An Initial Performance Analysis of Persistent Memory Allocators

Catalina Munoz[1], **Bruno Honorio**[1],
Lucas Bastelli[1], **Emilio Francesquini**[2], **Alexandro Baldassin**[1]

[1]Universidade Estadual Paulista "Júlio de Mesquita Filho" (UNESP)
Rio Claro – SP – Brazil

[2]Universidade Federal do ABC (UFABC)
SP – Brazil

Abstract. *Persistent Memory (PM) has recently resurfaced with the advent of new technologies that make it a competitive option in terms of performance compared to traditional storage, adding the benefit of durability. Because it is byte-addressable, the use of PM is similar to that of volatile RAM (DRAM). However, there are some differences related to data consistency and read/write latency, which makes it necessary to have special memory allocators. This paper presents an initial performance analysis of the impact of two persistent memory allocators, PMDK and Ralloc, using three typical data structures, namely, i) Linked List, ii) Skip List, and iii) Hash Map. Although the study is in its initial phase, we could observe the impact of PM allocator in performance.*

1. Introduction

Persistent memory (PM) is a byte-addressable memory technology, similar to typical volatile RAM (DRAM) [Baldassin et al. 2021]. Its use has recently become popular since these memories share the same data bus as the DRAM. Moreover, Intel have recently commercialized persistent devices (Optane DC) [Tyson 2019] and the CXL (Compute Express Link) interconnect [Jung 2022] has added support for PM. Even though it is non-volatile storage, which is typically slower, current PM performs considerably better than storage devices such as solid-state drives, due to lower read/write latencies. For instance, PM read latency can be as low as 350 nanoseconds, compared to 10,000 nanoseconds for an SSD [Izraelevitz 2019]. This characteristic makes PM a viable option for high-performance computing that requires data durability with a relatively low cost in terms of performance.

Despite the benefits of persistent memory, optimally using it is a non-trivial task. Unlike DRAM, the storage space is much higher. However, the durability characteristic makes it necessary to guarantee data consistency in case of a power loss. Therefore, programmers are required to guarantee that the stored data can be adequately recovered and accessed following a system crash. Aside from typical store instructions, cache flushes must be regularly executed to ensure that persistent data is not lost during a sudden system failure. A persistent memory allocator must specially consider how allocated data will be recovered in case of system crashes (e.g., power failure). Several allocators have been proposed to date [Scargall 2020] [Bhandari et al. 2016] [Cai and Wen 2020], with the main differences in memory layout, metadata used for data recovery, and handling of memory deallocation and release (e.g., using garbage collectors). However, it is still

not clear how the different design choices affect the overall performance. In this short paper, we show initial performance results using two well-known persistent allocators: 1) PMDK (Persistent Memory Development Kit), an allocator implemented by Intel and part of its development kit for programming with PM [Scargall 2020]; and 2) Ralloc, a state-of-the-art allocator [Cai and Wen 2020]. In this initial report we make use of three simple data structures: a linked list, a skip list, and a hash set. We show how the allocator used plays an important role in performance, particularly during deallocation, with the number of operations per second going between 4x and 20x higher with Ralloc compared to PMDK.

2. Persistent Memory Allocators

Dynamic memory allocation is among the most expensive and common operations in software development. Studies conducted with a group of heap-intensive applications have shown that, on average, 30% of the total execution time is spent on dynamic memory management [Tiwari et al. 2010]. Compared to traditional ones, PM allocators have further requirements, such as the need for data consistency (allocator metadata needs to be persistent) and different design tradeoffs. For instance, some designs are optimized to reduce write endurance of PM [Yu et al. 2015]. In this initial study, we concentrate on two important persistent allocators: PMDK [Scargall 2020] and Ralloc [Cai and Wen 2020]. In the following we provide a short introduction, highlighting the main characteristics of each allocator.

PMDK: PMDK is a set of libraries developed by Intel [Scargall 2020] for C and C++ that enable persistent memory usage, including allocation and deallocation. In order to use the persistent memory space with PMDK, the user must first allocate a root pointer, from which the applications' memory region will be reachable. PMDK separates the entire memory region into zones of specific size, and each zone into chunks of variable size. For allocation, a blocks' size is rounded up to match the next size of a free list. The malloc method provided by PMDK finds a free block and attaches it to previously allocated blocks. This operation occurs atomically to avoid possible memory leaks during a system crash. Additionally, redo and undo logs are kept for allocation atomicity and transactional snapshotting of a memory region, respectively. For deallocation, the free method gets the pointer to the block and puts it back into the free list. To manage the free address space of the application, PMDK maintains in the DRAM a vector of pointers to persistent memory blocks of specified size.

Ralloc: Ralloc is a persistent memory allocator developed by Wentao et al. [Cai and Wen 2020], which reorganizes the typical memory layout to enable recoverability. Ralloc separates memory in superblocks of the same size, mapped in a single region to keep the applications' data. To guarantee consistency and recoverability, two more regions are created in the address space to maintain metadata of superblocks size, state, and allocated data types. In the main memory of the application, superblocks are divided in an array of subblocks, where a free/allocated identifier is kept, and a free subblock points to the next free subblock on the list. Each superblock holds a state that indicates if such superblock is free, fully or partially allocated. Inside the actual memory region, objects are arranged according to their sizes by keeping blocks of equivalent sizes inside the same superblock. Therefore, during the allocation of an object of a specific size, a superblock for such a size class is selected, and a pointer for the next free block inside the superblock is

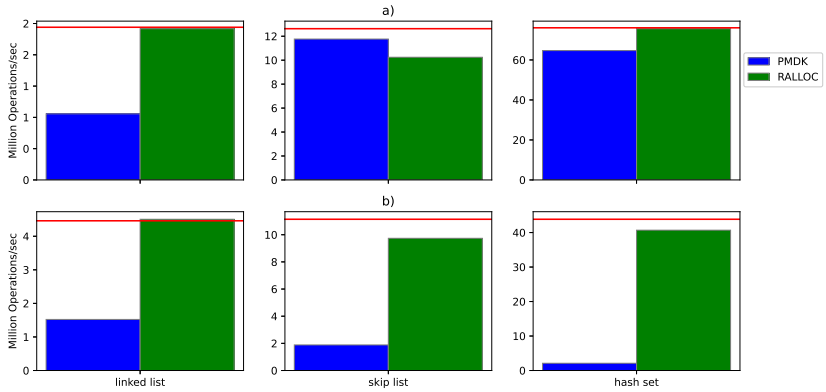

Figure 1. Add (a) and Update (b) operations per second for linked list, skip list, and hashset.

returned. For deallocation, a simple change from "full" to "empty" in the state descriptor of the block is required. The memory reclaim process consists of pushing the block from a "full" or "partial" list to a "partial" or "free list", according to the state transition of the block. Superblock reclamation is delayed during a malloc operation when no superblock is available for allocation. This causes the deallocation process to be much faster.

3. Experimental Setting and Performance Results

To carry out the performance analysis, we use three data structures on which `delete` and `add` operations require allocation and deallocation, and a `search` operation only reads from persistent memory. The memory access patterns depend on each data structure used as follows. A **Linked List** is a linear structure in which each element points to the next and/or the previous one. For this experiment, we use a simple linked list; in a **Skip List**, similar to linked lists, each of the elements points to the next one. However, these lists tend to be more efficient search operations since the list is arranged in levels that allow ordering the data; a **Hash Set** is a structure that facilitates the search for non-repeating elements by arranging them in semi-ordered groups (called buckets) using a hashing mechanism.

The used data structures are allocated exclusively in PM. Other metadata (volatile) used during execution is allocated in DRAM. Applications were executed in a machine powered by an Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz processor with 196GB of RAM, and 128GB of Intel Optane DC Persistent Memory. We compare the PMDK and Ralloc allocators and used LRmalloc [Leite and Rocha 2019], a DRAM allocator, as a performance reference (red line in the figures). Performance results are presented in terms of the number of operations per second executed by each application for the respective data structures. All sets are initialized with 256 randomly generated elements of integer type. Each application is executed 10 times with a 10 seconds duration each time. Results of each run are averaged to obtain the number of ops per second (ops/sec).

The top row of Figure 1 shows the number of `add` operations per second performed for each set. Each element is randomly generated and added if the value does not previously exist in the set. As can be seen in the figure, the difference between Ralloc and PMDK is large, particularly for the linked list, where 128% more set add operations

are performed per second using Ralloc than PMDK. For the hash set, 16% more operations are executed using Ralloc, compared to PMDK. However, a particular case occurs with the skip list, wherein PMDK outperforms Ralloc. PMDK performs 16% more `add` operations than Ralloc. This is possibly a consequence of the memory layout in PMDK, which benefits skip list lookup over Ralloc.

The bottom part of Figure 1 shows the number of `update` operations performed per second. This operation is composed of additions and removals of an element in the set (with equal probability). A wider performance difference can be seen between PMDK and Ralloc, with Ralloc performing better for all the studied sets. In the case of Ralloc, the garbage collector allows delayed memory recovery. As Rallocs' implementation is based on LRmalloc, the difference in performance is expected to be solely given by the differences in latency between PM and DRAM. In most of the tests, Ralloc behaves relatively similarly to LRmalloc, making it evident that PM is competitive for high-performance computing.

4. Conclusions

Our initial results show that persistent memory allocators play an important part in performance. Although in general Ralloc displayed much better results, we also spotted a scenario in which the PMDK provided better results. We are now checking more precisely the causes for the performance gain of PMDK with the skip list (`add` operations) and conducting an evaluation on more representative benchmarks (real cases), as well as adding comparisons with other available PM allocators.

5. Acknowledgments

This research has been supported by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), grant numbers: 2018/15519-5, 2023/04969-8, 2023/04971-2.

References

Baldassin, A., Barreto, J., Castro, D., and Romano, P. (2021). Persistent memory: A survey of programming support and implementations. *ACM Comput. Surv.*, 54(7).

Bhandari, K., Chakrabarti, D. R., and Boehm, H.-J. (2016). Makalu: Fast recoverable allocation of non-volatile memory. *SIGPLAN Not.*, 51(10):677–694.

Cai, W. and Wen, e. (2020). Understanding and optimizing persistent memory allocation. In *Proc. of the ISMM*, page 60–73.

Izraelevitz, J. et al. (2019). Basic performance measurements of the Intel Optane DC persistent memory module. *CoRR*, abs/1903.05714.

Jung, M. (2022). Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proc. of the 14th ACM Hotstorage*, page 45–51.

Leite, R. and Rocha, R. (2019). Lrmalloc: A modern and competitive lock-free dynamic memory allocator. In *Proc. of the 13th VECPAR*, pages 230–243.

Scargall, S. (2020). *Programming Persistent Memory - A Comprehensive Guide for Developers*. Apress, 1st edition.

Tiwari, D., Lee, S., Tuck, J., and Solihin, D. (2010). MMT:exploiting fine-grained parallelism in dynamic memory management. In *Proc. of IPDPS*, pages 1–12.

Tyson, M. (2019). Intel Optane DC PMEM launched. Retrieved from hexus.net/tech/news/storage/129143-intel-optane-dc-persistent-memory-launched/.

Yu, S., Xiao et al. (2015). WAlloc: An efficient wear-aware allocator for non-volatile main memory. In *Proc. of the 2015 IEEE IPCCC*, pages 1–8.