A Study of Memory Placement on Hardware-assisted Tiered Memory Systems

Wonkyo Choe, Jonghyeon Kim, and Jeongseob Ahn
Ajou University

Abstract—Recent advances in memory technology, memory hierarchy is becoming diverse with performance-differentiated memory such as high bandwidth memory (HBM) and non-volatile memory (NVM) in modern computer systems. However, the current memory placement has been designed with the assumption that all the memory has the same capabilities based on DRAM. In this paper, we analyze memory placement schemes in state-of-the-art Linux systems to the hardware-assisted tiered memory system. Our analysis is conducted on the real system equipped with Intel's Optane Persistent Memory, enabling tiered memory with DRAM (fast) and Optane (slow) memory. We observe that tiered memory augmented on traditional NUMA form do not exhibit that accessing the local memory provides better performance than that of the remote memory because local Optane memory is slower than remote DRAM. Due to this distinct characteristic, there are several inefficiencies in the commodity operating systems. To make use of tiered memory systems efficiently, this paper explores the design space of practical software solutions, which can be currently applicable in the Linux system.

Index Terms—Tiered Memory, Memory Placement, NUMA.

1 Introduction

As the non-volatile memory comes to a part of the main memory [4], the design of memory management in the operating systems should be revisited. Traditional operating systems have been designed with the assumption that all the memory in the system has the same capabilities based on DRAM. However, with the proliferation of performance-differentiated memory in modern server systems, the current operating systems, including Linux, fail to properly take into account the full benefits of tiered memory systems. Fig. 1 shows a typical two-socket system augmented with tiered memory. Traditional operating systems consider that the local memory is always faster than remote memory. In tiered memory systems, however, this assumption is no longer valid because the performance of accessing remote DRAM is faster than that of local non-volatile (Optane) memory. This performance characteristic is distinct from traditional NUMA systems and violates the design assumption in several aspects.

In this paper, we first analyze state-of-the-art Linux memory placement techniques to hardware-assisted tiered memory systems based on Intel's Optane memory. In HW-assisted tiered memory systems, the DRAM works as a HW-managed cache, automatically placing frequently accessed data on that while the rest of the data is kept on the large capacity but slow Optane memory. The DRAM is no longer visible to software, but the Optane memory is exposed to software as the main memory instead. Such HW-assisted tiered memory systems are transparent to existing software. A distinct characteristic is that the caching area is limited to the memory controller, which DRAM DIMMs and Optane DIMMs share. As depicted in Fig. 1, there are two DRAM caches physically separated on the basis of the memory controllers.

Since the default memory placement is designed and optimized for localizing memory access, it does not utilize remote DRAM caches before the local Optane memory is used up. As a result, the current design is not sufficient to take advantage of all the separated DRAM caches in the system. Also, this change to the memory hierarchy breaks many NUMA-aware optimization techniques. For instance, the AutoNUMA

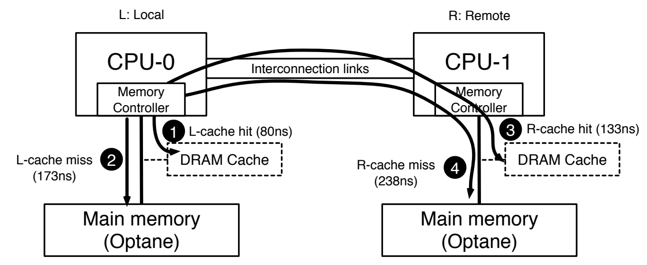

Fig. 1: HW-assisted tiered memory system (Latency is measured on a real system by using Intel MLC [3])

balancer [5] can lead to degradation of system performance. If the local DRAM cache is fully occupied, migrating cached pages in the remote DRAM into the local Optane memory can exacerbate the contention of the local DRAM cache.

The root cause of such inefficiency is the lack of a proper memory placement for tiered memory hierarchy. To mitigate the inefficiency, this paper explores the design space of memory placement schemes by considering the distinct performance characteristics of HW-assisted tiered memory systems. First, this paper proposes a new memory allocation policy to take advantage of the fast DRAM entirely, and then make use of the slow Optane memory. This approach is straightforward and intuitive. We evaluate the new policy on HW-assisted tiered memory systems. Second, we revise the AutoNUMA balancer for HW-assisted tiered memory to reduce the DRAM cache contention, bringing remote pages into local memory opportunistically. The extended AutoNUMA considers the available space of the DRAM cache as well as the memory access locality.

We implement our proposed memory placement schemes on top of the state-of-the-art Linux kernel 5.3. The experimental results show that the proposed memory allocation policy can utilize the remote DRAM cache effectively. The average latency and throughput are significantly improved when the required memory for applications can be served on the local and remote DRAM cache rather than Optane memory. As future work, we will examine the current design of memory management including page migration, promotion, and demotion mechanisms and policies.

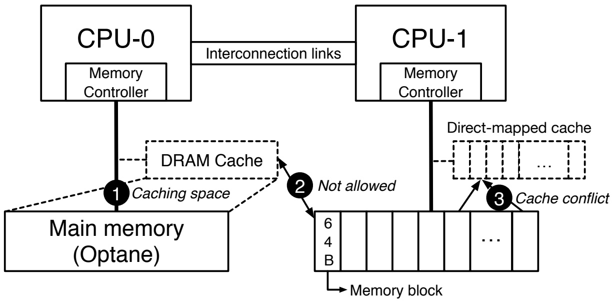

Fig. 2: Direct-mapped DRAM caches are separated across memory controllers

2 Background and Motivation

With the proliferation of performance-differentiated memory in computer systems, we need to revisit the design and implementation of the current memory placement schemes. We examine memory allocation schemes along with the NUMA management in the Linux kernel because the combination of local and remote memory resembles the tiered memory in terms of the memory hierarchy. There have been significant efforts to optimize NUMA systems on the commodity operating systems, but mostly DRAM-only systems have been considered [1]. Since the remote memory accesses lead to significant degradation of system performance, the default memory allocation policy (`local-first`) is designed to allocate pages on the local memory where the requester is currently running unless there is no free space. Thus, it eliminates the time accessing remote memory if the working set of applications fits well on the local memory. If the amount of required memory for applications cannot be served on the local memory, the Linux kernel looks up free pages on the remote memory according to the fallback node list (called `fast-path` in memory allocation). If it still fails to find free memory, the page reclamation is performed, such as direct compaction, direct reclamation, OOM killer, and kswapd (called `slow-path`).

Although this design approach has been well established in the Linux operating system, this has not been considered appropriate for tiered memory hierarchy. In tiered memory systems, it cannot be guaranteed that accessing the local memory provides better performance than that of the remote memory. Since slow memory is now part of the memory hierarchy, the design assumption is not valid on tiered memory systems. As depicted in Fig. 1, each CPU socket has both fast memory (DRAM) and slow memory (Optane) in its local domain. Typically, DRAM in the remote domain (❸) is faster than Optane (❷) attached to the local domain. Nevertheless, the default policy still allocates memory pages on slow memory in the local domain even if fast memory in the remote domain has enough free space. Although this is intended to reduce the remote memory traffic when considering DRAM-only NUMA systems, it has a negative performance impact on tiered memory systems.

3 Characterization of HW-assisted Tiered Memory Systems

In this section, we describe distinct characteristics and performance inefficiency of current memory placement schemes for hardware-assisted tiered memory systems. These observations motivate us to explore the design space of memory placement for tiered memory hierarchy.

Observation 1. We found that the DRAM cache is physically separated and managed by each memory controller where the DRAM DIMM is attached. Each DRAM cache can only hold

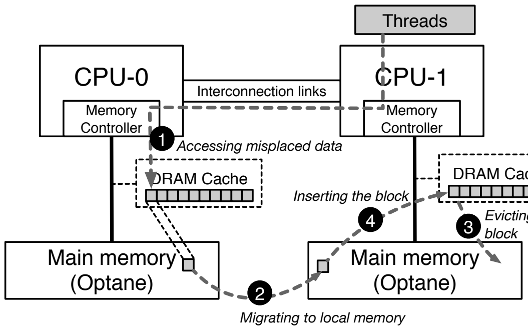

Fig. 3: Automatic NUMA balancing migrating pages cached in remote DRAM cache to local Optane memory

data that resides in the Optane memory attached to the same memory controller. Fig. 2 shows that the DRAM cache (❶) attached to the CPU-0 socket is not able to cache data stored in the Optane DIMM (❷) of the CPU-1 socket because they do not sit on the same memory controller.

Since the operating system has been designed to allocate pages on the local memory as much as possible, the local memory can be highly overloaded while the remote memory becomes idle. Note that the available remote memory space will be utilized once the local memory is fully occupied. This is intended for DRAM-only systems. When considering the separated DRAM caches in the system, this approach is not reasonable because it misses the opportunity of utilizing the remote DRAM cache. If the memory footprint of applications does not fit on the local DRAM cache, the operating system spends time back and forth between the DRAM cache and the Optane main memory while the remote DRAM cache is idle.

Observation 2. Another challenge is to minimize conflict misses within a memory node. Each DRAM cache is organized as a direct-mapped cache where each memory block (here 64Byte) is mapped to exactly one cache location. Two more memory blocks cannot be mapped to a single cache set, as depicted in Fig. 2 (❸). If some memory blocks that an application is frequently accessing are mapped to the same cache set, it causes significant performance degradation due to the excessive conflict misses. To reduce the number of conflict misses, a possible software solution is to allocate pages that are unlikely to incur the cache conflicts. However, as of writing, the DRAM cache indexing scheme has not been disclosed. We leave implementing a conflict-aware memory allocation as our future work.

Observation 3. We figured out that the AutoNUMA balancer is considered harmful to tiered memory systems because that is designed for DRAM-only NUMA systems. If application threads access the remote memory frequently, the AutoNUMA balancer migrates the remote pages to the local memory node [5]. This can minimize the remote memory traffic so that performance can be optimized in traditional memory systems. However, it may degrade performance on tiered memory systems. Fig. 3 shows the flow of balancing NUMA on the tiered memory system. This example assumes that part of the memory for the application is allocated in the remote memory node and cached on that remote DRAM cache. Once the application accesses the remote page frequently (❶ of Fig. 3), the remote pages would be migrated into the local memory (❷) unless there is no free space. We can eliminate the remote memory traffic. However, if the local DRAM cache does not have enough space, this incurs a cache eviction (❸) and insert the accessed memory block (❹). Due to the lack of local DRAM cache capacity, the application can experience frequent DRAM cache misses while not utilizing the remote DRAM cache.

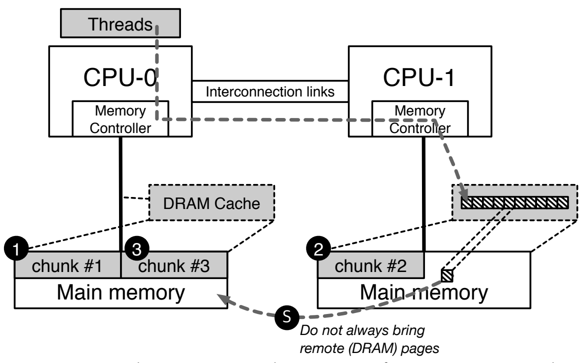

Fig. 4: Proposed memory placement for HW-assisted tiered memory

4 Memory Placement for HW-assisted Tiered Memory

In this section, we suggest that the memory allocator should be able to utilize the separated DRAM caches in the system entirely. As observed in Section 3, the default memory allocation policy used in the state-of-the-art Linux kernel is not able to exploit the full potential of separated DRAM caches in the system. The easiest way is to use the `page-interleave` policy in such tiered memory systems. However, it causes unnecessary remote memory accesses if the working set can fit on the local DRAM cache. In this study, we propose a new memory allocation policy (called `dram-first`). Fig. 4 shows the `dram-first` memory allocation flow as the memory consumption increases. The chunk size is identical to the size of the DRAM cache. ❶ This policy allocates pages on the local memory as much as the local DRAM cache (chunk size) can afford. Once the usage of the local DRAM cache reaches the maximum, we stop allocating memory locally and ❷ then use the remote memory to avoid expensive local Optane memory accesses while utilizing the remote DRAM cache in the system. If all the DRAM caches become full, ❸ then the memory allocator returns to the default policy, which can minimize remote memory accesses. This technique does not necessarily replace the default memory allocation policy. Instead, this technique can provide users with an alternative option to extract the full capabilities of tiered memory.

Next, we propose to extend the AutoNUMA balancer, opportunistically bringing the remote pages into local memory based on the DRAM cache usage. If the working set of applications can fit on the separated DRAM caches, we disallow the AutoNUMA balancer to migrate remote pages into the local memory. It can achieve the performance benefit by utilizing the remote DRAM cache, instead of expensive local slow memory accesses. As depicted in Fig. 4 Ⓢelective, we prevent to bring the remote pages if the local DRAM cache is full, but the remote DRAM cache can serve that. On the other hand, if the working set is larger than the aggregate size of all the DRAM caches so that the DRAM cache miss rate is significant, we allow the AutoNUMA balancer to migrate remote pages into local memory. It can minimize the cost of accessing remote slow memory. However, we observe that balancing memory across memory nodes improves performance in such DRAM cache designs. We leave a sophisticated technique to tune the AutoNUMA balancer as our future work.

5 Evaluation

5.1 Experimental Environment

Our evaluation platform has two Intel Xeon Gold 5218 processors, each of which has 16 physical cores. Each CPU socket has two integrated memory controllers (iMC), each of which has a 16GB DDR4-2666 DIMM and a 128GB Intel Optane DIMM on the same channel. We have a total of 64GB DRAM and 512GB Optane memory in the system. To minimize the measurement variability, we disable Hyper-Threading, DVFS, Turbo-Boost, and HW prefetchers. The Linux kernel 5.3 is used as our baseline and implement proposed schemes on top of the kernel.

Benchmarks: The existing memory latency and throughput benchmarks such as Intel's MLC [3] and open source X-mem [2] are not accurate to measure the performance of tiered memory systems. Such benchmarks are not designed to touch the given working set entirely. Thus, the sampling approach cannot represent the performance of tiered memory systems. We extend the X-mem to enforce that all the memory accesses are made at least once. We also evaluate our system with a real-world benchmark, `Memcached`, from CloudSuite.

5.2 Results

Latency: Fig. 5 shows the average sequential read, random read, and random write latency with the DRAM cache miss rate as the working set size increases. Each stacked bar indicates the local and remote DRAM cache miss rates for `default`, `page-interleave`, and `dram-first` in that order. The performance trend is not much difference across the access patterns. The average random access latency is slightly longer than the average sequential access latency. With the default memory allocation policy, the use of remote DRAM cache is limited to the case where the working set size is larger than the size of local memory. For the sequential read pattern (leftmost), the latency with the `default` policy is significantly affected when the memory footprint cannot fit on the local DRAM cache due to the lack of DRAM cache. The bars presenting the DRAM cache miss rate are supporting these phenomena. On the other hand, the `page-interleave` policy can minimize the latency for the 64GB working set size because it can utilize the local and remote DRAM caches. However, the `page-interleave` policy is not efficient at the small working set sizes from 16GB to 32GB because it causes remote memory accesses while the local DRAM cache is not fully occupied. For the large working set sizes from 128GB to 384GB, the DRAM cache misses frequently occur on both the local and remote DRAM cache so that it incurs additional overheads of accessing Optane memory.

On the other hand, our proposed scheme (`dram-first`) can extract the full performance of tiered memory from 16GB to 384GB. In the 64GB working set size, the local and remote DRAM miss rates are nearly zero because the working set can fit on them. It can achieve the best of the `default` and `page-interleave` policies. For the large working set sizes (256GB and 384GB), all the policies, including `dram-first`, spend the expensive cost of accessing remote (Optane) memory. Both the `default` and `page-interleave` policies are not designed for the tiered memory hierarchy so that their approach makes the performance sub-optimal in terms of latency.

We present the average latency for random read (middle) and write patterns (rightmost) for the range of 32GB to 128GB that is the boundary of DRAM caches and Optane memory. From 64GB to 96GB working sets, our `dram-first` and the `page-interleave` policy show a similar performance because both schemes utilize the entire local and remote DRAM caches. Beyond 96GB, the performance gap is increasing as the `dram-first` makes memory allocation on the local memory node while the `page-interleave` allocates pages across local and remote memory nodes.

Throughput: In Fig. 6, we look at the throughput performance from 32GB to 128GB of the working set with fine-grained scope.

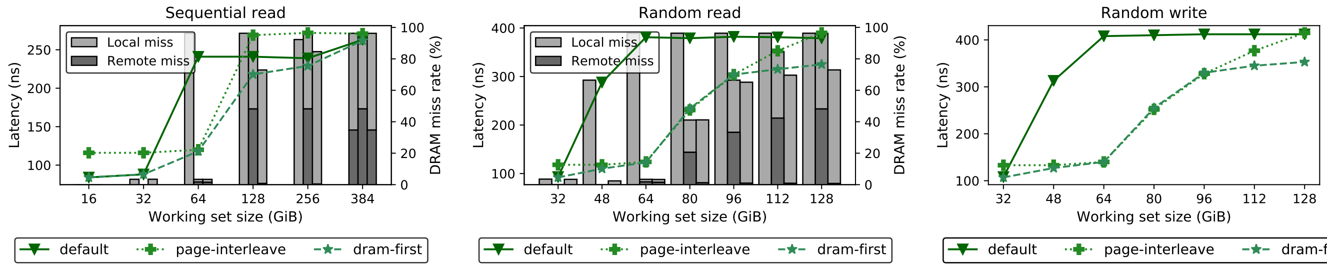

Fig. 5: Latency results for three access patterns (Lower is better)

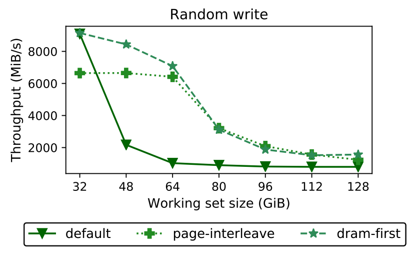

Fig. 6: Throughput (Higher is better)

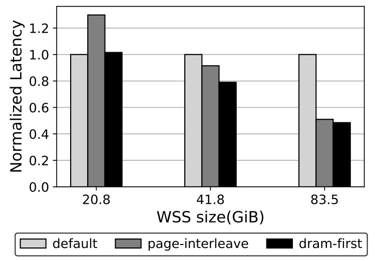

Fig. 7: Memcached with `dram-first`

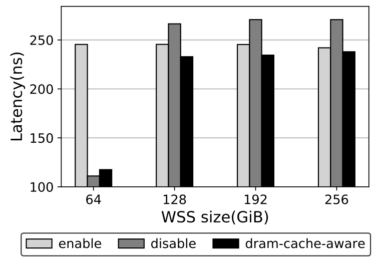

Fig. 8: Microbench with AutoNUMA

Our `dram-first` shows better performance than the `default` and `page-interleave` policies for the range of working sets. For the 32GB working set, the `default` and `dram-first` policies show a similar performance. From the 48GB working set, however, the local DRAM cache miss rates are significantly increasing in the `default` policy. For the `page-interleave` policy, the throughput decrease can be seen when using more than 64GB because it starts to use the remote DRAM cache. On the other hand, our `dram-first` scheme can efficiently utilize the local and remote DRAM caches so that it can maximize the DRAM throughput as much as possible.

Real-world benchmark: We evaluate the three policies, `default`, `page-interleave`, and `dram-first` with Memcached. We run a Memcached client on a different physical server within the same 10G network. Fig. 7 presents the average latency of Memcached. The baseline is the default policy (`local-first`). With the `page-interleave` and `dram-first`, the average latency is significantly reduced by utilizing the remote DRAM cache. For the `20.8GB` dataset, the `default` and `dram-first` show a similar performance because all the working set can be placed on the local DRAM cache. With the `page-interleave` policy, the latency is significantly affected because it generates the remote access traffic even though the local DRAM cache is not fully occupied. In the `41.8GB` dataset, we can improve the average latency by 21% compared to the baseline. Our `dram-first` can maximize the use of remote DRAM cache. On the other hand, the `page-interleave` policy evenly distributes the data across two memory nodes so that it spends more time in accessing data remotely. With the `83.5GB` dataset, our `dram-first` and `page-interleave` show a comparable performance because both schemes can utilize the remote DRAM cache.

Effect of AutoNUMA balancer: To figure out the performance impact of the AutoNUMA balancer, we run a microbenchmark we built. The microbenchmark allocates the given data half and a half to the local and remote memory and measures the average access latency. As Fig. 8, AutoNUMA (`enable`) shows the worst performance at the 64GB working set size. Even though the data can fit in local and remote DRAM caches, AutoNUMA brings almost all remote data into the local memory. As a result, it incurs excessive DRAM cache misses, leading to performance degradation. On the other hand, our scheme (`dram-cache-aware`) does opportunistically migrate the remote pages to local memory to minimize the cache contention. It does not allow to migrate remote data to local memory when the local cache does not have a free room while it can stay in the remote DRAM cache. From the 128GB to 256GB working set sizes, `dram-cache-aware` leaves part of data in the remote memory so that it is beneficial from the remote DRAM cache while the rest of them is migrated to the local memory. Thus, our scheme shows improved latency, compared to AutoNUMA (`enable` and `disable`).

6 Conclusions and Future Work

In this work, we conducted a detailed analysis of Linux memory placement to HW-assisted tiered memory systems. We found several inefficiencies in the current design that should be reconsidered by understanding the distinct characteristics of tiered memory systems. To alleviate the issues, we explored the design space of memory placement aspects to extract the full capabilities of the tiered memory.

As our future work, we plan to revisit the design and implementation of commodity OSes in terms of page allocation, reclamation, migration, promotion, and demotion because those mechanisms and policies are not well explored in tiered memory architecture.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2019R1C1C1005166).

References

[1] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth, "Challenges of memory management on modern numa systems," *Communication of ACM*, Nov. 2015.
[2] M. Gottscho, S. Govindan, B. Sharma, M. Shoaib, and P. Gupta, "X-mem: A cross-platform and extensible memory characterization tool for the cloud," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.
[3] Intel, "Intel memory latency checker v3.7," 2019. [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker
[4] L. Looi and J. J. Xu, "Intel optane data center persistent memory," in *HotChips : A Symposium on High-Performance Chips (HotChips)*, 2019.
[5] R. van Riel and V. Chegu, "Automatic numa balancing," 2014. [Online]. Available: https://www.redhat.com/files/summit/2014/summit2014_riel_chegu_w_0340_automatic_numa_balancing.pdf