Performance Evaluation of Intel Optane Memory for Managed Workloads

SHOAIB AKRAM, Australian National University, Canberra

Intel Optane memory offers non-volatility, byte-addressability, and high capacity. It suits managed workloads that prefer large main memory heaps. We investigate Optane as the main memory for managed (Java) workloads, focusing on performance scalability. As the workload (core count) increases, we note Optane's performance relative to DRAM. A few workloads incur a slight slowdown on Optane memory, which helps conserve limited DRAM capacity. Unfortunately, other workloads scale poorly beyond a few core count.

This paper investigates scaling bottlenecks for Java workloads on Optane memory, analyzing the application, runtime, and microarchitectural interactions. Poorly scaling workloads allocate objects rapidly and access objects in Optane memory frequently. These characteristics slow down the mutator and substantially slow down garbage collection (GC). At the microarchitecture level, load, store, and instruction miss penalties rise. To regain performance, we partition heaps across DRAM and Optane memory, a hybrid that scales considerably better than Optane alone. We exploit state-of-the-art GC approaches to partition heaps. Unfortunately, existing GC approaches needlessly waste DRAM capacity because they ignore run-time behavior.

This paper also introduces performance impact-guided memory allocation (PIMA) for hybrid memories. PIMA maximizes Optane utilization, allocating in DRAM only if it improves performance. It estimates the performance impact of allocating heaps in either memory type by sampling. We target PIMA at graph analytics workloads, offering a novel performance estimation method and detailed evaluation. PIMA identifies workload phases that benefit from DRAM with high (94.33%) accuracy, incurring only a 2% sampling overhead. PIMA operates stand-alone or combines with prior approaches to offer new performance versus DRAM capacity trade-offs. This paper opens up Optane memory to a real-life role as the main memory for Java workloads.

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

Additional Key Words and Phrases: Intel Optane memory, Java, Performance, Scalability, Estimation, Analytics

ACM Reference Format:
Shoaib Akram. 2021. Performance Evaluation of Intel Optane Memory for Managed Workloads. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (January 2021), 25 pages. https://doi.org/10.1145/3451342

1 INTRODUCTION

Today, DRAM satisfies our main memory needs in clouds and data-centers. Unfortunately, DRAM faces scaling limitations [31, 43]. Its pricing, as a result, has remained unchanged since 2016 [29, 33, 36, 67]. Scalable and low-cost memories are needed to support ever-growing users and datasets in clouds and data-centers.

Author's address: Shoaib Akram, Australian National University, Canberra.

XXXX-XXXX/2021/1-ART1
https://doi.org/10.1145/3451342

Emerging memory technologies promise byte-addressability and excellent scalability [30, 69]. These scalable technologies retain information for many years, making them non-volatile. Non-volatile memory (NVM) technologies offer fast persistent storage and complement DRAM in expanding main memory capacity. The most promising among the recently introduced NVM technologies is Intel Optane memory.

Exploiting Optane memory requires substantial efforts. Current efforts focus on native libraries for accessing persistent data [32] and managed language implementations for automatically persisting data [50, 60]. (Managed languages are ones with rich support, especially for automatic memory management and concurrent activity.) Extensive prior art explores software support for NVM in emulation [18, 19, 21, 57, 61–63, 71]. Supporting persistence is foundational to Optane's success. Equally crucial is, however, understanding Optane's performance as a real-life main memory for popular software. Contemporary software practice favors managed languages, such as Java, C#, Python, Ruby, and Rust, because they improve productivity. Unexplored in prior literature, this paper investigates Optane's performance as the main memory for real-world managed workloads.

We gather 17 diverse Java workloads and observe their performance scalability on Optane memory with increasing core count. Following best practice guidelines for Optane use [41, 64], we find that (1) for many workloads, including a relational database, Optane memory substitutes DRAM without significantly degrading performance irrespective of core count, (2) up to eight cores, the average performance is close to a DRAM system for the remaining workloads, (3) for the same workloads, beyond eight and 16 cores, depending on the workload, performance degrades rapidly. We validate these findings on widely used research and a production Java Virtual Machine (JVM).

We analyze poorly scaling (Optane-Sensitive) workloads on Optane memory. These workloads often demand large heaps, allocate memory rapidly, and frequently access objects in main memory. Optane memory, in turn, differs from DRAM on latency, bandwidth, and access granularity. We analyze the interaction of Java workload characteristics with Optane memory at the virtual machine and microarchitecture level. We find that the mutator (application) slows down by 2× on average and garbage collection (GC) by more than 3×. (GC relieves programmers from manually freeing memory, an essential service in managed runtimes.) We tease apart the reasons for GC slowdown and find that over a DRAM-based system, (1) scanning live young objects in Optane memory to locate reachable objects incurs a 150% overhead, (2) copying young survivors to the mature heap incurs a 35% penalty, and (3) remaining tasks such as object finalization incur a 15% slowdown. Mixing scan and copy operations in the same memory type also adds significantly (38%) to the overhead. These overheads shift the GC response time distribution which we analyze and mitigate.

Unlike GC, which performs liveness analysis and compaction, the mutator's functionality is workload-dependent, and so is its slowdown on Optane memory. We use hardware performance counters and find that mutator slowdown manifests as a large increase in pipeline stalls due to long-latency loads, a fully occupied store buffer, and instruction misses. We mitigate this increase in pipeline stalls. We also analyze and optimize for non-uniform memory access (NUMA) architectures.

Our analysis motivates hybrid DRAM-Optane memory, encouraging a heap organization with young objects in a DRAM nursery, old frequently accessed (hot) objects in a mature DRAM space, and the remaining objects in a mature Optane space. High instruction miss latencies encourage DRAM placement of dynamically optimized code. Prior literature exploits GC to manage hybrid memories, improving NVM endurance [5, 6] and reducing DRAM energy consumption [58]. (They use emulation.) They place the nursery and hot mature objects in DRAM. We evaluate the performance of prior GC approaches on real hybrid DRAM-Optane memory, improving upon prior heuristics to predict hot objects. Hybrid DRAM-Optane memory scales substantially better than Optane alone, and on par with DRAM for most workloads. Furthermore, GC allocation heuristics provide Pareto trade-offs between performance and DRAM capacity.

Our evaluated GC approaches differ in the fraction of heap they place in DRAM. Beyond a specific DRAM capacity, we observe performance either improves or stays unchanged for some workloads. Our graph workloads, written on top of the same GraphChi framework [39], offer an example. A particular GC, Crystal Gazer (CGZ), exploits allocation site profiling to identify *hot* objects, allocating those in DRAM. It improves the performance of two workloads, Page Rank and Connected Components, by 60%, and places 50% of the heap in DRAM. For the third workload, CGZ delivers only 14% performance gain for ALS Matrix Factorization, but places 50% of the heap in DRAM. A different GC, namely Kingsguard-nursery, performs similarly to CGZ for ALS, but only allocates 6.25% of the heap in DRAM. This unpredictable performance gain is because existing GCs profile objects and their allocation sites in a run-time-independent manner. They ignore co-running applications, shared resource contention, and memory locality and type.

Conserving limited DRAM capacity in hybrid memories requires adapting to the workload (e.g., core count) and run-time behavior. This paper introduces performance-impact guided memory allocation (PIMA) that exploits online sampling to estimate the performance impact of allocating mature objects in DRAM versus Optane memory. (Mature objects are ones that GC occasionally recycles.) In a sampling phase, the GC first promotes nursery survivors to DRAM and then to Optane memory. The JVM compares the performance on either memory type, as reported by the application, picking the best-performing one for long-term allocation. Sampling-based memory allocation, especially in a managed context, brings new challenges that we tackle, ensuring clean memory semantics and safety of sharing non-volatile Optane media.

We implement PIMA in the GraphChi framework [39] and the Java runtime, targeting modern analytics workloads. To estimate the performance impact of memory types, GraphChi informs the JVM, the total vertices it processes during sampling on DRAM and Optane memory. PIMA operates stand-alone (all mature objects in DRAM or Optane memory) and combines with GC heuristics to identify hot (DRAM) objects. Because existing heuristics require offline profiling, we propose programmer-demarcated *Criticality Hints* to obviate profiling. We evaluate PIMA in various ways, including with prior approaches and criticality hints. PIMA exploits Optane memory for low-core-count graph workloads, shifting to DRAM allocation at high core count, for workloads that benefit from DRAM, with 94.33% accuracy. Its sampling and bookkeeping overhead is only 2%. Overall, PIMA improves the average performance by 6% when optimized for best performance and saves DRAM capacity by 25% when optimized for preserving DRAM capacity than state of the art. In summary, the contributions of this paper are:

- performance characterization of Java workloads on Intel Optane memory, including NUMA effects, scalability, and application, and virtual machine interactions with Optane memory (Section 4);
- evaluation of hybrid DRAM-Optane memory, including microarchitectural analysis, their management by GC, profiling-based heuristics to identify hot objects, and identification of hot objects through programmer-demarcated criticality hints (Section 5); and
- run-time-adaptive and sampling-based memory allocation for hybrid memories, including a novel performance estimation methodology for graph analytics workloads, implementation in a Java runtime environment, and a detailed evaluation (Section 6).

2 BACKGROUND AND MOTIVATION

We first provide background on Intel Optane media, DIMMs, and operation modes, then on garbage collection in managed languages. Also, we motivate maximizing Optane's utilization in data-centers.

2.1 Intel Optane Memory

Non-Volatile DIMM (NVDIMM). An NVDIMM connects to the memory bus, similar to conventional DRAM. Prior NVDIMMs combine DRAM and a battery backup, e.g., AGIGA Tech's

AGIGARAM4, or connect to the memory bus as Flash-based DIMMs, e.g., IBM's eXFlash. Their capacity and cost scaling and endurance are limited. Flash-based DIMMs are more scalable than DRAM but expose a slow block-based interface that makes them impractical for use as main memory. Their prior use is in storage systems for write caching, and journaling.

Optane DIMM. Intel Optane memory is the most scalable and cost-effective NVDIMM to date. Optane exploits a new storage medium called 3D XPoint, which stores information as a change in the material's bulk resistance [31]. 3D XPoint is more scalable than DRAM, and the current DIMM capacity is up to 512 GB (8× that of DRAM.) Communication with Optane DIMM is mediated by the processor's integrated memory controller (iMC). iMC uses a new DDR-T (64-byte) interface. DDR-T enables asynchronous command and data timing. Once a request reaches the Optane DIMM, a module controller reads and writes to the actual media at a 256-byte line granularity. Optane's limited access granularity is due to its media limitations [1, 45, 64]. The module controller intercepts data and address signals (unlike DRAM) to allow for wear-leveling, error correction, and encryption. A 16 KB write combining buffer merges adjacent lines to mitigate high media latency [64].

Operation modes. Optane memory has two operation modes. The *memory* mode turns DRAM into a direct-mapped cache for Optane, and the host memory controller transparently manages the DRAM cache. The DRAM is invisible to software, which sees only the limited physical (Optane) memory space. Unlike the memory mode, the *App Direct* mode exposes the hybrid of DRAM and Optane memory to the software. It also offers persistent storage. The Optane media is abstracted by a light-weight filesystem [59, 63]. This work concerns the performance scalability of App Direct mode for storing managed heaps.

Latency, bandwidth, and endurance. Optane in App Direct mode is slower than DRAM because: (1) 3D XPoint media has higher latency. (2) All requests go through the module controller. Recent work characterizes Optane's latency and bandwidth using carefully written microbenchmarks [64]. They find that Optane is between 2×-3× slower than DRAM. More slow are random accesses compared to sequential accesses. The majority of latency is due to slower media. Next, Optane's bandwidth is lower than DRAM. The read-to-write bandwidth ratio is higher for Optane than DRAM. Optane's bandwidth increases proportionally to DIMM count through interleaving. Finally, Optane's media endurance is finite, but unlike Flash, enough to survive continuous operation as the main memory for a few years [9]. To spread writes out across the Optane media, the module controller maintains an address indirection table.

2.2 Garbage Collection in Managed Languages

Managed languages offer garbage collection (GC) to relieve programmers from manually reclaiming virtual memory. To automatically reclaim memory, GC performs at least two steps. (1) It identifies live roots, e.g., global variables, the stack, and registers. (2) It *scans* live roots for references and traces the references to discover reachable objects. Unreachable heap memory is reclaimed. High-performance collectors *copy* reachable objects to new locations to compact the heap. Copying collectors eliminate fragmentation and improve memory locality.

Production collectors exploit the generational hypothesis that many objects die young. To improve collection yield, they divide the heap into a small contiguous nursery and a large mature space [51]. The nursery is where the application (mutator) allocates new objects. When the application exhausts the nursery space, a *minor* collection promotes (copies) live nursery objects to the mature space. The collector leaves a forwarding pointer in place of the copied object, updating references to previously forwarded objects. After copying the live nursery to the mature space, it reclaims the nursery for fresh allocation. The GC remembers mature-to-nursery object pointers using a write barrier. Remembered pointers are treated as live roots. One way to remember pointers is to store them in a sequential store buffer [65]. Major collections collect the entire heap.

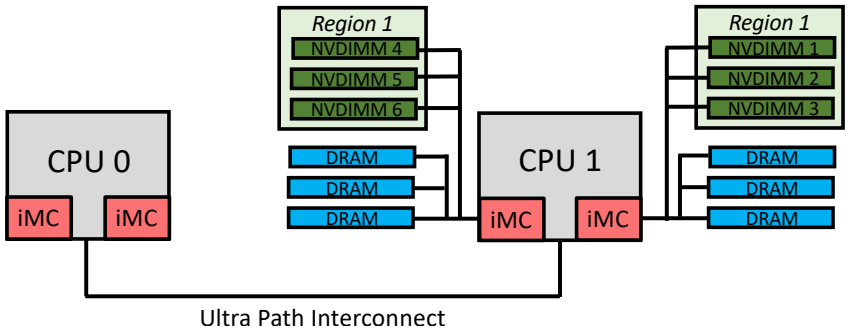


Fig. 1. Evaluation platform with Intel Optane memory. Each CPU attaches to six DRAM DIMMs and six Optane NVDIMMs.

2.3 Motivation: Maximizing Optane Utilization

The main memory, DRAM, contributes to data-center energy and costs significantly [8]. Energy consumed by DRAM translates into high cooling bills. Its idle energy and poor scaling exacerbate the problem. A new memory scaling roadmap is vital, and fortunately, Optane exhibits excellent capacity scaling. It also consumes no idle energy. By maximizing Optane's utilization while satisfying users' performance, we free up DRAM for latency and bandwidth-critical workloads. Limiting reliance on DRAM in this fashion reduces memory-related expenditures. Low expenditures translate into more users and low-cost cloud offerings.

3 EXPERIMENTAL METHODOLOGY

We discuss our evaluation platform, workloads, metrics, and how we follow prior advice to use Optane memory. We discuss our choice of JVMs, including our added support for Optane memory.

Experimental platform. Our server is a dual-socket Dell PowerEdge R740 (see Figure 1). Each processor is an Intel Xeon Gold 6242 operating at 2.8 GHz with 32 logical cores and a 24 MB shared Level-3 cache. Each host iMC supports six memory channels. Each memory channel is attached to a 32 GB Micron DDR4 DIMM and a 128 GB Intel Optane DIMM. The system thus has 400 GB of DRAM and 1.5 TB of Optane memory. The server runs the Ubuntu 18.04.2 Linux OS that uses the 5.1.0 kernel. We evaluate three memory systems: (1) DRAM-Only, (2) Optane-Only, and (3) hybrid DRAM-Optane memory. Unless stated otherwise, we use CPU 1 for execution and allocate heaps in memory attached to it. We always use Region 1 of Optane memory.

Best practice guidelines for Optane use. Two recent works offer best practice guidelines to use Optane memory [64]. They both emphasize avoiding non-uniform Optane accesses, improving locality, and interleaving Optane memory. We investigate and optimize for memory non-uniformity. We use heap settings that maximize memory locality. We use interleaved Optane memory (6 DIMMs). Yang et al. [64] further recommend limiting thread count accessing an Optane DIMM. We use limited mutator and GC threads for each application instance. Mason et al. [41] advise optimizing for page size. Huge pages improve performance regardless of the memory type but expose a new trade-off between page fragmentation and total performance. We use OS defaults and leave optimizing for page size to future work.

Java Virtual Machine (JVM). We use the open-source Jikes Research VM (RVM) v3.1.2 and OpenJDK v14.0.1 to evaluate Optane memory. Jikes RVM is a Java-in-Java JVM with an optimizing compiler of hot code and a memory management tool kit (MMTk) [10]. The vanilla JVM creates an anonymous mapping between heap chunks and DRAM. We first modify MMTk to inform the C and OS allocators to map chunks to either DRAM or Optane memory. Then, to support Optane memory,

	DRAM-Only Variation %	Optane-Only Variation %	Heap Size MB	DRAM-Only allocation rate GB/s	Optane-Only allocation rate GB/s	nursery survival rate %	DRAM KG-N %	DRAM KG-W %	DRAM CGZ-F1 %	DRAM CGZ-D1 %	DRAM CH %	DRAM PIMA %	DRAM CH+PIMA %
Eclipse	0.11	0.15	1280	0.37	0.01	14	3	25	51	22			
Avrora	0.29	0.44	512	0.11	0.03	15	4	14	5	5			
Fop	0.35	2.36	640	0.40	0.09	20	5	18	8	6			
Luindex	0.46	1.20	352	0.08	0.01	22	9	34	15	14			
Antlr	3.85	6.15	384	0.33	0.01	15	8	28	24	19			
Hsqldb	3.91	0.53	2032	1.13	0.84	60	2	13	10	2			
ALS	0.11	0.11	8192	2.34	0.04	63	6	20	53	35	42	6.25	6.25
Sunflow	0.60	0.53	864	3.96	0.81	2	4	14	21	7			
Pmd	0.73	0.34	784	2.84	1.52	23	4	26	25	9			
Pjbb	4.23	2.86	3200	2.89	0.24	20	1	19	10	4			
CC	2.32	0.74	8192	3.71	0.14	24	6	41	54	24	42	100	42
Pmd.Scale	0.31	0.31	784	3.55	3.38	27	4	18	20	9			
Lu.Fix	1.25	0.48	544	5.57	1.91	2	6	20	7	7			
Bloat	1.31	0.02	2112	7.23	0.48	4	6	22	9	8			
Xalan	0.51	0.51	864	4.50	0.86	14	4	23	17	8			
PR	1.59	0.20	8192	5.42	0.15	36	6	42	59	18	44	100	44
Lusearch	0.43	1.51	544	13.97	0.77	4	6	20	65	66			
Avg G-Chi	1.34	0.35	8192	3.82	0.11	41	6	34	55	26	43	69	30
Avg Op-Sens	1.22	0.69	2972	5.10	0.94	20	5	24	31	18			
Avg All	1.39	1.14	2488	3.40	0.66	21	5	23	27	15			

Table 1. The 32-core experimental variation, heap sizes, allocation and survival rates, and DRAM statistics.

we create shared mappings between chunks and Optane-resident files on a light-weight filesystem. The *shared* mode avoids copy-on-write semantics of *private* mode, allowing direct access of Optane memory. OpenJDK has limited built-in support for Intel Optane memory. Unless otherwise stated, we show results with Jikes RVM. (See Section 7 and **Appendix A** for more details.)

Direct Access (DAX) Filesystem. To expose Optane memory to the JVM, we use the DAX-aware EXT4 filesystem [59]. A DAX-aware filesystem bypasses the page cache and provides direct memory access. It also disables data journaling and eliminates other metadata writes, a necessity for storage devices. We pre-populate Optane memory with 1 MB files in Region 1 (Figure 1), which the JVM maps as chunks on-demand in its virtual address space during execution. We zero Optane-resident files before each experiment to comply with Java safety semantics.

Java benchmarks and workloads. We evaluate Optane's performance scalability using multi-programmed Java workloads. Multiple virtual machines on a multicore server are commonplace in data-centers. We use 15 Java benchmarks from three sources: 11 from DaCapo [12], pseudojbb2005 (Pjbb) [13], and three graph applications from GraphChi [39]. Graph applications include page rank (PR), connected components (CC) and ALS matrix factorization (ALS). We use lu.Fix that optimizes memory allocation [66]. We also use a new version of pmd namely pmd.scale that uses a balanced dataset to improve benchmark scalability [20]. We run the multithreaded DaCapo benchmarks, Pjbb, and graph benchmarks with four threads. In our evaluation, a four or eight-core experiment means one or two instances of a multithreaded benchmark. Alternatively, it means four or eight instances of a single-threaded benchmark. We show results with up to 32 cores. We use two stop-the-world collector threads. We use the default datasets for DaCapo benchmarks. We use the LiveJournal online social network for PR and CC, and the training set of the Netflix Challenge for ALS. We limit processing to ten million edges for graph workloads. For Pjbb, we use 4 warehouses and 50 K transactions. Table 1 lists the benchmarks, heap settings, and experimental statistics.

Performance measurement methodology. We use the maximum normalized turnaround time (MNTT) as a measure of performance [23]. For a single-programmed workload, MNTT translates into execution time. Because multiprogrammed workloads introduce run-to-run experimental variation, we perform each experiment eight times and report the average. We also use replay compilation for deterministic execution of Java benchmarks [28]. Replay compilation happens in

two iterations. The first iteration compiles each method to a pre-determined optimization level. The second iteration does no recompilation. We take measurements during the second iteration. Each instance of a multiprogrammed workload start the second iteration at the same time. We clear the page cache, the directory cache, and inodes before each experimental run. Also, we reinitialize the benchmark's datasets before each experiment.

Garbage collector and heap setting. Jikes RVM has several state-of-the-art garbage collectors for DRAM [10, 11, 14], and hybrid DRAM-NVM memories [2, 5, 6]. Our baseline collector in all experiments is the production collector in Jikes, generational Immix (GenImmix) [14, 49]. GenImmix uses a contiguous nursery and copies live objects to a *mark-region* mature space. Regions form a hierarchy of coarse-grained blocks and fine-grained lines. Blocks are multiples of the page size. Our block size is 32 KB. We use a line size of 256 bytes to match Optane's media line size (see Section 2). A nursery collection first copies objects into free lines in partially-filled blocks, then in completely unoccupied blocks. A full-heap collection marks regions and reclaims completely free blocks and completely free lines. Like other high-performance GCs, GenImmix allocates large objects in a specialized non-moving mature heap to avoid large copying overheads. In this work, we first allocate large objects ($\geq$ 8 KB) in the nursery and copy the surviving large objects in a separate non-moving Large Object Space (LOS) [5, 6]. We use nursery sizes similar to prior work: 4 MB for DaCapo and Pjbb; and 32 MB for GraphChi applications. Our heap sizes are twice the minimum heap size for DaCapo and Pjbb [4], and four times the minimum heap size for GraphChi. In Table 1, we show heap sizes for 32 core workloads.

4 PERFORMANCE ANALYSIS

We first explore the optimal NUMA setting for analyzing Optane memory. We then discuss its performance scalability for Java workloads, diving into workloads' interaction with Optane memory.

4.1 Optimal NUMA Setting

Sound experimental design for analyzing a new memory technology requires optimizing non-uniform memory access (NUMA) effects. In NUMA servers, remote memory accesses incur high latencies. To avoid them, Linux offers automatic NUMA balancing [54]. (We call it Auto.) Linux also allows binding threads and memory to specific sockets manually. Automatic balancing not only precludes manual intervention but also spreads out compute load and memory across sockets. Spreading out compute allows workloads to exploit core and cache resources fully.

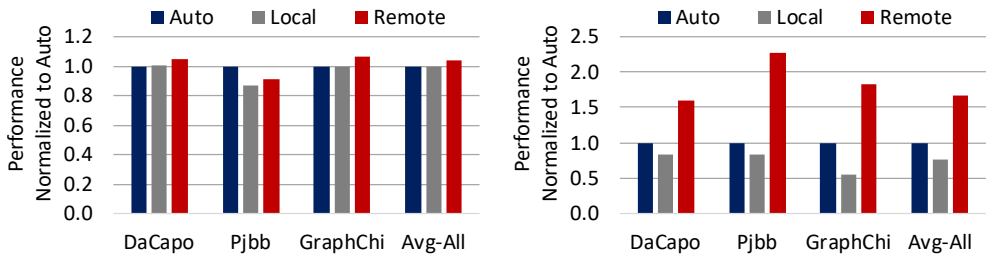


Fig. 2. Performance impact of non-uniform memory access (NUMA) settings. Auto is Linux's default NUMA balancer. Local allocates threads and memory in the same socket. *Local is best for both (a) DRAM-Only and (b) Optane-Only. (The vertical axes use different ranges.) Optane-Only is more sensitive to remote accesses.*

To analyze the performance impact of remote accesses, we use Linux's numactl to create Local and Remote configurations. Local binds threads and DRAM to CPU 1. Remote uses CPU 0 for

execution. Figure 2 (a) and (b) show the 32-core performance of Local and Remote for DRAM-Only and Optane-Only, normalized to Auto. We show harmonic means across three Java workload suites.

Remote accesses slow down DRAM-Only by 5% on average. For Pjbb, Remote-DRAM-Only performs 9% better than Auto. In this specific instance, binding threads on a socket and the resulting cache locality is more beneficial than spreading the load across sockets. Remote accesses slow down Optane-Only much more significantly (up to 128%). The Pjbb and GraphChi workloads use large heaps and incur a more significant slowdown than DaCapo. For Pjbb, the performance of Remote-Optane-Only degrades by 66% compared to Auto. Auto is unaware of Optane's role as the main memory but still performs better than Remote-Optane-Only. Auto's better performance is because it distributes memory allocation across sockets. As a result, many accesses resolve locally.

Compared to Auto, Local-Optane-Only improves average performance by 23% and up to 44% for GraphChi. For two workloads, however, Local-Optane-Only is worst than Auto. They are Lusearch and Sunflow, and both benefit from increased cache capacity. Overall, Local-Optane-Only is the best performing on average, but Auto enables better load balancing for some workloads. Unless stated otherwise, we use Local-Optane-Only for analysis in upcoming sections. We believe that future work should make Auto more aware of Optane's role as volatile main memory.

4.2 Scalability

For emerging memories to complement DRAM, they must scale in performance with an increasing workload. In Figure 3, we show the performance of Optane memory for several Java workloads with increasing core count. We normalize to DRAM-Only with similar core count. It is encouraging that for many workloads, with four and eight cores, Optane-Only performs on par with DRAM-Only. Compared to DRAM-Only, the four and eight-core average performance degradation is 5% and 7%. With eight cores, only the graph workload, Page Rank, incurs a notable slowdown of 1.49×. The left-most workloads are less sensitive to Optane properties and deliver scalable performance up to 32 cores. These Optane-insensitive workloads include a variety of popular software. For instance, Hsqldb (16% slower with 32 cores) is a relational database system [27], supporting in-memory and disk-based transactional support. Eclipse is a well-known integrated development environment [24]. Other workloads, including Fop, Antlr, and Luindex, process and transform textual data. For these workloads, Optane memory can replace DRAM as the main memory without significantly degrading performance. Unfortunately, beyond a specific core count, the right-most workloads in Figure 3, labeled as Optane-Sensitive, perform considerably worst than DRAM-Only. Most of these workloads begin to incur a notable slowdown beyond eight cores. The average 16-core slowdown for Optane-Sensitive workloads is 1.32×, and up to 2.25× for the worst affected workload. The performance further degrades from 16 to 32 cores, sometimes rapidly, and up to 7.5×.

In Table 1, we show the 32-core heap sizes and allocation rates for all workloads. There is a strong correlation between a workload's allocation rate on DRAM-Only and its scaling behavior on Optane memory. Poorly scaling workloads exhibit high allocation rates. Allocation rates on DRAM-Only reach up to 14 GB/s, and the average across Optane-Sensitive workloads is 5.1 GB/s. Unlike some Optane-Sensitive workloads, graph workloads with the largest heap sizes in our workload suite allocate less rapidly. They allocate numerous small objects representing the graph vertices and then perform heavy-weight computation on the allocated vertices. Unlike DRAM-Only, the average allocation rate on Optane-Only is less than 1 GB/s. Limiting the allocation rate is Optane's limited read and write bandwidth. The average heap size of poorly scaling workloads is larger than the Optane-insensitive workloads. Despite this, heap size is not strongly correlated with scalability behavior on Optane memory. Both Eclipse and Hsqldb, for instance, exhibit larger heaps than several Optane-Sensitive workloads.

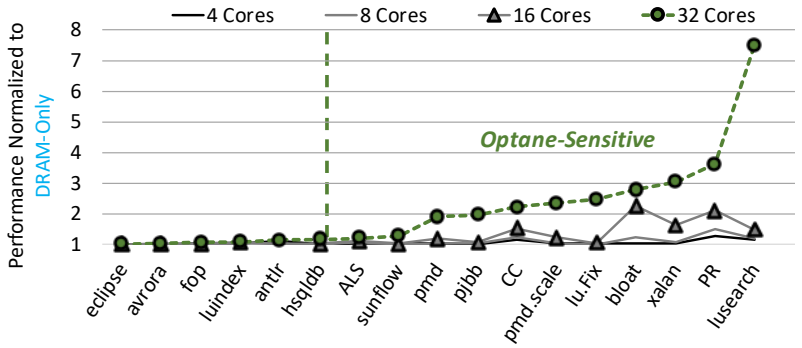


Fig. 3. Performance scalability of Optane-Only for Java workloads. The lower, the better. *Optane-sensitive* workloads are ones with more than 20% slower execution (32 cores) on Optane-Only. *For left-most and low-core-count workloads, Optane-Only performs close to DRAM-Only. It scales poorly for the remaining workloads.*

Highly allocating workloads stress the memory system. Each newly allocated object generates read and write memory traffic. The object is first brought in the processor's cache for initialization, which is on the memory allocator's critical path. Later, dirty objects, their corresponding cache lines are written back to memory. In Optane memory, each writeback turns into a read-modify-write operation. Many write operations, mostly random, fill the write combining buffer, slowing down subsequent writes. They also hurt read locality and interfere with read operations, increasing their latency. These effects are amplified due to the Optane's high latency and limited access granularity.

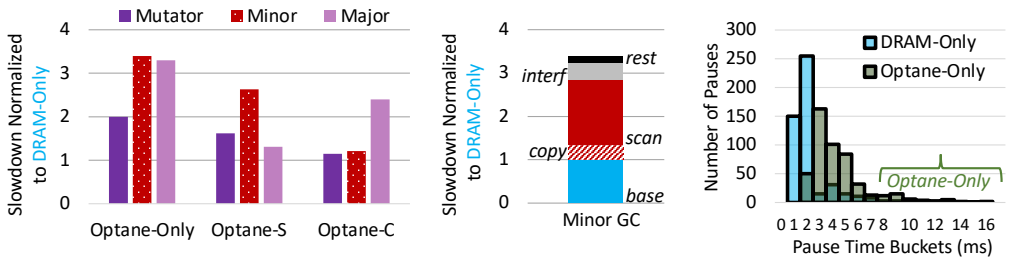
In Table 1, we also show the nursery survival rates for our workloads. We do not see a correlation between survival rates and the scaling behavior on Optane memory. Despite this, the survival rate contributes to the memory traffic on Optane memory. The GC copies survivors to the mature heap, which generates traffic similar to the new allocation. The higher the survival rate, the more the nursery-to-mature copies. These copies also generate traffic indirectly when GC adjusts references to copied objects elsewhere in a heap. In addition to the GC traffic, the object's mutator accesses also contribute to memory traffic.

We find it critical to evaluate new memories for diverse workloads and processor usages. Evaluations that under-utilize processors do not uncover true overheads. A focus solely on the opposite paints an overly pessimistic picture of potential DRAM replacements. Diverse evaluations gauge the specific role of emerging technologies, such as Optane memory, more clearly.

We discuss next the statistical significance of our evaluation, and after that, take a deeper dive into the interaction of mutator and GC with Optane memory.

4.3 Statistical Significance

Multiprogrammed Java workloads suffer from run-to-run variation in performance. Our evaluation is statistically significant, exhibiting low relative variability in performance for DRAM-Only and Optane-Only. On average, across all workloads, the coefficient of variation is less than 2%. To keep variation low requires following best practice guidelines for evaluating Java workloads [28, 34] and configuring Optane memory [64]. Table 1 shows the 32-core variability for each workload. We also measure fairness among the multiple instances in our workloads. We use the relative variability in execution times between workload instances to quantify fairness [52]. Fairness is close to one for both memory types. We observe a slightly lower fairness of 0.96 for transaction processing workloads, Hsqldb and Pjbb, with Optane-Only. Transaction processing exposes the higher and unpredictable latency of Optane memory for random accesses. We use MNTT to show scalability in Figure 3. Due to high fairness, average NTT (ANTT) reveals similar trends.



(a) Mutator and GC Slowdown (b) GC Slowdown Stack (c) Pause Time Distribution

Fig. 4. Mutator and GC characterization on Optane memory. *(a) GC slows down more than the mutator, (b) slow scanning overshadows slow copying, and (c) Optane pause time distribution has a long tail.*

4.4 Mutator and GC Characterization

The mutator and the GC both contribute to a managed workload's performance. We first tease apart mutator and GC slowdown on Optane memory at high core count and then isolate primary reasons for GC slowdown. The resulting analysis reveals Optane's reaction to distinct memory access patterns. It also motivates heap organizations for improved performance scaling.

For highly allocating workloads on Optane-Only, both the mutator and the GC incur a slowdown. However, the GC incurs a higher slowdown because it stresses the memory system more than the mutator. In Figure 4 (a), we see the 32-core average mutator slowdown of 2×, normalized to DRAM-Only, compared to the minor and major GC slowdown of more than 3×.

We analyze the reasons for GC slowdown by focusing on minor GC, which consumes an average of 20% of the execution time. It *scans* live nursery to find reachable objects and *copies* nursery survivors into the mature heap. To tease apart the GC slowdown due to slow scanning and copying, we create two configurations, Optane-S, and Optane-C. In Optane-S, the GC allocates and scans nursery objects in Optane memory, copying the nursery survivors into DRAM. In Optane-C, the GC allocates (scans) objects in DRAM, copying survivors into Optane memory. In Figure 4 (a), we observe that Optane-S speeds up minor GC by 1.3× over Optane-Only, implying that the speedup due to copying into DRAM instead of Optane memory is moderate. Scanning of the nursery in DRAM (Optane-C) speeds up minor GC by a large 1.8× over Optane-Only. Thus, slow scanning is the primary cause of GC slowdown on Optane-Only.

We decompose the minor GC time into four components to propose the *GC slowdown stack* in Figure 4 (b). To compute the stack, we first split the GC time into two parts: (1) scanning and copying young objects (δ), (2) the remaining time (ψ). Each component in the stack reveals an overhead on top of *base*, the GC time on a DRAM-Only system. Slow copying on Optane memory adds 35% on top of the base time, as shown by *copy* ($\delta_{Optane-S} - \delta_{DRAM-Only}$). Slow scanning is represented by *scan* and adds a substantial 150% overhead ($\delta_{Optane-C} - \delta_{DRAM-Only}$). The scan component is followed by *interf*, the interference between scanning and copying on Optane memory. Copying leads to writebacks that interfere with read requests due to scanning. We estimate *interf* as the increase in δ from Optane-Only to DRAM-Only, minus the sum of (isolated) copy and scan increases due to Optane memory. Interference adds 38% to the slowdown stack ($\delta_{Optane-Only} - \delta_{DRAM-Only} - (copy + scan)$). Finally, *rest* adds 15% to the stack ($\psi_{Optane-Only} - \psi_{DRAM-Only}$), and represents the slowdown in identifying global roots, calling finalizers, and manipulating meta-data.

The cost of copying objects into Optane memory is low because: (1) more than 90% of objects are up to 64 bytes, and (2) objects are copied to a mostly-contiguous mature heap. On the Optane media, each copy opens a 256-byte line, and lines reside in a 16 KB write combining buffer. The heap locality and hardware write combining keeps the copying overhead low. On the other hand,

scanning on Optane memory is extremely slow because GC scans the heap in a breadth-first manner. It pops objects from a shared work queue, reads their reference fields, copies the referents to new locations, and populates the queue with newly copied objects. The resulting random read operations incur a significant slowdown on Optane memory, which considerably slows down scanning.

It is imperative to mitigate the high GC times on Optane memory because they shift the pause time distribution significantly, inducing response time violations. In Figure 4 (c), we show the pause time distributions for the user-facing Pjbb workload on DRAM-Only and Optane-Only. With DRAM-Only, the peak in pause times appears at two milliseconds (ms). For Optane-Only, the peak is at 3 ms. The Optane distribution has a long tail, with maximum pause times doubling. Eliminating this tail requires either relocating the nursery in DRAM or retuning the nursery for Optane-Only.

The performance of three Optane variants in Figure 4 (a) gives an insight into the slowing down of the mutator and major GC on Optane memory. In particular, Optane-S copies the nursery survivors into DRAM and thus places the mature heap in DRAM. The fast access to long-lived mature objects in DRAM speeds up the mutator in Optane-S by 23% over Optane-Only. On the other hand, the mutator in Optane-C observes a much larger 74% gain in performance. This gain is obtained by allocating new objects in a DRAM nursery. We also observe that placing the mature heap in DRAM (Optane-S) significantly speeds up major GC cycles. On a surprising note, major GC cycles in Optane-C happen more quickly than in Optane-Only, although the mature objects are in Optane memory in both cases. Major GC also scans the nursery. The DRAM placement of the nursery in Optane-C is the reason for faster major collections.

This section's analysis motivates GC-managed hybrid memories. Specifically, allocating young objects in a small DRAM nursery avoids frequent scanning in Optane memory and speeds up the mutator. As part of the young generation, collectors such as OpenJDK's Parallel Scavenge [25], place survivor spaces in front of the nursery to avoid tenuring garbage [25]. It is interesting to explore the placement of survivor spaces in hybrid memory. Their size and placement are likely to be a trade-off between scanning overheads and DRAM capacity. In this work, we focus on the placement of nursery and mature objects instead and discuss the resulting hybrid memory approaches next.

5 DRAM-OPTANE HYBRID

So far, we have evaluated Optane's performance and scalability as the only main memory, observing it scales poorly for several Java workloads. We now evaluate hybrid DRAM-Optane memory, including scalability, heap partitioning, and microarchitectural analysis. We discuss profiling-based partitioning heuristics and propose *criticality hints* to exploit the programmer's knowledge in partitioning heaps over hybrid memory.

5.1 Scalability

The performance of highly allocating workloads scales poorly on Optane memory because Optane's bandwidth acts as the bottleneck against their memory access patterns. These patterns can be transformed by GC allocating a fraction of the heap in DRAM. Prior work proposes different GC approaches for managing hybrid memories [5, 6]. The state-of-the-art approach is Crystal Gazer. In Figure 5 (a), we exploit it and show the scaling behavior of the resulting hybrid DRAM-Optane memory for Optane-Sensitive workloads. We show results with two NUMA settings. At low core count, the average performance of DRAM-Only, Optane-Only, and hybrid memory is close to each other. At high core count, especially with 32 cores, hybrid memory narrows the substantial 2.2× difference in performance between DRAM-Only and Optane-Only (Local) to 1.12×. We also observe that, compared to Local, Auto-Optane-Only scales more poorly. Again, hybrid memory bridges the performance gap from 2.8× to 1.17×.

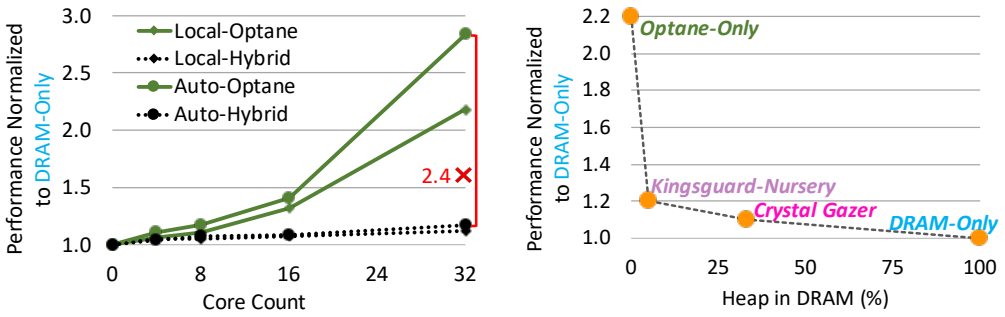


Fig. 5. Performance scalability and DRAM requirements of GC-managed hybrid memory. *(a) Hybrid memory scales substantially better than Optane alone, and (b) Different GC approaches expose a Pareto frontier between performance and DRAM capacity.*

The DRAM-Optane hybrid performs better than Optane alone. However, to exploit Optane's capacity, the GC must limit the percentage of the heap it allocates in DRAM. In Figure 5 (b), for 32-core Optane-Sensitive workloads, we compute the Pareto frontier of performance (Y-axis) and DRAM capacity (X-axis). DRAM capacity is quantified by the percentage of the heap in DRAM. On the Pareto frontier are two GC approaches from prior literature. The Kingsguard-nursery (KG-N) allocates new objects in a DRAM nursery and promotes nursery survivors to an Optane mature heap [5]. The nursery is sized to bound GC pause times, and it is a small, 5% on average, of the entire heap for our workloads. Placing 5% of the heap in DRAM improves the performance over Optane-Only by 1.8×. Next on the Pareto frontier is Crystal Gazer (CGZ) that also places the nursery in DRAM. It then classifies nursery survivors into hot and cold categories. The predicted-hot objects are copied into a DRAM mature heap and the rest into an Optane heap [6]. (The scaling behavior in Figure 5 (a) uses CGZ.) CGZ delivers an average (harmonic mean) 7% performance improvement over KG-N, placing 31% of the heap in DRAM. In particular, CGZ significantly improves the scaling of PR, improving its 32-core performance by 30%. Also, Pjbb benefits substantially, observing a 15% performance improvement.

There are multiple reasons for the better scaling of Optane-Sensitive workloads with KG-N and CGZ. They allocate new objects in DRAM, diverting many writebacks away from Optane memory. In addition, CGZ places hot objects in DRAM, mitigating Optane's high read latency. To predict hot objects, CGZ exploits offline profiling of allocation sites, using heuristics that expose performance versus DRAM capacity trade-offs. We will comprehensively analyze GC heuristics to identify hot allocation sites. However, before that, we present a microarchitectural analysis of DRAM-Only, Optane-Only, and hybrid memory. This analysis provides insight into Optane memory-related stalls and their mitigation by GC-managed hybrid memory.

5.2 Microarchitectural Analysis

We now analyze memory scaling behavior from the processor microarchitecture perspective. To access the processor's performance counters, we use Linux's perf utility. We configure perf to obtain the aggregated cycles across all cores during a workload's execution. We further break down cycles into four components: (1) all cycles during which the core is stalled and waiting for load requests to resolve (Load), (2) the core is stalled due to a fully occupied store buffer (Store), (3) the core is stalled due to an instruction cache miss (I-Cache), and (4) the remaining cycles (Rest). The remaining cycles constitute mostly the useful work the processor performs. We analyze in detail

two workloads, most sensitive to Optane's performance, one from DaCapo, xalan, and one from GraphChi, PR, at low (four) and high (32) core count. The remaining Optane-Sensitive workloads reveal similar trends, although the specific changes in cycle count vary. The Optane-insensitive workloads manifest a small change in cycle count with and without Optane memory.

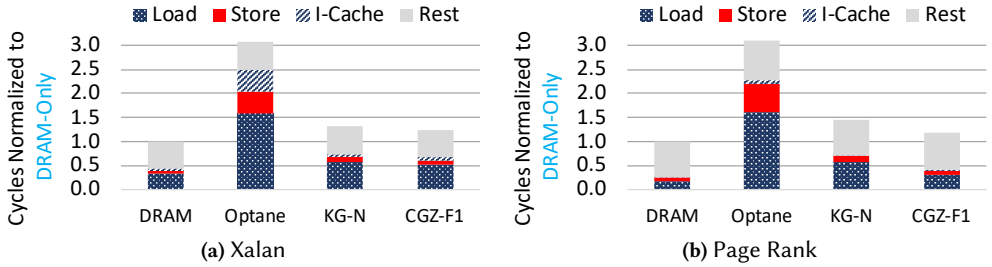


Fig. 6. Microarchitectural characterization of DRAM-Only, Optane-Only, and hybrid memory for (a) Xalan and (b) PR. *Compared to DRAM-Only, long-latency memory loads, limited store buffer capacity, and high instruction miss penalties induce almost an order of magnitude more stall cycles in Optane-Only.*

For the four-core xalan and PR workloads on top of DRAM-Only, the Load, Store, and I-Cache stalls constitute 30% and 23% of the cycle count. These stall components indicate significant main memory activity. For the same two workloads on Optane-Only, we observe an increase of 5% and 12% in Load stalls. Compared to DRAM-Only, this increase does not impact MNTT significantly. Also, the increase in stalls is mitigated by CGZ-F1.

Next, with 32 cores, the memory-related stalls increase by an average 9% for DRAM-Only. Compared to DRAM-Only, however, for Optane-Only, we see in Figure 6, a huge increase in the Load and Store stalls for both xalan (a) and PR (b). The Load stalls, in particular, increase by 4.9× and 8.9×. This increase is due to high Optane media latency for frequently accessed objects, especially those newly allocated. KG-N allocates new objects in DRAM, mitigating the increase in load latencies due to Optane memory. CGZ further reduces the Load stalls up to 2× by placing hot, frequently accessed objects in DRAM.

A significant reason for Optane's poor scaling is Load stalls from the processor's perspective. The Load stalls reduce once GC places new and hot objects in DRAM. The next big reason is Store stalls due to a fully occupied store buffer. These stalls have been observed in prior literature for managed workloads [3, 4]. Specifically, prior work reports three reasons for Store stalls: (1) new allocation, (2) GC copying, and (3) memory zeroing. Together, they result in store bursts, filling the core's store buffer, preventing subsequent stores from issuing, and halting the pipeline. Store bursts are more likely to degrade performance with Optane memory due to its high latency. We observe Store stalls in Figure 6 on DRAM-Only, but they constitute a small fraction of all stalls. On Optane-Only, Store stalls increase by almost 8× for both xalan and PR. In Figure 6, it is evident that GCs for hybrid memory eliminate a large percentage of the Store stalls.

We have observed that high Load and Store stalls pose a scaling bottleneck on Optane-Only. Next to them is another bottleneck, the I-Cache component. Several Java workloads incur a high instruction miss penalty on Optane-Only because the JVM places the per-method optimized code in Optane memory. This high penalty is especially visible for xalan in Figure 6 (a). Even without Optane memory, instruction miss rates for managed workloads are higher than native ones. The reasons include small method sizes and difficulty in predicting frequent control flow changes [44]. However, I-Cache is a small percentage consuming between 1% to 5% of total cycles on DRAM-Only. On Optane-Only, it grows to 14% for xalan and a similar level for other workloads (not shown). The

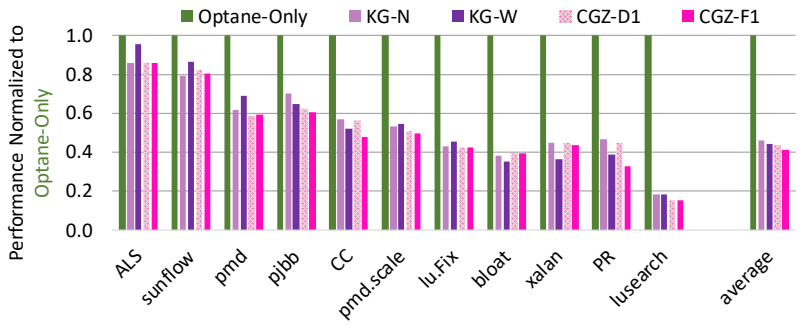


Fig. 7. Performance of Kingsguard (KG-N and KG-W) and Crystal Gazer (CGZ-D1 and CGZ-F1) for 32-core Optane-Sensitive workloads. *Compared to Optane-Only, hybrid DRAM-Optane memory improves the average performance by 59%.*

per-workload increase varies and depends on the average method size and control flow behavior. I-Cache stalls are eliminated by informing the GC to place the optimized code in DRAM.

The Load, Store, and I-Cache stalls are responsible for the poor scaling of Optane-Sensitive workloads from the processor core's perspective. GC-managed hybrid memory mitigates these stalls significantly by allocating new and hot objects and optimized code in DRAM.

5.3 GC Allocation Heuristics

We have observed that for hybrid memories, the Kingsguard and Crystal Gazer collectors trade off DRAM capacity for better performance. They allocate in DRAM, objects they predict as highly written. Their intent is on mitigating wear-out [5, 6]. Fortunately, object writes also predict their access frequency or hotness (sum of reads and writes). Thus we observe reduced load and store-related stall in hybrid memories. We now discuss and evaluate GC heuristics to identify hot objects.

The KG-N collector has a dynamic variant, namely, Kingsguard-writers (KG-W), that copies nursery survivors in a new DRAM *observer* space. Objects in this space are dynamically monitored by KG-W. On an observer collection, KG-W keeps highly written objects in DRAM and copies the rest in Optane memory. Monitoring incurs an overhead. To eliminate it, CGZ statically profiles object-writes on a per allocation-site basis, classifying sites into hot (DRAM) and cold (Optane). It classifies and labels sites as DRAM using two heuristics, Frequency (FREQ) and Density (DENS). FREQ measures per-object write frequency for identifying write-intensive sites. Instead of frequency, DENS uses the ratio of writes to the object size. DENS favors large objects with a few writes for Optane allocation, exploiting its capacity better.

In Figure 7, we compare the performance of KG-N, KG-W, and two CGZ variants against Optane-Only. For CGZ, we use density and frequency thresholds of one, leading to CGZ-D1 and CGZ-F1. KG-N offers the most attractive trade-off, a 54% improvement in performance (reduction in MNTT) on average, placing only 5% of the heap in DRAM. In Table 1, we show the percentage of heap various GCs place in DRAM. KG-N's dynamic variant, KG-W, improves the performance of some workloads, up to 19% (Xalan), and on average, by 8%. However, its performance is worst than KG-N for other workloads. The reason is the high overhead of monitoring object writes. KG-W does accurately identify hot objects, placing 23% (average) of the heap in DRAM. Next, the CGZ-D1 collector only improves performance by 6% on average but allocates a small, 18% heap in DRAM. The best performing collector is CGZ-F1, improving performance by 10% on average, and up to 30% for the graph workloads, PR. It places 31% (average) of the heap in DRAM. For 32-core workloads, the delta in performance between DRAM-Only and hybrid memory is 12% with CGZ-F1, making GC-managed hybrid memory appealing instead of DRAM and Optane memory alone.

We observe that, in Figure 7, CGZ-F1 benefits Pjbb and PR more than other benchmarks. CGZ-F1 profiles each benchmark in an architecture-independent manner [6]. It uses profiling-based prediction at run-time that benefits each benchmark differently. Specifically, the benefit depends on each benchmark's interaction with the architectural details, e.g., cache sizes.

The CGZ-F1 collector improves performance over Optane-Only but sometimes needlessly wastes DRAM capacity. For the graph workloads, PR, CC, and ALS that run on top of GraphChi, the average performance improves by 44% over Optane-Only. To obtain this performance, it places 55% of the heap in DRAM. We observe in Figure 7 that the performance of ALS is unchanged with CGZ-F1. Both KG-N and CGZ-F1 improve performance by 14% over Optane-Only. Therefore, in the case of ALS, the additional 47% of the heap CGZ-F1 places in DRAM on top of KG-N's 6% is wasted. Always executing graph workloads with KG-N results in an efficient operating point for ALS, but unfortunately, hurts the performance of other graph workloads. Similar is the behavior of KG-W for graph workloads. The performance gain relative to DRAM allocation is dictated by workload characteristics and run-time factors, such as memory locality, shared resource contention, and main memory properties. What we need is an approach that allocates objects in DRAM only if it improves performance. We will discuss one such approach in detail in Section 6.

We also explore new performance-centric heuristics. These heuristics profile object-reads in addition to writes on a per allocation-site basis. The best performing heuristic uses a ratio (e.g., 20%) to divide objects into two quadrants, hot and cold. It then uses a hotness cut-off that places objects from a site in the top 20% of all hot objects. If more than a threshold (homogeneity-threshold) of objects from a site are hot, we classify the site as DRAM, marking its objects as hot at nursery allocation time. The GC then promotes those objects into a DRAM mature space. We set the homogeneity-threshold to 1% similar to prior work [6], and use the hotness ratio of 20%. On average, we observe a 2% performance gain over CGZ-F1, and up to 4%. Because object writes predict their hotness, our new heuristics deliver limited performance gain over CGZ-F1.

5.4 Allocation Site Analysis

The best performing collector for hybrid memory, Crystal Gazer, and its variants, exploit offline profiling of allocation sites, making them impractical for unseen workloads. We analyze allocation-site advice of CGZ-F1 and find that: (1) the total number of DRAM-labeled sites vary between 100 to more than 600, (2) unlike class libraries and virtual machine sites, DRAM allocation of sites in the application code account for most of the performance benefit. A few tens of sites in the application code are DRAM-labeled. In the GraphChi framework, for instance, only 40 allocation sites are classified as DRAM by CGZ-F1. Turning DRAM into a measuring tool on our platform, we narrow the performance benefits of CGZ-F1 for PR and CC to a *single* allocation site. This allocation site parses the input graph, creating and allocating new vertices in memory.

Criticality Hints. The allocation site analysis above reveals an opportunity for programmers to identify and annotate hot (DRAM) allocation sites. We propose Criticality Hints (CH) that programmers attach to allocation sites during development. We add a new annotation to Java [26], @perf_critical, which programmers use to attach hints to allocation sites. These hints contain semantic information, informing the runtime of the high criticality (hotness) of the objects allocated from CH-attached sites. The runtime and GC act upon the semantic information, placing objects from CH-attached sites in DRAM.

6 PERFORMANCE IMPACT-GUIDED MEMORY ALLOCATION

In hybrid memories, it is challenging to utilize Optane memory efficiently. Its performance for many workloads is on par with DRAM at low core count. Performance usually degrades with increasing core count, especially for highly allocating workloads. For these workloads, exploiting

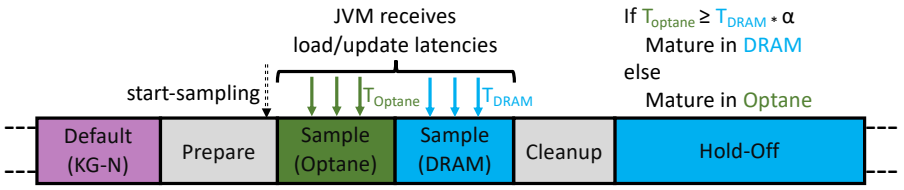


Fig. 8. The execution of a managed workload under PIMA: phases, application-runtime communication, and algorithm for choosing the long-term memory type.

GC to partition the workload's heap across the DRAM-Optane hybrid improves over Optane-Only. The resulting GC approaches trade off DRAM capacity for good performance. Unfortunately, they sometimes waste DRAM capacity due to their run-time-agnostic nature. For instance, they ignore co-executing applications, heap locality, shared resource contention, and memory characteristics. A run-time-adaptive approach for managing hybrid memories is desirable, an approach that adjusts automatically to low versus high processor usage and workload characteristics, including the degree to which those characteristics interact with the memory system to degrade overall performance. We now propose a scheme for managing hybrid memories that aim for DRAM allocation: (1) if there is enough DRAM capacity, and (2) placing objects in DRAM delivers a performance boost.

6.1 High-Level Overview

Our approach is called Performance Impact-guided Memory Allocation (PIMA) because it estimates at run-time the performance impact of placing mature objects in different memory types. It can estimate the impact of placing all mature objects in a memory type or originating from CH-attached sites or sites in profiling-based advice. To estimate the performance impact, PIMA relies on sampling. During the sampling phase, the JVM and GC first allocate the mature objects in one memory type and then in the other. At the end of sampling, the application informs the JVM of the end-user performance on each memory type. Based on this, the JVM chooses either DRAM or Optane memory for stable (long-term) allocation. Because both play a role, we co-design the application framework and the JVM for implementing PIMA.

PIMA requires a high-level application metric to estimate the performance impact of different memory types and semantic signals from the application to identify sampling intervals. It benefits all managed applications, but the implementation effort is especially justified for analytics frameworks that support a large application base. We target PIMA at graph analytics, implementing it in GraphChi and Jikes RVM. GraphChi allows developing various iterative analytics algorithms in which a similar computation is repeatedly performed on graph vertices till a convergence condition is met. Our approach generalizes to other analytics frameworks, for instance, GraphLab [40] and Apache Spark [70]. Next, we describe phases of execution in PIMA, then performance estimation, and then a detailed evaluation.

6.2 Phases

PIMA divides the execution of a managed workload into five phases, shown in Figure 8.

Default. Each workload begins execution in default mode, with the nursery in DRAM and the mature space in Optane memory. Other spaces, such as the code and the meta-data space, are also placed in DRAM. Their volume as a fraction of the total heap is small.

Prepare. In the prepare phase, the JVM and the application (GraphChi) first perform a handshake, the JVM communicating its intent to sample heap allocation on different memories. Upon receiving a start-sampling signal from GraphChi, the JVM prepares and exposes on-heap buffers where the application stores end-user performance measurements during sampling. In case multiple JVMs are executing, they coordinate the beginning of the sampling phase.

Sample. In the sample phase, the GC continues to evacuate the nursery to Optane memory for a pre-determined sampling period. Assuming the example execution in Figure 8, it then switches to evacuating the nursery into DRAM. First, the JVM migrates mature live objects to Optane memory. It then informs the GC to begin evacuating nursery survivors to Optane memory. During migration and evacuation, the JVM picks an available zeroed file from a pool (shared queue) to map heap chunks into Optane memory. During sampling on either memory type, GraphChi communicates the graph load and update latencies to the JVM, which the JVM stores in on-heap buffers.

Cleanup and Hold-off. In the clean-up phase, the JVM chooses a memory type for long-term allocation during the hold-off period. It does so by comparing the workload's performance on DRAM and Optane memory. The JVM then relocates live objects to the chosen memory type. This relocation is not required if the current and chosen memory types are the same. One way to keep the relocation overhead low is to align clean-up with a major GC cycle. After the hold-off period expires, the JVM samples again to adjust to each workload's phase behavior. The JVM also zeroes and unmaps unwanted virtual memory in the background, making it available for other processes. In Optane's case, the unmapped files are enqueued into the shared pool.

6.3 Performance Impact Estimation

It is challenging to estimate the performance impact of a specific memory type. A memory type's impact manifests as a change in GC time or a change in Load, Store, and I-Cache stalls. The most accurate measure of impact is, however, at the end-user level. After observing the execution of graph analytics workloads, we extract two metrics to quantify memory type's performance impact.

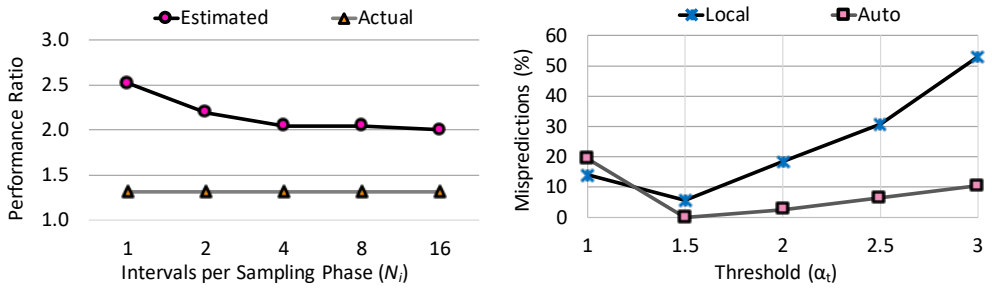
GraphChi divides each learning iteration of an analytics workload into intervals. Each interval processes a sub-graph (interval). Associated with an interval is a shard that stores its incoming edges. The outgoing edges are stored in the remaining shards. Each iteration's interval *loads* the sub-graph and its shard into memory, and *updates* the graph interval's vertices by executing a user-defined update function. The per-vertex updates propagate throughout the graph. We use as a measure of performance, the load, and update latencies for the n_{th} interval, $load^n$, and $update^n$. The sum of $load^n$ and $update^n$ quantifies an interval's performance. Several intervals, N_i, make up a sampling phase. The estimated performance during a DRAM or an Optane sampling phase is denoted by T_{DRAM} and T_{Optane}. Either one equals to the average of the aggregated load and update latencies across N_i intervals. This is mathematically equivalent to, $\sum_{n=1}^{Ni} (load^n + update^n)/N_i$. With this formulation, the ratio of T_{Optane} to T_{DRAM} is the slowdown due to Optane memory. Conversely, this ratio represents the speed-up from allocating the mature heap in DRAM. GraphChi informs the JVM of the interval id, n, and the per-interval load and update latencies during sampling.

6.4 Evaluation Results

We now evaluate PIMA across three dimensions: (1) accuracy of predicting optimal memory type, (2) performance, and (3) DRAM capacity.

Accuracy. To predict the best memory type for mature objects, PIMA first estimates the Optane-to-DRAM performance ratio. In Figure 9 (a), we compare the estimated to the actual ratio. The actual ratio is obtained with the mature heap in DRAM (DRAM-Only) versus Optane memory (default mode). The estimated ratio is computed by randomly selecting N_i consecutive intervals across hundreds of workloads' runs, and computing the ratio of T_{Optane} to T_{DRAM}. We show averages across three workloads. First, the estimated ratio is higher than the actual because T_{DRAM} and T_{Optane} exclude the I/O latencies. I/O is required to read and update graph vertices. It limits the performance impact of faster DRAM. Second, the estimation accuracy is better with a larger N_i.

The estimated Optane-to-DRAM performance ratio is a proxy for the actual ratio. To use the estimated ratio to pick the optimal memory type, we introduce a user-defined threshold of α_t.



(a) Performance Ratio **(b)** Prediction Accuracy

Fig. 9. Evaluation of the performance estimation model in PIMA. *(a) The estimated performance ratio is higher than the actual ratio. (b) The rate of mispredictions (as a percentage) is low across different NUMA settings.*

If the estimated ratio, i.e., slowdown, is higher than α_t, PIMA chooses DRAM, and otherwise, it chooses Optane memory. We compare PIMA's accuracy in predicting the optimal memory type to *static-optimal*, which places the mature heap in DRAM if the actual ratio is higher than one. If PIMA chooses a memory type that differs from the static-optimal, we count it as a misprediction. We observe the rate of mispredictions across several hundred sampling phases. In Figure 9 (b), we show the results by varying α_t from one to three, and using an N_i of two.

Misprediction rate (as a percentage) is low. Accuracy is highest for α_t of 1.5. We find that optimally setting this threshold once, empirically, works across many scenarios. We observe high prediction accuracy for Local and Auto. Misprediction rate increases with large thresholds because PIMA begins to place the mature heap in Optane memory, even for workloads that benefit from DRAM allocation. For Auto, though, because actual performance ratios are higher than Local, even at high thresholds, PIMA continues to place the mature heap in DRAM. Finally, manually tuning α_t to satisfy a different performance target is straightforward. Automating its tuning is future work.

Performance and DRAM capacity. We first evaluate PIMA for the challenging 32-core graph workloads. The goal is to maximize Optane usage without sacrificing performance. We run PIMA stand-alone, and in combination with criticality hints, comparing its performance to prior static and dynamic approaches for managing hybrid memories. We show the results of our performance evaluation in Figure 10. In (a), we show results for Local. We observe that performance degrades by almost 2× with Optane-Only. The performance of KG-W, the state-of-the-art dynamic approach, is still 1.21× worst on average than DRAM-Only. In fact, for ALS, KG-W is only 9% better in performance than Optane-Only, placing 20% of its heap in DRAM (see Table 1). In absolute terms, this translates to 1.6 GB DRAM. Overall, KG-W degrades performance compared to PIMA's default mode because of dynamic monitoring. PIMA delivers performance competitive to DRAM-Only, placing the mature heap in DRAM for PR and CC, and for ALS, just the nursery, accurately predicting its DRAM needs. PIMA improves performance over KG-W by 19% on average. The improvement is the highest for PR, which is memory-intensive, allocating more rapidly than the other two workloads (see Table 1).

Compared to DRAM-Only, PIMA is only 2% slower on average, and up to 3% for PR. The sampling and book-keeping overhead are low and determined by the number of graph intervals (N_i) in a sampling phase. For workloads that benefit from mature heap allocation in DRAM, a large N_i hurts performance. On the other hand, it also provides an opportunity to align full-heap collections in normal execution, with major GC cycles required by PIMA. Those major cycles are required during the switching of memory types and at the end of the clean-up phase. We use two intervals in a sampling phase, balancing the accuracy and sampling overhead. Indeed, we sometimes require

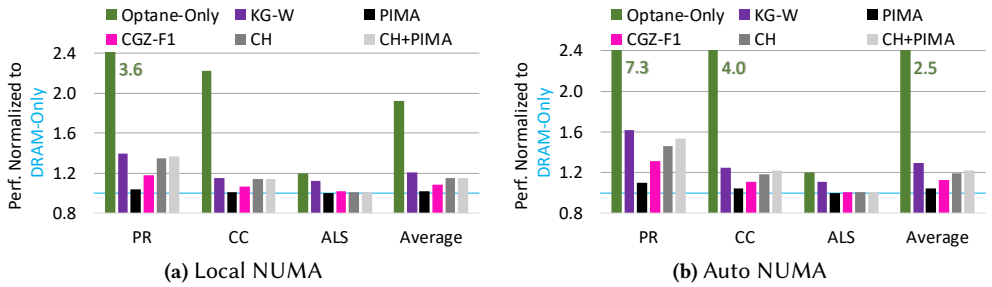


Fig. 10. Performance of 32-core graph workloads with Optane-Only, GC-managed hybrid memory (KG-W and CGZ-F1), PIMA, CH, and their composite for (a) Local, and (b) Auto. *The fully automatic PIMA delivers performance comparable to DRAM-Only. Combined with CH, CH+PIMA requires a small programmer's effort but delivers good performance and high Optane memory utilization.*

additional collections. We measure their overhead to be 40 milliseconds for DRAM to Optane switching, and 130 milliseconds for the reverse. The high overhead of Optane to DRAM transfers is due to the high scanning overhead on Optane memory. Fortunately, the overhead of extra collections amortizes over long hold-off periods. We do a sensitivity study by varying hold-off between 10 and 50 seconds. The later setting works the best for our workloads.

Comparing PIMA to the profiling-based, CGZ-F1, it outperforms CGZ-F1 by 6% on average. However, PIMA places 69% of the heap in DRAM, in contrast with 55% for CGZ-F1. To preserve DRAM capacity, we combine CGZ-F1 with PIMA. The composite consumes 15% less DRAM capacity than CGZ-F1 but slows down execution by 2% over CGZ-F1. Next to CGZ-F1 in Figure 10 (a) is the performance of CH, only 7% worst than CGZ-F1, and 15% worst than DRAM-Only. Attaching CH to only one allocation site in GraphChi delivers a huge performance boost compared to Optane-Only. On the programmer's side, the small effort of predicting hot allocation site(s) is worth the performance gain. CH places 43% (average) of the heap in DRAM and 42% for ALS. Comparing CH to PIMA, PIMA consumes a much larger DRAM capacity but performs 12% better than CH. For ALS, it places 6.25% of the heap in DRAM. Combining CH and PIMA, as shown in Figure 10 (a), delivers performance 15% worst than DRAM-Only, and conversely, 40% better than Optane-Only. In Table 1, we see that CH+PIMA places 30% of the heap in DRAM. This DRAM consumption is 25% lower than the best-performing CGZ-F1 collector. Thus, CH+PIMA brings new Pareto optimal performance versus DRAM capacity trade-offs, and on top of it, obviates offline profiling of allocation sites.

We also evaluate PIMA with Auto NUMA setting, allowing us to report PIMA's benefits across various scenarios. We show results in Figure 10 (b) with unchanged N_i of two and α_t of 1.5. With Auto, the performance gap between Optane-Only and DRAM-Only is higher, and we expect the performance benefits of PIMA to be higher. First, Kingsguard and Crystal Gazer collectors mitigate performance degradation due to Optane memory. However, PIMA delivers the best performance, resulting in only a 5% average slowdown compared to DRAM-Only, and up to 9% for PR. The overhead is due to sampling, and it is higher compared to Local because the additional mature GCs that PIMA requires take longer. Another reason is the remote Optane accesses during the sampling phase. Despite the overheads, compared to the dynamic KG-W collector, PIMA is more than 50% better for PR, and 24% better on average. Compared to Local, DRAM utilization is unchanged. Moreover, similarly to Local, the performance of CH is only 6% worst than CGZ-F1. Finally, CH+PIMA delivers good performance and maximizes the utilization of Optane memory.

So far, we have evaluated PIMA at high core count. At low core count, PIMA brings substantial benefits. With eight cores, execution happens with the mature heap in Optane memory, except during sampling and in case of mispredictions. With 16 cores, PIMA places the mature heap in DRAM

for PR. We measure the sampling and book-keeping overheads to be less than 1%. In summary, PIMA prevents the waste of DRAM at low processor usage, without sacrificing performance.

7 THREATS TO VALIDITY

① Future Optane generations will offer higher bandwidth. The sequential read and write bandwidth of the second generation is 16% and 34% higher. This growth rate may eventually bridge its performance gap with DRAM for highly allocating workloads. Improved Optane technology over time would require adjusting PIMA's critical parameters, such as N_i, α_t, and hold-off. ② Future big data workloads will demand more enormous heaps. Our heaps approach multiple gigabytes, large enough to not fit in the processor's caches, providing us a fair assessment of Optane's performance relative to DRAM. ③ This work focuses on homogeneous multithreaded workloads. We expect heterogeneous workloads to benefit from PIMA, depending on their interaction at the cache and memory-level. We leave a detailed analysis for future work. ④ A different runtime will deliver better or worst performance on Optane memory. We use Jikes RVM and its production GC (widely used in literature) to ease profiling, teasing apart scaling bottlenecks, and implementing a new memory manager. We observe scaling trends on Optane memory with OpenJDK and its Parallel Scavenge GC, similar to those we report with Jikes. In **Appendix A**, we analyze the performance scaling of graph workloads with OpenJDK on different memory types. We leave a detailed evaluation with different JVMs to separate work due to space constraints.

8 RELATED WORK

Intel Optane DIMMs were made available two years ago. Few efforts report Optane's performance as the main memory. Yang et al. [64] use native microbenchmarks to characterize its latency and bandwidth. They also study its persistent properties for embedded data stores. Mason et al. [41] analyze the interaction of page sizes with filesystem extensions for Optane memory. Patil et al. [46] use small datasets to explore Optane as the main memory for computational kernels. In contrast with prior works, we are the first to characterize real-world Java workloads on Optane memory.

Related to this work are GC approaches that exploit scalable NVM to expand physical memory. Their pro-active and fine-grained nature outperforms OS and hardware solutions to manage hybrid memories [5, 6, 58]. Wang et al. [58] use emulated NVM to store big data heaps on hybrid memory. They store infrequently accessed resilient distributed datasets (RDDs) [70] in NVM and the rest in DRAM. Due to production NVM's limited endurance, prior work proposes write-rationing collectors to mitigate their wear-out in hybrid memories [5, 6]. We re-evaluate these prior collectors on DRAM-Optane hybrid with a focus on performance. Kolokakis et al. [38] store RDDs in PCIe-attached Optane drives for scalable RDD caching in Spark. Our workload variety and focus on performance scalability invite broader applicability of Optane DIMMs as the main memory.

In addition to GC-based approaches, prior work proposes hardware, software, and cooperative approaches for exploiting hybrid memories. These approaches generalize to all languages and runtimes. The early works focus on managing die-stacked DRAM as a cache for the much large off-package DRAM [16, 17, 35, 55, 68]. (See Mittal et al. [42] for full coverage.) These caching approaches incur meta-data overhead for storing tags. They also expose a limited address space to the OS. Others propose to manage a hybrid of DRAM and a slower but scalable technology, such as phase change memory [15, 22, 37, 47, 48]. They offer a flat address space, migrating coarse-grained pages between the two memory types. Most recently, Vasilakis et al. [56] combine caching and migration in a single approach. Specifically, they use a small DRAM cache for predicting the most suitable migration candidates. In contrast to these approaches, GC-managed hybrid memories require no hardware support and adapt to fine-grained object granularity at the software level.

Exploiting the non-volatility of Optane memory requires crash-consistent heaps. To make managed heaps, crash-consistent, reachability-based NVM frameworks [50, 60] exploit GC to copy the transitive closure of *persistent roots* into Optane memory. These frameworks divide the managed heap into volatile (DRAM) and non-volatile (Optane) regions. Our performance analysis motivates allocating a fraction of the volatile heap in Optane memory in addition to its use as fast persistent storage. Our heap partitioning approaches exploit GC to discover object access patterns. Extending them and PIMA for reachability-based NVM frameworks is straightforward future work.

The sampling-based memory allocation, PIMA, relies on estimating the performance impact of DRAM and Optane as the main memory. PIMA is inspired by multicore scheduling approaches. Van Craeynest et al. [52, 53] analytically determine the most appropriate core for scheduling heterogeneous multicores. On the managed language side, Akram et al. [3] use analytical models to optimize the processor's voltage and frequency for Java workloads. They also propose a sampling-based approach for scheduling concurrent GCs on heterogeneous multicores targeting a reduction in energy consumption [7]. By contrast, we are the first to exploit GC to demonstrate low-overhead and sampling-based memory allocation for hybrid memories.

9 OUTLOOK AND CONCLUSION

As DRAM approaches scaling limits, how best to complement it with emerging memory technologies is an open question. This paper analyzes Intel Optane memory for Java workloads, focusing on performance scalability. We find that Optane substitutes DRAM without significantly hurting performance for a few critical workloads. For other highly allocating workloads, performance degrades substantially at high core count. For them, Optane complements DRAM by storing a fraction of heap objects. The resulting hybrid DRAM-Optane memory, managed by garbage collection (GC), scales on a par with DRAM. GC exploits profiling to identify and allocate *hot* objects in DRAM, delivering Pareto trade-offs between performance and DRAM capacity. Our evaluated GC approaches are workload and run-time-agnostic and sometimes waste DRAM capacity.

Understanding and mitigating scaling bottlenecks require substantial efforts at the application, virtual machine, and microarchitecture level. We find that GC slows down more on Optane memory than the application. The GC slowdown is primarily due to slow object scans on Optane memory. We also discover and mitigate the high penalty of load, store, and instruction misses for the application.

To exploit Optane memory without sacrificing performance requires workload and run-time-adaptive memory allocation approaches. This paper proposes performance impact-guided memory allocation (PIMA) that automatically chooses DRAM versus Optane memory for heap allocation. The critical problems in PIMA are estimating the performance impact of memory types and the safe sharing of Optane memory. Combining PIMA with new static and existing dynamic approaches to identify hot objects delivers new Pareto optimal trade-offs for hybrid memories. More research into emerging memories—their behavior as main memory—the resulting inefficiencies, and disruptive approaches to maximize their use will lead to more scalable main memory systems in the future.

ACKNOWLEDGEMENT

The author is grateful to J. Eliot B. Moss at the University of Massachusetts Amherst for providing access to the experimental server with Intel Optane persistent memory.

REFERENCES

[1] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. 2011. Onyx: A Protoype Phase Change Memory Storage Array. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*.

[2] Shoaib Akram, Kathryn S. McKinley, Jennifer B. Sartor, and Lieven Eeckhout. 2018. Managing Hybrid Memories by Predicting Object Write Intensity. In *Proceedings of the Conference Companion of the International Conference on Art,*

Science, and Engineering of Programming (Programming'18 Companion).

[3] S. Akram, J. B. Sartor, and L. Eeckhout. 2016. DVFS performance prediction for managed multithreaded applications. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).*

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

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

[6] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2019. Crystal Gazer: Profile-Driven Write-Rationing Garbage Collection for Hybrid Memories. *SIGMETRICS Perform. Eval. Rev.* 47, 1 (Dec. 2019), 21–22.

[7] Shoaib Akram, Jennifer B. Sartor, Kenzo Van Craeynest, Wim Heirman, and Lieven Eeckhout. 2015. Boosting the Priority of Garbage:Scheduling Collection on Heterogeneous Multicore Processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT).*

[8] Luiz Andre Barroso and Urs Hoelzle. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* (1st ed.). Morgan and Claypool Publishers.

[9] Brian Beeler. 2019. Intel Optane DC Persistent Memory Module (PMM). https://www.storagereview.com/news/

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

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

[12] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA).*

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

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

[15] S. Bock, B. R. Childers, R. Melhem, and D. Mossé. 2016. Concurrent Migration of Multiple Pages in software-managed hybrid main memory. In *IEEE 34th International Conference on Computer Design (ICCD).*

[16] Chia-Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2014. CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[17] Chia-Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2015. BEAR: techniques for mitigating bandwidth bloat in gigascale DRAM caches. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA).*

[18] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[19] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP).*

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

[21] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys).*

[22] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys).*

[23] S. Eyerman and L. Eeckhout. 2008. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro* 28, 3 (2008), 42–53.

[24] Eclipse Foundation. 2020. Desktop IDEs. https://www.eclipse.org/ide/

[25] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2013. A Study of the Scalability of Stop-the-World Garbage Collectors on Multicores. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[26] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[27] The HSQL Development Group. 2020. HyperSQL. http://hsqldb.org

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

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

[30] Jim Handy. 2018. Emerging Memories Today: The Technologies: MRAM, ReRAM, PCM/XPoint, FRAM, etc. https://thememoryguy.com/

[31] Jim Handy. 2018. Emerging Memories Today: Why Emerging Memories are Necessary. https://thememoryguy.com/

[32] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

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

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

[35] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*.

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

[37] Dmitry Knyaginin, Vassilis Papaefstathiou, and Per Stenström. 2018. ProFess: A Probabilistic Hybrid Main Memory Management Framework for High Performance and Fairness. In *IEEE International Symposium on High Performance Computer Architecture, HPCA*.

[38] Iacovos G. Kolokasis, Anastasios Papagiannis, Polyvios Pratikakis, Angelos Bilas, and Foivos Zakkak. 2020. Say Goodbye to Off-heap Caches! On-heap Caches Using Memory-Mapped I/O. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

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

[40] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.* 5, 8 (April 2012), 716–727.

[41] T. Mason, T. D. Doudali, M. Seltzer, and A. Gavrilovska. 2020. Unexpected Performance of Intel® Optane™ DC Persistent Memory. *IEEE Computer Architecture Letters* 19, 1 (2020), 55–58.

[42] S. Mittal and J. S. Vetter. 2016. A Survey Of Techniques for Architecting DRAM Caches. *IEEE Transactions on Parallel and Distributed Systems* 27, 6 (2016), 1852–1863.

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

[44] Priya Nagpurkar, Harold W. Cain, Mauricio Serrano, Jong-Deok Choi, and Chandra Krintz. 2007. Call-Chain Software Instruction Prefetching in J2EE Server Applications *(PACT '07)*. IEEE Computer Society, USA, 140–149.

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

[46] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance Characterization of a DRAM-NVM Hybrid Memory Architecture for HPC Applications Using Intel Optane DC Persistent Memory Modules. In *Proceedings of the International Symposium on Memory Systems (MemSys)*.

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

[48] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta. 2017. Automating the Application Data Placement in Hybrid Memory Systems. In *IEEE International Conference on Cluster Computing (CLUSTER)*.

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

[50] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: An Easy-to-Use Java NVM Framework Based on Reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 316–332.

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

[52] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. 2013. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *Proceedings of the international conference on Parallel architectures and compilation techniques (PACT)*.

[53] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[54] Rik van Riel. 2014. Automatic NUMMA Balancing. https://www.redhat.com/files/summit/2014/summit2014_riel_chegu_w_0340_automatic_numa_balancing.pdf

[55] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. 2019. Decoupled Fused Cache: Fusing a Decoupled LLC with a DRAM Cache. *ACM Trans. Archit. Code Optim.* 15, 4 (2019), 65:1–65:23.

[56] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. 2020. Hybrid2: Combining Caching and Migration in Hybrid Memory Systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[57] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[58] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[59] Matthew Wilcox. 2014. Add Support for NV-DIMMs to Ext4. https://lwn.net/Articles/613384/

[60] Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, and Haibing Guan. 2020. GCPersist: An Efficient GC-Assisted Lazy Persistency Framework for Resilient Java Applications on NVM. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)*. 1–14.

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

[62] Jian Xu and Steven Swanson. 2016. 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)*.

[63] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.

[64] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*.

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

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

[67] Jung Yoon, Ranjana Godse, and Andrew Walls. 2018. 3D NAND Technology Scaling helps accelerate AI growth. In *Proceedings of the Flash Memory Summit (FSM)*.

[68] Vinson Young, Chia-Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2018. ACCORD: Enabling Associativity for Gigascale DRAM Caches by Coordinating Way-Install and Way-Prediction. In *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA*.

[69] S. Yu and P. Chen. 2016. Emerging Memory Technologies: Recent Trends and Prospects. *IEEE Solid-State Circuits Magazine* 8, 2 (2016), 43–56.

[70] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*.

[71] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. 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)*.

A ANALYSIS WITH OPENJDK

In this appendix, we show the performance scaling of graph workloads with OpenJDK for different memory types. We compare the resulting trends with those observed with the Jikes RVM. In Figure 11, we show the scaling of Optane-Only and hybrid DRAM-Optane memory relative to DRAM-Only for PR, CC, and ALS with increasing core count. In (a), we show the scaling trends with OpenJDK, and in (b), we show the trends with Jikes RVM. The figures show harmonic means with two NUMA settings. The main findings of our work still hold with OpenJDK.

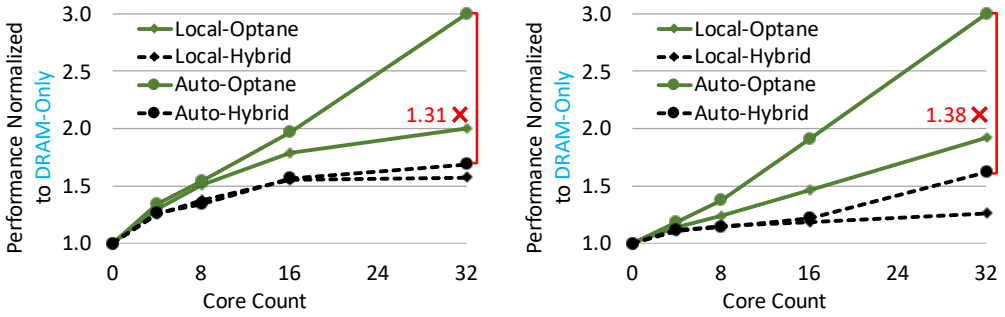


Fig. 11. Performance scalability of graph workloads with (a) OpenJDK and (b) Jikes RVM. Relative to DRAM-Only, an Optane-Only system scales poorly in performance regardless of the JVM choice. Hybrid memory narrows the performance gap between the two memory types in slightly different ways.

We use the Parallel Scavenge (PS) GC in OpenJDK and configure it to mirror Jikes's heap organization and settings. We use a 32 MB young generation and set the MaxTenuringThreshold to zero. The mutator and GC thread count and other settings are similar to Jikes's. We observe that regardless of the JVM choice, Optane scales poorly relative to DRAM for the highly allocating graph workloads. The 32-core slowdown with Auto is 3× with both JVMs. Hybrid memory (KG-N) bridges the performance gap by 1.31× with OpenJDK and 1.38× with Jikes. Improving the support for hybrid memory in OpenJDK is left to future work. Compared to Jikes, at low core count, the slowdown due to Optane memory is higher with OpenJDK. We suspect this difference is because: (1) OpenJDK exploits multicore parallelism better and saturates the memory system at low core count. (2) In contrast with no interpreter in Jikes, it uses an interpreter that results in additional memory traffic at low core count. Finally, we validate the scaling trends across a variety of PS settings and also with the production G1 collector in OpenJDK.

Received December 2020; revised January 2021; accepted February 2021