JPDHeap: A JVM Heap Design for PM-DRAM Memories

Litong You[1†], Tianxiao Gu[3], Shengan Zheng[2], Jianmei Guo[3*], Sanhong Li[3], Yuting Chen[1*], and Linpeng Huang[1]
[1]Shanghai Jiao Tong University, [2]Tsinghua Univeristy, and [3]Alibaba Group
Email: [1]{litong.you, ytchen, lphuang}@sjtu.edu.cn, [2]venero@tsinghua.edu.cn,
[3]{tianxiao.gu, jianmei.gjm, sanhong.lsh}@alibaba-inc.com

***Abstract*—Real-world e-commerce systems need large cache capacities. Persistent memory (PM) can be employed to enlarge JVMs' cache capacities, meanwhile they incur heavy write slowdowns and garbage collection overheads. This paper proposes JPDheap, a JVM heap design for PM-DRAM memories. A JPDheap is composed of a standard Java heap on DRAM and another heap on PM. The core insight is to separate heap objects and store them on DRAM or PM, allowing objects to be accessed much more efficiently. Our evaluation shows that JPDheap outperforms state-of-the-art heap designs by up to 115.96% in increasing applications' throughput and by up to 87.03% in decreasing the average latency.**

***Index Terms*—JVM Heap; Hybrid Memory; Persistent Memory**

I. Introduction

Real-world e-commerce systems need large cache capacities. For example, the transaction peak reached 540,000 transactions per second in the 2019 Alibaba's shopping festival [18]. The performance of the e-commerce system needs to be guaranteed for such an event, particularly during the transaction peak period. The rapidly increasing data in recent years requires JVMs to provide large cache capacities to applications, aiming at achieving high cache hit rates for heavy workloads. Conventional JVMs use DRAM as cache, which suffers from its limited capacity, high power consumption, and high cost. Meanwhile, DRAM is not scalable enough: a JVM's heap memory is constrained by the main memory, whose size is eventually constrained by the memory DIMMs.

Some recent researches use persistent memory (PM) [2], [11], [23] for enlarging JVMs' cache capabilities and running e-commerce systems [7]. PM has many advantages, such as low cost, large capacity, and near-DRAM access latency. PM also has the potentials of significantly reducing the monetary and energy cost in large-scale data centers. By adopting PM into the JVM, applications and their JVMs are able to obtain much larger cache capacities and higher cache hit rates at runtime.

Although it is promising to run JVMs on PM, the idea has not yet been fully explored. One problem remaining unsolved is:

Problem Description: How should a JVM's heap be designed such that it leverages PM to improve the application's throughput and as well reduce performance overheads they raised?

*Corresponding authors.
†This work was conducted during the research internship at Alibaba Group.

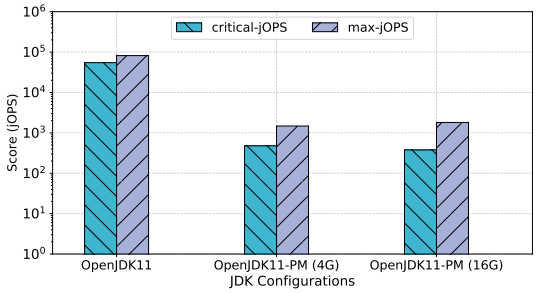


Fig. 1. Performance of running SPECjbb 2015 on OpenJDK's HotSpot VMs (with/without PM).

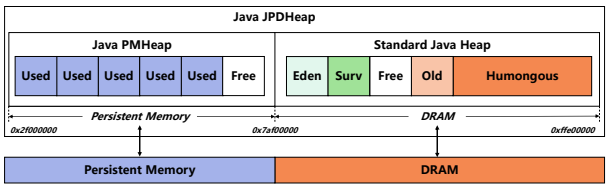
In particular, running JVMs on PM incurs two types of performance overheads:

Write slowdowns. PM has lower write bandwidth than DRAM, and thus PM-based heaps are unfriendly to write-intensive applications. For instance, Intel's Optane DC persistent memory has higher latency (346 ns [12]) than DRAM but lower latency than SSD. Unlike DRAM, its bandwidth is asymmetric: for a single DIMM, the maximal read bandwidth is 2.9× of the maximal write bandwidth, while DRAM has a smaller gap (1.3×) between its read and write bandwidths [12].

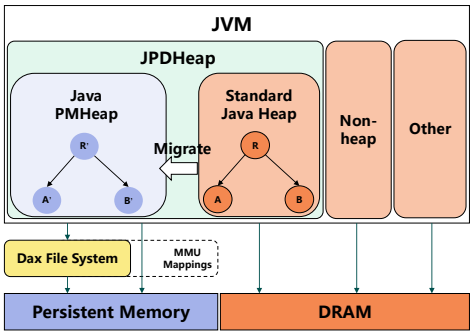
A PM-based JVM heap [2], [14] also suffers from write slowdowns. We run the SPECjbb 2015 benchmark suite on OpenJDK's HotSpot VMs with PM [14]. As Figure 1 shows, the benchmarks' critical- and the max-jOPS drop sharply, compared to those running on HotSpot VMs with DRAM. The results indicate that naively replacing DRAM cache with PM leads to slowdowns to the applications. A JVM's heap should be redesigned for the unique characteristics of PM.

GC overheads. A standard Java heap can be paused shortly, but frequently for garbage collection (GC) when running read-intensive workloads. Meanwhile, collecting heap garbages in PM can lead to significant performance overheads—many write operations do exist during garbage collections, and thus it is much slower to collect garbages in PM than in DRAM; a PM can store many more heap objects than a DRAM, and thus it spends much time in marking and sweeping heap garbages. GC overheads need to be reduced for enhancing the JVMs' performance.

To fully exploit the benefit of PM and address the above issues, we propose JPDHeap, a JVM heap design for PM-DRAM memories. A JPDHeap is composed of a JSH (a



(a) Memory layout.



(b) Memory management. PMHeap is not subject to Java GC; the standard Java heap is on DRAM and subject to Java GC.

Fig. 2. An overview of JPDHeap.

standard Java heap on DRAM) and a PMHeap (a large capacity heap on PM). The core insight is to separate heap objects and store them on different heap regions—read-intensive objects are mainly stored in the PMHeap regions and the other objects in the JSH regions. It allows heap objects to be accessed much more efficiently.

This paper makes the following contributions:

- **Memory layout.** JPDHeap is a JVM heap design for PM-DRAM memories. It enlarges a JVM's cache, significantly improving an application's cache throughput. The PMHeap, rather than the off-heap storage, stores read-intensive objects, reducing serialization overheads.
- **Memory management.** We design an approach to manage heap at runtime. This approach divides objects into read-intensive objects and the others, storing them into PMHeap and JSH, respectively. This design reduces the number of write operations on PM, and also reduces the frequencies of runtime garbage collections.
- **Implementation and evaluation.** We evaluate JPDHeap on the Intel's Optane DC PM devices. The evaluation clearly shows the strengths of JPDHeap: compared with state-of-the-art heap designs, JPDHeap increases the applications' throughput by 25.55∼115.96%, and decreases the average latencies by 15.66∼87.03%. JPDHeap has been deployed to real production environments; the CPU consumption is reduced by 5%.

II. Design

A. Overview

JPDHeap is designed as a JVM heap for PM-DRAM memories. It stands for Java PM-DRAM heap, a hybrid memory combining a heap on PM and another on DRAM. As Figure 2 shows, JPDHeap is composed of a JSH on DRAM and a PMHeap on PM. It allows JPDHeap to hold two features.

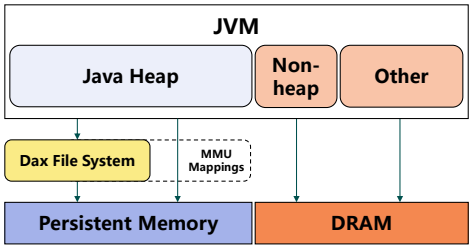


Fig. 3. JPH. The heap resides on PM and the other parts reside on DRAM. This design does not change the behavior of the JVM [2].

TABLE I
Typical PMHeap operations.

TYPE	OPERATION

Object-Related	① void check(obj); //Check if *obj* is in PMHeap
	② void getSize(*obj*); //Calculate the size of a PMHeap object *obj*
	③ void move(*obj*); //Move *obj* from the JSH into the PMHeap

Memory-Related	④ unsigned long freeMemory(); //Return the currently available memory in PMHeap
	⑤ void compact(); //Reclaim free memory if there is no sufficient unused memory in PMHeap

1) Large capacity. JPDHeap extends the original JVM heap with a managed, and usually large capacity heap residing on PM. A PMHeap stores a number of Java objects, which improves cache hit rate.
2) Low overhead. Heap objects are stored on different memory regions: read-intensive objects are mainly stored in the PMHeap and the other objects in the JSH, which significantly reduces write slowdowns. Only the objects in the JSH can be garbage collected, and the PMHeap objects are free from garbage collections. JPDHeap keeps all cached data, if possible, on-heap but off-storage, reducing serialization overheads. The serialization task is usually performed for moving objects from the memory into the storage devices, saving the memory space at runtime.

B. Memory Layout

The JPDHeap memory is split into two sets of fix-sized regions on DRAM and PM, which are managed by JSH and PMHeap, respectively. In particular,

- Each region of JPDHeap is a contiguous range of a virtual memory, where the PMHeap is mapped to the low address of the JPDHeap, and the JSH is allocated from the high address space. Thus, the PMHeap and the JSH are separated by their memory addresses.
- PMHeap is mapped into several regions. The DRAM regions are still mapped into logical representations of Eden, Survivor, Old, Free, and Humongous regions.
- PMHeap is exclusively used to store read-intensive or long-lived Java objects. Each size of the region is set during the JVM startup, ranging 1∼32MB.

```
1  private static void testJPDHeap() {
2    List<Object> lists = new ArrayList<>();
3    List<Object> test = new ArrayList<>();
4    for (int i = 0; i < 10000; i++) {
5      lists.add("test" + i);}
6    // Read-intensive objects
7    List<Object> sublists = lists.subList(500, 5000);
8    test = move(sublists);
9  }
```

Fig. 4. A code example. Objects are created in JSH, and the read-intensive ones are then moved into PMHeap.

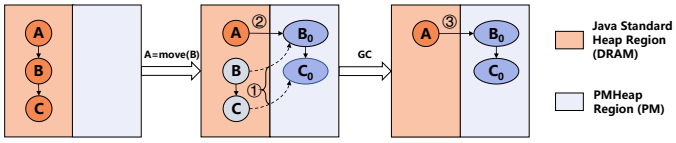


Fig. 5. An example of migrating JPDHeap objects.

We compare JPDHeap against JPH, a state-of-the-art heap design on DRAM-PM memories. As Figure 3 shows, JPH is a heap running on PM, whose object allocations and garbage collections are performed only on PM. Comparatively, a JPDHeap is hybrid, containing a PMheap on PM and a Java standard heap on DRAM.

C. Memory Management

JPDHeap extends the JVM's memory management in managing JSH objects and PMHeap ones. Two types of operations (object- and memory-related operations), as Table I shows, are defined to support memory allocation, object migration, and memory compaction.

Memory allocation. A PMHeap region is still a heap region, and can be normally accessed. JPDHeap employs the Garbage-First (G1) Garbage Collector [8] to organize memory and allocate heap regions on DRAM and PM. It employs G1 to partition the heap into a set of equal-sized heap regions; G1 is also employed for performing a concurrent global marking at runtime.

In JDPHeap, each region is divided into several 512-byte cards. Each card has a one-byte entry in a global card table that can be used to track which cards are modified by the mutator threads. Subsets of these cards are tracked, and referred to as Remembered Sets (RS) [8].

During the memory allocation phase, the allocated object are allocated on a number of cards that are contiguous in their physical addresses.

Object migration. JPDHeap allows programmers to move Java objects from the JSH to the PMHeap. During this process, the JVM creates and manages a ThreadLocal forwarding table that keeps a list of forwarding pointers. Each value in the table is the address of an object under migration.

A move operation needs to be invoked explicitly for migrating objects. The migration of an object (obj) is typically performed as follows:

- JPDHeap checks whether obj is in the PMHeap by comparing obj's address against the right boundary of the PMHeap. obj can be moved into the PMHeap only when obj is outside the PMHeap.
- Object movement is performed during the non-GC phase. The JVM localizes obj and traverses all of the reachable objects of obj. The results are stored in the forwarding table.
- The object move operation is performed for moving (copying) obj from the JSH to the PMHeap. The forwarding table is deleted when the object migration is completed.

Algorithm 1 Compacting PMHeap

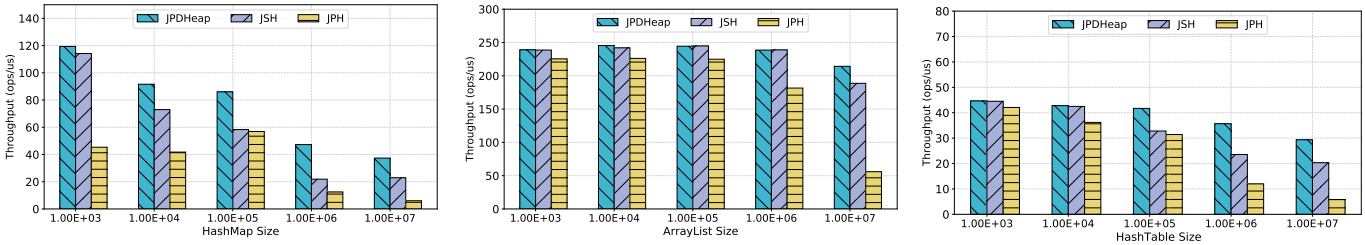
```
for pm_region in PMHeap do
  if sflag is INUSE then
    if rflag is TRUE then
      continue
    end if
    if liveness <threshold then
      for obj in pm_Rset do
        Append copies of obj and all its reachable objects
into UNUSED pm_region
        Update the reference of obj
        Delete obj and all its reachable objects
      end for
    end if
    sflag ← UNUSED
  end if
end for
```

Figure 5 shows an example of migrating a JPDHeap object B from JSH to PMHeap. Since there is no pointer from PMHeap to JSH, all objects that are B-reachable must be migrated to PMHeap after move(B) is called. Then, the JVM creates B_0 (a copy of B) and C_0 (a copy of C) in the PMHeap(①). After that, the reference of the object A needs to be updated such that it points to B_0 (②). Thereafter, there are five objects (A, B, C, B_0, and C_0) in the JPDHeap. Among them, B and C are marked as garbage objects, and can be reclaimed by G1. B and C are further replaced with B_0 and C_0 in PMHeap, respectively (③). During migration, an object may have two copies in the JSH and the PMHeap (e.g., B and B_0).

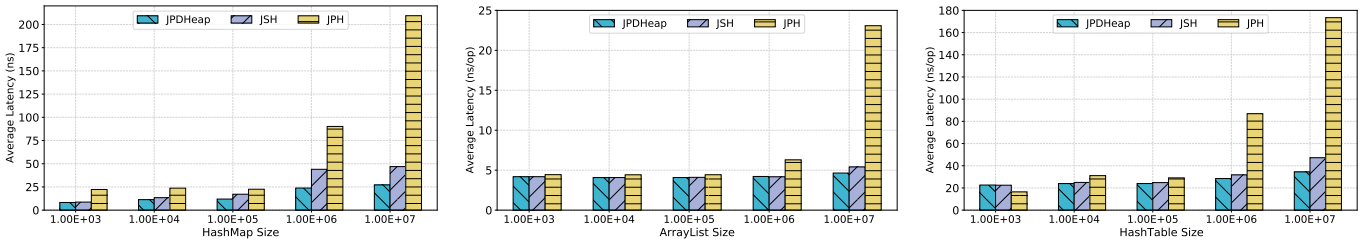
Memory compaction. GC on JSH is light-weight and highly efficient, while it is not feasible for PMHeap because PMHeap stores many more long-lived objects.

We implement compact-GC, an extension of G1 in OpenJDK's HotSpot. Compact-GC can compact the PMHeap regions and reclaim the unused memory. When PMHeap does not have enough memory for migrating objects, compact-GC starts to compact it. If compact-GC fails, a full GC is needed.

Algorithm 1 shows the memory compaction algorithm of compact-GC in PMHeap. We modified the region in PMHeap by adding two 1-bit flags. $sflag$ indicates whether the region is empty; $rflag$ indicates whether the region has forward pointer from JSH. pm_Rsets are per-region entries tracking whether there are external references pointing to objects in a PMHeap region. Each pm_region has its own pm_Rset. $liveness$ shows the ratio of live objects in a PMHeap region. The



(a) Throughput for HashMap. (b) Throughput for ArrayList. (c) Throughput for HashTable.



(d) Latency for HashMap. (e) Latency for ArrayList. (f) Latency for HashTable.

Fig. 6. Comparing JPDHeap with JSH and JPH.

$threshold$ is adjusted by the Pause Prediction model [8] in G1 GC. We employ the STAB algorithm [9] to maintain a snapshot for live objects during compact-GC. Consequently, once a system crashes, PMHeap can be recovered to a consistent state.

III. Evaluation

We have implemented JPDHeap on OpenJDK 8 and evaluated it against some state-of-the-art heap designs. Our evaluation is aimed at answering two questions:

- **RQ1.** How efficient is JPDHeap in storing and loading Java objects?
- **RQ2.** How much overhead can be reduced by JPDHeap?

A. Setup

The evaluation is set up as follows:

Heap designs. We evaluated JPDHeap against a pure JSH supported in JDK 8 (on DRAM), a Java heap on PM supported in JDK 11 (JPH). We also evaluated JPDHeap against an off-heap storage on an anonymous production platform. The size of JPDHeap, JPH and JSH are set 32 GB; the ratio of PM and DRAM in JPDHeap is set 3:1.

Some PM-based heap designs were not compared with JPDHeap because they were not publicly available.

Metrics. Three metrics are used: (1) Throughput. It calculates how many operations are executed per second. (2) Latency. It measures how many seconds each benchmark spends in running. (3) GCoverhead. It measures the average GC pause time w.r.t. each benchmark.

Benchmarks. We have implemented three micro-benchmarks: HashMap(size), ArrayList(size), and HashTable(size). In order to emulate read-intensive workloads, we let the three commonly-used data structures be equipped with fixed size objects. The sizes are 10^4, 10^5, 10^6, and 10^7. Java Microbenchmark Harness (JMH) [16] is employed as the harness.

We also evaluated JPDHeap on SPECjbb2015, a popular benchmark developed for measuring performance based on the latest Java application features*.

Configurations. We let the JVM with JPDHeap be run on a server with six Optane DCPMM (256 GB per DIMM, 1.5 TB in total), 128 GB of DRAM memory, and two 2.5 GHz Intel Xeon Platinum 8269CY CPUs (104 cores in total). The DCPMMs are exposed to user-space applications via the DAX-file system interface [21]. Correspondingly, an Ext4-DAX file system on PM is mounted to support PM accesses using the AppDirect mode.

B. Throughput and Latency

1) Running Single-Threaded Benchmarks: The evaluation on the three micro-benchmarks clearly shows the strengths of JPDHeap. JPDHeap outperforms JSH and JPH in that it increases the applications' throughputs and reduces the write latencies. As Figure 6(a) shows, the HashMap's throughput decreases when its size increases; JPDHeap outperforms JSH and JPH by 25.55∼115.96% and 51.24∼521.36%, respectively. JPDHeap decreases average latencies by 15.66∼87.03%, compared with the other two designs. JPDHeap also outperforms JSH and JPH on ArrayList and HashTable with different sizes of objects. For example, for ArrayList(10^6), JPDHeap outperforms JPH by 31.40∼282.32% in the benchmark's throughout, and has an average latency close to JPH; for HashTable(10^5), JPDHeap outperforms JSH and JPH by 27.14∼51.37% and 17.76∼407.18% in the throughput, respectively, and decreases the average latency by 17.44∼80.13%.

2) Running Multi-Threaded Benchmarks: We also compared the heap designs using multi-threaded benchmarks. JPDHeap outperforms JSH when the number of threads increases. Let HashMap(10^7) be chosen as an example. Let the application be run with 1∼7 threads. As Figure 7 shows, along

* https://www.spec.org/jbb2015/

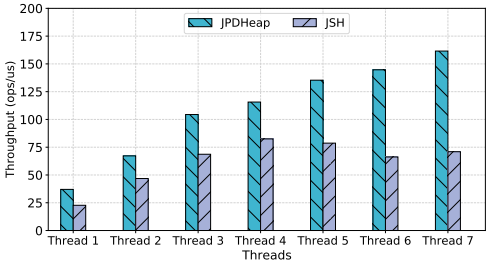


Fig. 7. Performance of multi-threads on JPDHeap and JSH.

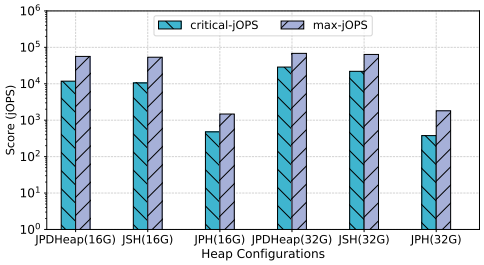


Fig. 8. Running SPECjbb 2015 on JPDHeap, JSH and JPH.

with the increase of the number of threads, the benchmark's throughput increases when it is run on JPDHeap. Comparatively, when the benchmark is run on JSH, its throughput decreases if the number of threads is greater than 4.

We compared JPDHeap with JSH and JPH on SPECjbb 2015. As Figure 8 shows, with two different sizes of heap, the critical and the max jOPS of JPDHeap are greater than JSH and JPH.

Therefore, we have the answer to RQ1:

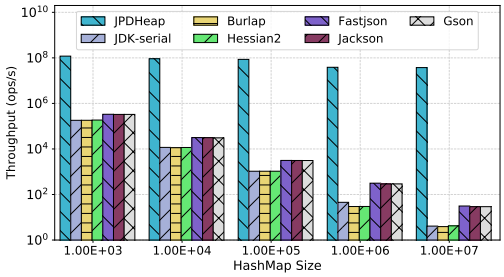
> **Efficiency.** For read-intensive workloads, JPDHeap outperforms JSH & JPH in increasing applications' throughout and reducing the average latency. The benchmarks' throughputs can be further enhanced when the heap objects are accessed by multi-threads.

C. Overhead

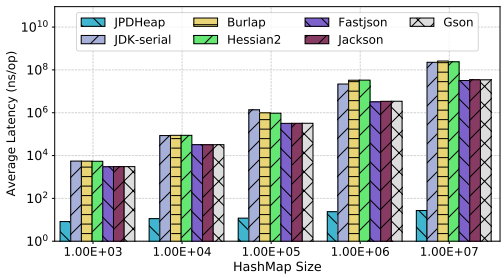
1) Serialization Overhead: We employed DirectByteBuffer as an off-heap memory [6], with six serialization frameworks (JDK-Native, Burlap, Hessian2, Fastjson, Gson, and Jackson).

The micro-benchmark's throughput on the off-heap memory decrease sharply (see Figure 9). The HashMap(10^7)'s throughput on the off-heap memory is 4.096 ops/s, and that on the JPDHeap is 3.62×10^7 ops/s. As Figure 9 shows, the average latency on JPDHeap is far less than that on state-of-the-art Java serialization frameworks, and the speedup ranges from $101\times$ to $962\times$.

2) GC Overhead: We perform a number of *get* operations on HashMap on JPDHeap, JSH, and JPH. As Figure 10(a) shows, the JVM's GC time decreases when JPDHeap is employed. The GC time is decreased by 6.50∼90.32%. With the increase of the size of data-set, JSH and JPH must utilize young-GC to traverse the live objects. which significantly increases the GC time of JSH and JPH. Since JPDHeap explicitly disbale young-GC on PMHeap, the GC time in



(a) Throughput for HashMap.



(b) Latency for HashMap.

Fig. 9. Comparing JPDHeap against Java serialization frameworks. The Y-axis is exponential. The throughput and latency of JPDHeap increase linearly.

JPDHeap is decreased. We also use GCViewer [1] to parse the GC logs. Latency spikes can be detected. In Figure 10(b), the pauses spike is up to 0.2713 seconds. There are many green spikes in Figure 10(b), indicating that the get operations on HashMap(10^7) running on JSH lead to heavy GC overheads.

Therefore, we have the answer to RQ2:

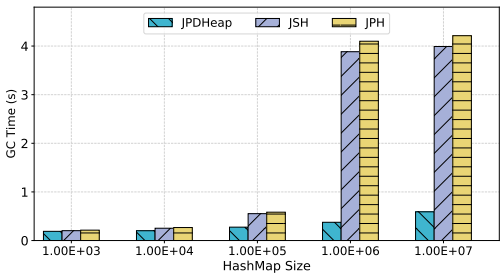
> **Overhead reduction.** JPDHeap significantly reduces serialization overheads and the GC time.

IV. Related Work

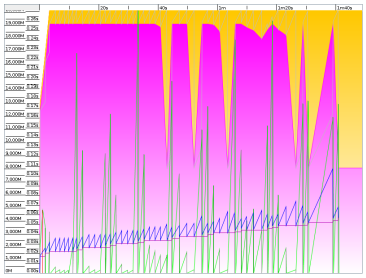
We discuss two strands of related work.

A. PM-Based Heap

As PM technology becomes mature, an increasing number of systems start to use PM. Makalu [4] provides a crash-consistent heap allocator for PM. Espresso [23] employs PM to provide a general design called Persistent Java Heap (PJH) for managing persistent objects. Intel proposes Java support for PM [2]–The JVM uses new keywords to allocate all the Java objects created by the application on the Java heap. JDK 10+ provides facilities for hosting Java heap on the Intel's Optane DC PM [14]—a JVM's flag `-XX:AllocateHeapAt` directs the allocation of the JVM's heap on PM. Write Rationing [3] is a GC technique that places highly mutated objects in DRAM and mostly read objects in PM to increase PM's lifetime. AutoPersist [17] directly manages Java objects in PM to improve the access performance, and GCPersist [22] further extends it to support resilient applications with trivial overheads. Some other efforts, such as RaPPs page access profiling technique [15] and Zhous technique [24], are not open sourced or rely on specific hardware environments, making them not comparable with JPDHeap.



(a) GC time of JPDHeap, JSH and JPH.



(b) GCViewer result of HashMap on JSH.

Fig. 10. Java GC overhead comparison.

Comparatively, JPDHeap is designed as a hybrid heap on DRAM-PM memories. It is designed to use PM to improve applications' throughputs and eliminate their serialization overheads; it also stores Java objects in different memory regions, which guarantees the applications' performance.

B. Framework and Language Facilities

Many studies have been conducted for boosting the performance of persistence management of native code [10], [13]. However, how PM can be exploited by high-level programming languages is less studied. Tian et al. [19], [20] prove that the performance gap between persistent heap and persistent storage mainly comes from serialization (and deserialization). They propose a framework for unifying accesses to persistent objects both in persistence storage and in temporary memory. Java provides a POJO persistence model for object-relational mapping called Java Persistence API (JPA) [5]. JPA transforms persistent objects between Java applications and relational database management systems (RDBMS). Some libraries provide engineers with supports (e.g., command-line options and memory allocation statements) to build and run applications on PM. For example, Persistent Collections for Java (PCJ) [11] is a Java library allowing developers to design or retrofit their applications around PM; it also implements a new persistent type system on PM, managing heap objects using the Intel's PMDK (Persistent Memory Development Kit). Meanwhile, PCJ does not eliminate the serialization overhead, since it stores persistent data as native off-heap objects.

On the contrary, JPDHeap leverages the G1 garbage collector to manage the heap space. Allocating read-intensive objects on JPDHeap also helps reduce the GC's overhead as well.

V. Conclusion

JPDHeap is a JVM heap designed for PM-DRAM memories. JPDheap allows a JVM to store heap objects in different memory regions, speeding up memory accesses and reducing write slowdowns; PMHeap is GC-insensitive and has large capacity, which helps reduce GC frequencies and serialization overheads. The evaluation results show that JPDHeap provides notable performance enhancement on various workloads. JPDHeap has been practically used in production environments, providing high throughput for Java applications.

VI. Acknowledgment

We sincerely thank our shepherd for helping us improve the paper. We also thank the anonymous reviewers for their insightful feedback. This work is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003302), the National Natural Science Foundation of China (Grant No. 62032004, 61772200). The work is also partially supported Alibaba Group. Shengan Zheng is supported by China Postdoctoral Science Foundation (Grant No. 2020M680570).

References

[1] GCViewer. https://github.com/chewiebug/GCViewer/.
[2] Java* Support for Intel Optane DC Persistent Memory. https://software.intel.com/en-us/articles/java-support-for-intel-optan e-dc-persistent-memory/.
[3] Shoaib Akram, Jennifer B Sartor, et al. Write-rationing garbage collection for hybrid memories. *ACM SIGPLAN Notices*, 2018.
[4] Kumud Bhandari, Dhruva R Chakrabarti, et al. Makalu: Fast recoverable allocation of non-volatile memory. *ACM SIGPLAN Notices*, 2016.
[5] Heiko Böck. Java persistence api. In *The Definitive Guide to NetBeans Platform 7*. Springer, 2012.
[6] Langshi Chen, Bo Peng, Bingjing Zhang, Tony Liu, et al. Benchmarking harp-daal: High performance hadoop on knl clusters. In *CLOUD*, 2017.
[7] Ian Cutress and Billy Tallis. Intel launches optane dimms up to 512gb: Apache pass is here, 2016.
[8] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *ISMM*, 2004.
[9] Christine H Flood, Roman Kennke, et al. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *PPPJ*, 2016.
[10] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, et al. Nvthreads: Practical persistence for multi-threaded applications. In *Eurosys*, 2017.
[11] Intel. Persistent collection for Java. https://github.com/pmem/pcj.
[12] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.
[13] Sanketh Nalli et al. An analysis of persistent memory use with whisper. In *ACM SIGARCH Computer Architecture News*, 2017.
[14] OpenJDK. Heap Allocation on Alternative Memory Devices. https://openjdk.java.net/jeps/316.
[15] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *ICS*, 2011.
[16] Aleksey Shipilev. Jmh: Java microbenchmark harness.
[17] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: An easy-to-use java nvm framework based on reachability. In *PLDI*, 2019.
[18] The Washington Post. Chinese online shoppers shatter Cyber Monday records on Double 11.
[19] Yuchuan Tian and Fang Wang. A unified access manner for storage-class memory. In *ICPADS*, 2016.
[20] Yuchuan Tian and Fang Wang. Managing persistent objects with a unified access framework in persistent memory. In *ICPADS*, 2017.
[21] Matthew Wilcox. Dax: Page cache bypass for filesystems on memory storage. 2014.
[22] Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, and Haibing Guan. Gcpersist: an efficient gc-assisted lazy persistency framework for resilient java applications on nvm. In *VEE*, 2020.
[23] Mingyu Wu, Ziming Zhao, Haoyu Li, et al. Espresso: Brewing java for more non-volatility with non-volatile memory. In *ASPLOS*, 2018.
[24] Ping Zhou, Yu Du, Youtao Zhang, and Jun Yang. Fine-grained qos scheduling for pcm-based main memory systems. In *IPDPS*, 2010.