
# Exploring the Design Space of Page Management for Multi-Tiered Memory Systems

Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn, *Ajou University*

https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the
2021 USENIX Annual Technical Conference
is sponsored by USENIX.

# Exploring the Design Space of Page Management for Multi-Tiered Memory Systems

Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn
*Ajou University*

## Abstract

With the arrival of tiered memory systems comprising various types of memory, such as DRAM and SCM, the operating system support for memory management is becoming increasingly important. However, the way that operating systems currently manage pages was designed under the assumption that all the memory has the same capabilities based on DRAM. This oversimplification leads to non-optimal memory usage in tiered memory systems. This study performs an in-depth analysis of page management schemes in the current Linux design extending NUMA to support systems equipped with both DRAM and SCM (Intel's DCPMM). In such multi-tiered memory systems, we find that the critical factor in performance is not only the *access locality* but also the *access tier* of memory. When considering both characteristics, there are several alternatives to page placement. However, current operating systems only prioritize access locality. This paper explores the design space of page management schemes, called *AutoTiering*, to use multi-tiered memory systems effectively. Our evaluation results show that our proposed techniques can significantly improve performance for various workloads, compared to the stock Linux kernel, by unlocking the potential of the multi-tiered memory hierarchy.

## 1 Introduction

With the advent of in-memory computing, such as data analytics, key-value stores, and graph processing, the demand for high-density DRAM has been steadily increasing in recent years [27]. However, due to the challenge of scaling DRAM density, a new class of memory has received attention to bridge the performance gap between DRAM and SSD. For example, Intel recently unveiled its non-volatile memory based on 3D Xpoint technology, called Optane DC Persistent Memory Module (`DCPMM`) that provides more density than DRAM while outperforming flash-based SSDs [23]. Cloud vendors such as Google, Oracle, Microsoft, and Baidu have adopted such storage class memory (SCM) in their cloud services [4, 14, 17, 25].

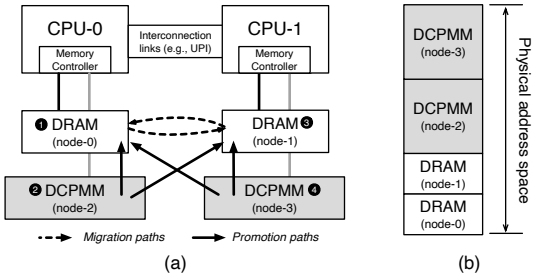



Figure 1: Software-managed tiered memory system augmented on the NUMA architecture

Since modern server systems are built with the Non-Uniform Memory Access (NUMA) architecture, future large-memory systems will take the shape of tiered memory augmented on traditional NUMA architecture, called *multi-tiered memory*. Figure 1 presents a real-world multi-tiered memory system used throughout this study. Each compute chip has two types of memory: DRAM (`upper-tier`) and Intel's DCPMM (`lower-tier`). We configure both DRAM and DCPMM to be fully exposed to software as memory.

This paper presents that the recent advancement in Linux [15] and tiered memory studies [16, 20, 35] do not lead to optimal page placement in multi-tiered memory systems. As the new class of memory becomes part of the main memory, the critical factor in performance is not only the *access locality* but also the *access tier* of memory. However, current page placement schemes have been established for DRAM-only NUMA architecture and only consider locality between threads and memory [2, 8, 12, 13, 21, 38]. As a result, the current design is far from exploiting the potential benefits of multi-tiered memory systems. For example, suppose the local DRAM becomes full when promoting pages from the lower-tier (DCPMM) to the upper-tier (DRAM) memory. In this case, the current state of the art leaves the page on the lower-tier memory, regardless of the availability of the remote DRAM (of the upper-tier). Such a decision is reasonable for DRAM-only NUMA systems because there is no difference between alternatives. However, in multi-tiered memory sys-

tems, we cannot consider every possible alternative equivalent to the *access tier*. When placing pages, the access tier of memory should be considered before the access locality because the access tier has a more significant impact on performance.

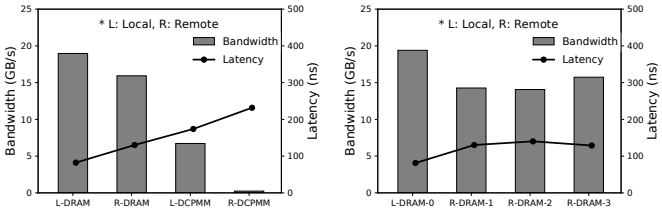
This limitation motivates us to revisit the page management schemes of the commodity OSes and explore the design space of page management for multi-tiered memory systems. In this study, we introduce a set of new page management schemes. Our first scheme, called *AutoTiering-CPM*, conservatively looks for promotion or migration alternatives using the access tier and locality metric when failing to find the best memory node (e.g., local DRAM).

Although this conservative approach can achieve better performance by considering alternatives, such a design does not unlock the full potential of software-managed tiered memory. To effectively utilize the limited capacity of upper-tier memory, we design a page reclamation scheme tailored to multi-tiered memory systems. Our second technique is opportunistic page promotion or migration, called *AutoTiering-OPM*, which judiciously demotes pages from the upper-tier memory. To reclaim effective, we predict the *least accessed page* as a victim in the upper-tier memory by estimating the access frequency of pages. When deciding on which page to promote, our OPM compares the page with the victim to determine which is relatively more accessed. With OPM, we can achieve better effectiveness of the upper-tier memory while reducing the memory accesses to the lower-tier memory.

Unless there is a free space in the upper-tier memory, a promotion operation waits until the completion of a demotion operation. To hide the latency of demoting pages from the critical path, we reserve a set of free pages in the upper-tier memory to serve the promotion requests immediately. When the number of reserved pages exceeds a threshold, our *kdemoted* wakes up and reclaims the least accessed page to the free page pool in the background. kdemoted differs from the traditional reclamation because it is only responsible for demoting pages to the lower-tier memory and not storage.

In this study, we implement our proposed schemes on top of Linux kernel v5.3. We take advantage of the AutoNUMA facility, which periodically scans memory pages and marks them inaccessible to capture non-local DRAM accesses. Once the pages are reaccessed, it incurs a page fault, called *NUMA hinting page fault*. We take the NUMA faults as demand signals for the page promotion from the DCPMM nodes or the migration from the remote DRAM node. We build the access history per page with the fault-based facility and use this information when demoting pages.

The experimental results show that our *AutoTiering* can significantly improve the performance of various applications. `GraphMat` and `graph500` show performance increases around 2.3× and 6.9×, respectively, compared with the baseline Linux kernel [15]. Most of the SPECAccel workloads show a 2× speedup. Compared to Intel's recent approach [36], our performance improvement shows up to a 3.5× speedup.




(a) Multi-tiered memory (b) DRAM-only memory

Figure 2: Memory access latency and bandwidth for multi-tiered and DRAM-only memory systems

## 2 Background and Motivation

### 2.1 Large Memory Systems

Data centers typically employ multi-chip NUMA architecture to scale up the performance of commodity servers with high core counts and memory capacity. Although this can increase the number of DIMM slots per server, scaling DRAM density is still a significant obstacle. It poses challenges in cost-effectively constructing large memory systems. Meanwhile, since SCM offers byte-addressable and non-volatile properties, it is gaining traction to bridge the performance gap between DRAM and SSD. Intel recently released the 3D XPoint non-volatile memory (DCPMM) that can be installed on DIMM without modification [23]. Many cloud vendors such as Google, Microsoft, Oracle, and Baidu have adopted Intel's DCPMM in their cloud services [4, 14, 17, 25]. Recently, Samsung has revealed a CXL (Compute Express Link) based DRAM module attached to the system, forming tiered memory systems [1]. Since such new type of memory is not as fast as DRAM, they cannot replace DRAM entirely. Instead, future computer systems will offer a form of tiered memory architecture with DRAM and SCM.

In this study, we take advantage of DCPMM as a new tier between DRAM and SSD. Intel DCPMM provides two types of tiered memory systems that can be categorized as `hardware-assisted` or `software-managed`. In hardware-assisted mode, DCPMM is exposed to software as the main memory while DRAM acts as a hardware-managed cache, non-visible to the software. The memory controller automatically places frequently accessed data on the DRAM cache, while the rest of the data is kept on a large capacity but slow DCPMM. On the other hand, with the operating system support, both DRAM and DCPMM can be exposed as normal memory and visible to software, tiering memory into fast and slow [15]. We call this a software-managed tiered memory system. In this environment, operating system support is supposed to effectively use both DRAM and DCPMM because the full control is given to software. This paper focuses on system software aspects of tiered memory systems by understanding how the hardware is organized.

## 2.2 Performance Characteristics

We describe the distinct performance characteristics of a software-managed tiered memory system that runs with the Linux operating system. Figure 1 presents the system organization used in this study. There are two CPU sockets in the system. For each CPU socket, one DRAM node and one DCPMM node are attached. The entire physical address space is comprised of both the DRAM and DCPMM nodes.

In multi-tiered memory systems, the critical factors in performance are not only the *access locality* but also the *access tier* of memory. Figure 2a shows read access latency and bandwidth for each of the four memory nodes measured from `MLC` [18]. Accessing the local DRAM outperforms the other three memory nodes, which is well established in traditional NUMA architecture. On the other hand, we observe that local DCPMM (`L-DCPMM`) is slower than that of remote DRAM (`R-DRAM`) due to the device characteristics. This is in stark contrast with the conventional wisdom that local memory is always faster than remote memory. Note that we also observe a similar pattern in a bandwidth measurement.

Similarly, Figure 2b shows the same type of evaluation over the four CPU (Intel Xeon Gold 6242) sockets with DRAM-only systems. There is no significant difference in latency and bandwidth for access to any remote DRAM nodes. Due to this, placing pages on the remote DRAM nodes in DRAM-only systems is a relatively simple task.

These distinct characteristics motivate us to explore the design space of page management in operating systems. The operating systems need to have the ability to (re)locate memory efficiently and dynamically by understanding the performance characteristics of multi-tiered memory systems. Unlike the DRAM-only systems, not all the remote memory nodes can be considered equal due to the access tier.

## 2.3 OS Support of Multi-Tiered Memory

The multi-tiered memory hierarchy that is the focus of this paper is distinct from traditional two-tiered memory. Nevertheless, the current Linux still relies on the existing NUMA framework to support multi-tiered memory systems. Such limited OS support makes the page placement sub-optimal in multi-tiered memory architectures. Although we can redefine the NUMA distance table according to the access latency, it does not exploit the full potential of the multi-tiered memory systems. First, Linux classifies memory nodes as either local or remote in a binary way. When promoting or migrating pages, several alternatives among the remote nodes are not considered at all. Second, the Linux does not support demoting (or reclaiming) pages from the upper-tier to the lower-tier memory. In this study, we revisit page placement strategy by considering performance characteristics across access-tier as well as access-locality.

# 3 Analysis of Page Management to Multi-Tiered Memory Systems

In this section, we investigate existing page management techniques of Linux that have been designed for DRAM-only NUMA systems. Then, we identify the lack of sufficient support for tiered memory systems. Although AutoNUMA [33] can be used for such multi-tiered memory, we observe that it fails to take full advantage of the multi-tiered memory.

## 3.1 Initial Page Placement

With the introduction of storage class memory (e.g., `Optane DCPMM`) in main memory, conventional page placement based only on the access locality has a negative impact on performance because the most critical factor in performance is not only the locality but also the memory tier. Meanwhile, current page placement schemes in operating systems have been well established for DRAM-based NUMA architecture, considering the access locality *only* between threads and memory [8, 13, 21]. In Linux, the default page allocation policy tries to use local memory as much as possible to minimize the performance penalty incurred by accessing remote memory. Only if there is no free space in the local memory, the memory allocator looks for free space on a remote memory node known as a `fallback` path [9].

As a result, the default (`local-first`) allocation policy is considered harmful in multi-tiered memory systems. The numbers in Figure 1a present the order used in the default fallback path when a thread runs on CPU-0, considering only the physical distance. If the `local-DRAM` node does not have enough free space, the memory allocator examines the fallback path to determine which memory node the allocation request should be sent. We anticipate that the allocator should ask the `remote DRAM` node to get a free page because this node provides better performance than the local DCPMM node. Surprisingly, however, the fallback path in the state-of-the-art Linux kernel indicates the local DCPMM (lower-tier). It had not taken into account the distinct characteristic that the memory type is more sensitive to performance than the access locality in multi-tiered memory systems.

**Problem:** If the local DRAM is full, the fallback prioritizes allocating from the local DCPMM (lower-tier), even though the remote DRAM (upper-tier) performs better.

## 3.2 Dynamic Placement

Although initial page placement plays a crucial role in the given memory space, the decision may not represent optimal performance because it depends on the memory access traffic at runtime. To adjust the placement of pages according to access patterns, the AutoNUMA facility has been included as part of Linux, automatically migrating pages to a mem-

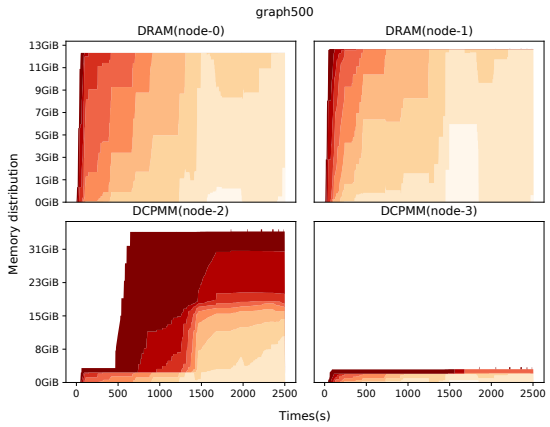



Figure 3: Page distribution and access intensity of `graph500` across memory nodes (Darker colors refer to pages that are accessed relatively more frequently.)

ory node closer to the thread running at runtime [33]. The operating system examines the access locality to find whether the accessed page is placed on the local memory or remote memory. If this is on the remote memory, the page is migrated to the local memory to avoid the remote accesses for subsequent requests. This approach improves the performance of applications running on DRAM-only NUMA systems.

However, we find that the current design does not exploit the advantages of the multi-tiered memory hierarchy. Figure 3 presents memory usage across the memory nodes for `graph500` with the 128GB dataset as time goes by. The detailed experimental setup is explained in Section 5. First, we notice that the upper-tier memory is *ineffectively* used because more frequently accessed pages (dark red) mainly reside in the lower-tier memory (`node-2`). In contrast, less frequently accessed pages are placed on the upper-tier memory [1]. The primary reason for this is that the current memory management does not allow page promotion or migration to the upper-tier memory when there is no free space. Although such a design decision is reasonable for DRAM-only systems, we need to reconsider this assumption for multi-tiered memory systems. Figure 4 depicts three cases whereby AutoNUMA encountered page promotion or migration failure due to lack of free space in local DRAM.

Even though the page promotion or migration cannot be made to the best memory node satisfying the access tier as well as the access locality, there are effective alternatives to placement in multi-tiered memory systems. When page promotion fails from remote DCPMM to local DRAM, for example, we have two possible workarounds: placing the page either on the remote DRAM or on the local DCPMM.

**Problem:** Pages in the lower-tier are not promoted when the upper-tier is fully utilized.

[1] Section 4.2.1 explains how we estimate the access frequency.

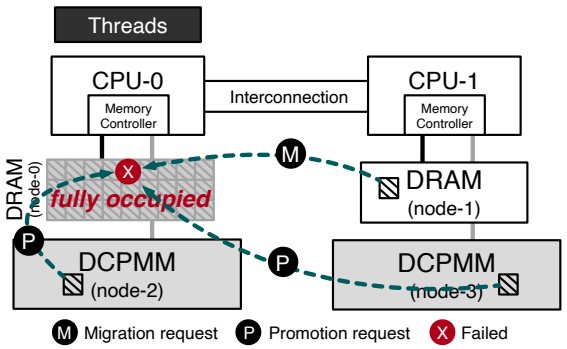



Figure 4: Page promotion and migration failure cases (`Promotion`: lower-tier → upper-tier, `Migration`: movement between the same tier)

Second, we observe the *skewed* page distribution across the memory nodes in the lower-tier memory. This is because the page movement to CPU-less nodes (`DCPMM`) is not considered in the current Linux operating systems. Since the traditional OSes were designed under the assumption that memory access performance is highly affected by the access locality between CPU and memory nodes, moving pages to CPU-less nodes does not occur. Only if the destination upper-tier memory has free space, the pages residing in the CPU-less node (lower-tier memory) can be promoted through AutoNUMA. Figure 1a shows arrows all the possible page promotion and migration paths that the stock Linux kernel currently supports. Even though the upper-tier memory is full, so that the operating system cannot place pages on the preferred access tier, we need to be able to preserve the access locality in the lower-tier memory by freely allowing page movement across any CPU-less nodes.

**Problem:** Pages are never migrated to the CPU-less (lower-tier) nodes due to a NUMA policy that does not apply to multi-tiered memory systems.

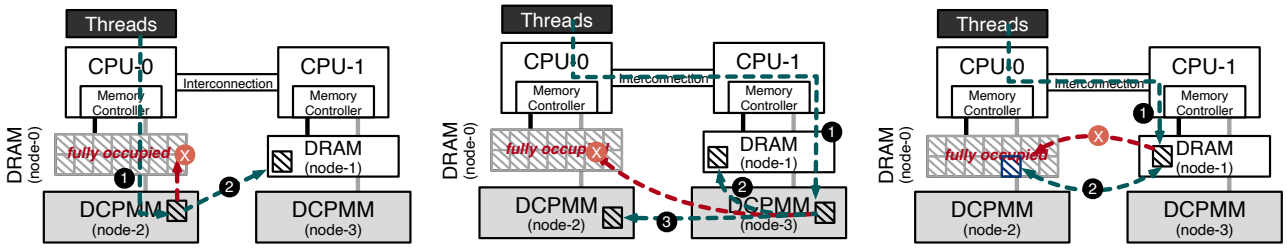
## 3.3 Page Reclamation

The current page reclamation is also designed for DRAM-only systems backed by storage-based swap devices rather than tiered memory systems. Traditionally, `kswapd` reclaims the inactive pages in memory directly to the storage regardless of the memory tier when the memory node is exhausted. It would make sense to reclaim pages in the lower-tier memory to the storage device. However, this is not a desirable solution for the upper-tier memory pages when the lower-tier has enough space. This limitation is intertwined with the two problems explained in Section 3.2.

**Problem:** Frequently accessed pages from the lower-tier cannot be promoted without demoting less frequently accessed pages from the upper-tier.

| Problem / limitation | Our solution | Section |
|---|---|---|
| Allocation fallback does not consider the access tier<br>Pages are not promoted when upper-tier is full | Promotion or migration to alternatives | 4.1 |
| Pages are never demoted or reclaimed to lower-tier memory | Demotion for the least-accessed pages | 4.2 |
| Page classification is too coarse-grained (binary) | Fine-grained access history estimation | 4.2.1 |
| Page reclamation to lower-tier & to storage needs to be decoupled | Foreground promotion & background demotion | 4.3 |

Table 1: Problems with current page management implementation, and our solutions




(a) Page promotion (LDCPMM → RDRAM)

(b) Page promotion or migration (RDCPMM → RDRAM or LDCPMM)

(c) Page exchange between DRAM nodes

Figure 5: Our conservative design: exploiting multi-tiered memory hierarchy (L: Local, R: Remote)

In Linux operating systems, part of the virtual address space for each process can be mapped, either the `file-backed` or the `anonymous` region. The pages in the file-backed region contain the contents of an existing file(s) on memory so that subsequent file I/O operations of the same file can be replaced with memory access. On the other hand, pages belonging to the anonymous region do not represent any file contents. This is used for keeping arbitrary data on memory (e.g., `malloc`). For each memory region, the Linux operating system classifies memory pages into `active` or `inactive`. The kernel has inactive lists containing pages that might not be in use while keeping recently accessed pages on the active list. However, the current page classification is too conservative to precisely differentiate which pages are frequently or less frequently accessed from the active and inactive lists.

**Problem:** Binary page classification (either active or inactive) is too coarse-grained to be used for tiering.

Last, current page reclamation is very carefully performed in the background to hide the cost of accessing the storage devices from the critical path. In tiered memory systems, however, the cost of demoting pages to the lower-tier memory is cheaper than that of swapping out pages to the storage. We need to decouple the demotion to the lower-tier memory from traditional reclaim to the storage.

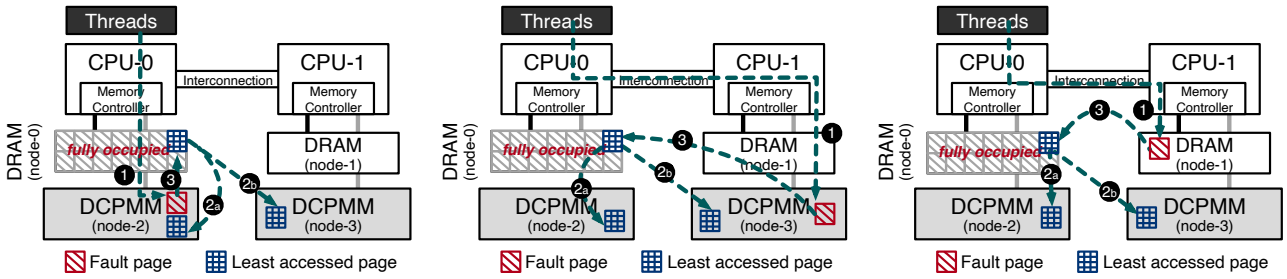
# 4 Automatic Multi-Tiered Memory

This section explores the design space of page management to tiered memory systems. The goal of our page management is to extract the full advantage of multi-tiered memory to improve the performance of large-memory applications. To keep our design simple, we base our design on the AutoNUMA facility. Table 1 summarizes the supported mechanisms for each design space. In the following subsections, we explain the design and implementation of each scheme.

## 4.1 Exploiting Multi-Tiered Memory

As depicted in Figure 4, we take the NUMA fault as a demand signal for page promotion or migration from the DCPMM nodes or the remote DRAM node. Note that the current AutoNUMA deals with the promotion and migration requests only if the upper-tier (local DRAM) memory has free space. Otherwise, the request is discarded, and the faulted page remains in the original memory. When the local DRAM is fully occupied, our proposed design allows the pages to be promoted or migrated into the next best memory node in the multi-tiered memory hierarchy. This approach can exploit the advantage of the multi-tiered memory hierarchy, providing higher performance than the stock Linux kernel.

Figure 5 describes how we react when the local DRAM is full. There are three sources of demand for page migration or promotion to local DRAM in the multi-tier memory system. The multi-tier hierarchy opens up new opportunities to design memory placement. First, (5a) when the faulted page resides in the local DCPMM (❶), we promote the page to the remote DRAM as the second best location (❷). Since the remote DRAM provides lower latency and higher bandwidth than local DCPMM, we can improve the performance of applications to which memory was initially allocated in the lower-tier memory. Second, (5b) if the faulted page resides in the remote DCPMM (❶), we have two alternatives to ex-




(a) Eviction & Promotion (LDCPMM → LDRAM)

(b) Eviction & Promotion (RDCPMM → LDRAM)

(c) Eviction & Migration (RDRAM → LDRAM)

Figure 6: Our progressive design: opportunistic page promotion and migration with page eviction (L: Local, R: Remote)

ploit the advantage of the multi-tier memory hierarchy. We attempt to promote the page to the remote DRAM (❷). If the remote DRAM also does not have free space, we try to migrate the page to the local DCPMM (❸). Unlike the stock Linux kernel, our modified kernel supports migrating the page to CPU-less nodes (`local DCPMM`). We call this *AutoTiering Conservative Promotion and Migration (CPM)*. Last, (5c) the faulted page is in the remote DRAM (❶). This means that the page is already in the second best location. The existing AutoNUMA is unable to complete the page migration operations between the upper-tier memory nodes. As a result, it leads to sub-optimal performance. In such a case, we consider the page exchange option to satisfy the demand for memory affinity (❷). The prior study proposed the page exchange mechanism for tiered memory [35]. We repurpose the mechanism to resolve the migration failures between the same tier memory nodes as well. Internally, for each memory node, we keep track of which pages fail to be migrated. We then leverage this information to determine the migration demand that can be resolved with the exchange operation. We call this *AutoTiering CPM with Exchange (CPMX)*.

Since this design does not require significant changes in the existing Linux operating system, it is easily integrated on top of the AutoNUMA facility. We anticipate that our conservative design can be a practical solution for such software-managed tiered memory systems.

## 4.2 Opportunistic Promotion and Migration

As we design a conservative approach for finding the best alternative, this is limited to extracting the full performance benefit of software-managed tiered memory. In our conservative design, frequently used pages can reside in the lower-tier (DCPMM) memory while the upper-tier (DRAM) memory holds infrequently accessed data. To relieve such undesirable memory placement, we explore a progressive strategy, opportunistically demoting a page from the upper-tier memory to create free space. This is the main difference between conservative and progressive designs. By demoting a page, the request for page promotion can be successful. For page demotion to be effective, we need to have the ability to select a page that is highly unlikely to be reused within a short time. Otherwise, the wrong selection can have a negative impact on performance. We explain how we select a page for the demotion in the following subsection (4.2.1).

Figure 6 depicts how our progressive approach works with page demotion. When the NUMA page fault occurs (❶), we find the least accessed page from the upper-tier (`local DRAM`) memory and compare the access frequency of the least accessed page with the faulted page. If the selected page is relatively less frequently accessed than the faulted page, we demote the selected page to place the faulted page on the higher-tier memory node. Otherwise, we prevent page promotion or migration requests to keep the upper-tier memory with more frequently accessed pages.

To promote a page from either local DCPMM (6a) or remote DCPMM (6b), we demote the least accessed page selected to the lower-tier memory (❷a or ❷b). The destination of the demoted page depends on where the page was previously accessed to preserve the locality. After that, (❸) we can finally promote the page to the local DRAM node. In addition, Figure 6c shows how the page migration request from the remote DRAM is made. We call this *AutoTiering Opportunistic Promotion and Migration (OPM)*.

We further optimize our progressive design by fusing demotion and promotion (or migration) into one exchange operation, called *AutoTiering OPM with Exchange (OPMX)*. When the destination of the demotion is equal to the source of the promotion or migration, we leverage the exchange operation instead of individual promotion and migration. For example (6a), if we need to demote a selected page into the local DCPMM (❷a) and promote the page to the local DRAM (❸), two individual operations are fused. The exchange operation eliminates unnecessary page allocation and free operations.

In case we cannot find a page to be demoted from the upper-tier memory, we try to promote the page to the next best location - the remote DRAM - as would normally be the case. Since we are conducting page promotion and migration opportunistically, we can reduce excessive page promotion

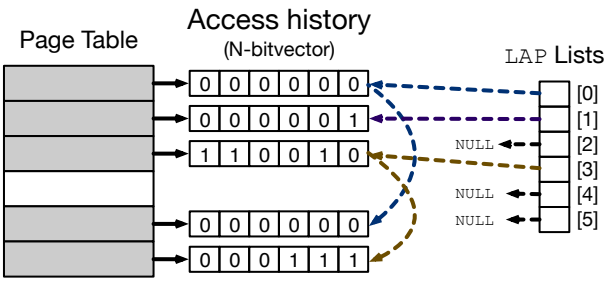



Figure 7: Maintaining least accessed page lists

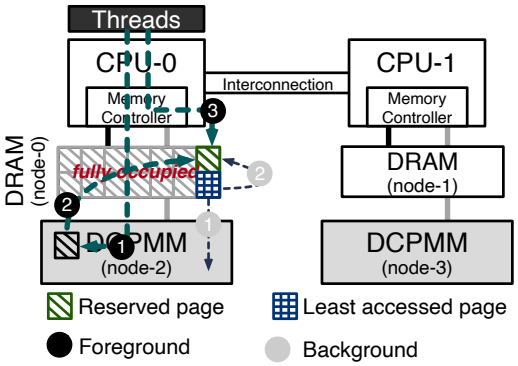



Figure 8: Hiding latency of page demotion with our `kdemoted`

and demotion that incurs performance overhead associated with page table manipulation and TLB shootdowns.

#### 4.2.1 Predicting the Least Accessed Page

To make the progressive design effective, our goal is to find the least accessed page from the upper-tier memory. As explained in Section 3.3, the Linux operating system separates memory into file-backed and anonymous pages as LRU lists. When page promotion or migration fails due to lack of free space on the upper-tier memory, we investigate the pages from the file-backed region preferentially and move on to the anonymous region if we are unable to find the least accessed page from the file-backed region.

① **File-backed pages:** We examine whether we can make free space by demoting a page belonging to the file-backed region. As file-backed pages are maintained in two LRU lists, active and inactive, we regard the oldest page in the inactive list as the least accessed page. Whenever the file-backed pages are reaccessed (e.g., `sys_read` or `sys_write`), the operating system marks the page as accessed and moves it into the active list. Since the operating system can track the access of file-backed pages, we estimate the least accessed page by looking at the inactive list. If not, we move on to the active list. Note that we preserve the portion of the page cache configured in the kernel parameter (e.g., `vfs_cache_pressure`). If we cannot find a reusable space in the file-backed region, then we look for a page in the anonymous region as a fallback path.

② **Anonymous pages:** On the other hand, we keep the access (fault) information per page to select the least accessed page in the anonymous region judiciously. To minimize the monitoring overhead, we leverage the page scan facility used for AutoNUMA, keeping track of whether the pages are accessed or not during a given time window. Then we build the access history for each page with an `N` bit-vector. This means that we maintain up to the last `N`-time access history. We set `N` to 8. Based on the access history, we classify the pages into `N` levels (`Least Accessed Page` lists), where N is the number of bits that are set, as shown in Figure 7. Once page demotion is required, we find one of the pages in the `LAP[0]` list because those pages have not been accessed the last `N` times. If the `LAP[0]` list is empty, then we try to find a page from the `LAP[1]` list, and so on. After that, we can select the least accessed page in the upper-tier memory and conduct page demotion to the lower-tier memory.

### 4.3 Hiding Latency of Page Demotion

To enforce page promotion, we are supposed to demote a page from the target node. Before completing the page demotion, we cannot proceed with the promotion request due to a lack of space. The fault handling time is the sum of the two operations in the critical path: page demotion and promotion. To remove page demotion from the critical path, we explore a software optimization technique.

We keep a page pool of a few reserved pages. We empirically determine the reserved pages to 16 and 4 for 4KB and 2MB, respectively. The reserved pages allow us to immediately serve the promotion request without requiring the demotion process, even though the upper-tier memory is full. This approach is more cost-effective than the page exchange scheme [35] because it hides the latency of the page demotion in the critical path. Page demotion to the lower-tier memory takes longer than page promotion to the upper-tier memory because the storage-class memory used for lower-tier memory provides better read performance than write performance. To efficiently demote pages in the background, we maintain a new kernel thread called `kdemoted`, demoting the least accessed pages in a batch. Once the number of reserved pages reaches below a certain threshold, we wake the kernel thread to start the demotion process. The threshold is set to 4 through sensitivity studies.

Figure 8 depicts how simple optimization hides the latency of the page demotion. For every promotion request (❶), the page can be promoted even when the upper-tier memory is full (❷). After completing the promotion, the NUMA fault handler is returned without the demotion process, and future accesses to the page (❸) will take place on the upper-tier memory. Meanwhile, `kdemoted` demotes the least accessed pages as needed in a batch (①) to reclaim the memory pool (②). We call this *OPM-BD (Background Demotion)*.

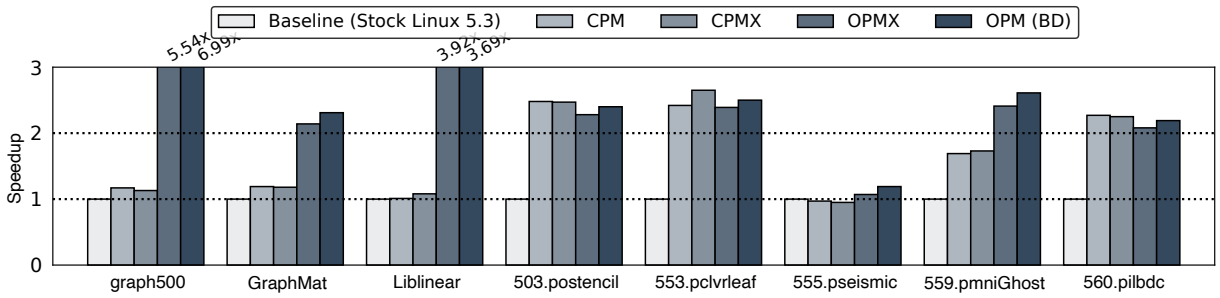



Figure 9: Speedup of our conservative (`CPM` and `CPMX`) and progressive (`OPMX` and `OPM(BD)`) schemes

# 5 Evaluation

## 5.1 Experimental Setup

To evaluate our proposed scheme, we use a NUMA server equipped with two Intel Xeon Gold 5218 processors and compose a multi-tiered memory hierarchy with a 16GB DDR4-2666 DIMM and a 128GB Intel Optane DC Persistent Memory (DCPMM) for each CPU socket. The server system has a total of 32GB DRAM and 256GB DCPMM as the main memory. To minimize measurement variability, we disable HW features, including Hyper-Threading, DVFS, Turbo-Boost, and prefetches. We use the Linux kernel 5.3 and Ubuntu 18.04 server as our baseline and implement our proposed schemes on top of the kernel. Our code is available at `https://github.com/csl-ajou/AutoTiering`. We run benchmarks from graph500, SpecACCEL (OpenMP), GraphMat [32], and Liblinear [22] used in recent large memory systems [2, 35]. We configure them for all the benchmarks to use all 32 cores across two sockets and more than 64GB of memory to sufficiently stress the multi-tiered memory system. Since the page size can affect performance in various ways, we evaluate performance for the large page (2MB) as well as the base page (4KB).

## 5.2 Experimental Results

**Performance with our conservative approach (CPM):** Figure 9 presents the speedup results over the stock Linux kernel (first bar). Note that AutoNUMA is enabled in the default Linux kernel. The second bar in Figure 9 presents the speedup with our conservative promotion and migration (`CPM`), and the third bar shows performance changes when the conservative exchange is applied (`CPMX`). For most of the workloads we evaluate, we can see significant performance improvement with our `CPM`. In `503.postencil`, `553.pclvrleaf`, and `560.pilbdc`, the speedup is over 2x, compared to the baseline. Also, `559.pmniGhost` shows 1.6x performance improvement. Our conservative design (`CPM`) can promote pages from the lower-tier (DCPMM) memory nodes to the remote DRAM node even though the locality is not preserved because the remote DRAM is faster than the local DCPMM. We can also migrate pages between the two DCPMM nodes to better access locality when accessing the lower-tier memory. As a result, we can utilize the multi-tiered memory systems more efficiently, leading to performance improvement.

For `graph500` and `GraphMat`, we look at that the speedup is around 17% and 19%, respectively. `GraphMat` and `Liblinear` read the large dataset from file to its in-memory data structure. The file-backed pages occupy a significant portion of the DRAM nodes, while the crucial data structures reside in the DCPMM nodes. After that, it performs the Pagerank algorithm for analytics. Unfortunately, we could not observe that `kswapd` is invoked to free the inactive file-backed pages. Thus, our `CPM` and `CPMX` do not have the opportunity to utilize the DRAM nodes effectively.

On the other hand, we observe that the performance of `555.pseismic` is not improved at all. This shows that the memory placement is well balanced across the upper-tier and lower-tier memory in the baseline. As a result, we could not exploit the promotion and migration opportunities.

**Performance with our progressive approach (OPM):** We look at further improvement with our progressive design described in Section 4.2, which finds the least accessed page from the upper-tier (local DRAM) node and demotes that to the lower-tier memory. As shown in Figure 9, `OPMX` exhibits better performance than `CPMX` for most of the workloads. In particular, `graph500`, `GraphMat`, `Liblinear`, and `559.pmniGhost` show significant speedup over `CPMX`. The performance of `503.postencil`, `553.pclvrleaf`, and `560.pilbdc` is slightly degraded, by about 7 to 8%. `560.pilbdc` fails to amortize the overhead of page promotion and demotion, and `553.pclvrleaf` shows that the opportunity for page exchange is reduced.

We analyze the running behavior of `graph500` to understand performance improvement. Before performing the core graph algorithm, it generates a lot of intermediate data that occupies the DRAM space. As a result, the baseline and our `CPMX` spend most of the time accessing the lower-tier memory while running BFS (Breadth-First Search). Interestingly, it exhibits the memory access locality on a small part of the address space, even though the whole address space is huge. With `OPMX`, we can demote the less frequently accessed data

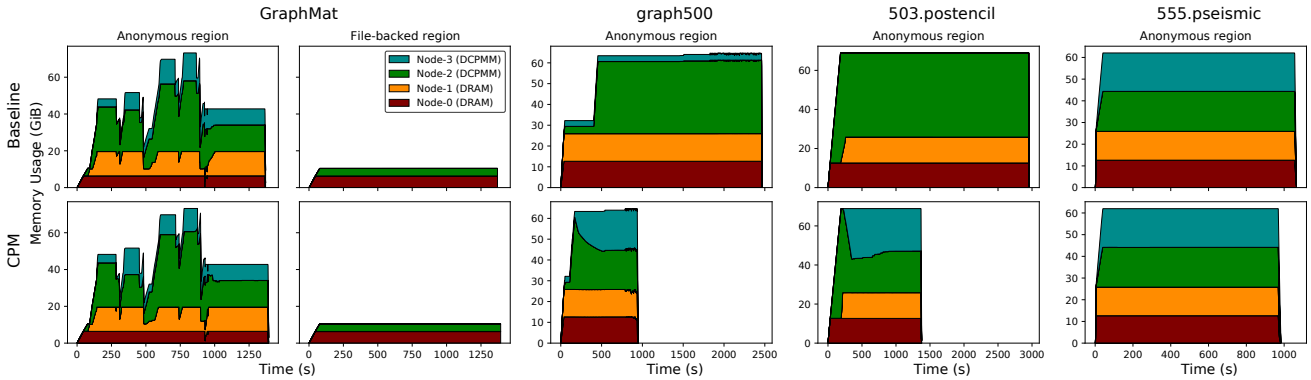



Figure 10: Memory usage across DRAM and DCPMM nodes (Baseline vs. CPM)

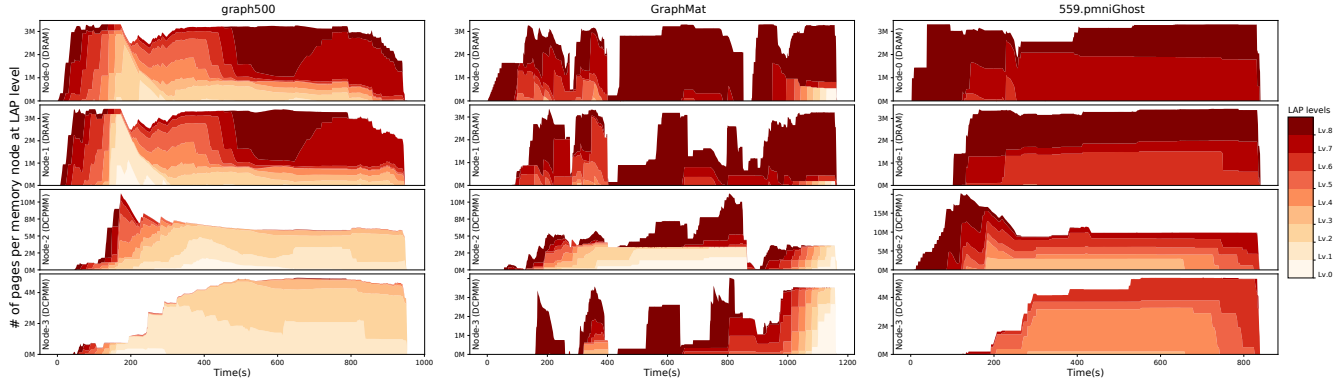



Figure 11: Estimated access frequency of pages corresponding to the LAP levels

to the lower-tier memory and utilize the upper-tier memory space for more frequently accessed data. Therefore, we can increase performance drastically.

For GraphMat, we see significant performance improvement comes from the demotion of file-backed pages. By default, the file-backed page is initially placed on the inactive list. After the page is reaccessed, it moves from the inactive to the active list. By demoting the least accessed page from the file-backed pages, we can improve the utilization of the upper-tier memory.

With OPM(BD), we can further improve the performance of most workloads. The improvement for graph500 and GraphMat is considerable compared to OPMX. 555.pseismic and 559.pmniGhost show marked improvement, and the other SPEC workloads also restore the degraded performance from the exchange version. Meanwhile, Liblinear shows degraded performance slightly.

**Distribution of memory usage:** We analyze how multi-tiered memory is utilized as time goes by. Figure 10 compares the memory usage for the baseline and our progressive design for selected workloads, which are beneficial from our CPM. For the three SPEC workloads, the lower-tier memory is more well balanced with CPM. This is because we allow pages to be migrated to the CPU-less node. Even though CPM is unable to satisfy the required access tier, it can preserve the access locality in the lower-tier memory.

For GraphMat, the performance improvement is relatively small compared to the SPEC workloads, and 555.pseismic is not improved. The reason is that the baseline already keeps the memory balance across lower-tier DIMMs. In the next paragraph, we look at the effectiveness of our OPM-based schemes for those two workloads.

**Effectiveness of LAP classification:** To evaluate the effectiveness of our LAP scheme, we decompose all the pages into each LAP level and build a histogram as time goes by. Figure 11 presents the selected three workloads that show significant improvement with OPMX compared to CPM. For graph500, GraphMat, and 559.pmniGhost, the relatively frequently accessed pages (dark red) corresponding to level 7 or 8 are placed on DRAM nodes, while DCPMM nodes serve the relatively less accessed pages. This result shows that our OPMX can be effective for applications whose working set fits in the upper-tier memory.

The additional space for keeping the access history (8b), two pointers for lists (16B), page frame number (8B), and last faulted CPU number (1B) is required for each page to enable our LAP scheme. Compared to the baseline, we require 32 bytes of metadata per page due to 8 bytes alignment, and it

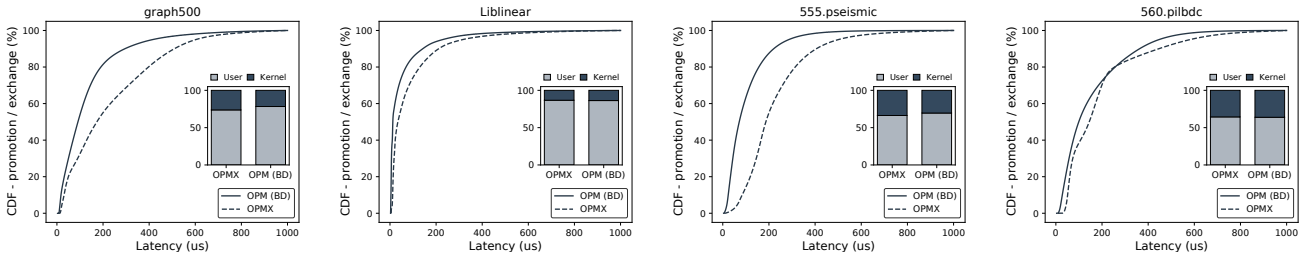



Figure 12: Cumulative distribution function (CDF) of page promotion and migration with `OPMX` and `OPM(BD)`

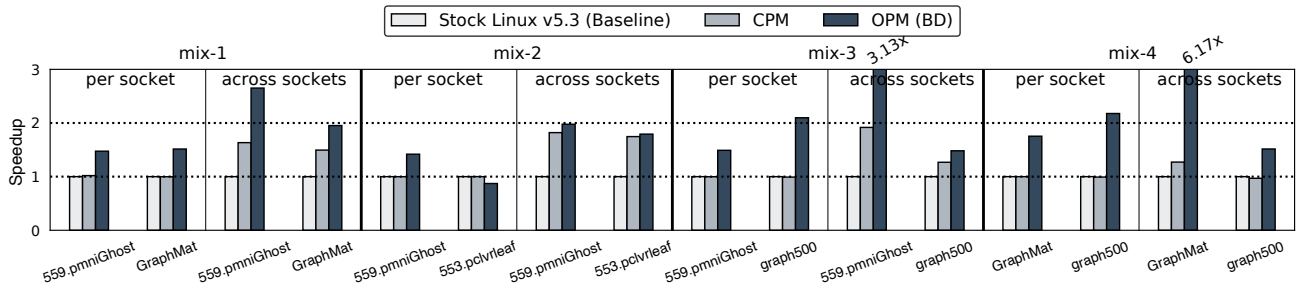



Figure 13: Speedup of multi-programmed execution scenarios

results in an extra 288MB for the system memory of 32GB DRAM with 256GB DCPMM. It slightly decreases the effective DRAM memory space by 0.91%.

**Latency hiding with background demotion:** We measure promotion and exchange latency distribution in the page fault path with `ftrace` and the kernel and user time. Figure 12 presents the latency CDF serving a page promotion for `OPM(BD)` and a page exchange for `OPMX`. The solid line is for the distribution, including the latency hiding scheme (`OPM(BD)`), and the dashed line is for the `OPMX` scheme. The fault latency varies for both schemes. We observe that `OPM(BD)` shows better latency distribution for all the workloads than the exchange version because the demotion is off the critical path. Especially, `graph500` and `555.pseismic` are beneficial to the background demotion.

On the other hand, the end performance of `Liblinear` is not improved with the background demotion shown in Figure 9. Even though the latency is reduced, the kernel execution time is not significantly changed. This is due to the possibility of incurring memory access contentions across the application threads and the background kernel thread. We plan to resolve the undesired background demotion case with rigorous scheduling of kdemoted in our future work.

**Performance with multi-programmed workloads:** We evaluate how well our proposed schemes work for multi-programmed workloads in two scenarios, `per socket` and `across sockets`. Figure 13 shows the speedup with `CPM` and `OPM(BD)` when two applications run on the same server. We mimic four multi-programmed scenarios (`mix-1` to `4`) with the combination of the workloads. When isolating the applications based on sockets (`per socket`), `CPM` does not provide any performance improvements as expected because it lacks the opportunity to exploit the multi-hierarchy of memory. On the other hand, we can observe significant performance improvement with `OPM(BD)` for all cases except for `553.pclvrleaf` in `mix-2` as more frequently accessed pages can be placed on the upper-tier memory. When allowing the applications to run across sockets, these may not effectively utilize the upper-tier memory nodes because the memory usage across the threads of applications is not evenly distributed. In the `across sockets` setting, we observe that `CPM` can be useful by exploiting the multi-tiered memory hierarchy. In addition, `OPM(BD)` can further improve performance, compared to `CPM`, for all the cases.

**Working-set sensitivity:** Figure 14 presents the performance by varying the working-set size from 32GB to 160GB for selected workloads. We observe that the performance of `graph500` and `559.pmniGhost` is significantly improved by both `CPM` and `OPM(BD)`. For `503.postencil` and `553.pclvrleaf`, however, `OPM(BD)` and `CPM` show similar performance improvements as the working-set size increases. As explained in the LAP classification paragraph, most of the pages are evenly accessed in those benchmarks less beneficial to `OPM(BD)`. Note that the speedup with our approach is significant and still effective compared to stock Linux (Baseline). Since the other workloads, such as `GraphMat` and `Liblinear`, have fixed problem input sizes, we could not evaluate the working-set sensitivity for the workloads.

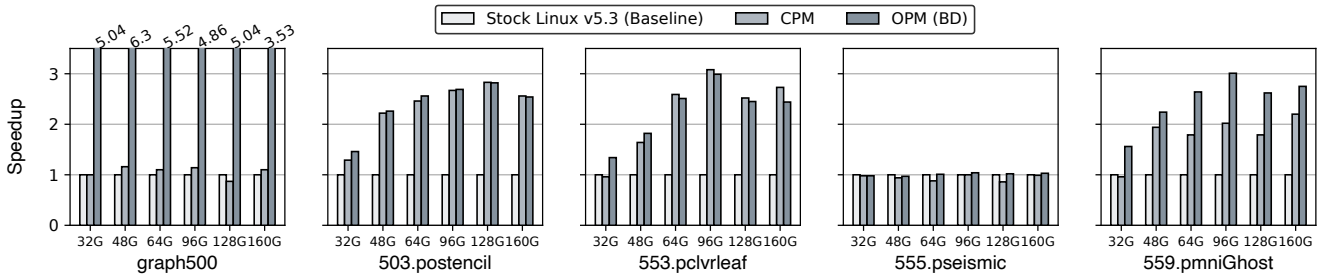



Figure 14: Performance speedup by varying the working-set size from 32GB to 160GB

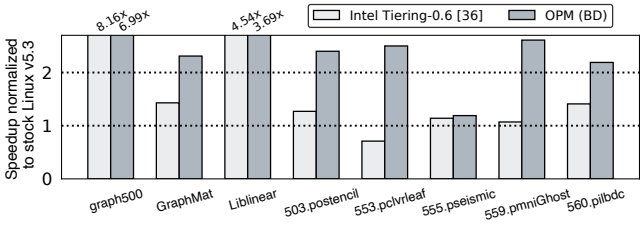



Figure 15: Performance comparison: `OPM(BD)` and `Tiering v0.6` [36]

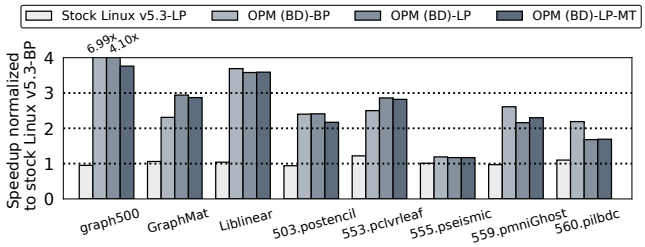



Figure 16: Speedup with large page (LP) over base page (BP)

**Performance comparison with prior studies:** Figure 15 presents the performance comparison results with a recent proposal from Intel called `Tiering v0.6` [36]. Note that `Tiering v0.6` is based on Linux kernel version 5.9 but not merged into the mainline. We show that our `OPM(BD)` outperforms the performance of `Tiering v0.6` for most of the workloads.

For the anonymous memory region, `Tiering v0.6` supports the page promotion to the upper-timer memory by extending the AutoNUMA framework. Through the monitoring facility, they investigate whether the page is accessed during the last two consecutive scans or not. If so, they consider the page is hot and promote it. On the other hand, our `OPM(BD)` maintains access history for the last N (8) times. The decision is more accurate than looking at the previous two accesses.

Besides, `Tiering v0.6` inherits the same limitation from the traditional AutoNUMA. Once the local DRAM becomes full, it is allowed to promote the page to neither the remote DRAM nor the local DCPMM. Instead, `kswapd` is triggered to reclaim pages to lower-tier memory. In contrast, our LAP scheme makes the upper-tier memory better utilized by opportunistically performing promotion and demotion.

For `graph500`, we observe that `Tiering v0.6` performs better than `OPM(BD)`. Our scheme outperforms the time for building graphs before execution, but while traversing the graph, `graph500` with `OPM(BD)` accesses more pages in the lower-tier memory compared to `Tiering v0.6`. In `Liblinear`, it shows that `Tiering v0.6` more aggressively demotes the file-backed pages, leading to better utilization of the upper-tier memory.

**Performance with large page:** To minimize the overhead of TLB misses for large memory applications, modern computer systems provide large page options. Figure 16 shows the speedup results when we leverage the large page (2MB) instead of the base page size (4KB). For `GraphMat` and `553.pclvrleaf`, the performance improvement with large pages can be observed in the `OPM(BD)` scheme. On the other hand, most of the workloads do not present a significant difference. `559.pmniGhost` and `560.pilbdc` exhibit degraded performance. When analyzing the LAP histogram for the SPEC workloads with the large page, our scheme can quickly separate pages into the LAP levels compared to the base page. However, there is performance degradation because the overhead of page demotion increases when moving the large pages. To reduce the overhead, we take advantage of the multi-threaded (MT) version of copying pages [35]. `559.pmniGhost` only shows improved performance, but the other workloads become even worse. As mentioned in the earlier paragraph (*Latency hiding*), we will further investigate how our scheme can be extended to mitigate the performance overhead in our future work.

## 6 Related Work

There have been significant efforts throughout hardware and software to use tiered (or heterogeneous) memory systems effectively. We compare our scheme with prior approaches. First, most previous studies focused on designing page migration mechanisms and policies on two-tiered memory systems [3, 10, 11, 19, 20, 24, 26, 29, 35]. Unlike the prior work, this study extends the problem space to the tiered memory

augmented on traditional NUMA architecture, called multi-tiered memory systems. As there are several alternatives to page placement in multi-tiered memory systems, this paper exploits such opportunities when placing, promoting, and demoting pages. Second, *AutoTiering* does not differentiate pages into hot and cold with a predefined threshold used in prior work [3, 19, 20]. In this study, promotion and demotion decisions are made on the relative access frequency and recency across the memory tier. To estimate page access activities, we rely on the AutoNUMA facility. Last, we use a real-world infrastructure to evaluate our proposed ideas based on Intel's Optane Persistent Memory (DCPMM), which has attracted recent attention. Prior work emulated or simulated two-tiered memory systems based on DRAM [3, 11, 19, 20, 35]. Although real-world storage-class memory (SCM) shows asymmetry in read and write performance, it was not correctly modeled in emulating tiered memory with DRAM.

Below, we describe previous software efforts in the OS community. To understand the heterogeneity of memory systems in Linux, the ACPI 6.2 specification introduced heterogeneous memory attributes tables (HMAT) to provide users with performance information for various memory types [39]. Since the Linux kernel 5.0-rc1, the persistent memory (here, Intel's DCPMM) can be used as volatile main memory, although it is slower than DRAM [15]. It can provide abundant main memory space, but the policy and mechanism supporting tiered memory hierarchy are in infancy. Recently, a new memory allocator for hardware-managed DRAM cache known as shuffle page allocator introduced in the Linux kernel [34]. However, it is not enough to extract the full performance of tiered memory because it does not consider the distance between memory nodes and NUMA typologies. Also, there have been efforts to efficiently support page migration between fast and slow memory, but these still rely on active and inactive list management [5, 16, 30] and have not been merged into the mainline of the Linux kernel until now. In the Windows operating system, they measure the cost of various page operations when the system is initialized through `MiComputeNumaCost` and build a table like the NUMA distance of Linux, but this reflects the access costs [37]. Due to limited information for Windows OS, we could not find how they work for multi-tiered memory.

Researchers in the architecture community introduced hardware techniques to effectively utilize two-tiered memory systems while minimizing the performance overhead in estimating access frequency [7, 28, 29, 31]. Choe et al. suggested memory allocation schemes for the hardware-assisted multi-tiered memory systems where the DRAM nodes are invisible to software [6]. Except for that, none of the work considered the case of multi-tiered memory systems. The overheads such as tracking and migrating pages can be reduced significantly by architectural support, but the flexibility of making decisions for promotion and demotion considering placement alternatives is limited.

## 7 Conclusion

This work explored a set of new page management schemes called *AutoTiering*, which benefit from multi-tiered memory systems. We found that the Linux operating system focuses on the access locality posed by NUMA, rather than the memory tier. However, in multi-tiered memory systems, the cost of accessing memory is not proportional to solely the locality. We comprehensively addressed the diverse aspects of utilizing the multi-tiered memory systems by considering two factors: the access tier and locality. We built a proof of concept with a real-world tiered memory system. Our evaluation showed significant performance improvements in various benchmarks than the stock Linux kernel version 5.3 and the previous approach from Intel's Tiering v0.6.

Future tiered-memory systems are expected to be more diverse and heterogeneous. To make our approach more general, we can maintain a table describing possible alternatives to placement. While initializing memory in the operating system, we can measure actual performance across memory nodes. With such a new table, *AutoTiering* can adjust where pages need to be promoted, demoted, or migrated adaptively, as explained, without a static decision.

## Acknowledgments

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

## References

[1] Samsung unveils industry-first memory module incorporating new cxl interconnect standard, 2021. https://bit.ly/3uBo27J.

[2] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[3] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[4] Microsoft Azure. Introducing new product innovations for sap hana, expanded ai collaboration with sap and more, 2019. https://azure.microsoft.com/ko-kr/blog/introducing-new-

product-innovations-for-sap-hana-expanded-ai-collaboration-with-sap-and-more/.

[5] Keith Busch. Page demotion for memory reclaim, Mar 2019. https://lwn.net/Articles/783672/.

[6] Wonkyo Choe, Jonghyeon Kim, and Jeongseob Ahn. A study of memory placement on hardware-assisted tiered memory systems. *IEEE Computer Architecture Letters (CAL)*, 19(2), 2020.

[7] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[8] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[9] Linux Kernel Documentation. What is numa?, June 2019. Available at https://www.kernel.org/doc/Documentation/vm/numa.

[10] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A hybrid memory page scheduler with machine intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019.

[11] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[12] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (ATC)*, 2014.

[13] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern numa systems. *Communication of ACM*, November 2015.

[14] Google. Available first on google cloud: Intel optane dc persistent memory, 2019. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory.

[15] Dave Hansen. Allow persistent memory to be used like normal ram, Jan. 2019. https://lwn.net/Articles/776921/.

[16] Dave Hansen. [RFC] Memory Tiering, 2019. Available at https://lore.kernel.org/linux-mm/c3d6de4d-f7c3-b505-2e64-8ee5f70b2118@intel.com/.

[17] Intel. Baidu feed stream services restructures its in-memory database with intel optane technology, 2019. https://newsroom.intel.com/wp-content/uploads/sites/11/2019/08/baidu-feed-case-study.pdf.

[18] Intel. Intel memory latency checker v3.9, 2020. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[19] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[20] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[21] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2015.

[22] Chih-Jen Lin. Liblinear – a library for large linear classification. https://www.csie.ntu.edu.tw/~cjlin/liblinear/.

[23] Lily Looi and Jianping Jane Xu. Intel optane data center persistent memory. In *HotChips : A Symposium on High-Performance Chips (HotChips)*, 2019.

[24] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for nvm+dram hybrid main memory. In *Proceedings of*

*the 12th Conference on Hot Topics in Operating Systems (HotOS)*, 2009.

[25] Oracle. Oracle and intel collaborate on optane dc persistent memory performance breakthroughs in next generation oracle exadata x8m, 2019. https://www.oracle.com/corporate/pressrelease/oow19-oracle-intel-partner-optane-exadata-091619.html.

[26] Mark Oskin and Gabriel H. Loh. A software-managed approach to die-stacked dram. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[27] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramcloud. *Communication of ACM*, 54(7), July 2011.

[28] Moinuddin K. Qureshi and Gabe H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[29] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2011.

[30] Yang Shi. Migrate mode for node reclaim with heterogeneous memory hierarchy, Jun 2019. Available at https://lwn.net/Articles/791174/.

[31] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. Transparent hardware management of stacked dram as part of memory. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[32] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.

[33] Rik van Riel and Vinod Chegu. Automatic numa balancing, 2014. Available at https://www.redhat.com/files/summit/2014/summit2014_riel_chegu_w_0340_automatic_numa_balancing.pdf.

[34] Dan Williams. mm: Randomize free memory, January 2019. https://lwn.net/Articles/776228/.

[35] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[36] Huang Ying. tiering-0.6, 2020. Available at https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/commit/?h=tiering-0.6.

[37] Pavel Yosifovich, Mark Russinovich, David Solomon, and Alex Ionescu. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more (Developer Reference)*. Microsoft Press, 2017.

[38] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[39] Ross Zwisler. Add support for heterogeneous memory attribute table, June 2017. https://lwn.net/Articles/724562/.