

Emulating Hybrid Memory on NUMA Hardware

Shoaib Akram
Ghent University

Jennifer B. Sartor
Ghent University

Kathryn S. McKinley
Google

Lieven Eeckhout
Ghent University

Abstract—Non-volatile memory (NVM) has the potential to disrupt the boundary between memory and storage, including the abstractions that manage this boundary. Researchers comparing the speed, durability, and abstractions of hybrid systems with DRAM, NVM, and disk to traditional systems typically use simulation, which makes it easy to evaluate different hardware technologies and parameters. Unfortunately, simulation is extremely slow, limiting the number of applications and dataset sizes in the evaluation. Simulation typically precludes realistic multiprogram workloads and considering runtime and operating system design alternatives.

Good methodology embraces a variety of techniques for validation, expanding the experimental scope, and uncovering new insights. This paper introduces an emulation platform for hybrid memory that uses commodity NUMA servers. Emulation complements simulation well, offering speed and accuracy for realistic workloads, and richer software experimentation. We use a thread-local socket to emulate DRAM and the remote socket to emulate NVM. We use standard C library routines to allocate heap memory in the DRAM or NVM socket 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, and use 15 applications from three benchmark suites with various datasets and workload configurations. We show emulation enhances simulation results. The two systems confirm most trends, such as NVM write and read rates of different software configurations, increasing our confidence for predicting future system effects. In a few cases, simulation and emulation differ, offering opportunities for revealing methodology bugs or new insights. Emulation adds novel insights, such as the non-linear effects of multi-program workloads on write rates. We make our software infrastructure publicly available to advance the evaluation of novel memory management schemes on hybrid memories.

I. INTRODUCTION

Systems researchers and architects have long pursued bridging the speed gap between processor, memory, and storage. Despite many efforts, the increase in processor performance has consistently outpaced memory and storage speeds. Recent advances in memory technologies have the potential to disrupt this speed gap.

On the storage side, emerging non-volatile memory (NVM) technologies with speed closer to DRAM and persistence similar to disk promise to narrow the speed gap between processors and storage. Recent work engineers new filesystem abstractions, storage stacks, programming models, wear-out mitigation schemes, and prototyping platforms to integrate NVM in the storage hierarchy [1], [2], [3], [4], [5], [6], [7].

On the main memory side, NVM promises abundant memory. DRAM is facing scaling limitations [8], [9], and recent work combines DRAM and NVM to form hybrid main

memories [10], [11]. DRAM is fast and durable whereas NVM is dense and has low energy. Hardware mitigates NVM wear-out in both its storage and memory roles using wear-leveling and other approaches [11], [10], [12], [13], [14], while the OS keeps frequently accessed data in DRAM [15], [16], [17], [18], [19], [20]. Recent work also explores managed runtimes to mitigate wear-out [21], [22], tolerate faults [23], and keep frequently read objects in DRAM [24]. Collectively, prior research illustrates the substantial opportunities to exploit NVM across all layers including software and language runtimes.

In this paper, we expand on the methodologies for evaluating NVM and hybrid memories. The dominant evaluation methodology in prior work is simulation; see for example [11], [10], [12], [13], [14], [15], [16], [17], [19]. A few researchers have complemented simulation with architecture-independent measurements [25], [21], [22], but these measurements have limited value because they miss important effects such as CPU caching. This paper shows emulation confirms the results of simulation and architecture-independent analysis and enables researchers to explore richer software configurations.

The advantage of simulation is that it eases modeling new hardware features, revealing how sensitive results are to architecture. Its major limitation is that it is many orders of magnitude slower than running programs on real hardware. Because time and resources are finite, it thus reduces the scope and variety of architecture optimizations, application domains, implementation languages, and datasets one can explore. Popular simulators also trade off accuracy to speed up simulation [26], [27]. Furthermore, frequent hardware changes, microarchitecture complexity, and hardware’s proprietary nature make it difficult to faithfully model real hardware.

Other research evaluations are increasingly embracing em-

	Simulation	Architecture Independent	Emulation
Speed	Slow	Fast	Native
Hardware Diversity	High	N/A	Modest
Workload Diversity	Low	High	High
Production Datasets	✗	✓	✓
Full System Effects	✗	✗	✓
Realistic Hardware	✗	✗	✓

TABLE I: Comparing the strengths and weaknesses of evaluation methodologies for hybrid memories. *Emulation* enables native exploration of diverse workloads and datasets on realistic hardware.

ulation. For instance, emulating cutting-edge hardware on commodity machines to model: asymmetric multicores using frequency scaling [28], [29], die-stacked and hybrid memory using DRAM [18], [24], [1], and wearable memory using fault injection software [23]. Recent work using emulation for exploring hybrid memory is either limited to native languages [18], [1], or is limited to simplistic heap organizations in the case of managed languages [24] (See also Section VI). Table I compares the methodologies for evaluating hybrid memories, showing all can lead to insight and that emulation has distinct advantages in speed and software configuration.

We present the design, implementation, and evaluation of an emulation platform that uses widely available commodity NUMA hardware to model hybrid DRAM-NVM systems. We use the local socket to emulate DRAM and the remote socket to emulate NVM. All threads execute on the DRAM socket. Our heap splits virtual memory into DRAM and NVM virtual memory, which we manage using two free lists, one for each NUMA node by explicitly specifying where to allocate memory in the C standard library. We expose this hybrid memory to the garbage collector, which directs the OS where in memory (which NUMA node) to map heap regions. Contrary to most prior work, our platform handles both manual memory management routines from the standard C library and memory management using an automatic memory manager (garbage collector). We redesign the memory manager in the popular Jikes research virtual machine (RVM) to add support for hybrid memories. Our software infrastructure is publicly available at <link-anonymized-for-blind-reviewing>.

We evaluate this emulation platform on recently proposed write-rationing garbage collectors for hybrid memories [21]. Write-rationing collectors keep highly mutated objects in DRAM in hybrid DRAM-NVM systems to target longer NVM lifetime. We use 15 applications from three benchmark suites: DaCapo, Pjbb, and GraphChi; two input datasets; seven garbage collector configurations; and workloads consisting of one, two, and four application instances executing simultaneously. We find emulation results are very similar to simulation results and platform-independent measurements in most cases, but we can generate a lot more of them in the same amount of time and explore much richer software configurations and workloads. The emulator reveals trends not identified previously by simulation and platform-independent measurements. We summarize our key findings below.

- Simulation, emulation, and architecture-independent analysis reveal similar trends in write rate reductions and other characteristics of garbage collectors designed for hybrid memories, increasing our confidence in all the evaluation methodologies.
- Managed workloads use a lot of C/C++ code. Garbage collection strategies for hybrid memories should protect against both writes to the managed heap and writes to memory allocated using explicit C and C++ allocators.
- Executing multiple applications simultaneously super-linearly increases NVM write rates due to LLC interference, a configuration that is not practical to explore in simulation. A

major portion of the additional writes to memory are due to nursery writes. Kingsguard collectors isolate these writes on DRAM and thus are especially effective in multiprogrammed environments.

- Modern graph processing workloads use larger heaps and their write rates are also higher than widely used Java benchmarks. Future work should include such benchmarks when evaluating hybrid memories.
- Addressing large objects' behaviors are essential to memory managers for hybrid memories. Graph applications can see huge reductions in write rates when using Kingsguard collectors, because they have a lot of large objects that benefit from targeted optimizations.
- Changing a benchmark's allocation behavior or input changes write rates. Future work should eliminate useless allocations and use a variety of inputs for evaluating hybrid memories.
- LLC size impacts write rates. Future work should use suitable workloads with emulation on modern servers with large LLCs, or report evaluation for a range of LLC sizes using simulation.
- Graph applications wear PCM out faster than traditional Java benchmarks. Multiprogramming workloads can also wear PCM out in less than 5 years. Write limiting with Kingsguard collectors brings PCM lifetimes to practical levels.

II. BACKGROUND

This section briefly discusses characteristics of NVM hardware and the role of DRAM in hybrid DRAM-NVM systems. We then discuss write-rationing garbage collection [21] that protect NVM from writes and prolongs memory lifetime. We will evaluate write-rationing garbage collectors in Section V using our emulation platform.

A. NVM Drawbacks and Hybrid Memory

A promising NVM technology currently in production is phase change memory (PCM) [30]. PCM cells store information as the change in resistance of a chalcogenide material [31]. During a write operation, electric current heats up PCM cells to high temperatures and the cells cool down into an amorphous or a crystalline state that have different resistances. The read operation simply detects the resistance of the cell. PCM cells wear out after 1 to 100 million writes because each write changes their physical structure [10], [11], [31]. Writes are also an order of magnitude slower and consume more energy than in DRAM. Reading the PCM array is up to 4× slower than DRAM [10].

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 system. The large PCM capacity reduces disk accesses which compensates for its slow speed.

B. Garbage Collection

a) Generational 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 [32]. 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 exploits garbage collectors to manage hybrid memories [21], [23] and to improve PCM lifetimes.

b) *Write-Rationing Garbage Collection*: Write-rationing collectors keep frequently written objects in DRAM in hybrid memories to improve PCM lifetime [21]. They come in two main variants: The *Kingsguard-nursery* (KG-N) collector allocates nursery objects in DRAM and promotes all nursery survivors to PCM. The nursery is highly mutated and KG-N reduces write rates significantly compared to PCM-Only which leads to a longer PCM lifetime. *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 some nursery survivors but further improves PCM lifetime over KG-N.

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. Large Object Optimization (LOO) in KG-W allocates some large objects, chosen using a heuristic, in the nursery giving them time to die. Like standard collectors, 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. MetaData Optimization (MDO) places PCM object metadata in DRAM to eliminate garbage collector writes to object metadata.

Kingsguard collectors build on the best-performing collector in Jikes RVM: generational Immix (GenImmox) [33]. GenImmox uses a copying nursery and a mark-region mature space.

III. DESIGN AND IMPLEMENTATION

This section describes the design and implementation of our emulator for hybrid memory systems. It includes a hybrid-memory-aware memory manager built on top of the standard NUMA hardware platforms widely available today.

A. Heap layout and management

We allocate memory using the Linux OS calls for specifying a memory allocation on a 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. We use a NUMA specific version of the C memory allocator to call these

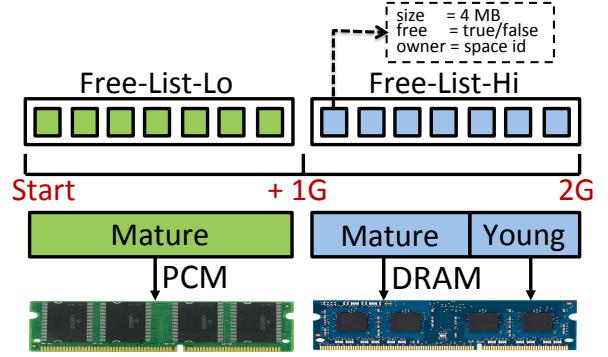


Fig. 1: The organization of our heap in hybrid memory. *Memory composition is exposed to the language runtime. Two free lists keep track of available virtual pages in DRAM and PCM.*

routines. We modify the Java Virtual Machine to call the C routines for DRAM and PCM allocation. Figure 1 shows the high-level layout of our heap in hybrid memory.

We use Jikes RVM, but our approach generalizes to other JVMs. Jikes RVM is a 32-bit virtual machine, and each program has 4 GB of virtual memory. The Linux OS and system libraries use the low virtual memory for its own purposes. We use the upper 2 GB heap for the Java heap. This memory is sufficient for our applications, although it is possible to use more than 2 GB. We partition the heap into two parts. Each 1 GB portion is logically divided into 4 MB chunks and managed independently by a free-list data structure. Figure 1 shows Free-List-Hi and Free-List-Lo that keep track of free DRAM and PCM memory respectively. Each entry in the free-list contains meta-information about the chunk: (1) size of the chunk, (2) status of the chunk (free or in use), and (3) the current owner of the chunk. The lower 1 GB portion in virtual memory maps to PCM, and the upper portion maps to DRAM.

Jikes RVM includes a memory management tool kit (MMTk) to manage the Java heap. Standard MMTk configurations flexibly manage portions of the heap using different allocation and collection mechanisms. Each such portion is called a space in MMTk terminology. For example, the nursery is a contiguous space and uses a bump pointer allocator, and its memory is reclaimed using copying collection. Each space in our implementation reserves virtual memory by requesting the allocator associated with Free-List-Lo or Free-List-High. 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 on DRAM or PCM.

Alternative approaches are possible although their efficiency might be low. For instance, a monolithic Java heap with a single free-list would require unmapping free chunks from the physical memory. Because otherwise, 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.

Spaces such as the nursery have their address ranges reserved at boot-time. On the other hand, mature spaces use a request mechanism to acquire chunks. These spaces share the pool of available chunks with other spaces. Both types of spaces can be placed in either DRAM or PCM. Each space specifies DRAM or NVM as a flag in its constructor.

Similar to the baseline design, we place the young generation (nursery) at one side of the 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.

MMTK uses `mmap()` for reserving virtual memory if none is available as indicated by the free lists. To bind a virtual memory range to a particular socket, we call `mbind()` with the socket number after each call to `mmap()`.

B. Emulation on NUMA Hardware

Hardware requirements: Our hardware requirement is a commodity NUMA platform with two sockets. We require both sockets 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 emulates PCM.

Space to Socket Mapping: Table II shows the space to socket mapping in three of the 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 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.

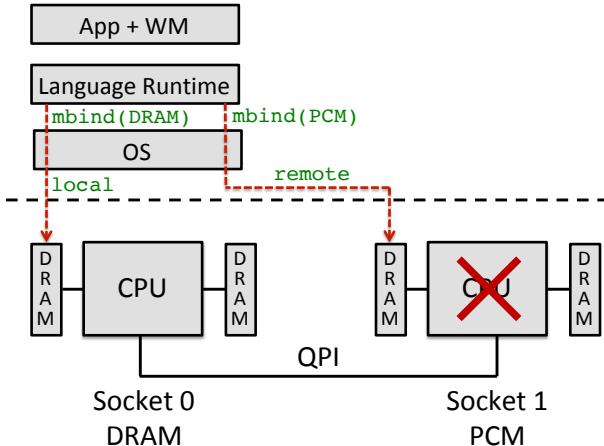


Fig. 2: Our platform for hybrid memory emulation. The application and write rate monitor (WM) runs on socket 0. The memory on socket 0 is DRAM and socket 1 is PCM.

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

TABLE II: Spaces in Kingsguard collectors and their mapping to socket 0 (S0) or socket 1 (S1). S0 is DRAM and S1 is PCM. KG-N does not use an observer space. KG-W uses a mature, large, and metadata space in both DRAM and PCM. *Our virtual heap layout enables a range of collector configurations for hybrid memory.*

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

Thread to Socket mapping: When a particular thread uses the C or C++ library to allocate memory, the OS places that memory on the socket where the thread is executing. Thus we have to control to which socket each thread is mapped. Our JVM calls down to these C and C++ libraries for allocation. For the Kingsguard configurations, we always bind threads, including application and JVM service threads, to socket 0 (see Figure 2). When emulating a PCM-Only system, 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.

This work focuses on PCM lifetimes. PCM lifetime in years depends directly on its write rate. We measure write rates on our emulation platform using a write rate monitor (WM in Figure 2) that also runs on socket 0. Threads are not pinned to specific cores and we use the default OS scheduler. We experimentally find out that scheduling WM on socket 0 leads to more deterministic write rate measurements. When scheduled on socket 1 and all allocation isolated to socket 0, we continue to observe memory traffic on socket 1.

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 [34], [35], [36], [37]. As a comparison point, it took Hotspot [38], [39] 10 years from the publication of the G1 collector [40] to its release. 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 [41], [33], [42]. Its memory management tool kit (MMTk) [41] 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 [43] which makes it easy to implement a range of heap organizations.

Evaluation Metrics: We use two metrics to evaluate write-rationing garbage collectors: write rate and execution time. PCM lifetime is directly proportional to its write rate [11], [44], [45]. We report execution time both for single application and multiprogrammed workloads. Our multiprogrammed workloads consist of multiple instances of the same application. On a properly provisioned platform, all instances should finish execution at the same time. However, due to shared resources, there is variation in the execution time of individual instances. We find the variation on our platform to be low.

Measurement Methodology: We use best practices from prior work for evaluating Java applications on our emulation platform [46], [47]. To eliminate non-determinism due to the optimizing compiler, we use replay compilation as used in prior work. 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 Intel’s Performance Counter Monitor framework for measuring write rates. We use the `pcm-memory` utility in the framework for measuring write rates. 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.

Java Applications: We use 15 Java applications from three diverse sources: 11 DaCapo [48], `pseudojbb2005` (`Pjbb`) [49], and 3 applications from the GraphChi framework for processing graphs [50]. The GraphChi applications we use are: (1) `page rank` (`PR`), (2) `connected components` (`CC`), and (3) `ALS` matrix factorization (`ALS`). Compared to recent work [21], 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 [51], and `pmd.S` eliminates a scalability bottleneck in the original version due to a large input file [52]. 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 [53], 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.

Even though we do not include C and C++ benchmarks in this work, many of our Java benchmarks exhibit the common behavior of mixing some C/C++ with Java because Java

standard features, such as IO, use C implementations. For example, the DaCapo benchmarks execute a lot of C code [54].

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.

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. KG-B and its variants use a 12 MB nursery for DaCapo and `Pjbb`, and a 96 MB nursery for the GraphChi applications. The reason to use KG-B is to understand if simply using large nurseries, equal to the sum of nursery and observer space in KG-W, could reduce PCM write rates similar to KG-W.

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 and 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 to have the observer space twice as large as the nursery. Prior work reports this to be a good compromise between tenured garbage and pause time.

We compare to PCM-Only with the baseline generational Immix collector [33]. We configure the baseline collector similar to prior work [21]. All our experiments use two garbage collector threads.

Nursery and Heap Sizes: Nursery size has an impact on performance, response time, and space efficiency [55], [41], [56], [57]. Similar to prior work [21], we use a nursery of 4 MB 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 GraphChi applications. We use a modest heap size that is twice the minimum heap size. Our heap sizes reflects those used in recent work [58], [33], [59], [57], [60].

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 in both sockets. The 20 MB LLC on each processor is shared by all cores. 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 by QPI links that support up to 8 GT/s. We use Ubuntu 12.04.2 with 3.16.0 kernel.

V. RESULTS

This section evaluates Kingsguard collectors using emulation. We first compare emulation results to results of simulation and architecture-independent analysis. Emulation results match prior results boosting our confidence in our newly proposed methodology. We use emulation to explore a richer space of software configurations and workloads. This enables us to discuss previously unseen writes to memory due to interference patterns in multiprogrammed workloads, simpler heap organizations for graph applications, and the implications of production datasets. We conclude this section with reporting raw write rates and PCM lifetimes in years.

A. Quantitative Comparison of Evaluation Methodologies

Each evaluation methodology for hybrid memories has its strengths and weaknesses. Simulation models real hardware features but limits evaluation to a few Java applications. Architecture-independent (ARCH-INDP) studies are fast and improve application diversity but assume unrealistic hardware. ARCH-INDP counts the writes to virtual memory without taking cache effects into account [21]. Our goal in this section is to explore if the major conclusions hold regardless of the evaluation methodology, including the reduction in PCM write rates. Lacking PCM hardware, we can not compare accuracy.

We first compare emulation results to simulation results. We reproduce simulation results from previous work [21]. 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 use two configurations of simulated hardware: (1) 4 cores and 4 MB LLC, and (2) 4 cores and 20 MB LLC, which more closely matches the emulation platform.

Table III shows the percentage reduction in PCM write rates reported by the three methodologies. Intuition suggests Kingsguard collectors should be more effective with a smaller LLC. Smaller LLC absorbs fewer writes which increases the writes to PCM memory. The results from two simulated systems in Table III confirm this intuition.

LLC pressure is high on the simulated system with a 4 MB LLC and the emulation platform running a multiprogrammed workload with four instances. The nursery size of the seven simulated benchmarks is 4 MB, and the nursery is highly mutated, so using an emulation system with four applications creates more LLC interference and thus more PCM writes, similar to the simulated system with a 4 MB LLC. The average reduction in PCM write rate for both cases is similar.

Simulated results with a 4 MB LLC are different from emulation results with one and two program workloads. This is because the nursery is a major source of writes which are absorbed by the 20 MB LLC in the emulation platform.

We observe that simulation results for KG-N and a 20 MB LLC report a 4% reduction in PCM writes. On the contrary, emulation results report a 29% reduction in PCM writes. This discrepancy is due to full-system effects in the emulation platform. When emulating KG-N, we place explicit memory allocations by the C and C++ libraries in DRAM. In PCM-Only,

	SIM 4 MB LLC	SIM 20 MB LLC	EMU N = 1	EMU N = 2	EMU N = 4
KG-N	81%	4%	78%	29%	45%
KG-B	85%	-7%	80%	41%	54%
KG-W	91%	62%	97%	66%	73%

TABLE III: Comparing PCM write reduction using simulation (SIM), architecture independent (ARCH-INDP) analysis, and emulation (EMU) on 7 simulated benchmarks. N is the number of program instances in our multiprogrammed workloads. *Simulation results confirm emulation results. The differences are due to cache sizes and full system effects.*

	ARCH-INDP	EMU N = 1	EMU N = 2	EMU N = 4
KG-N	70%	47%	58%	77%
KG-B	76%	44%	60%	80%
KG-W	94%	77%	84%	90%

TABLE IV: Comparing PCM write reduction using ARCH-INDP and EMU for all benchmarks. *ARCH-INDP over-reports the reduction in PCM writes for 1 and 2 program workloads. For a balanced workload that utilizes all cores on Socket 0, EMU results match ARCH-INDP results.*

these allocations are placed in PCM, which is why emulation reports a larger reduction in PCM writes for KG-N.

We observe another discrepancy between the simulation and the emulation results for KG-B with a 20 MB LLC. Simulation results suggest increasing the nursery size to 12 MB leads to more PCM writes compared to a 4 MB nursery. On the other hand, emulation results suggest a 41% reduction in PCM write rate. This discrepancy opens up opportunities for future investigations. Bugs in one or both environments could misreport PCM writes with larger nurseries. Alternatively, the discrepancy could arise because of full system effects – the simulation environment isolates Java heap allocations from OS and native library allocations. We leave investigating this discrepancy further to future work.

ARCH-INDP results over predict reductions when compared to emulation results with a single program instance. They over predict because ARCH-INDP counts successive writes to PCM virtual memory as writes to PCM physical memory. In reality, some of those writes are filtered by the CPU caches. When workloads exhibit large LLC interference, such as multiprogrammed workloads with four instances, emulation results are in the ballpark of ARCH-INDP. We observe the same behavior when comparing ARCH-INDP and emulation results for all benchmarks in Table IV.

Finding 1. *LLC size impacts PCM write rates. Simulation and emulation results converge with similar nursery to LLC size ratios. Full-system effects may cause discrepancies for some configurations.*

Finding 2. *Architecture-independent metrics over-report the reduction in PCM write rates.*

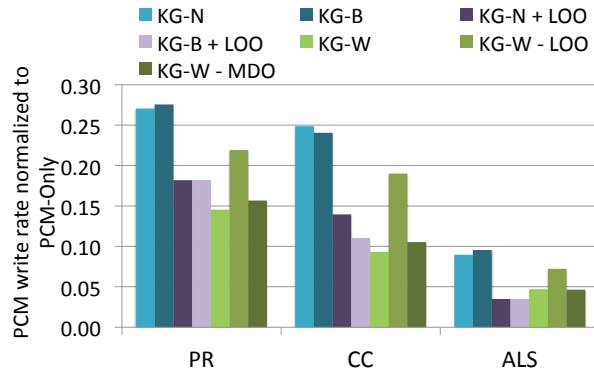


Fig. 3: PCM write rates with various Kingsguard collectors normalized to PCM-Only for GraphChi applications. *KG-B + LOO* works well for graph applications.

B. Workload Analysis Using Emulation

This section evaluates write-rationing garbage collectors much more fully than prior work. All trends previously observed for a narrow set of applications and datasets are confirmed by emulation on a richer workload space. These results make the case for using PCM as main memory even stronger.

a) *Garbage Collection Strategies for GraphChi*: Contrary to prior work, emulation provides us the opportunity to evaluate Kingsguard collectors for GraphChi applications. We find these applications allocate large objects frequently. We show that combining the optimization for large objects in KG-W with the heap organization of KG-N is effective. This configuration simplifies heap management and improves performance. GraphChi applications have more mature space collections than DaCapo and Pjbb. We tease apart the impact of the metadata optimization here by evaluating KG-W-MDO. We also evaluate KG-W-LOO to tease apart the impact of the large object optimization from KG-W.

We show the write rates for single-program workloads normalized to PCM-Only in Figure 3. The absolute write rates increase for multiprogrammed workloads but the normalized trends remain the same. GraphChi applications benefit from KG-N and KG-B that allocate new objects in a DRAM nursery. This reduces write rates by 74%, 75%, and 91% for PR, CC, and ALS respectively. KG-B uses a bigger nursery compared to KG-N but still reduces write rates similar to KG-N. This confirms previous findings with simulated benchmarks that we need novel heap organizations and other optimizations to reduce PCM write rates further.

The graph applications allocate large objects and some follow the generational hypothesis and thus benefit from the LOO optimization. KG-N + LOO and KG-B + LOO both reduce write rates on top of KG-N and KG-B respectively. KG-N + LOO reduces write rate by up to an additional 11% compared to KG-N. KG-B + LOO is even more effective for PR and CC: a 3% additional reduction in write rate. KG-N + LOO and KG-B + LOO are effective and have smaller execution time

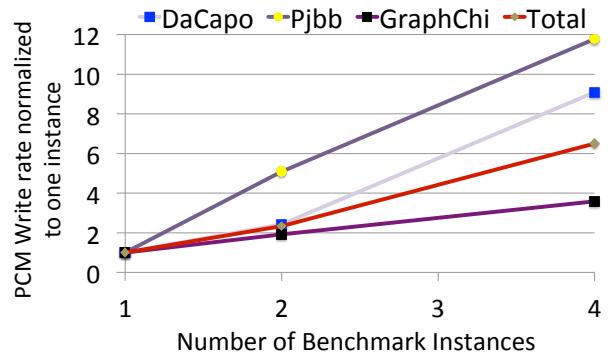


Fig. 4: Average PCM write rates with PCM-Only normalized to single-program write rates. *Writes rates increase with the number of benchmark instances. The write rates of Pjbb and DaCapo grow super-linearly from 1 to 4 instances.*

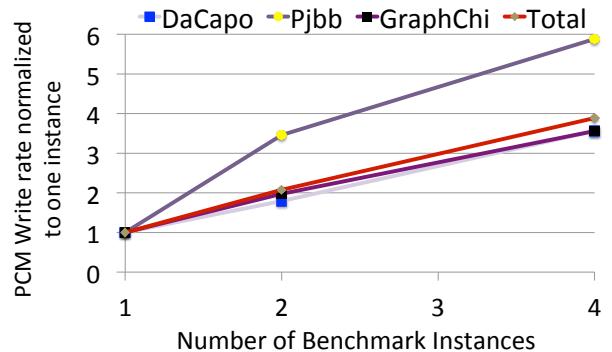


Fig. 5: Average PCM write rates with KG-W normalized to single-program write rates. *The growth in write rates is close to linear except Pjbb.*

overhead compared to KG-W. Nevertheless, KG-W reduces write rates to PCM more than KG-B + LOO for PR and CC.

Excluding LOO from KG-W (KG-W-LOO) increases the write rate because of large object allocation in PCM of short-lived objects. Large object allocation in PCM further fills up the heap quickly leading to more frequent mature collections. Mature collections are a source of PCM writes because of the collector updates to the object mark states. The write rate increases by 3.3× for PR, 2.6× for CC, and 1.5× for ALS.

Without the metadata optimization (MDO), PCM write rates increase, proving that eliminating metadata writes during mature collections is effective. With one instance of PR, the write rate increases by 1.32× for PR and 1.13× for CC. MDO benefits multiprogrammed workloads even more (not shown).

Finding 3. *Graph applications allocate many large objects that benefit greatly from Kingsguard collectors that use the large object optimization.*

b) *Interference in Multiprogrammed Workloads*: Long simulation times impede the evaluation of hybrid memories for multiprogrammed Java workloads. The native execution speed of these workloads on our emulation platform reveals interference patterns in the LLC which results in writes to PCM memory. Figure 4 and Figure 5 shows the growth in average PCM write rates for PCM-Only and KG-W for each

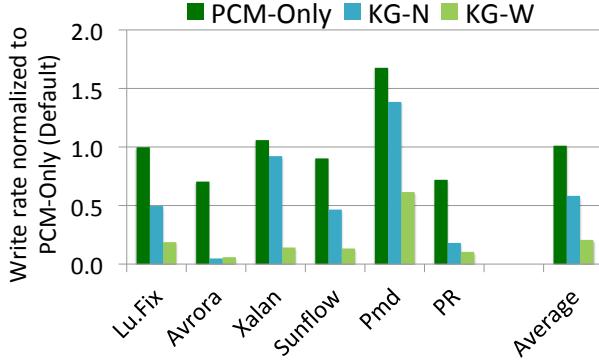


Fig. 6: Write rates with large datasets normalized to write rates with default datasets (PCM-Only). Normalized write rates may change with production datasets. PCM write rate still reduces with KG-N and KG-W.

benchmark suite and for all benchmarks.

We observe a variety of trends in write rates from the three suites. On average for PCM-Only, the increase in write rate from 1 to 2 program instances is $2.3\times$, which is as we expect, but from 1 to 4 instances, the increase is non-linear at $6.4\times$. DaCapo applications encounter high interference in the LLC. The average increase in write rate from 1 to 4 instances for DaCapo is $9\times$ ($2.4\times$ from 1 to 2 instances). The increase for Pjbb is even higher. From 1 to 2 instances, the write rate increases by $5\times$, and from 1 to 4 instances, the write rate increases by $12\times$. GraphChi applications on average show a linear trend. The increase in write rate from 1 to 4 instances is $1.9\times$, and $3.5\times$ from 1 to 4 instances.

Contrary to PCM-Only, KG-N and KG-W exhibit a linear increase in write rates from 1 to 2 and 4 program instances across the three suites: with 2 instances, the increase is $1.8\times$, $2.8\times$, and $2.6\times$ for DaCapo, Pjbb, and GraphChi; with four instances, the increase is $3.1\times$, $4.8\times$, and $4.7\times$ respectively. With KG-W, the increase is less than linear except for Pjbb, which increases $6\times$ with 4 program instances.

Finding 4. *PCM write rates grow super-linearly with the number of concurrently running program instances for two popular Java benchmark suites. Write rationing garbage collection significantly reduces the growth in write rates.*

c) *Modest versus Production Datasets:* The native speed of emulation admits larger datasets, previously unexamined. The normalized writes rates for PCM-Only with large datasets shown in Figure 6 follow three trends: The write rates of lusearch.fix and xalan stay the same. The write rates of avrora, sunflow, and PR for PCM-Only decrease by $0.7\times$, $0.9\times$, and $0.72\times$ compared to default datasets. The compute-to-write ratio of these applications increases with larger inputs. Conversely for pmd, the write rate increases by $1.7\times$. Although absolute write rates change for some benchmarks, we observe a similar reduction in PCM write rates with KG-N and KG-W.

Finding 5. *Production datasets sometimes shift the balance between compute and memory-writes, changing PCM write rates.*

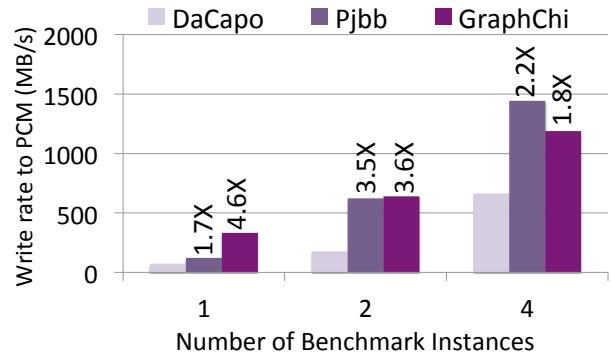


Fig. 7: Average write rates in MB/s for DaCapo, Pjbb, and GraphChi with PCM-Only. The numbers on top of Pjbb and GraphChi bars show the PCM write rate normalized to DaCapo. Write rates increase for multiprogrammed workloads. Pjbb and GraphChi have greater write rates compared to DaCapo.

d) *Classical versus Modern Suites:* The DaCapo benchmark suite is the dominant choice for prior research in garbage collection and some studies use Pjbb. We compare the average write rates of Pjbb and GraphChi to DaCapo in Figure 7. The average write rates of Pjbb and GraphChi are greater than DaCapo. Pjbb and GraphChi also have the largest heap sizes of all of our benchmarks. The average heap size of DaCapo is 100 MB, Pjbb is 400 MB, and GraphChi is 512 MB. Pjbb has $1.7\times$ the write rate of DaCapo with single-program workloads. Although we expect both Pjbb and GraphChi to have higher write rates than DaCapo, it is interesting that GraphChi has a $4.7\times$ higher write rate than DaCapo.

Finding 6. *Future studies on hybrid memories should use a diversity of applications, including large heaps.*

The write rates of Pjbb and GraphChi are higher even for multiprogrammed workloads. The difference is less pronounced compared to single-program workloads because with four instances, DaCapo applications on average experience greater interference in the LLC. The DaCapo rates increase super-linearly whereas the increase is less than super-linear for Pjbb and GraphChi.

The difference in write rates between DaCapo and GraphChi is less pronounced with KG-N and KG-W. 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 $1.5\times$ and $1.6\times$ higher than DaCapo. Thus, a large reason for the gap in write rates of DaCapo and GraphChi for PCM-Only is the nursery writes.

The write rates of Pjbb with KG-N and KG-W are still higher compared to the average DaCapo rates. For instance, with KG-W and single-program workloads, the write rate of Pjbb is $3\times$ that of DaCapo ($2\times$ for KG-N). For two and four instances with KG-W, Pjbb incurs a $5.7\times$ and $5\times$ higher write rate compared to DaCapo.

Finding 7. *Pjbb and GraphChi have higher write rates than DaCapo.*

e) *PCM Write Rates and Implications for Lifetime:* We now show the raw write rates to PCM for our benchmarks

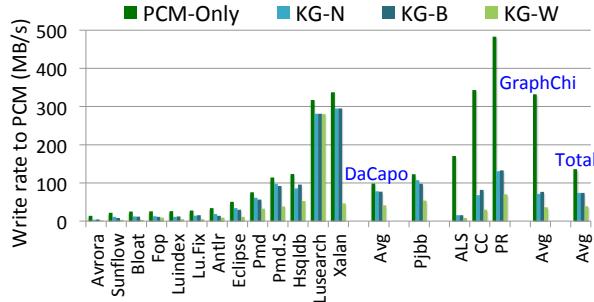


Fig. 8: Write rates in MB/s for one instance workloads. Benchmarks exhibit a range of write rates. Applications that allocate large objects frequently have the highest write rates.

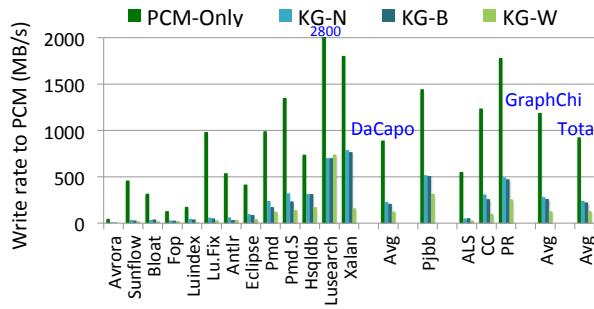


Fig. 9: Write rates in MB/s with 4 instance workloads. Write rates reduce significantly across all benchmarks with Kingsguard collectors.

and reduction in write rates using KG-W. We also discuss PCM lifetime in years for our workloads. Lifetime is a linear function of write rate and PCM cell endurance. We compute PCM lifetimes similar to prior work assuming a PCM write endurance of 10 M writes per cell [10], [11], [12], [21]. We assume a 32 GB PCM system with hardware wear-leveling that delivers endurance within 50% of the theoretical maximum [12]. Table V shows worst-case PCM lifetimes in years for the three benchmark suites. We choose the shortest lifetime of all benchmarks for DaCapo and GraphChi. We only consider the fixed version of lusearch in the worst-case lifetime analysis.

Figure 8 shows the write rates for PCM-Only and three Kingsguard configurations for single programs. The average PCM write rate for PCM-Only is 126 MB/s and write rates vary from 14 MB/s for *avroa* to 480 MB/s for *PR*. *lusearch* is excluded from the average. Higher write rates of GraphChi applications limit memory lifetime of PCM-Only to only 10.5 years with single programs. The worst-case lifetime in DaCapo is 14 years for *xalan*. Wear-leveling and write filtering by LLC alone can make PCM last for 41 years when running a single instance of *Pjbb*.

Finding 8. Graph processing applications wear out PCM much more quickly than DaCapo and *Pjbb*.

Of all the DaCapo benchmarks, *lusearch* has the highest write rate of 320 MB/s. Interestingly, *lusearch.fix* fixes an allocation bug in the original *lusearch* and has a write rate of only 27 MB/s. We also observe a change in write rate between the two versions of *pmd*. The original benchmark has a write

rate of 75 MB/s on our platform. The version of the benchmark that removes an input file for better scaling with the number of threads, *pmd.S*, has a write rate of 114 MB/s. The execution time of *pmd.S* reduces significantly compared to *pmd* leading to this higher write rate.

A widely used application in the DaCapo suite, *eclipse*, has a write rate of 50 MB/s. This write rate is less than transaction (*hsqldb* and *Pjbb*) and graph processing applications. On the other hand, it is higher than applications that do lexical analysis such as *antlr* and *bloat*. *ALS* with 170 MB/s has the lowest write rate of the three GraphChi applications.

Finding 9. Applications from different benchmark suites and from different domains within a suite exhibit a variety of PCM write rates. Applications that allocate large objects abundantly have higher write rates.

Finding 10. Allocation behavior and input sets influence write rates.

Kingsguard collectors significantly reduce PCM write rates across the three benchmark suites. KG-N reduces the average write rate by 50% for single programs. The average write rate of KG-N is 60 MB/s. KG-B with its bigger nursery results in the same PCM write rates as KG-N. GraphChi applications write much less to PCM with KG-N. This shows that the nursery is highly mutated even in modern graph processing applications. The benchmarks that do not benefit a lot from KG-N are those that: (1) profusely allocate large short-lived data structures such as *lusearch* and *xalan*, and (2) have more mature-object writes than nursery writes, such as *Pjbb* [21].

Finding 11. Simply using DRAM for larger nurseries does not reduce PCM write rates in hybrid memory systems.

KG-W reduces the average write rate by 80% and the raw average write rate is 24 MB/s. These low write rates greatly improve PCM lifetime. PCM lifetime with KG-W for single programs is practical across all three benchmark suites. For instance, GraphChi applications with KG-W will wear out PCM after 72 years. PCM will also be used for persistent storage which can have many more writes. The lifetimes shown in Table V are therefore optimistic.

Figure 9 shows PCM write rates of 4-program workloads. Write rates increase a lot and up to 2.8 GB/s for *lusearch*. KG-N reduces the write rate significantly for the 4-program *lusearch* workload. The increase in write rates for multiprogrammed workloads has implications for PCM lifetimes. The

	N=1		N=4	
	PCM-Only	KG-W	PCM-Only	KG-W
DaCapo	15	109	2.8	31.5
Pjbb	41	94	3.5	16
GraphChi	10.5	72	2.9	20

TABLE V: PCM lifetime in years for single-program (N=1) and 4-program (N=4) workloads. *GraphChi* and multiprogrammed workloads quickly wear out PCM. Kingsguard collectors make PCM practical for all workloads.

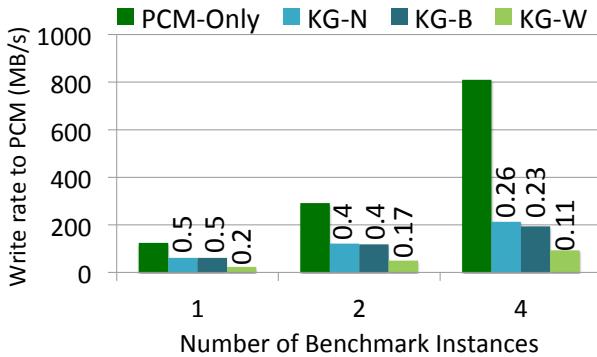


Fig. 10: Average write rates in MB/s with varying number of benchmark instances. The numbers on top of KG-N, KG-B, and KG-W bars show write rates normalized to PCM-Only. *Write rates increase with increasing number of benchmark instances. Kingsguard collectors are more effective for multiprogrammed workloads.*

average lifetime for the DaCapo suite is not even 5 years. Lifetimes of Pjbb and GraphChi are worse.

Finding 12. *Multiprogramming workloads can wear out PCM memory in less than 5 years.*

Figure 10 compares average write rates of single-program and multiprogrammed workloads with PCM-Only, KG-N, and KG-W. Write rates reach close to 1 GB/s with two program instances and up to 2.8 GB/s with four program instances. Fortunately, write rates to PCM drop to less than 100 MB/s on average with KG-W across all workloads. Figure 10 also shows that KG-N, KG-B, and KG-W are more effective in normalized terms for multiprogrammed workloads. Write rates increase due to interference in the LLC and most of the interference is due to nursery writes. Contrary to KG-N reducing the write rate to PCM by 50% with one instance, KG-N reduces the write rate to PCM by 80% with 4 program instances.

Finding 13. *Concurrently running applications in multiprogrammed environments incur LLC interference due to nursery writes. Kingsguard collectors are highly effective in such environments.*

Table V shows that Kingsguard collectors bring PCM lifetimes to practical levels for multiprogrammed workloads. The worst-case lifetime is more than 15 years for DaCapo, Pjbb, and GraphChi. Software and hardware approaches together can make PCM a practical replacement for DRAM.

f) Execution Time: Overall across single-program and multiprogrammed workloads and compared to KG-N, KG-B slightly reduces the execution time, and KG-W increases the execution time. The average reduction with KG-B is 3%, and average increase with KG-W is 10% for single programs. The results are in the ballpark for multiprogrammed workloads. hsqldb suffers the highest overhead of 28%. An exception with KG-W is bloat whose execution time reduces up to 12%. The low survival rate of observer collections leads to fewer mature collections which improve overall application performance.

Finding 14. *There is a price to pay for severely limiting writes to PCM. KG-W's overhead ranges from 0-28%.*

VI. RELATED WORK

Now we discuss related work on methods to evaluate hybrid memories, and managed runtimes for emerging hardware.

A. Evaluation methodologies

Prior work uses emulation to evaluate emerging memories [18], [24], [1]. Oskin et al. use a NUMA platform for emulating die-stacked DRAM [18]. Their evaluation only considers applications written in C. Dulloor et al. emulate hybrid DRAM-NVM memory on a NUMA platform but use it to evaluate filesystems for persistent object storage.

Two platforms today enable executing Java applications on top of simulated hardware in a reasonable time: (1) Jikes RVM on top of Sniper [59], and (2) Maxine VM on top of ZSim [61]. Both Sniper and ZSim are a cycle-level multicore simulators that trade off some accuracy for speedy evaluations [26], [27]. In their publicly available versions, both platforms lack support for full-system simulation; favoring speed over detail.

Cao et al. use emulation to evaluate managed runtimes for hybrid memories 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 to measure write rates of Java workloads that are run using replay compilation [46], [47].

B. Managed runtimes for emerging hardware

Prior work has looked into tailoring the managed runtime for hybrid memories. Wang et al. use DRAM in hybrid DRAM-NVM systems for allocating frequently read objects [24]. 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.

Gao et al., use the managed runtimes to tolerate PCM failures [23]. The hardware informs the OS of defective lines which in turn communicates faulty lines to the garbage collector. The garbage collector masks the defective lines and moves data away from them.

We discussed write-rationing garbage collection for hybrid memories [21] proposed by Akram et al., in Section II. Recent work predicts write-intensive objects using offline profiling to reduce the overheads of online monitoring [22].

VII. CONCLUSIONS

Advances in non-volatile memory (NVM) technologies have implications for the whole computing stack. Researchers need fast and accurate methodologies for evaluating NVM as memory and storage. This work introduces an emulation platform built using widely available NUMA servers to accurately measure read and write rates and the performance of hybrid memories that combine DRAM and NVM. This platform 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. We compare emulation to simulation and architecture-independent analysis, showing they have similar trends. With the 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, and benefit greatly from write-rationing collectors. Multiprogrammed environments see a super-linear growth in write rates compared to running single programs. This growth goes away with write-rationing collectors. Although simulation and emulation both have their place, emulation adds the ability to explore a richer software design and workload space.

REFERENCES

- [1] 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 Ninth European Conference on Computer Systems (EuroSys)*, 2014.
- [2] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, “Better i/o through byte-addressable, persistent memory,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.
- [3] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, “Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.
- [4] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, “Mojim: A reliable and highly-available non-volatile memory system,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.
- [5] J. Xu and S. Swanson, “Nova: A log-structured file system for hybrid volatile/non-volatile main memories,” in *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*, 2016.
- [6] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, “Nova-fortis: A fault-tolerant non-volatile main memory file system,” in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.
- [7] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.
- [8] 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 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.
- [9] O. Mutlu and L. Subramanian, “Research problems and opportunities in memory systems,” *Supercomputing Frontiers and Innovations*, vol. 1, no. 3, Oct 2014.
- [10] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable dram alternative,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.
- [11] 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.
- [12] 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 *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009.
- [13] 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 *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.
- [14] A. Seznec, “A phase change memory as a secure main memory,” *IEEE Computer Architecture Letters*, vol. 9, no. 1, Jan 2010.
- [15] L. E. Ramos, E. Gorbatov, and R. Bianchini, “Page placement in hybrid memory systems,” in *Proceedings of the International Conference on Supercomputing (ICS)*, 2011.
- [16] 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.
- [17] 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 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.
- [18] M. Oskin and G. H. Loh, “A software-managed approach to die-stacked dram,” in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015.
- [19] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, “Utility-based hybrid memory management,” in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.
- [20] L. Liu, H. Yang, Y. Li, M. Xie, L. Li, and C. Wu, “Memos: A full hierarchy hybrid memory management framework,” in *IEEE 34th International Conference on Computer Design (ICCD)*, 2016.
- [21] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, “Write-rationing garbage collection for hybrid memories,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018.
- [22] S. Akram, K. S. McKinley, J. B. Sartor, and L. Eeckhout, “Managing hybrid memories by predicting object/write intensity,” in *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming’18> Companion)*, 2018.
- [23] 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 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.
- [24] C. Wang, T. Coa, J. Zigman, F. Lv, Y. Zhang, and X. Feng, “Efficient management for hybrid memory in managed language runtime,” in *IFIP International Conference on Network and Parallel Computing (NPC)*, 2016.
- [25] S. V. den Steen, S. Eyerman, S. D. 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.
- [26] 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.
- [27] D. Sanchez and C. Kozyrakis, “Zsim: Fast and accurate microarchitectural simulation of thousand-core systems,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.
- [28] 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.
- [29] 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 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.
- [30] 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>
- [31] 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.
- [32] D. Ungar, “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)*, 1984.
- [33] 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.
- [34] 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.
- [35] 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.

- [36] S. M. Blackburn, P. Cheng, and K. S. McKinley, “Oil and water? High performance garbage collection in Java with MMTk,” in *International Conference on Software Engineering (ICSE)*, May 2004.
- [37] 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 *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2009.
- [38] OpenJDK Group, “Hotspot VM.” [Online]. Available: <http://openjdk.java.net/groups/hotspot/>
- [39] M. Paleczny, C. Vick, and C. Click, “The java hotspot server compiler,” in *Usenix Java Virtual Machine Research and Technology Symposium (JVM)*, April 2001.
- [40] D. Detlefs, C. Flood, S. Heller, and T. Printezis, “Garbage-first garbage collection,” in *Proceedings of the 4th International Symposium on Memory Management (ISMM)*, 2004.
- [41] 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.
- [42] 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.
- [43] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking, “Barriers reconsidered, friendlier still!” in *Proceedings of the Eleventh ACM SIGPLAN International Symposium on Memory Management (ISMM)*, jun 2012.
- [44] M. K. Qureshi, “Pay-as-you-go: Low-overhead hard-error correction for phase change memories,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.
- [45] 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 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.
- [46] J. Ha, M. Gustafsson, S. M. Blackburn, and K. S. McKinley, “Microarchitectural characterization of production JVMs and Java workloads,” in *IBM CAS Workshop*, 2008.
- [47] 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.
- [48] 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.
- [49] 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>
- [50] A. Kyrola, G. Blelloch, and C. Guestrin, “Graphchi: Large-scale graph computation on just a pc,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.
- [51] 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.
- [52] 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.
- [53] J. Leskovec and A. Krevl, “SNAP Datasets: Stanford large network dataset collection,” <http://snap.stanford.edu/data>, 2014.
- [54] B. Lee, M. Hirzel, R. Grimm, and K. S. McKinley, “Debugging mixed-environment programs with blink,” *Softw. Pract. Exper.*, vol. 45, no. 9, 2015.
- [55] A. W. Appel, “Simple generational garbage collection and fast allocation,” *Softw. Pract. Exper.*, vol. 19, no. 2, Feb. 1989.
- [56] D. Ungar and F. Jackson, “An adaptive tenuring policy for generation scavengers,” *ACM Trans. Program. Lang. Syst.*, vol. 14, no. 1, Jan. 1992.
- [57] 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 *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2009.
- [58] 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.
- [59] J. B. Sartor, W. Heirman, S. M. Blackburn, L. Eeckhout, and K. S. McKinley, “Cooperative cache scrubbing,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, 2014.
- [60] 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 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.
- [61] A. Rodchenko, C. Kotselidis, A. Nisbet, A. Pop, and M. Luján, “Maxsim: A simulation platform for managed applications,” in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.