadata space in both DRAM and PCM.

The boot space contains the boot image runner that boots Jikes RVM and loads its image files. Except for a system with only PCM, we always place the boot image in DRAM because we observe a large number of writes to it.

Thread to Socket mapping: For the Kingsguard configurations, we always bind threads, including application and JVM service threads, to Socket #0 (see Figure 2). When emulating a system with only PCM, we bind threads to Socket #1 for accurately reporting write rates. We do not pin threads to specific cores and use the default OS scheduler.

We measure write rates on our emulation platform using a write rate monitor (WM in Figure 2) that also runs on Socket #0. We experimentally find out that scheduling WM on Socket #0 leads to more deterministic write measurements.

IV. Experimental Methodology

Java Virtual Machine: We use Jikes RVM 3.1.2 because it uses software practices that favor ease of modification, while still delivering good performance [4], [5], [8], [19]. Jikes RVM is a Java-in-Java VM with both a baseline and a just-in-time optimizing compiler, but lacks an interpreter. Jikes RVM has a wide variety of garbage collectors [7], [11], [46]. Its memory management tool kit (MMTk) [7] makes it easy to compose new collectors by combining existing modules and changing the calls to the C and OS allocators. Jikes RVM also offers easy-to-modify write barriers [54] which makes it easy to implement a range of heap organizations.

Evaluation Metrics: We are mainly interested in accurately measuring PCM writes. Write rates sometimes reveal more insight than raw writes, for example, when comparing the impact of changing the input dataset or understanding lifetime implications. PCM lifetime in years is inversely proportional to its write rate [40]. Therefore, whenever appropriate, we also show PCM write rates. In this work, the observed write rates on the remote socket equal the PCM write rates.

Workload Formation: Multiprogrammed workloads reflect real-world server workloads because (1) a single application does not always scale with more cores, and (2) multiprogramming helps amortize server real-estate and cost.

Our multiprogrammed workloads consist of two and four instances of the same application. We do not restart applications after they finish execution. To avoid non-determinism due to sharing in the OS caches in multiprogrammed workloads, we use independent copies of the same dataset for the different instances.

Measurement Methodology: We use best practices from prior work to evaluate Java applications on our emulation platform [21], [23]. To eliminate non-determinism due to the optimizing compiler, we use replay compilation as used in prior work [12]. 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. The second measured iteration does not recompile methods leading to steady-state behavior. We perform each experiment four times and report the arithmetic mean.

We use the pcm-memory utility in the Intel's Performance Counter Monitor framework to measure PCM writes. We make modest modifications to support multiprogrammed workloads and to make it compatible for use with replay compilation. In a multiprogrammed workload, all applications synchronize at a barrier and start the second iteration at the same time.

Applications: We use 15 Java applications from three diverse sources: 11 DaCapo [9], pseudojbb2005 (Pjbb) [10], and 3 applications from the GraphChi framework for processing graphs [25]. The GraphChi applications we use are: (1) page rank (PR), (2) connected components (CC), and (3) ALS matrix factorization (ALS). Compared to recent work [2], we drop jython as it does not execute stably with our Jikes RVM configuration. To improve benchmark diversity, we use updated versions of lusearch and pmd in addition to their original versions. lu.Fix eliminates useless allocation [55], and pmd.S eliminates a scalability bottleneck in the original version due to a large input file [16]. Similar to recent prior work, we run the multithreaded DaCapo applications, Pjbb, and GraphChi applications with four application threads.

Unless otherwise stated, we use the default datasets for DaCapo and Pjbb. Our default dataset for GraphChi is as follows: for PR and CC, we process 1 M edges using the LiveJournal online social network [27], and for ALS, we process 1 M ratings from the training set of the Netflix Challenge. The DaCapo suite comes packaged with large datasets for a subset of the benchmarks. Our large dataset for GraphChi consists of 10 M edges and 10 M ratings.

We use the C++ implementations of the GraphChi applications to characterize and compare C++ and Java memory management and their implications for hybrid memory management.

Nursery and Heap Sizes: Nursery size affects performance, response time, and memory space efficiency [6], [7], [49], [58]. Similar to prior work [2], we use a 4 MB nursery for DaCapo and Pjbb. Although recent prior work uses a 4 MB nursery for GraphChi applications, we find a 32 MB nursery improves performance, and we use this size for our experiments with the GraphChi applications. We use a modest heap size that is twice the minimum heap size. Our heap sizes reflect those used in recent work [1], [11], [35], [44], [58].

Garbage Collectors and Configurations: We explore seven write-rationing garbage collectors. Our collector configurations include KG-N, and a variant called KG-B, that uses a bigger nursery than KG-N. We use KG-B to understand if simply using a large nursery, equal to the sum of nursery and observer space in KG-W, could reduce PCM writes similar to KG-W. KG-B and its variants use a 12 MB nursery for DaCapo and Pjbb, and a 96 MB nursery for the GraphChi applications.

For the GraphChi applications, we evaluate KG-N and KG-B with the Large Object Optimization (LOO) to form KG-N+LOO and KG-B+LOO. We include the original KG-W along with two variants: one that removes LOO to form KG-W-LOO and one that removes the MetaData Optimization (MDO) to form KG-W-MDO. We configure the Kingsguard collectors with an observer space that is twice as large as the nursery. Prior work reports this to be a good compromise between tenured garbage and pause time [2]. We compare to PCM-Only with the baseline generational Immix collector [11]. All of our experiments use two garbage collector threads.

Hardware Platform: Figure 2 shows the NUMA platform we use to emulate hybrid memory. Each socket contains one Intel E5-2650L processor with 8 physical cores each with two hyperthreads, for 16 logical cores. The platform has 132 GB of main memory. Physical memory is evenly distributed between the two sockets. We use all the DRAM channels on both sockets. The 20 MB LLC on each processor is shared by all cores. The maximum bandwidth to memory is 51.2 GB/s, which is more than the maximum bandwidth consumed by any of our workloads. The two sockets are connected via QPI links that support up to 8 GB/s. We use Ubuntu 12.04.2 with a 3.16.0 kernel.

Simulated Hardware: We compare the results of emulation and simulation to establish confidence in our newly proposed methodology, and to confirm the findings in prior work [2]. We compare emulation against simulated hardware using the Sniper multicore simulator [15]. We make a best effort to choose hardware parameters of the simulated system to be the same as our emulation platform. More specifically, we consider 8 out-of-order cores with 256 KB private L2 caches, and a shared 20 MB L3 cache. Unfortunately, the simulated hardware does not model hyper-threading. To account for this difference, we disable hyper-threading on our emulation platform when comparing emulation versus simulation.

V. Emulation versus Simulation

This section validates emulation against simulation for the Kingsguard collectors in terms of the reported reduction in PCM writes. Lack of full-system support and long simulation times limit evaluation using the simulator to 7 DaCapo benchmarks: lusearch, lu.Fix, avrora, xalan, pmd, pmd.S and bloat. We compare emulation versus simulation for three Kingsguard collectors: KG-N, KG-B, and KG-W, see Table II. We consider the following reference setup to emulate a PCM-Only system *and* to isolate system-level effects. We consider the

	Simulator	Emulator
KG-N	4%	8%
KG-B	11%	13%
KG-W	64%	62%

TABLE II: Percentage reduction in PCM writes for three Kingsguard collectors compared to a PCM-Only system using simulation versus emulation. *Emulation and simulation report similar reductions in PCM writes.*

baseline GenImmix as the collector, bind all the heap spaces to the remote socket S1 (PCM), and measure the number of writes to the local socket S0 (DRAM). The writes to S0 are due to system-level activities (e.g., kernel and write rate monitor) since all the program memory is on the remote socket S1 (PCM). We now consider the three Kingsguard collectors relative to this PCM-Only reference setup.

KG-N: Recently published work reports an 81% reduction in PCM writes using KG-N [2]. This result was obtained through simulation of a system with a 4 MB L3 cache. The processors in our emulation platform on the other hand feature a 20 MB L3 cache. We perform new experiments using the simulator by matching the L3 cache size in simulation and emulation. Compared to a PCM-Only system, the newly simulated system reports a 4% reduction of PCM writes with KG-N. The large 20 MB L3 cache absorbs most of the nursery writes which limits the benefit of KG-N in reducing the number of writes to PCM.

We now evaluate whether emulation yields a similar reduction in PCM writes. To emulate KG-N, we bind the nursery to the local socket S0 (DRAM) and leave all other heap spaces on the remote socket S1 (PCM). This results in an increase in DRAM writes and a corresponding reduction in PCM writes compared to the reference PCM-Only setup which puts all heap spaces on S1. The difference in PCM writes between the two setups is the reduction in PCM writes because of KG-N. Emulation reports an average reduction in PCM writes of 8%. In other words, we note a discrepancy of 4 percentage points between emulation and simulation.

KG-B: The second Kingsguard collector we consider is the KG-B collector which effectively is the KG-N collector with a bigger nursery space (12 MB versus 4 MB). Simulation reports an average 11% reduction in PCM writes versus 13% through emulation — a discrepancy of 2 percentage points on average. We note that only two benchmarks, namely lusearch and xalan, experience a reduction in PCM writes with a bigger 12 MB nursery, further validating emulation versus simulation.

Intuition suggests that a large nursery could potentially increase the total number of writes to memory, because a small nursery has better L3 cache locality. L3 cache lines containing mature and large objects are more likely to face evictions when sharing the limited cache space with a larger nursery. Nevertheless, we were surprised that on average for 7 DaCapo benchmarks and using the simulator, the total number of writes to memory increases by 1.98× for KG-B compared to KG-N. Through emulation we find an increase in total memory writes with KG-B to be 2.2×. This further validates emulation trends against simulation.

KG-W: The third Kingsguard collector we consider is KG-W. We emulate KG-W by following the heap organization as shown in Table I: we bind the nursery and observer spaces to the local socket S0, and compute the reduction in PCM writes compared to the reference PCM-Only system. The average reduction in PCM writes reported by simulation and emulation is close: simulation reports an average 64% reduction in PCM writes whereas emulation reports a 62% reduction — a discrepancy of 2 percentage points.

Finding 1. *Emulation matches simulation when measuring the reduction of PCM writes on hybrid memories due to write-rationing garbage collection. Emulation must isolate system-level effects when assessing the benefits of managed language approaches to manage hybrid memories. A large L3 cache absorbs most of the nursery writes.*

Performance: So far, we validated emulation against simulation in terms of PCM writes. We now consider performance when validating emulation versus simulation. We do not (and cannot) accurately model PCM access latency on our emulation platform. However, we can accurately measure the performance overhead of KG-W versus KG-N because of the observer space and various other optimizations to improve PCM lifetime. While simulation reports an average performance overhead of 7%, emulation measures an average 10% overhead. This further confirms our conclusion that emulation is a viable and accurate evaluation complement to simulation.

VI. Hybrid Memory for Managed Workloads

We now evaluate hybrid memory with managed language workloads using our newly proposed emulation framework. Specifically, we focus on understanding the number of writes to PCM for a rich workload and software configuration space. We focus on writes because they determine PCM lifetime, and whether PCM is a practical DRAM replacement. We discuss the implications of programming language, multiprogrammed workloads, benchmark suite, and input dataset size for PCM writes. We further report write rates to PCM to analyze PCM's practicality in terms of its lifetime. Finally, the native speed of emulation allows us to report on the effectiveness of recently proposed Kingsguard collectors for graph processing workloads.

A. Programming Language Implications for PCM Writes

The two memory management techniques in wide use today are manual and automatic memory management. C++ offers manual memory management whereas Java offers automatic memory management through garbage collection. We quantify the impact of the programming language and the choice of memory management technique on PCM writes. The GraphChi developers provide C++ and Java versions for PR, CC and ALS. Emulation makes it easy to compare both versions in terms of PCM writes.

Figure 3 compares the number of PCM writes that we observe with the C++ and Java implementations of the

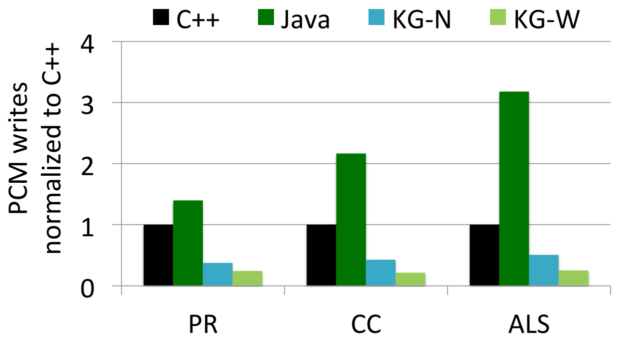


Fig. 3: Comparison of PCM writes in C++ and Java implementations of the GraphChi applications. *Java applications write more to PCM than C++ applications in a PCM-Only system. KG-N and KG-W result in fewer PCM writes than the C++ implementations in a hybrid memory system.*

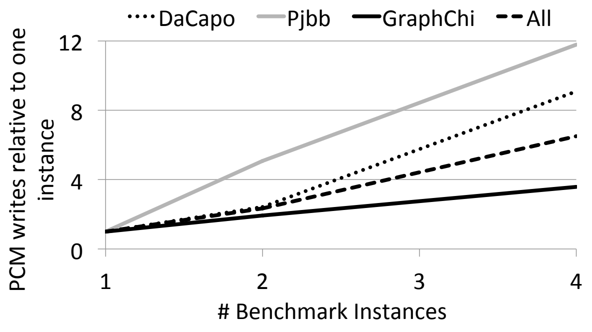
GraphChi applications on a PCM-Only system. In a PCM-Only system (left two bars), the Java versions of GraphChi applications write up to 3.2× more to PCM than their C++ counterparts. There are three reasons for the additional writes of the Java applications: (1) high allocation rates [58], (2) object copying during garbage collection, and (3) zero-initialization to guarantee memory safety.

This finding is further supported by measuring the total amount of allocation in GB for both the C++ and Java versions of GraphChi. We instrument the allocation sequence in Jikes RVM to measure the amount of allocation for the Java applications. For measuring the allocation in the C++ applications, we used the memcheck tool in the Valgrind distribution [34]. Java applications allocate more than their C++ counterparts: 1.34× for PR, 1.6× for CC, and 2× for ALS. The larger volume of allocation in Java applications contributes to more writes to PCM.

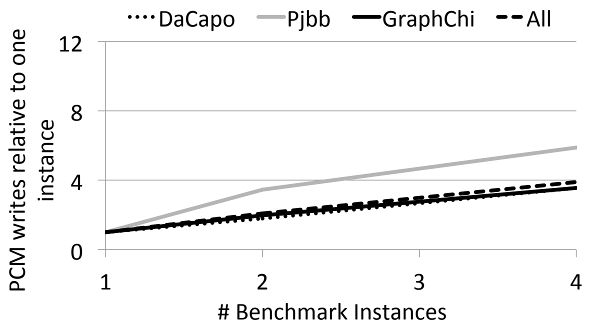
The C++ applications also consume less maximum heap memory. We use the massif tool in Valgrind to measure the peak heap memory during execution. Although we configure the C++ heap equal to the size of the Java heap, i.e., 512 MB, the maximum heap used by the three graph applications is approximately 400 MB. The Java versions, on the other hand, trigger full-heap collections that implies a peak heap usage of 512 MB.

Next, we discuss PCM writes for a hybrid memory system. The Java runtime has two advantages that help limit writes to PCM: (1) newly allocated and old objects are segregated in coarse-grained spaces in the heap, and (2) the garbage collector can move highly written objects from PCM to DRAM. The malloc-based allocations in C++ on the other hand rely on a free-list allocator to manage heap memory. Fresh allocation is thus scattered throughout the heap. Furthermore, there is no possibility to move fine-grained objects between PCM and DRAM.

Figure 3 shows PCM writes of Java applications in a hybrid memory system with the KG-N and KG-W collectors. For the Java applications with KG-N, the number of PCM writes is less than half the number of writes for the C++ applications on average. The nursery captures a large fraction of the writes due to fresh allocation, and KG-N places this in DRAM. KG-N piggybacks on generational garbage collection to eliminate a large number of PCM writes for Java applications. In contrast, fresh allocations in C++ are not localized to a specific region of the heap. In addition to keeping the nursery in DRAM, KG-W keeps highly written objects in DRAM. We observe that KG-W even further reduces the number of writes to PCM compared to C++.



(a) PCM-Only



(b) KG-W

Fig. 4: Average PCM writes with PCM-Only and KG-W normalized to PCM writes with a single instance. *The growth in PCM writes is super-linear for many multiprogrammed workloads.*

Finding 2. *Java applications allocate more memory and write more than C++ applications in a PCM-Only system. With hybrid memory, the generational heap organization of Java's garbage collectors enables capturing writes due to fresh allocation in DRAM. This brings the number of writes to PCM for Java workloads below C++.*

B. PCM Writes for Multiprogrammed Workloads

Prior work on Java workload evaluation on PCM memory is limited to single-program workloads [2], [20]. This limitation is because long simulation times impede the evaluation of PCM and hybrid memories for multiprogrammed Java workloads. Our emulation platform allows us to run four instances of a program at the same time, at native execution speed. Multiprogrammed workloads incur interference patterns in the last-level cache that results in writes to PCM memory. Figure 4(a) and Figure 4(b) show the growth in the average number of PCM writes on a PCM-Only system, as well as on a hybrid memory system with KG-W, respectively.

We observe a variety of trends in PCM writes. On average for PCM-Only, the increase in PCM writes from 1 to 2 program instances equals 2.3×, as expected. However, from 1 to 4 instances, we observe a super-linear increase of 6.4×. DaCapo applications encounter high interference in the LLC. The average increase in PCM writes from 1 to 4 instances for DaCapo is 9× (2.4× from 1 to 2 instances). The increase for Pjbb is even higher. From 1 to 2 instances, the number of writes to PCM increases by 5×, and from 1 to 4 instances, the number of writes increases by 12×. The GraphChi applications exhibit a linear trend. The increase in PCM writes equals 1.9× and 3.5× for 2 and 4 instances, respectively.

We analyze PCM writes further in a PCM-Only system to understand the super-linear trend. In particular, we isolate nursery writes to one socket and the mature writes to another socket. Our analysis shows that writes to the nursery are responsible for the super-linear increase in PCM writes for all DaCapo benchmarks and Pjbb. For example, from 1 to 4 instances, nursery writes in DaCapo increase by 30×. The mature writes on the other hand increase only by 3×. Figure 4(b) highlights this further by showing the increase in PCM writes with KG-W. Contrary to PCM-Only, KG-W (and KG-N, not shown) exhibit a linear increase in PCM writes from 1 to 2 and 4 program instances across the three suites.

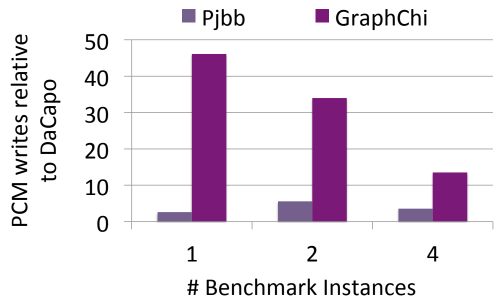
Finding 3. *PCM writes grow super-linearly with the number of concurrently running program instances in a PCM-Only system. Write-rationing garbage collection significantly dampens the increase in the number of writes to PCM.*

C. DaCapo versus Emerging Java Benchmark Suites

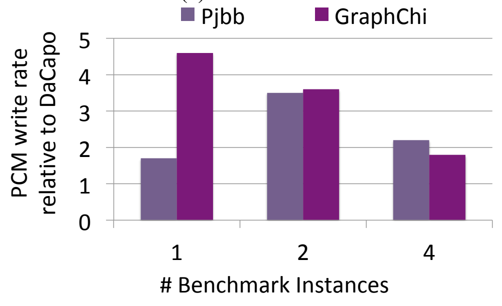
The DaCapo benchmark suite is the dominant choice for prior research on garbage collection; some studies use Pjbb. Emerging application domains such as graph processing use managed languages due to ease of implementation. They differ from traditional Java benchmark suites in a number of ways: (1) they use larger heaps, (2) they allocate larger and more objects in the mature heap, and (3) they perform full-heap garbage collections more often. This section analyzes PCM writes for the GraphChi applications and compares the results to DaCapo and Pjbb. The average heap size equals 100 MB for DaCapo, 400 MB for Pjbb, and 512 MB for GraphChi.

We show the average number of PCM writes with a PCM-Only system in Figure 5(a) for multiprogrammed Pjbb and GraphChi applications. We normalize to writes obtained with DaCapo. Pjbb writes on average 2× more relative to DaCapo when running a single program instance. GraphChi applications on average write to PCM 46× more often than DaCapo. This relative gap in writes between DaCapo and GraphChi narrows for multiprogrammed workloads. This is because DaCapo applications incur greater LLC interference in a multiprogrammed setting.

The execution times of different benchmarks vary, and GraphChi applications run longer than both DaCapo and Pjbb. Thus, comparing PCM writes alone could be misleading. The compute-to-write ratio determines how quickly PCM wears out as a DRAM replacement. Figure 5(b) shows the average PCM write rates with Pjbb and GraphChi relative to DaCapo. The average write rates of Pjbb and GraphChi are 1.7× and 4.7× higher than DaCapo. This shows that although the volume of writes to PCM by graph applications is an order of magnitude larger, the increase in write rate is less dramatic.



(a) PCM Writes



(b) PCM Write Rates

Fig. 5: Average raw PCM writes and write rates (MB/s) for Pjbb and GraphChi relative to DaCapo for a PCM-Only system. *Pjbb and GraphChi have more writes and higher write rates compared to DaCapo.*

The difference in write rates between DaCapo and GraphChi is less pronounced with KG-N and KG-W (not plotted). GraphChi has the same average write rate as DaCapo with KG-N and single-program workloads. The write rate with two and four instances is 1.5× and 1.6× higher than DaCapo. This shows that the higher write rate for GraphChi is due to nursery writes.

Finding 4. *Future studies on hybrid memories should use a diversity of applications. Pjbb and GraphChi incur a larger number of PCM writes and higher write rates than DaCapo.*

D. PCM Write Rates and Practicality as Main Memory

We now discuss the PCM write rates that we observe for our applications. We show write rates for PCM-Only and three Kingsguard collectors. Write rates are important to observe because hardware vendors recommend a maximum write rate for PCM to work reliably during its warranty period. We derive the maximum PCM write rate from recent work [18]. Specifically, they use a real NVM prototype of 375 GB with a limit of 30 drive writes per day (DWPD). DWPD limits the number of times the entire PCM memory (or drive) can be written per day. A DWPD of 30 translates to a recommended write rate of 140 MB/s. Figure 6 shows the write rates for our applications in MB/s. Focusing first on PCM-Only, we observe

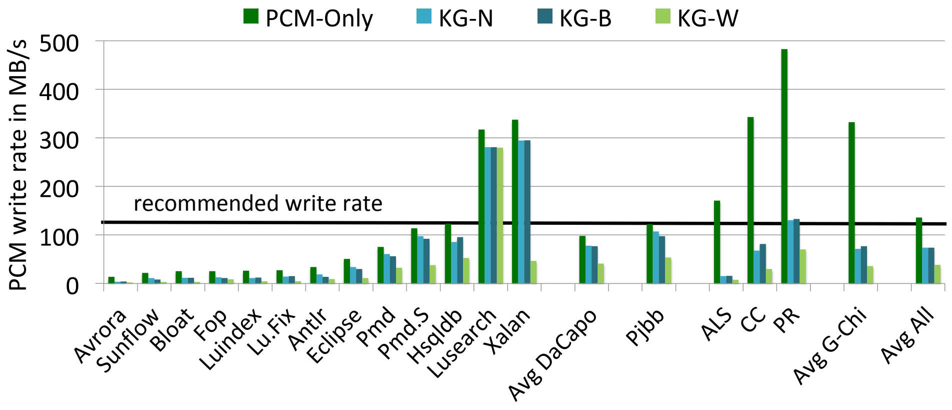


Fig. 6: PCM write rates in MB/s for all of our benchmarks. *PCM alone is impractical for many Java applications.*

two distinct behaviors. One set of applications writes to PCM at a rate below the recommended write rate, including almost all of the DaCapo benchmarks and Pjbb. On the other hand, two DaCapo and all of the graph processing applications have very high PCM write rates. When these applications execute on a PCM-Only system, they will wear-out PCM quickly, making it impractical as main memory. For many applications, Kingsguard collectors and especially KG-W reduces the PCM write rate significantly. However, even with KG-W, PCM write rate for one application is above the recommended write rate. Future work should continue to investigate novel ways to reduce PCM write rates for Java applications.

Finding 5. *PCM in itself cannot replace DRAM. Applications from different domains, and especially the ones that process huge graphs, will wear out PCM very quickly. More work is needed to limit PCM write rates and make it a practical DRAM replacement.*

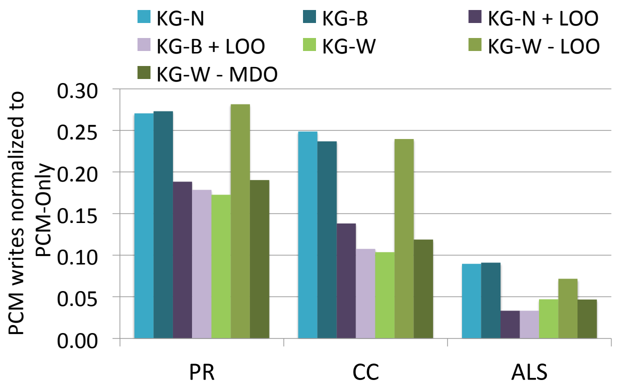


Fig. 7: PCM writes with various Kingsguard collectors normalized to PCM-Only for the GraphChi applications. *Graph applications write less to PCM when the nursery is placed in DRAM, and the large objects are first allocated in the nursery.*

E. Write-Rationing Garbage Collection for GraphChi

Recent work has explored write-rationing garbage collectors (Kingsguard) for Java applications with moderate heaps [2]. Here, we evaluate the Kingsguard collectors for applications from the GraphChi framework. The goals of this evaluation are to understand the impact on PCM writes of (1) larger nurseries, (2) allocating large objects first in the nursery (LOO), (3) write monitoring in KG-W using an extra observer space, and (4) placing meta-data in DRAM (MDO).

Figure 7 shows the results of our evaluation. Specifically, the figure shows the reduction in PCM writes for single instances of GraphChi applications. We observe that using KG-N to place the nursery in DRAM substantially reduces writes to PCM. Simply increasing the nursery size with KG-B has a negligible impact on PCM writes. We investigated the reason for this further. Large objects in KG-B are allocated directly in the large object space in PCM. Increasing the nursery size takes virtual memory away from the large object space leading to more full-heap collections. This is because the large object allocation more quickly fills up the heap in PCM. Based on this analysis, we explore two new Kingsguard configurations in this work. We combine KG-N and KG-B with LOO. Combining LOO with both KG-N and KG-B is effective in further reducing the writes to PCM. KG-N+LOO and KG-B+LOO both reduce PCM writes by up to 11% and 13% on top of KG-N and KG-B, respectively.

We also observe in Figure 7 that KG-W reduces PCM writes similar to KG-N+LOO. To find the feature of KG-W that has the largest impact on reducing PCM writes, we explore two additional Kingsguard configurations. First, we evaluate KG-W-LOO to tease apart the impact of the large object optimization from KG-W. Excluding LOO from KG-W (i.e., KG-W-LOO) increases PCM writes because of allocation of short-lived large objects in PCM. Large object allocation in PCM fills up the heap quickly, leading to more frequent mature collections. Mature collections are a source of PCM writes because of the collector updating the objects' metadata to mark them live. As a result, the number of writes to PCM increases by 1.6× for PR, 2.3× for CC, and 1.5× for ALS.

Next, we tease apart the impact of the metadata optimization by evaluating KG-W-MDO. The GraphChi applications have more mature space collections than DaCapo and Pjbb. Without the metadata optimization (MDO), PCM writes increase for KG-W, but only marginally. Specifically, the writes to PCM

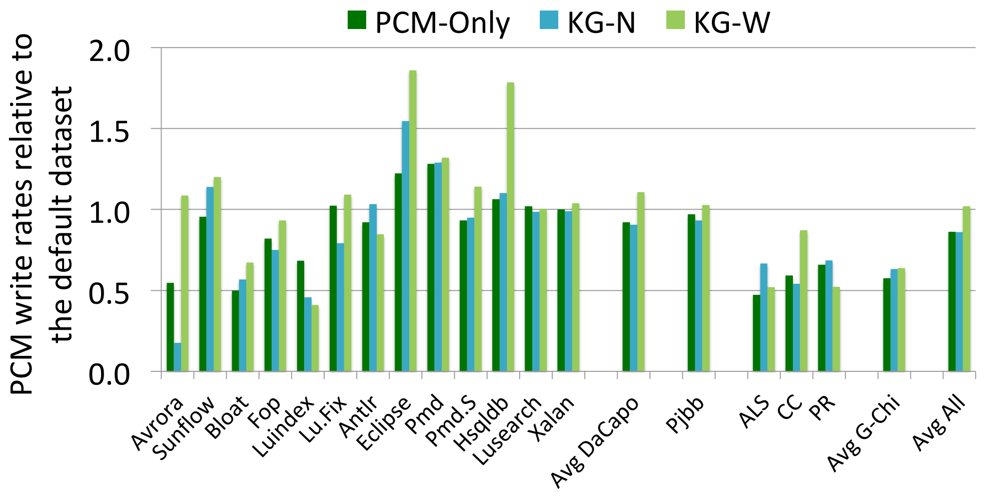


Fig. 8: PCM write rates with large datasets normalized to write rates with the default datasets. *The input dataset size could increase/decrease the PCM write rates.*

increase by 1.14× for both PR and CC. Without MDO, the writes to PCM of multiprogrammed workloads increase more than the increase with single-program workloads (not shown).

We did a similar analysis for the DaCapo applications and Pjbb. We do not show all the results due to space constraints. Based on the analysis, we conclude that compared to DaCapo and Pjbb, dynamic monitoring of objects in KG-W has limited benefits for GraphChi applications. These applications benefit the most from optimizations that target fresh allocation, nursery writes, and large objects.

Finding 6. *Allocating the nursery in DRAM and using the large object optimization are effective ways to reduce the number of writes to PCM for modern graph applications.*

F. The Impact of Larger Input Dataset Sizes

Simulation of managed workloads precludes large datasets because they make simulation times too long. The native speed of emulation enables evaluation with large datasets. Here, we evaluate the impact of large datasets on PCM writes and write rates. We observe that, not surprisingly, large datasets increase PCM writes by 3.4× on average and up to 10× because the execution times are longer. Our analysis shows that the number of writes to both the nursery and the mature spaces increases with a larger dataset.

Write rates reveal more insight about the impact of large datasets. Figure 8 shows PCM write rates for PCM-Only, KG-N, and KG-W normalized to the default datasets. We observe three trends for PCM write rates: (1) they stay unchanged, (2) they increase by up to 1.5×, and (3) they decrease by up to 80%. When PCM write rates stay unchanged, the compute-to-write ratio does not depend on the size of the datasets. If write rates increase/decrease, there is a disproportional decrease/increase in compute relative to the size of the dataset. Interestingly, the PCM write rates of graph applications reduce by 60% when the size of the graph increases by 10×. We also observe that large inputs lead to high PCM write rates for some DaCapo applications with KG-W. This opens up opportunities for future work and optimizations.

Finding 7. *Large datasets sometimes shift the balance between compute and memory-writes, changing PCM write rates.*

G. Study of PCM Lifetimes

We now discuss PCM's lifetime in years based on our earlier findings regarding write rates. PCM's lifetime in years depends on a number of factors, primarily the rate of writes to PCM, which varies with the processor frequency, cache setup, number of co-running applications, etc. We use the analytical model from prior work to report lifetimes for our set of applications running on our emulation platform [13], [40]. More specifically, PCM lifetime in years before failing is estimated as follows:

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

We assume the size (S) of PCM main memory to be 32 GB. We consider three PCM endurance (E) levels used in prior work [37], [38]: 10 M, 30 M, and 50 M writes per PCM cell. Finally, B is the write rate of an application during execution. Equation 1 assumes perfect wear-leveling which is unrealistic. We assume a PCM system with hardware wear-leveling that delivers endurance within 50% of the theoretical maximum [38]. Table III shows worst-case (shortest) PCM lifetimes in years across 15 applications.

We observe that running single-program workloads leads to practical PCM lifetimes without the need for Kingsguard collectors. This is true across the three PCM prototypes. On the other hand, running multiprogrammed workloads consisting of four instances would wear PCM out in two years given the widely assumed PCM endurance level of 10 M writes per cell. Write-rationing garbage collection is effective in improving PCM's lifetime by more than 3×. The higher the PCM cell endurance, the longer its lifetime. With an endurance of 50 M writes per cell, the worst-case PCM lifetime is 9 years, making a PCM-Only system useful as main memory. Using KG-W

Workload	Prototype 1 10 M writes/cell		Prototype 2 30 M writes/cell		Prototype 3 50 M writes/cell	
	PCM-Only	KG-W	PCM-Only	KG-W	PCM-Only	KG-W
N = 1	10	18	31	54	52	90
N = 4	2	7	5	20	9	34

TABLE III: PCM lifetime in years (worst-case across our benchmarks) for single-program (N=1) and 4-program (N=4) workloads. *Multiprogrammed workloads quickly wear out PCM. Kingsguard collectors make PCM useful across a broad set of applications.*

prolongs PCM's lifetime to more than 30 years. However, if more than four applications are running simultaneously, a PCM-Only main memory would become non-viable again. This analysis shows that software techniques such as write-rationing garbage collection promise to make persistent main memories last longer, potentially the same number of years as secondary storage devices such as disks.

VII. Related Work

We now discuss related work on evaluation methodologies for emerging hardware and hybrid memories.

A. *Evaluation Methodologies*

Prior work uses emulation to evaluate emerging hardware. Prior work uses commodity machines to emulate asymmetric multicores using frequency scaling [14], [22]. Other works focus more on cutting-edge memory technologies. Oskin et al. [36] use a NUMA platform for emulating die-stacked DRAM. Their evaluation only considers applications written in C. Dulloor et al. [17] emulate hybrid DRAM-NVM memory on a NUMA platform but use it to evaluate file systems for persistent object storage.

More closely related work emulates hybrid DRAM-PCM main memories [31], [51], [52]. The work closest to ours in terms of hardware setup is Quartz [51], which is a performance emulator and does not report writes nor write rates. On the software side, Quartz leaves it to programmers and application writers to spread program memory across DRAM and NVM in hybrid DRAM-NVM memories. Our proposed platform precisely reports writes which enables studies of PCM's practicality as main memory and provides a modified managed runtime for hybrid memories, thus easing future studies. Other related works target native applications written in C and C++, and focus only on emulating the latency of PCM on real hardware [31], [52].

Two platforms today enable executing Java applications on top of simulated hardware in a reasonable amount of time: (1) Jikes RVM on top of Sniper [44], and (2) Maxine VM on top of ZSim [42]. Both Sniper and ZSim are cycle-level multicore simulators that trade off accuracy for speed [15], [43]. In their publicly available versions, both platforms lack support for full-system simulation, favoring speed over detail.

B. *Managed Languages with Hybrid Memory*

Prior work has looked into tailoring the managed runtime to more efficiently take advantage of hybrid memories. Wang et al. [52] use DRAM in hybrid DRAM-NVM systems for allocating frequently read objects. They use an offline profiling phase to identify hot methods in the program. During runtime, all object allocation that happens from hot methods goes into DRAM. Unlike write-rationing collectors that target lifetime, their goal is performance. This work uses emulation for evaluation, but their infrastructure only supports simple heap organizations. Our platform is flexible and enables the evaluation of a range of collector configurations. We also provide a methodology for measuring the write rates of Java workloads under replay compilation [21], [23].

Gao et al. [20] use the managed runtime to tolerate PCM failures. The hardware informs the OS of defective lines which communicates faulty lines to the garbage collector. The garbage collector masks these lines and moves data away from them. This work uses a limited form of emulation on existing hardware with fault injection software to model failures.

We discussed write-rationing garbage collection for hybrid memories [2] in detail in Section II. More recent work introduces a write-rationing garbage collector that uses prediction of write-intensive objects from offline ahead-of-time profiling to reduce the overheads of online monitoring [3].

VIII. Conclusions

Advances in non-volatile memory (NVM) technologies have implications for the whole computing stack. Researchers need fast and accurate methodologies for evaluating future NVM and hybrid memory systems. This work introduces an emulation platform built using widely available NUMA servers to evaluate managed applications running on top of hybrid memories that combine DRAM and NVM. This platform measures writes to NVM and can be used to evaluate applications that use manual or automatic memory management. We evaluate our emulator with write-rationing garbage collectors that keep frequently written objects in DRAM to guard NVM against writes and improve its lifetime. We compare emulation to simulation, showing that they have similar trends. Our emulation platform reveals that Java applications allocate and write much more to NVM than their C++ equivalents. With emulation, we can and do explore large graph applications and multi-programmed workloads with large datasets. Emulation reveals new insights, such as that modern graph applications have much larger write rates than DaCapo benchmarks and benefit significantly from write-rationing collectors. Multiprogrammed environments see a super-linear growth in write rates to NVM compared to running single programs. Although simulation and emulation both have their place, emulation adds the ability to explore a richer software design and workload space.

Acknowledgements

We thank the reviewers for their thoughtful and valuable feedback. 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] S. Akram, J. B. Sartor, and L. Eeckhout, "DEP+BURST: Online DVFS performance prediction for energy-efficient managed language execution," *IEEE Trans. Comput.*, vol. 66, no. 4, Apr. 2017.

[2] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Write-rationing garbage collection for hybrid memories," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[3] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Crystal Gazer: Profile-driven write-rationing garbage collection for hybrid memories," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2019.

[4] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño virtual machine," *IBM Systems Journal*, vol. 39, no. 1, 2000.

[5] B. Alpern, S. Augart, S. M. Blackburn, M. A. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. A. Ngo, V. Sarkar, and M. Trapp, "The Jikes RVM Project: Building an open source research community," *IBM System Journal*, vol. 44, no. 2, 2005.

[6] A. W. Appel, "Simple generational garbage collection and fast allocation," *Softw. Pract. Exper.*, vol. 19, no. 2, Feb. 1989.

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Myths and realities: The performance impact of garbage collection," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2004.

[8] S. M. Blackburn, P. Cheng, and K. S. McKinley, "Oil and water? High performance garbage collection in Java with MMTk," in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2004.

[9] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.

[10] S. M. Blackburn, M. Hirzel, R. Garner, and D. Stefanović, "pjbb2005: The pseudojbb benchmark, 2005," 2010. [Online]. Available: http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005

[11] S. M. Blackburn and K. S. McKinley, "Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[12] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century," *Communications of the ACM*, vol. 51, no. 8, 2008.

[13] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy, "Phase change memory technology," *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, vol. 28, no. 2, 2010.

[14] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley, "The yin and yang of power and performance for asymmetric hardware and managed software," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2012.

[15] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, 2014.

[16] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout, "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)*, 2013.

[17] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2014.

[18] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti, "Reducing DRAM footprint with NVM in Facebook," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2018.

[19] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev, "Demystifying magic: High-level low-level programming," in *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2009.

[20] T. Gao, K. Strauss, S. M. Blackburn, K. S. McKinley, D. Burger, and J. Larus, "Using managed runtime systems to tolerate holes in wearable memories," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[21] J. Ha, M. Gustafsson, S. M. Blackburn, and K. S. McKinley, "Microarchitectural characterization of production JVMs and Java workloads," in *IBM CAS Workshop*, 2008.

[22] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, "Exploiting heterogeneity for tail latency and energy efficiency," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2017.

[23] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng, "The garbage collection advantage: Improving mutator locality." in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2004.

[24] ITRS, "Internatial technology roadmap for semiconductors 2.0: Beyond CMOS," 2015.

[25] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a pc," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[26] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[27] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[28] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-based hybrid memory management," in *Proceedings of the International Conference on Cluster Computing (CLUSTER)*, 2017.

[29] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[30] L. Liu, H. Yang, Y. Li, M. Xie, L. Li, and C. Wu, "Memos: A full hierarchy hybrid memory management framework," in *Proceedings of the International Conference on Computer Design (ICCD)*, 2016.

[31] S. Lloyd and M. Gokhale, "Evaluating the feasibility of storage class memory as main memory," in *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, 2016.

[32] A. Malventano, "How 3D XPoint Phase-Change Memory works," 2017. [Online]. Available: https://www.pcper.com/reviews/Editorial/How-3D-XPoint-Phase-Change-Memory-Works

[33] O. Mutlu and L. Subramanian, "Research problems and opportunities in memory systems," *Supercomputing Frontiers and Innovations*, vol. 1, no. 3, Oct 2014.

[34] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[35] K. Nguyen, L. Fang, G. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu, “Yak: A high-performance big-data-friendly garbage collector,” in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[36] M. Oskin and G. H. Loh, “A software-managed approach to die-stacked dram,” in *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[37] M. K. Qureshi, “Pay-As-You-Go: Low-overhead hard-error correction for phase change memories,” in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, 2011.

[38] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, “Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling,” in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, Dec 2009.

[39] M. K. Qureshi, A. Seznec, L. A. Lastras, and M. M. Franceschini, “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)*, 2011.

[40] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, “Scalable high performance main memory system using phase-change memory technology,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[41] L. E. Ramos, E. Gorbatov, and R. Bianchini, “Page placement in hybrid memory systems,” in *Proceedings of the International Conference on Supercomputing (ICS)*, 2011.

[42] A. Rodchenko, C. Kotselidis, A. Nisbet, A. Pop, and M. Luján, “MaxSim: A simulation platform for managed applications,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.

[43] D. Sanchez and C. Kozyrakis, “ZSim: Fast and accurate microarchitectural simulation of thousand-core systems,” in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[44] J. B. Sartor, W. Heirman, S. M. Blackburn, L. Eeckhout, and K. S. McKinley, “Cooperative cache scrubbing,” in *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, 2014.

[45] A. Seznec, “A phase change memory as a secure main memory,” *IEEE Computer Architecture Letters*, vol. 9, no. 1, Jan 2010.

[46] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley, “Taking off the gloves with reference counting Immix,” in *ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, oct 2013.

[47] TIOBE, “TIOBE Index for January,” https://www.tiobe.com/tiobe-index/, 2019.

[48] D. Ungar, “Generation scavenging: A non-disruptive high performance storage reclamation algorithm,” in *Proceedings of the ACM SIGSOFT-/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE)*, 1984.

[49] D. Ungar and F. Jackson, “An adaptive tenuring policy for generation scavengers,” *ACM Trans. Program. Lang. Syst.*, vol. 14, no. 1, Jan. 1992.

[50] S. Van den Steen, S. Eyerman, S. De Pestel, M. Mechri, T. E. Carlson, D. Black-Schaffer, E. Hagersten, and L. Eeckhout, “Analytical processor performance and power modeling using micro-architecture independent characteristics,” *IEEE Transactions on Computers*, vol. 65, no. 12, 2016.

[51] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, “Quartz: A lightweight performance emulator for persistent memory software,” in *Proceedings of the Annual Middleware Conference (Middleware)*, 2015.

[52] C. Wang, T. Coa, J. Zigman, F. Lv, Y. Zhang, and X. Feng, “Efficient management for hybrid memory in managed language runtime,” in *Proceedings of the IFIP International Conference on Network and Parallel Computing (NPC)*, 2016.

[53] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, “Exploiting program semantics to place data in hybrid memory,” in *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[54] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking, “Barriers reconsidered, friendlier still!” in *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, jun 2012.

[55] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley, “Why nothing matters: The impact of zeroing,” in *Proceedings of the ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[56] P. Zalden, M. J. Shu, F. Chen, X. Wu, Y. Zhu, H. Wen, S. Johnston, Z.-X. Shen, P. Landreman, M. Brongersma, S. W. Fong, H.-S. P. Wong, M.-J. Sher, P. Jost, M. Kaes, M. Salinga, A. von Hoegen, M. Wuttig, and A. M. Lindenberg, “Picosecond electric-field-induced threshold switching in phase-change materials,” *Phys. Rev. Lett.*, vol. 117, p. 067601, Aug 2016.

[57] W. Zhang and T. Li, “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)*, 2009.

[58] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao, “Allocation wall: A limiting factor of Java applications on emerging multi-core platforms,” in *Proceeding of the ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.