PMShifter: Enabling Persistent Memory Fluidness in Linux

Theodore Michailidis
University of California, San Diego
tmichail@eng.ucsd.edu

Steven Swanson
University of California, San Diego
swanson@eng.ucsd.edu

Jishen Zhao
University of California, San Diego
jzhao@eng.ucsd.edu

Abstract

Intel recently released the first commercial server-grade persistent memory (PM), Optane DC Persistent Memory. PM bridges the long-standing gap between volatile and non-volatile storage devices, as its byte-addressability and non-volatility allow it to be used partly as storage and memory simultaneously. One downside of conventional memory management design is that such distinction needs to be made at boot time or explicitly set through a user-level program. This limits the flexibility offered by the device, as data persistence requirements vary over time.

To address this issue, we propose **PMShifter**, which transparently and dynamically configures PM between memory and storage. To enable this flexible configuration, PMShifter also targets inefficient memory compaction, page migration and PM-oblivious NUMA policies. We evaluate PMShifter on micro-benchmarks and real-life workloads, showing up to 12.3× improved page migration throughput, faster retrieval of up to 12.77× more large physically contiguous memory segments during compaction, while running Redis displayed 64% reduced tail latency and 2.09× improved throughput.

CCS Concepts

• **Information systems → Phase change memory**; • **Software and its engineering → Memory management**.

Keywords

Persistent Memory, Non-volatile Memory, Operating Systems, Memory Management

ACM Reference Format:
Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2022. PMShifter: Enabling Persistent Memory Fluidness in Linux. In *13th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '22), August 23–24, 2022, Virtual Event, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3546591.3547523

1 Introduction

Intel recently released the first commercially available server-grade persistent memory, named Optane DC Persistent Memory [7], allowing researchers to investigate its practical design challenges. Studies have shown that Optane is up to 8× larger compared to DRAM, while having 2× higher latency and up to 7× lower bandwidth [28].

PM can be subdivided to main memory and direct access (DAX) storage chunks, maximizing its utilization by different type of workloads. Leveraging this feature is essential in data centers, where applications vary substantially [9] between DAX-intensive and memory-bound applications. Since memory accounts for approximately 40% of modern server costs [1], it is crucial to not only leverage any idle PM DAX segments as memory, but also to actualize it in a performant manner.

Previous work on hybrid memory systems has focused mostly on the efficient placement of pages in the right device [3, 4, 11, 14, 17, 19, 23, 27]. However, there is a lack of work on ***how*** and ***when*** to partition a dual-mode device like PM, and the associated costs. Since memory requirements change over time, there is no ideal configuration for PM. When the system is under memory pressure, leveraging on the fly idle DAX segments as memory could substantially reduce operational costs. We introduce the term ***shift*** to describe this online transition of a DAX PM segment to memory and vice versa.

Efficiently addressing PM partitioning is burdensome, due to substantial design challenges. **First**, the distinction has to be made either at boot time [8] or by explicitly using user-level programs during runtime[10]; both are impracticable in data centers, where interactions and restarts should be minimized.

Second, conventional memory management lacks support for PM as part of the main memory. These issues are related mainly to page migration, memory fragmentation and NUMA page placement, all of which are interconnected. Fast page migration and memory fragmentation have a cyclic dependency, since randomly migrating a page to another device can exacerbate memory fragmentation, while a fragmented memory can reduce page migration throughput. In addition, page migration goes through the memory allocation critical path, affecting both the system and page migration performance. Last, as recent work has shown [13], modern OS lack support for accurate NUMA page placement in hybrid memory systems. They are built upon the erroneous assumption that memories' attributes, such as latency and bandwidth, are similar.

We introduce **PMShifter**, the first practical work that enables online, transparent PM shifting. When under memory pressure, PMShifter utilizes unused DAX segments to extend the main memory, allowing to fully utilize the available space. We identify and propose fixes for the associated pathologies in the memory subsystem, optimizing both PM shifting and the overall performance of hybrid memory systems.

PMShifter consists of 2 main components, the **Shifter** and the **Migrator**, while introducing minimal modifications on NUMA PM page handling. The Shifter is responsible for memory pressure monitoring and shifting PM segments, employing an adjusted version of the Exponential Moving Average [26]. The Migrator enables faster page migration through its combined compaction and page migration scheme, while avoiding the memory allocator to alleviate increasing the pressure in the central allocation point.

At the same time, PMShifter makes lightweight changes to the existing DRAM memory compaction mechanism to address both the fragmentation and high compaction failure rates issues. PMShifter uses a slightly modified version for PM, as a collaborative method to allow fast, non-intrusive dynamic PM shifting from memory to DAX. Finally, PMShifter introduces simple changes to the NUMA logic to avoid the related performance issues.

2 Background

This section focuses on the Linux memory management background related to hybrid memory systems.

2.1 Memory Hotplug in Linux

Memory hotplug is a feature in Linux that allows increasing or decreasing the total amount of physical memory the system sees while it is running. Adding and removing pages is called *online* and *offline*, respectively, and offlined pages are invisible to the system, thus cannot be used.

In order to offline an allocated page, a new page has to be allocated and then populated with the contents of the old page. Then, the virtual-to-physical mapping has to be updated and the corresponding TLB entries invalidated. The last step is to update the page's metadata to reflect the offline status. This process is extremely costly, stalling the system and workloads that use offlined pages. Contrarily, offlining a free page involves only updating the metadata.

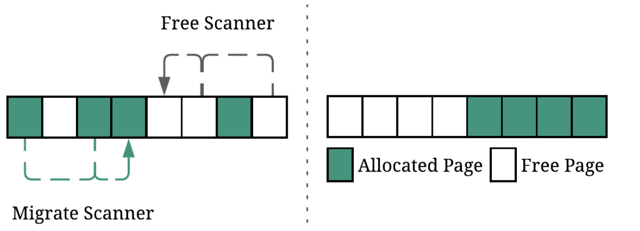

Figure 1: Compaction routine and target memory state

2.2 Memory Fragmentation

Memory fragmentation is considered a major memory management issue[24], with our measurements showing that a fragmented memory can increase allocation latency by up to 2.6×. To alleviate this problem Linux uses a memory compactor to defragment the memory. The compactor uses two scanners, termed *migrate* and *free*. The former scans for allocated pages from the left-most end of the memory (*migrate area*), while the latter browses for free pages, starting from the right-end (*free area*). The process of compaction (Figure 1) is the following: **(i)** The *migrate scanner* looks for a suitable pageblock (2 MB block) to migrate pages from. **(ii)** The *free scanner* tries to find one or more pageblocks that can accommodate the pages to be migrated from the previous step. **(iii)** The pageblocks' contents from step (i) are migrated to pageblocks from step (ii).

This process continues until the two scanners converge or the compactor determines that the cost of compaction will exceed a certain limit. In addition, during the first 2 steps, the allocated and free pages are held in two lists, termed *migrate list* and *free list*, respectively. While pages are in these lists, they are considered invisible to the rest of the system and cannot be utilized. Linux v5.9 introduced periodic compaction, as an effort to prevent extreme memory fragmentation.

2.3 Hot/cold pages in the kernel

When the demands exceed the memory capacity, the kernel resorts to swapping out cold pages. Linux identifies cold pages by maintaining two lists termed *active* and *inactive*, that hold hot and cold pages, respectively.

2.4 NUMA Policies

In a Non-Uniform Memory Access (NUMA) system, each node consists of a CPU with one or more memory nodes. The CPUs can access these local memory nodes faster than the remote nodes. We use the terms *NUMA memory node* and *memory node* interchangeably in the rest of the paper.

Allocation and Page Migration The OS sorts the NUMA memory nodes based on the NUMA topology, and uses that ordering during allocation or page migration to choose the closest target.

Automatic NUMA Balancing. The goal is to minimize or eliminate inter-NUMA accesses from processes. To accomplish that the kernel periodically tries to bring pages within the same node the processes are running to. This feature is known as *automatic NUMA balancing* [6],

3 Motivation

This section contains the related shortcomings we identified in Linux memory management. We show how compaction in Linux fails to address the fragmentation issue and limits its performance. In addition, we demonstrate how NUMA allocation and migration policies lead to easily-avoided performance degradation in hybrid DRAM/PM systems. We exhibit how page migration, a core function in a hybrid memory system, increases the pressure on the centralized allocation path and leads to excessive memory fragmentation.

3.1 PM wastage

Statically configuring PM leads to suboptimal use. Under memory pressure, using unutilized PM DAX regions can alleviate the pressure and eliminate the need for swapping. However, simply supporting dynamic shifting is not enough: we need to minimize shifting frequency, shift as early as possible, and avoid interfering with workloads' execution.

3.2 Memory Compaction Pathologies

Prematurely ending the compaction process is a significant source of overhead. The key reasons for that are: **(i)** The free scanner skips 2 MB and 4 MB pageblocks, to avoid breaking the physical contiguity in these *free area* pageblocks. This limits the space where the compacted pages can be moved, and also creates a mix of allocated and free pages in the *free area*, leading to a never-ending cycle of fragmenting and compacting the memory. **(ii)** The free scanner skips pageblocks that cannot accommodate entirely the pages to be migrated from the first step of compaction. **(iii)** Pages that are unavailable (e.g. pinned pages) during a compaction run, are excluded by both scanners in future subsequent compaction runs. Last, if the migrate scanner encounters an unmovable page in a pageblock, it has to release all previously collected pages from the *migration list*. This leads to wasted work and stalls workloads, since the pages in the list cannot be accessed throughout the entire process.

3.3 Inefficient Page Migration

We identify two main sources of inefficiency in the current page migration mechanism: **First**, since the allocator is one of the most crucial and centralized points in the kernel, frequent page migrations can slow down both the system and workloads. **Second**, the allocator always reserves the leftmost regions that fit the memory needs. This adds overhead to the memory compactor that will move them again to the rightmost side of the memory. Thus, the number of actual migrations is doubled, doubling TLB invalidations and other associated costs.

3.4 Challenges in NUMA policies

NUMA autobalancing is associated with the following major performance drawbacks: **First**, it unmaps pages, but cannot guarantee that page migrations will occur. This leads to unnecessary overhead, primarily due to page faults caused by the unmapping. **Second**, there is no intuition about ***which*** pages to migrate. Ideally, we would want to move hot pages to local DRAMs.
In addition, NUMA ordering and allocation policies in Linux fail to recognize the different performance attributes between DRAM and PM, leading to severe performance degradation. When Linux tries to migrate a page from a PM node, the ordering policy will prioritize a remote PM over the remote DRAM under the same NUMA node, leading to significant performance degradation.

4 PMShifter: NUMA-aware Dynamic Persistent Memory

PMShifter's goal is to fully leverage the available PM capacity by dynamically configuring the PM memory and storage segments, according to existing memory pressure. We tackle the aforementioned compaction pathologies by developing two different compactors (one for each memory device) as we show that the priorities in the two devices are different. PMShifter, combines DRAM compaction and PM page migration, as an effort to accelerate both operations, avoid the allocation critical path, and mitigate the fragmentation problem. In addition, we use a coordinated PM compaction-shifting scheme to reduce the cost of shifting. In terms of NUMA management, PMShifter adapts the NUMA memory ordering that is followed during page migration and allocation, to limit the performance degradation caused by erroneous NUMA page placements.

4.1 Migrator

The Migrator uses a lightweight holistic compaction-migration mechanism to combat fragmentation and boost both operations' performance. It accelerates page migration throughput and increases total physical contiguity.

4.1.1 Compaction. In an effort to maintain compatibility, we build the Migrator on top of the 3-step Linux compaction process, while introducting the following improvements: **(i) We increase the pageblock size from 2 MB to 4 MB**. This is the biggest-sized segments that the Linux memory allocator keeps track of [5]; aiming to keep the biggest segments

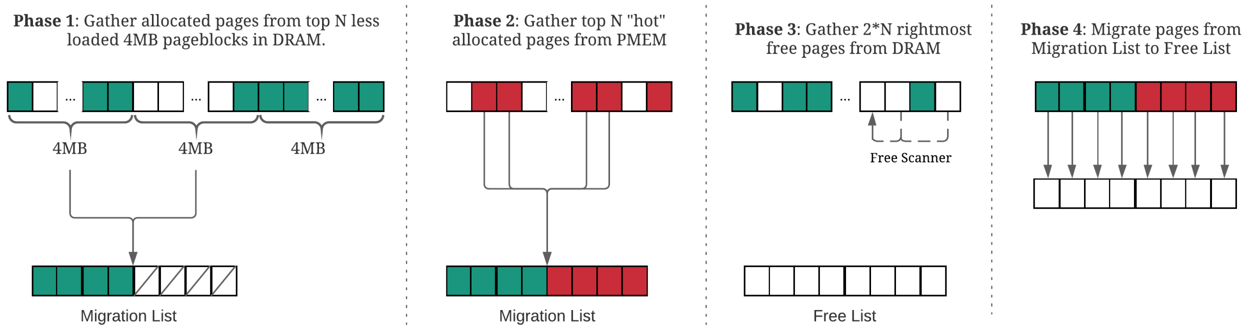
Phase 1: Gather allocated pages from top N less loaded 4MB pageblocks in DRAM.
4MB
4MB
4MB
Migration List
Phase 2: Gather top N "hot" allocated pages from PMEM
Migration List
Phase 3: Gather 2*N rightmost free pages from DRAM
Free Scanner
Free List
Phase 4: Migrate pages from Migration List to Free List

Figure 2: Combined DRAM compaction and page migration from PM in PMShifter

clean, increases the physical contiguous memory available, reduces the total allocation time due to the increased bigger segments' availability, and lowers the demand for synchronous compaction. **(ii) We skip pageblocks that contain unmovable pages**. We check this in O(1) since this information is provided as a flag field in each pageblock. This optimization prevents the memory compaction overhead described at the end of section 3.2. **(iii) The free page scanner in PMShifter does not skip empty 2 MB and 4 MB pageblocks**. This allows to efficiently use the whole space that is available in the *free area*, without adding any kind of overhead. **(iv) The PM and DRAM scanners do not keep pageblocks state between compaction runs**. As we described in 3.2, the Linux compactor skips unusable pageblocks for multiple runs. We empirically found that these pageblocks are limited in number, while this "unused" status does not persist between multiple runs.

In terms of *which* pageblocks to compact, we differentiate in DRAM and PM. In DRAM, the goal is to end up with the utmost amount of clean pageblocks. Since we compact a fixed amount of pageblocks, we prioritize the ones that have the fewest allocated pages in them, to maximize the number of clean pageblocks. We check the emptiest pageblocks by maintaining a bitvector that tracks utilized 4 KB pages in each pageblock. This is efficient both in terms of speed and space, since it only reserves 96 MB for a 3 TB memory. Our goal about PM compaction is different: we want to keep the leftmost end of the memory clean; the intuition about this decision will become clear in Section 4.2.

4.1.2 PM Page Migration.

PMShifter's goal is to avoid migrating pages that would cause memory pressure, setting a minimum free space that should always be available. We empirically set this free page threshold to be over 3× the number of pages to migrate. PMShifter uses the *active* list to determine which pages to migrate to DRAM. The advantage of this is that we avoid extra overhead for hot page tracking, while using a carefully designed and maintained hotness tracking mechanism.

4.1.3 Combined Mechanism.

The process for the combined migration is as follows: **First**, we assemble a migration list containing DRAM pages for compaction and PM pages for migration. **Second,** we scan the memory space and gather free pages in DRAM to accommodate the pages from the migration list. **Finally**, we do the actual migration between the migration and free lists. Figure 2 shows the entire process of our combined DRAM compaction and PM page migration mechanism. By combining migration and compaction in the Migrator, we achieve three things: **First**, we increase the rate of migration by batch migrating pages. Yan et al. [27] displayed that increasing the number of pages under migration, also increases the migration throughput. **Second**, we avoid contributing to the memory fragmentation by directly migrating pages to the right-most end of the memory. **Third**, we avoid the centralized allocation path by using the free page scanner during compaction to find free target pages.

If there are no PM pages for migration, then we only run the compaction mechanism. We use a background periodic kernel thread for this part, migrating up to 400 MB in each iteration to avoid adding significant overhead.

4.2 Shifter

In this section, we describe what the functionality of the Shifter and focus on the following 3 important matters: **(i)** when should the shifting occur, **(ii)** how to choose the PM areas that should undertake the shifting, to minimize the cost, and **(iii)** what happens if there is no available PM space to shift or if there need to increase the available DAX space.

4.2.1 Memory Pressure Classification.

The Shifter's goal is to accurately predict whether we need to shift memory according to memory pressure. We do this by adjusting the Exponential Moving Average [26] formula to measure the memory pressure at any time:

$$MP_t = a * free\_space + (1 - a) * MP_{t-1}$$

Where MP_t is the memory pressure at time t, *free_space* denotes the total free space in every DRAM-PM pair, and a is a smoothing factor which is used to remove the noise from short-term fluctuations.

The Shifter uses a periodic background thread that manages PM shifting and checks every second the existing memory pressure. If MP_t is greater than the given threshold, it shifts memory until the free available memory is increased by 5×. To prevent fluctuating shifting to both directions (memory to storage and vice-versa), we use a 5-second interval in-between different direction shifting. We fine-tune all the variables to accurately predict the need for shifting.

4.2.2 PM area selection. Choosing the right memory areas to undertake the memory to storage conversion is crucial to PMShifter's performance. The goal here is twofold: **(i)** maintain the contiguity, and **(ii)** minimize the cost. The former can be achieved by either pruning the start or the end of the PM address space. The latter dictates our decision of what to prune: since we want to reduce the cost, we need to offline and convert a region that probabilistically has fewer allocated pages. The intuition behind this is the fact that offlining multiple allocated pages at once is extremely costly and time consuming, as described in the end of section 2.1.

Trying to keep the leftmost PM memory regions clean helps us alleviate the cost of offlining allocated pages. This intuition is what led us to separate the logic between the DRAM and PM compactors: the former tries to maximize the total number of clean pageblocks, while the latter prioritizes keeping the left-most PM area clean. Another benefit is that we remove page migration from the critical path, avoiding disrupting workloads' execution.

Despite keeping clean the leftmost end of the memory, it might contain some allocated pages. To avoid the cost of offlining multiple allocated pages, we first check our bitvector for each pageblock to offline, and limit this operation to up to 100 MB. We also check if the free space in DRAM is enough to comfortably accommodate potential allocated pages that will be offlined. In our prototype, this check is the same as for page migration, where the amount of free space should be at least 3× the amount that we want to offline.

4.2.3 Insufficient PM space. When there is an absence of PM to utilize under memory pressure, PMShifter falls back to the default case, swapping. When there is need for DAX-enabled space, and some portion of PM is used as memory, we just shift and evict the pages. This eviction leads to page migrations from PM to DRAM or swapping, depending on if DRAM has enough space to accommodate these pages.

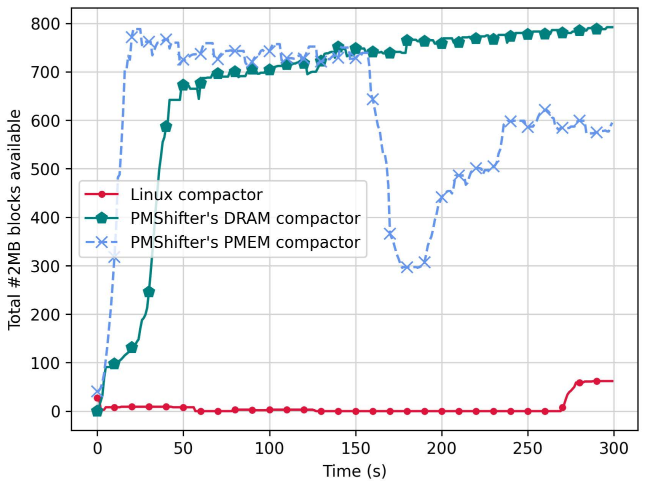

Figure 3: Recovery of 2 MB pageblocks for vanilla Linux and PMShifter.

4.3 NUMA Efficiency

We fix the NUMA pathologies that we described in 3.4 with the following changes: **(i)** We disable NUMA autobalancing for PM, allowing PMShifter to manage ***which*** pages should be migrated and ***how frequently***. This allows PMShifter to minimize the use of PM only when it is needed, while keeping the hottest pages in DRAM. **(ii)** We change the allocation/page migration ordering to prioritize the DRAMs over the PMs, for each remote DRAM-PM pair.

5 Evaluation

For our evaluation we use a mix of memory-intensive microbenchmarks, and real-world applications. The former are used to isolate and demonstrate how our modifications affect individual parts of the memory subsystem, while the latter to show the impact of our changes in real-world situations.

Our evaluation targets to address the following:

- Is Shifter elastic and proactive?
- How fast can our DRAM and PM compactors restore the physical contiguity, compared to vanilla Linux?
- How does the Migrator perform?
- What is the impact of our NUMA ordering changes?

5.1 Configuration

We performed our experiments in a dual-socket, 24-core per socket machine. Each socket has 6 memory channels, and the total amount of memory that we used includes 32 GB Micron DDR4 DIMM and 256 GB Intel Optane DC Persistent Memory. We built PMShifter on top of Linux 5.6.19, which we also use to evaluate our changes. We also modified the Linux compactor to reflect the changes introduced in v5.9.

5.2 Fragmentation

We use the modified periodic Linux 5.9 compactor and compare it with PMShifter's compactors. In a virtual machine with 12 GBs of memory, we use a kernel-level workload to severely fragment the memory[1], until there are nearly non 2 MB and 4 MB pageblocks available. We track how these numbers change over the course of 3 minutes to evaluate how fast each compactor can restore physical contiguity. We run this experiment for each compactor, each time in a fresh VM instance, and track the total number of 2 MB pageblocks, as the sum of 2 MB and 4 MB pageblocks. We track only these pageblocks, as the biggest contiguous segments that the Linux memory allocator keeps track of; a large number of these pageblock being available are a good memory fragmentation index. For each compactor we try to compact up to 200 MB every 5 seconds.

Figure 3 shows how fast the three compaction mechanisms can increase the total free 2 MB pageblocks. For the first 272 seconds, the default compactor recovered none, while PMShifter's compactors start almost immediately restoring these pageblocks. Regarding the PM compactor, there is a big plunge starting from second 160, but then it continues retrieving physically contiguous pageblocks. This is anticipated, since we do not set an upper limit on the regions that the PM migrate scanner looks for allocated blocks. After the first three minutes, we notice that the PM compactor continues recovering more 2 MB segments.

The result is that our DRAM and PM compactors reclaim up to 12.77× more 2 MB physical contiguous blocks compared to Linux. At the same time, our compactors initiate this 2 MB clean block retrieval much earlier.

5.3 Combined Page Migration

We measure the efficacy of PMShifter's combined page migration mechanism by comparing it with a compaction followed by a migration in vanilla Linux. We vary the size of total migrated pages between 4 MB and 1.6 GB, half of which are used for compaction in DRAM and half for migration from PM, and measure the throughput and number of failed page migrations. Figure 4 shows the corresponding rates of migration for each method. The average speedup of our method is 5.88× over the default method.

We also measure the number of failed migrations. In our case, the number of failed migrated pages is usually zero, but always less than 0.0083%, while for Linux it is between 41.4% and 49.9%. In PMShifter, a migration can only fail when there is a rapid change in the page status between the migrate list assembly and actual migration. For Linux, this high failure percentage is primarily attributed to the scanners' early meeting, described in subsection 2.2. At the same time, PMShifter migrates the pages on the right side of DRAM, avoiding any fragmentation exacerbation, which would lead to further future migrations.

[1]We use a VM instead of our machine, because this kernel workload can potentially corrupt the kernel memory

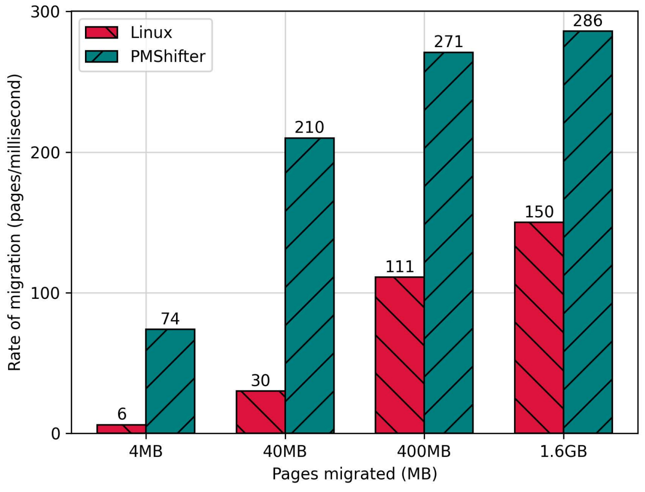

Figure 4: Migration throughput for combined page migration in vanilla Linux and PMShifter. Higher is better.

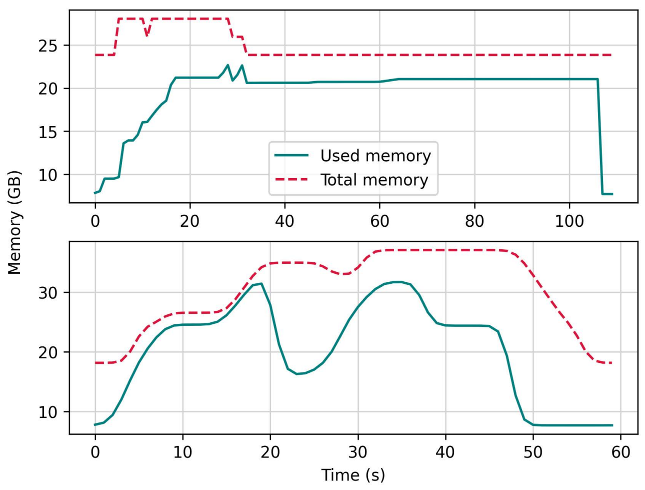

Figure 5: Total memory size and allocated memory in PMShifter: (i) when running connected components in Galois with a 65M node, 1.8B edges input graph (up), and (ii) for a microbenchmark with varying memory allocation/freeing rate (down).

5.4 Impact On Real-Life Workloads

We use Galois [15], an object-based parallelization system, to evaluate the PMShifter's performance and malleability. We run the Galois connected components implementation over a large input graph from Stanford Network Analysis Platform (SNAP) [18], with 65 million nodes and 1.8 billion edges. This

Operation	Mset	Set	Get	Lpush	Rpop
Throughput	2.02x	1.62x	1.92x	2.09x	1.92x
90th latency	0.49x	0.64x	0.59x	0.48x	0.57x
99th latency	0.49x	0.56x	0.48x	0.48x	0.36x
99.9th latency	0.50x	0.64x	0.55x	0.52x	0.39x

Table 1: Redis throughput and latency in PMShifter compared to vanilla Linux. For throughput higher is better, for latency lower is better

workload is divided in two phases: **(i)** reading and preprocessing the graph and **(ii)** executing the connected components algorithm. The total system and workload memory required is approximately 28 GB. We limit the local DRAM to 20 GB.

Upper Figure 5 shows the total amount of used memory and the total available amount of memory. We derive the following: **First**, in the initial stage (5s - 10s), where Galois is allocating most of the memory, PMShifter detects a rapid change in the allocation rate, and starts shifting PM storage segments, correctly predicting the amount of memory needed in the near future. **Second**, from second 18 till the end, the total amount of used memory is roughly stabilized; PMShifter notices that the allocation rate has dropped and the amount of memory needed is less, so it starts offlining the pages. **Third**, after PMShifter shifts back the PM segments, the total amount of memory and used memory are close, showing that PMShifter correctly identifies that this memory is not needed, avoiding wasting PM as memory at all. We note that the remote DRAM is not used at all, which is the intended case to avoid any performance degradation.

Next, we use Redis, an in-memory data store, to evaluate our NUMA modifications. We run `redis-benchmark` [22] with a memory consumption of 16 GB, and measure the latency and throughput for 5 different operations: `mset`, `set`, `get`, `lpush`, `rpop`. We fully consume the local DRAM, to ensure that it will not receive any pages during the experiment.

We first run redis-server on local PM, populate the server with the key values, offline the local PM node, and then run the Redis operations. Offlining the node forces the OS to migrate the allocated pages to the next memory node according to the policy. Since the local DRAM is full, the pages are migrated to remote PM in Linux, and remote DRAM in PMShifter. Table 1 contains Redis' throughput, and 90th, 99th, and 99.9th latency PMShifter compared to vanilla Linux, showing that PMShifter can offer up to 2.09x better throughput with approximately 50% reduction across all latencies.

5.5 Hybrid Memory Shifting Elasticity

We test PMShifter's elasticity with a microbenchmark that uses different allocation and freeing rates. Lower Figure 5 shows the corresponding total allocated and total size of memory. We notice that PMShifter responds accordingly to any allocation rate and absence of total free memory, while it avoids withholding unnecessary PM memory segments. We repeat this for a fixed allocation/freeing rate and observe similar elasticity, but omit the results due to space limitation.

6 Related Work

Dynamic Memory Storage Division. A dynamic distinction between memory and storage has been briefly discussed in the past by Song et al. [25], where PM was split between storage and a dynamically configured part. However, it does not take into account associated memory management issues and misses a prototype.

Compaction. Illuminator [21] modifies the memory allocator to avoid polluted pageblocks, containing both movable and unmovable pages. Ingens [16] uses the default memory compactor periodically. MEGA [20] compacts pageblocks based on their fullness and age of virtual-to-physical mappings. These solutions are orthogonal or complementary to PMShifter's compactors.

Identifying Hot Pages. Yan et al. [27] use the *active* and *inactive* lists to identify hot and cold pages. HeteroOS [11] and Ingens [16] use the kernel's idle page tracking feature [12], which is associated with frequent TLB invalidations. Ingens uses the EMA for past accesses to ameliorate this cost.

Thermostat [2] also uses the idle page tracking feature, but limits the overhead by using classification to predict which pages are accessed frequently.

7 Conclusion

This paper proposes PMShifter, the first complete work on dynamic PM configuration. Our evaluation shows that PMShifter adapts well to the memory needs of the system, working proactively and avoiding interfering with workloads' and the system's normal execution. PMShifter also increases the page migration throughput, both within and across different memory nodes, and fixes PM-unaware migration policies. Last, it improves compaction, both in terms of speed and total amount of physical contiguity restored.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This paper is supported in part by NSF grants 1829524, 1817077, and SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications.

In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[2] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. *SIGPLAN Not.* 52, 4 (April 2017), 631–644. https://doi.org/10.1145/3093336.3037706

[3] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mossé. 2014. Concurrent Page Migration for Mobile Systems with OS-Managed Hybrid Memory. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (Cagliari, Italy) *(CF '14)*. Association for Computing Machinery, New York, NY, USA, Article 31, 10 pages. https://doi.org/10.1145/2597917.2597924

[4] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. 2019. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/3307681.3325398

[5] Mel Gorman. Online; Accessed May, 2021. Chapter 6 Physical Page Allocation. https://www.kernel.org/doc/gorman/html/understand/understand009.html.

[6] Mel Gorman. Online; Accessed May, 2021. Foundation for automatic NUMA balancing. https://lwn.net/Articles/523065.

[7] Intel. Online; Accessed May, 2021. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[8] Intel. Online; Accessed May, 2021. Intel Optane DC Persistent Memory - Quick Start Guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf.

[9] Intel. Online; Accessed May, 2021. Reimagining Memory and Storage in the Data Center. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology/reimagine-memory-storage-in-the-data-center.html.

[10] PMDK Team Intel. Online; Accessed May, 2021. Documentation for ndctl and daxctl. https://pmem.io/ndctl.

[11] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 521–534. https://doi.org/10.1145/3079856.3080245

[12] The kernel development community. Online; Accessed May, 2021. Idle Page Tracking. https://www.kernel.org/doc/html/latest/admin-guide/mm/idle_page_tracking.html.

[13] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 715–728. https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

[14] A. Kokolis, D. Skarlatos, and J. Torrellas. 2019. PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 596–608. https://doi.org/10.1109/HPCA.2019.00012

[15] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic Parallelism Requires Abstractions. *SIGPLAN Not.* 42, 6 (June 2007), 211–222. https://doi.org/10.1145/1273442.1250759

[16] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 705–721.

[17] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, USA, 277–289.

[18] Jure Leskovec and Andrej Krevl. Online; Accessed May, 2021. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[19] Felix Xiaozhu Lin and Xu Liu. 2016. Memif: Towards Programming Heterogeneous Memory Asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 369–383. https://doi.org/10.1145/2872362.2872401

[20] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. 2019. MEGA: Overcoming Traditional Problems with OS Huge Page Management. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '19)*. Association for Computing Machinery, New York, NY, USA, 121–131. https://doi.org/10.1145/3319647.3325839

[21] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 679–692. https://doi.org/10.1145/3173162.3173203

[22] Redis. Online; Accessed May, 2021. Redis benchmark. https://redis.io/topics/benchmarks.

[23] Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. 2018. A Case for Granularity Aware Page Migration. In *Proceedings of the 2018 International Conference on Supercomputing* (Beijing, China) *(ICS '18)*. Association for Computing Machinery, New York, NY, USA, 352–362. https://doi.org/10.1145/3205289.3208064

[24] Nikolay Savvinov. Online; Accessed May, 2021. Memory fragmentation: the silent performance killer. https://savvinov.com/2019/10/14/memory-fragmentation-the-silent-performance-killer.

[25] Hyeonho Song and Sam H. Noh. 2018. Towards Transparent and Seamless Storage-As-You-Go with Persistent Memory. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/song

[26] Wikipedia contributors. Online; Accessed May, 2021. Exponential Moving Average — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Moving_average.

[27] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 331–345. https://doi.org/10.1145/3297858.3304024

[28] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang