A Study of Latency-Aware Data-Placement in Heterogeneous (PMEM) Memory Systems*

João Antão[1], João Barreto[2], and João M. Lourenço[1]

[1] FCT—NOVA University Lisbon & NOVA LINCS, Portugal
jp.antao@campus.fct.unl.pt joao.lourenco@fct.unl.pt
[2] IST Lisbon & INESC-ID, Portugal
joao.barreto@tecnico.ulisboa.pt

Abstract. Late demands for big data operation reveal shortcomings in the traditional relational model and consequently shift the distributed storage paradigm in favor of NoSQL databases. Such databases rely mainly on DRAM operation and weak consistency models to achieve high performance and availability.
In this work, we explore Persistent Memory (PMEM) hardware to enable large and cost-effective NoSQL deployments. More specifically, we analyze different data allocation strategies and how they impact system performance.
Our results show that DRAM is 2× and 1.5× and faster than PMEM for write and read operations respectively. This performance gap motivates system developers to adopt strategies that reduce the impact of PMEM slow performance while taking advantage of its cost-effective capacity. We show that static approaches are not flexible and do not adapt to the running workload. For example, we see that with a static allocation policy, we can be using a higher percentage of DRAM but still have worse throughput and latency than with a higher percentage of PMEM.

Keywords: Heterogeneous Memory · Presistent Memory · Latency · Data Placement · NOSQL Databases

1 Introduction

Distributed storage systems are critical to support large-scale online services. These services become increasingly demanding in terms of user experience. Such demands have exposed shortcomings in the traditional relational model and shifted the paradigm of distributed storage in favor of NoSQL databases, as they successfully leverage weak consistency models, and mainly in-memory operation to achieve high availability. Throughout this work, we will focus on the latter.

Computer systems hosting databases rely on a two-level memory hierarchy with contrasting technologies: *volatile main memory* devices (DRAM), providing

* Supported by Research Grants PTDC/CCI-COM/32456/2017 & LISBOA-01-0145-FEDER-032456.

fast access and small capacity; and *non-volatile storage* devices (SSD, magnetic disks), with much higher capacity than main memory, but also much slower. Moreover, storage devices are more cost-effective than main memory devices but, due to their lower performance, they are not connected to the processor's memory bus. This means that applications do not have direct access to data and have to go through the operating system [3].

However, recent advances in Persistent Memory (PMEM) [3,8] technologies have introduced a new type of memory that shares characteristics of both main memory and storage, namely high-capacity byte-addressable devices that are slower than main memory but much faster than storage at a fraction of the cost of memory. Intel's Optane Datacenter Persistent Memory Module (DCPMM) was the first PMEM module made commercially available in 2019 [3], providing two operation modes, namely *memory mode* and *app-direct mode*. In *memory mode*, DRAM becomes a hardware managed cache for PMEM, allowing the transparent use of both DRAM and PMEM without requiring modifications to the codebase. In *app-direct* mode, applications can explicitly access DRAM and PMEM, either through a DAX-based file system or exposing PMEM as separate NUMA nodes. DCPMM can provide up to 6 terabytes (TB) of memory on a single machine, however its access performance is substantially worse than DRAM, in particular w.r.t. write operations [11]. This performance gap motivates heterogeneous memory configurations with both DRAM and PMEM, and challenges programmers to develop data models that are capable of leveraging PMEM's larger capacities combined with DRAM's performance.

Our goal is to address such challenges and explore PMEM hardware to enable large and cost-effective NoSQL deployments. More specifically, we are interested in the usage of PMEM only as an extension to DRAM and focus on how to minimize the impact of the slower performance of PMEM devices in such setting trough data placement strategies. These strategies can be divided into **static** and **dynamic**: With a static approach, data objects have predetermined locations (e.g., all keys on DRAM and all values on PMEM); And with a dynamic approach, data objects move from one module to another according to a predefined policy (e.g., most accessed data objects are placed on DRAM). Note that dynamic data placement can be done at the application level and also at the OS level by monitoring page access patterns.

While static approaches are simple to reason about and implement, they are not flexible to application requirements in different scenarios, and hence do not achieve the best possible performance. Dynamic approaches, on the other hand, have the potential to achieve better performance but are complex and can include mechanisms such as periodic profiling [7], which in turn consume more resources. Moreover, dynamic strategies rely on data migration between different kinds of memories, which consumes bandwidth that could be used by normal application operation. This also includes prefetching data objects directly from PMEM to processor-cache, noting that this approach is extremely time sensitive due to cache evictions.

There are, in the literature, proposals that combine the aforementioned data placement strategies [11], however they do not consider the NoSQL domain which is particularly interesting because NoSQL databases benefit from larger capacities and performance is often a key aspect. In this work, we present an analysis on the impact of static data allocation strategies provided by a state-of-the-art NoSQL with support for PMEM operation.

The remainder of this document is organized as follows. In Section 2, we discuss our proposal. Section 3 outlines the work carried out. In Section 4, we present and discuss the obtained results.

2 Proposal

In our evaluation, we analyze four different static allocation strategies implemented on a NoSQL database. Two of such strategies consist in allocating data only to one kind of memory (i.e., either DRAM or PMEM). The other two use both kinds of memory by choosing where to allocate data based on a predefined parameter. More specifically, one of such strategies follows a *threshold* parameter, having all allocations equal or greater to said *threshold* target PMEM and all others DRAM. The other, allocates data to PMEM or DRAM following a *ratio* parameter (e.g., 30% data to DRAM and 70% to PMEM). These strategies do not make any distinction between application data and metadata. For example, system metadata in most cases consists of small objects that are rarely written but also very frequently read as shown in [11], hence should be placed on DRAM and with the abovementioned approaches could end up on PMEM. Moreover, they are not flexible and do not take into consideration the workload which the database is being submitted to. Also, we should expect that configurations of such parameters will behave worse somewhat proportionally to the PMEM usage. For this reason, we expect such strategies to not achieve optimal performance, leaving room for optimization.

As stated before and shown on Section 4, static approaches are simple, but their lack of flexibility makes them unsuitable for applications that can experience different workloads and access patterns such as NoSQL databases. Other, more promising, approaches that have been proposed in the literature [7,11] combine both dynamic and static data allocation policies. Such approaches leverage data migration between DRAM and PMEM (and even CPU cache, as we will discuss further in this Section). However, handling data migration poses some challenges. Namely, since our goal is to optimize performance, stopping the application to migrate data is not a viable option. Hence, data migration has to be carried out in the background, notwithstanding the fact that it consumes resources that could be critical for application execution. Furthermore, choosing the right data to move between memory modules dynamically requires either full knowledge of the application behavior and workload, or some kind of mechanism that monitors application access patterns and triggers the migration accordingly, which in turn also consumes resources that could be required by the application. Such challenges can be addressed with thorough application analysis to find out idle execution points where data migration would not impact application performance, or by lowering the overhead of the migration or profiling mechanisms.

Ideally, all data objects should be allocated to DRAM. However, if that is not possible, frequently accessed objects and short-lived objects should be prioritized. With this in mind, and considering the proposal by Jie Ren et al. in [11] we intend to explore the combination of static and dynamic data placement with processor-cache prefetching. To that end, we plan on breaking up the NoSQL data model. System's metadata (e.g., database hash table structure) and short-lived objects should be placed in DRAM. Database records should be dynamically placed in DRAM and PMEM. In NoSQL databases, the interval between the moment a client request reaches the system and the moment the ditto request is processed can be used to move the requested records from PMEM to DRAM if necessary. Our goal is to leverage this by record migration from PMEM do DRAM or, alternatively, by prefetching records directly to processor-cache. The processor-cache prefetching approach is simple but has some drawbacks, namely it has to be executed right before the record is going to be accessed in order to avoid being evicted due to cache conflicts. Moving data between PMEM and DRAM, on the other hand, is a more challenging and slow process but can have long-term improvements (e.g., when a record is frequently accessed, it is better to move such record to DRAM once instead of perfecting it to cache at every request). In summary, processor-cache prefetch presents a good option when the system is not saturated and the window between the request arrival and processing is short. However, if a subset of records is being frequently accessed and/or the system is saturated making the time window wider, which in turn raises the probability of the record being evicted from cache before it is accessed. Note that it might also be useful to differentiate between read and write requests (i.e., records subject to write requests should have priority in DRAM placement).

With this work, we propose to perform a thorough analysis of a NoSQL database to further help identify what objects should be statically placed on DRAM and what object should be subject through data migration either by fetching them from PMEM to DRAM or CPU cache under what circumstances. As future work, we plan on developing a mechanism based on this Section's proposal. for the NoSQL domain that optimizes memory usage performance in heterogeneous PMEM systems.

3 Approach

In order to achieve the goals stated in Section 2 we have to choose a NoSQL database to evaluate the performance impact of the data allocation policies. TieredMemDB[2] is a fork of Redis that supports PMEM operation, having a number of static data allocation policies already implemented that can be selected and parameterized through a configuration file, which is convenient in our use case. It is fully compatible with Redis and supports all its structures. TieredMemDB leverages the `Memkind` [6] library as an extensible heap manager built on top of `jemalloc`, to provide an implementation for the static data placement strategies, as described on Section 2. More specifically, there are four available and configurable memory allocation policies withing TieredMemDB that can be selected by setting the `memory-alloc-policy` parameter to any of the following options: `only-dram`, `only-pmem`, `threshold`, and `ratio`.

The `threshold` policy uses both DRAM and PMEM according to the value of the `static-threshold` parameter. When this policy is selected, the database will check the value of the `static-threshold` at each allocation and: If the allocation size is equal or greater than its value on bytes, it goes to PMEM. Else it goes DRAM. The `ratio` policy uses both DRAM and PMEM according to the `dram-pmem-ratio` parameter. With this policy, the database allocates part of the data in DRAM and part in PMEM based on a value of the `dynamic-threshold`. At runtime, the database periodically monitors DRAM and PMEM usage and changes the value of `dynamic-threshold` to achieve the configured ratio. For example, configuring the database with `dram-pmem-ratio 1 3` would approximately make 25% of the allocations to DRAM and 75% to PMEM. Additionally, there is also a parameter that specifies whether the hash table is placed on DRAM or PMEM (e.g., `hashtable-on-dram yes`).

In this work, we submit these policies to several workloads with the goal of comparing them and measuring how much they affect system performance. We use the *Yahoo! Cloud Serving Benchmark* (YCSB) framework and its standard workloads, measuring the latency and throughput for each combination of workload and policy. We also generated two different new workloads that execute 100% of each operation respectively: `insert` and `read`. Measuring the memory usage was carried out by periodically running the *numastat* system utility and extracting its output. The results of the tests are presented and discussed in the next Section.

4 Evaluation

4.1 Hardware Setup

The experiments were executed in one machine made available by Grid 5000 [4]. The machine used is located at the Grenoble site and is part of the troll cluster with a Intel Xeon Gold 5218 (Cascade Lake-SP, 2.30GHz, 2 CPUs/node, 16 cores/CPU), and 384 GiB + 1.5 TiB PMEM.

TieredMemDB accesses PMEM as separate memory-only NUMA node(s). We have configured the database to use one of the PMEM node with approximately 756 GB.

4.2 Workloads

For our experiments, we use the YCSB client and respective standard workloads as described in Table 1. Workloads A through F are the standard workloads of YCSB, and were run so that we could compare the results with evaluations in other works and assess whether the behavior for each workload in TieredMemDB

Work-load	Descrip-tion	Operation Ratio	Request Distributions	Application Example
A	Update heavy	read/update ratio: 50/50	zipfian	Session store recording recent actions.
B	Read mostly	read/update ratio: 95/5	zipfian	Photo tagging; Add a tag is an update, but most operations are to read tags.
C	Read only	read/update ratio: 100/0	zipfian	User profile cache, where profiles are constructed elsewhere.
D	Read latest	read/update/insert ratio: 95/0/5	latest	User status updates; people want to read the latest.
E	Short ranges	scan/insert ratio: 95/5	zipfian	Threaded conversations, where each scan is for the post in a given thread (assumed to be clustered by thread id).
F	Read-modify-write	read/read-modify-write ratio: 50/50	zipfian	User database, where user records are read and modified by the user or to record user activity.

Table 1: Standard YCSB workloads.

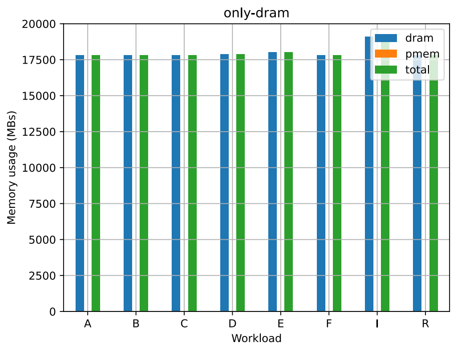
was similar to Redis. Additionally, we have generated two separate workloads: **workload I** executes 100% inserts and **workload R** executes 100% reads. The motivation behind these workloads was that they allow us to observe the difference between read and write operations and how they influence performance.

For each workload, we executed 1 million operation over a deployment with 10 million records with 1KB (i.e., 10 fields, 100 bytes each, plus key). In total 10 GB of records.

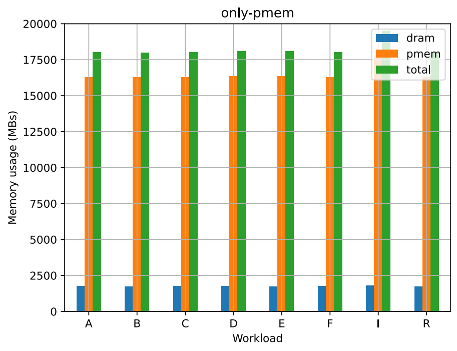
4.3 Results & Discussion

We present the average results of 4 runs comparing the maximum memory usage and overall performance of all the previously mentioned memory allocation policies. For the standard YCSB workloads, TieredMemDB follows the patterns of other Redis benchmarks [1,5,9,10] having workloads A, B, C, D and F with similar performance and workload E substantially worse. Also, note that workload E consumes more memory as the server allocates data resulting from the scan before sending it to the client.

The memory usage of `only-dram` and `only-pmem` can be observed in Figure 1. Note that in Figure 1b the `only-pmem` policy still uses a small fraction of DRAM



(a) *only-dram.*



(b) *only-pmem.*

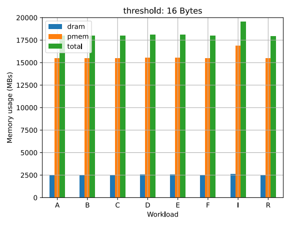
Fig. 1: *only-dram* and *only-pmem* memory usage.

due to the hash table placement configuration. In the experiments described in this Section, we configured the hash table to be allocated to DRAM.

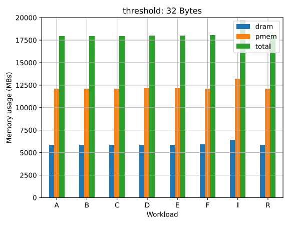
Threshold The `threshold` policy was configured with the following values (in bytes) for `static-threshold`: 16, 32, 64, 104, 105. Our goal was to gradually increase the `threshold` until the approximate value of the max allocation size used by the database (104 bytes).

We show in Figure 2 that there is noticeable difference in the memory usage pattern between 16 and 32 bytes: With a 32 bytes threshold, the database uses about 1.7× more DRAM. The 32 an 64 byte thresholds present similar memory usage values, and we also show that an 104 byte threshold uses roughly 1.1× more DRAM than the 64 and 32 byte thresholds. Finally, the 105 byte threshold places all allocations in DRAM, representing a 2.4× increase of DRAM usage when compared to the 104 byte threshold. We executed several tests with different threshold values from 64 bytes up to 104 bytes, and the difference in memory utilization for such was negligible. The records represent the majority of the database data (i.e., 10 GB) and they are split into fields of 100 bytes each, explaining the growth in memory usage between 104 bytes (Figure 2d) to 105 bytes (Figure 2d) which is about 10GB and represents the volume of the records. Note that we would expect the growth to happen at the 100 bytes threshold but in practice the allocations regarding record fields take 4 extra bytes.

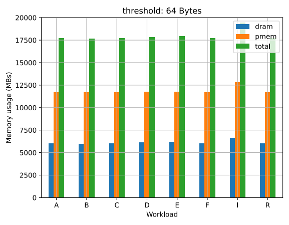
Performance wise, we separate the results for the standard workloads (Figure 3a and Figure 3b) and for our workloads (Figure 3c and Figure 3d). The figures show that the inverse relation between the latency and throughput, as expected. In the standard workloads there is large difference in performance from the workload E to the other workloads, due to the nature of scan operations. In the bottom half of the figure, we can clearly see the difference in performance between the `read` and `insert` operations: in average, workload R (`read`) has about 2.5× more throughput than workload I (`insert`), which in turn has about 2× higher latency. In workload I, using DRAM achieves 2× more through-



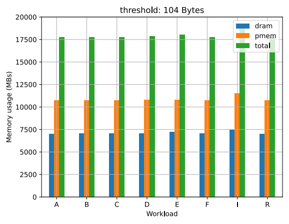
(a) 16 Bytes.



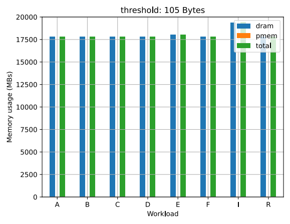
(b) 32 Bytes.



(c) 64 Bytes.



(d) 104 Bytes.



(e) 105 Bytes.

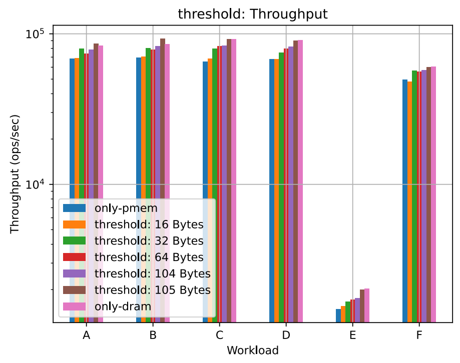
Fig. 2: *threshold memory usage.*

put and achieves 2× shorter latency than using PMEM. On the other hand, with workload R, using DRAM produces 1.3× more throughput and achieves 1.5× shorter latency than using PMEM. In summary, we can see that write operation produce a higher impact on performance than read operations and the more PMEM is used, the slower the database becomes.

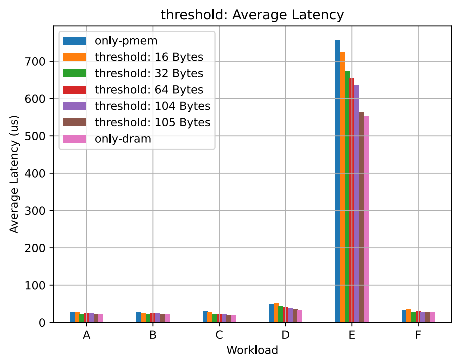
Ratio The `ratio` policy was configured with the following values (in percentage of DRAM usage): 17%, 25%, 33%, 50%, 67%, 75% and 83%. Our goal is to validate that the implementation matches the specification and to gradually increase the percentage of DRAM used while measuring the performance impact of such increases.

In Figure 4, we can see that the results approximately respect the ratio configuration validating TieredMemDB's implementation of the ratio policy.

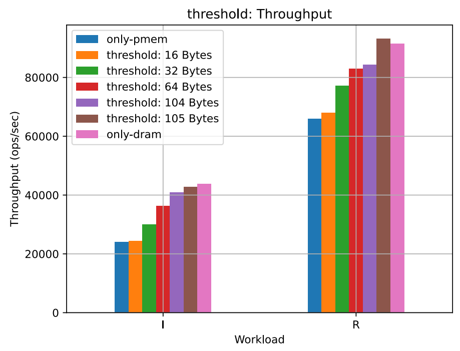
We can also see that the performance results in Figure 5 are somewhat similar to those of the `threshold` policy in Figure 3. The only difference is that with workload R (Figures 5c and 5d), increasing the portion of DRAM used does not improve performance in some cases (e.g., from 50% to 67%). With the `ratio` policy implementation as described on Section 3, it is possible that higher percentage of frequently accessed objects could be allocated to DRAM with 50% than with 67%, because the ratio policy disregards access frequency, the dynamic threshold value is changed periodically, and it is not easy to predict whether with different parameters the same data object will be allocated to different kinds of memory.



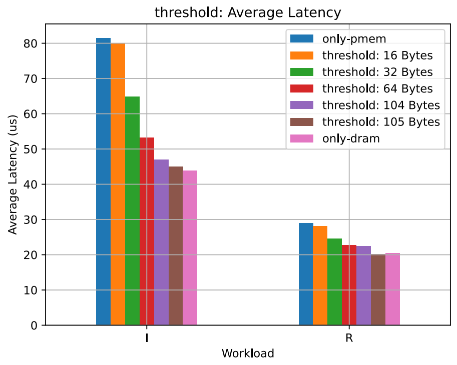
(a) Standard workloads *threshold* throughput.



(b) Standard workloads *threshold* latency.



(c) Operation workloads *threshold* throughput.



(d) Operation workloads *threshold* latency.

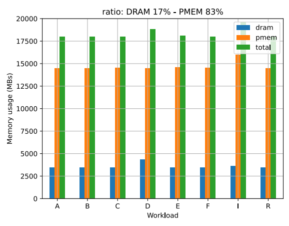
Fig. 3: Insert/Read *threshold* performance.

4.4 Summary

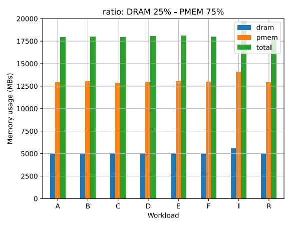
In this Section, we have shown that PMEM slow performance impacts the overall database performance negatively. However we also showed that performance degradation is not linearly proportional to PMEM usage, namely when requests comprise insert operations. For example, we can see in Figure 2 that the difference in memory usage between 32 bytes and 64 bytes is negligible, but in Figures 3c and 3d there is a noticeable difference in performance. This means that we can have the same PMEM usage and still achieve better performance depending on what we choose to allocate to each kind of memory. In summary, more flexible approaches that take into account the application access patterns can possibly achieve better performance than the static approaches

5 Related Work

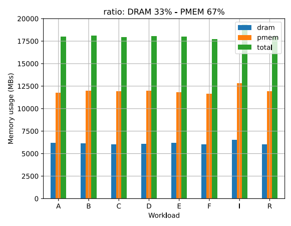
In this work, we have evaluated static data placement on the NoSQL domain and proposed to explore a new strategy for data allocation that combines static and



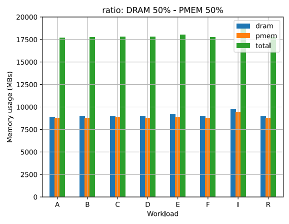
(a) 17% - 83%.



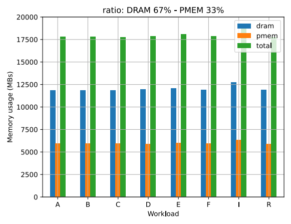
(b) 25% - 75%.



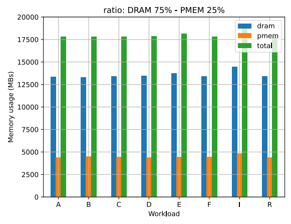
(c) 33% - 67%.



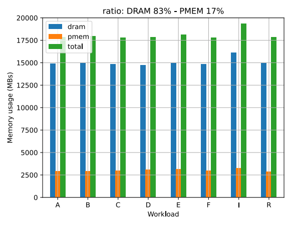
(d) 50% - 50%.



(e) 67% - 33%.



(f) 75% - 25%.

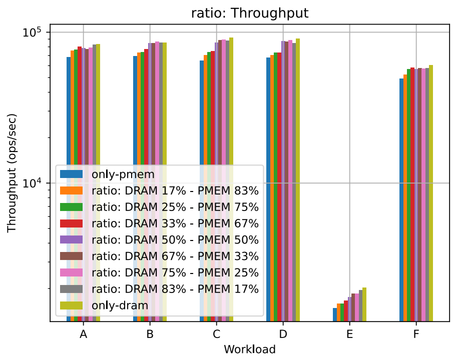


(g) 83% - 17%.

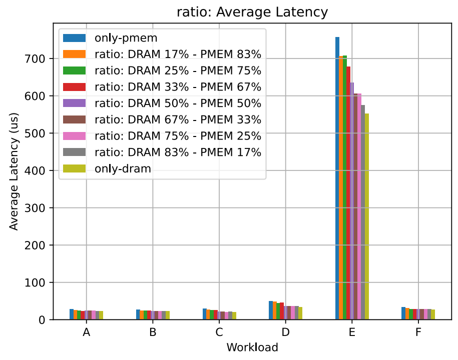
Fig. 4: *ratio* memory usage.

dynamic data placement with processor-cache prefetching. There is a particular work which inspired our proposal. That work is WarpX-PM by Jie Ren en. al [11].

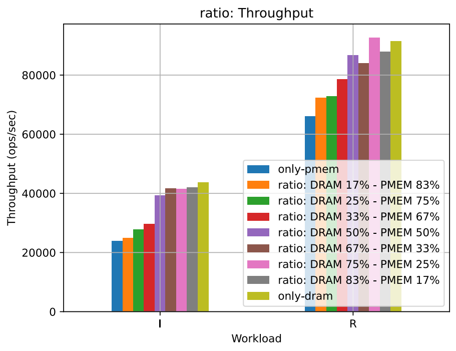
WarpX-PM WarpX is an advanced plasma simulation code which is mission-critical and targets future exascale systems. After a thorough analysis of WarpX, Ren et al. [11] propose the WarpX-PM runtime system, extended from WarpX to manage data placement between DRAM and PMEM automatically. In WarpX-PM, DRAM is partitioned into four spaces to store data objects with different characteristics and access patterns: Short-lived objects that are frequently allocated and freed, such as local variables in functions, go to the *temporary space*; The *metadata space* stores long-lived objects that are not frequently updated but are frequently accessed; the *migration space* is used to prefetch data from PMEM to DRAM before they are required by the computation; Finally, the *free space* is used to store the maximum possible field data. In these four spaces, WarpX-PM combines static and dynamic strategies, and processor-cache prefetch mechanism for data placement. The **migration space** is used for dynamic data placement and the remainder three spaces are used for static data placement. The authors



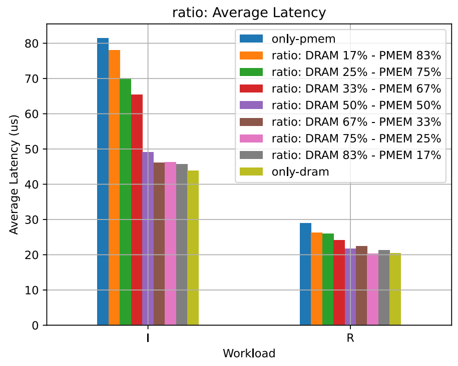
(a) Standard workloads *ratio* throughput.



(b) Standard workloads *ratio* latency.



(c) Operation workloads *ratio* throughput.



(d) Operation workloads *ratio* latency.

Fig. 5: *ratio* performance.

carefully study what size these spaces should have and also propose a performance model for the processor-cache prefetch and migration mechanisms.

WarpX-PM improved WarpX's execution on Optane-only by 66.4% and outperformed DRAM-cached, the NUMA first-touch policy, and a state-of-the-art HM solution by 38.8%, 45.1% and 83.3%, respectively.

6 Conclusions

The emerging large-capacity of PMEM enables large and cost-effective NoSQL deployments. However, leveraging PMEM for real world NoSQL deployments remains to be investigated. In this paper, we present an analysis of several static allocation policies provided by a NoSQL database called TieredMemDB [2] and their impact on system performance. Our results show that these approaches are not flexible, as they do not take into consideration the application domain and do not adapt to the workload.

With this in mind and as future work, we plan on exploring a new mechanism for memory allocation in Redis that combines both static and dynamic data placement with processor-cache prefetching, inspired on the work done by Ren et al. [11].

References

1. Benchmark (ycsb) numbers for redis, mongodb, couchbase2, yugabyte and bangdb, `http://highscalability.com/blog/2021/2/17/benchmark-ycsb-numbers-for-redis-mongodb-couchbase2-yugabyte.html`
2. TieredMemDB, `https://tieredmemdb.github.io/TieredMemDB/`
3. Baldassin, A., Barreto, J., Castro, D., Romano, P.: Persistent memory: A survey of programming support and implementations. ACM Computing Surveys (CSUR) **54**(7), 1–37 (2021)
4. Balouek, D., Carpen Amarie, A., Charrier, G., Desprez, F., Jeannot, E., Jeanvoine, E., Lèbre, A., Margery, D., Niclausse, N., Nussbaum, L., Richard, O., Pérez, C., Quesnel, F., Rohr, C., Sarzyniec, L.: Adding virtualization capabilities to the Grid'5000 testbed. In: Ivanov, I.I., van Sinderen, M., Leymann, F., Shan, T. (eds.) Cloud Computing and Services Science, Communications in Computer and Information Science, vol. 367, pp. 3–20. Springer International Publishing (2013). https://doi.org/10.1007/978-3-319-04519-1_1
5. Ben Seghier, N., Kazar, O.: Comparing NoSQL databases with ycsb standard benchmark. University of Eloued (2022)
6. Cantalupo, C., Venkatesan, V., Hammond, J., Czurlyo, K., Hammond, S.D.: memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2015)
7. Chen, Y., Peng, I.B., Peng, Z., Liu, X., Ren, B.: Atmem: Adaptive data placement in graph applications on heterogeneous memories. In: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization. pp. 293–304 (2020)
8. Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y.J., Wang, Z., Xu, Y., Dulloor, S.R., et al.: Basic performance measurements of the intel optane dc persistent memory module. arXiv preprint arXiv:1903.05714 (2019)
9. Lehmann, R.: Redis in the yahoo! cloud serving benchmark (2011)
10. Osemwegie, O., Okokpujie, K., Nkordeh, N., Ndujiuba, C., Samuel, J., Stanley, U.: Performance benchmarking of key-value store NoSQL databases. International Journal of Electrical and Computer Engineering **8**(6), 5333 (2018)
11. Ren, J., Luo, J., Peng, I., Wu, K., Li, D.: Optimizing large-scale plasma simulations on persistent memory-based heterogeneous memory with effective data placement across memory hierarchy. In: Proc. ACM Int. Conf. on Supercomputing (2021)