MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems

Adnan Maruf*, Ashikee Ghosh*, Janki Bhimani*, Daniel Campello†, Andy Rudoff‡, and Raju Rangaswami*
*Knight Foundation School of Computing and Information Sciences, Florida International University
†Google, ‡Intel Corporation

***Abstract*—**
The rapid growth of in-memory computing powered by data-intensive applications has increased the demand for DRAM in servers. However, a DRAM-based system can be limiting for modern workloads because of its capacity, cost, and power consumption characteristics. Hybrid memory systems, which consist of different types of memory, such as DRAM and persistent memory, can help address many of these limitations. One promising direction that has been explored in the recent literature involves introducing persistent memory devices as a second memory tier that is directly exposed to the CPU. The resulting tiered memory design must address the fundamental challenge of placing the right data in the right memory tier at the right time while minimizing overhead. We present MULTI-CLOCK, an efficient, low-overhead hybrid memory system that relies on a unique page selection technique for tier placement. MULTI-CLOCK's page selection captures both page access *recency* and *frequency*, and enables moving pages to appropriate tiers at the right time within hybrid memory systems. We implemented a Linux-based, NUMA-aware version of MULTI-CLOCK that is entirely transparent and backward compatible with any existing application. Our evaluation with diverse real-world applications such as graph processing and key-value stores shows that MULTI-CLOCK can improve the average throughput by as much as 352% when compared with several state-of-the-art techniques for tiered memory.

I. Introduction

Over the last several decades, DRAM performance and capacity have followed Moore's Law and thus kept up with advances in CPU technology. However, DRAM-based memory systems have two significant drawbacks — cost and power consumption. These drawbacks impact their usage in both enterprise servers and mobile systems. Most new generation applications are inherently memory-intensive, whereby workloads demand access to high-performance yet low-cost memory systems [1]–[4]. A complementary technological change is the imminent availability of higher capacity and lower power consuming byte-addressable persistent memory (PM) technologies [5]–[7]. These new memories offer latency and bandwidth for byte-addressable access that are within an order of magnitude of those for DRAM [8], [9], with power consumption being lower by 4-29x compared to DRAM [8]. These characteristics make the use of PM to extend the main memory attractive. When using PM as the main memory, its persistence capability becomes irrelevant, thereby entirely avoiding its biggest performance overhead [10].

One appealing use of persistent memory is as a new tier in a hybrid multi-tier memory system with tiers ordered from *high performance - low capacity* to *low performance - high capacity*. This approach allows applications to access their data directly from persistent memory without first paging into DRAM. However, managing persistent memory simply as additional available memory (i.e., static tiering) could compromise the effectiveness of the tiered memory system. Once an application has exhausted higher performance tier resources, future allocations for that application or any other application on the system will have to be serviced from lower performance tiers. Additionally, such allocations continue to reside in lower performance tiers regardless of how important the data becomes over its lifetime. Thus, the primary challenge in building an efficient hybrid memory system is the dynamic placement of pages in appropriate tiers. From a system design standpoint, addressing this challenge translates to understanding the relative importance of pages, identifying misplaced pages in either tier and moving such pages to their optimal tier, all while controlling the overhead of these operations. The major contributions of our work are:

- We design an efficient page selection method for dynamic page movement across memory tiers. This method enables identifying pages suitable for specific memory tiers in an online fashion, thereby adapting to the workload.
- We propose MULTI-CLOCK, a solution based on dynamic tiering that overcomes the limitations of static tiering and extends the system's memory with improved performance without sacrificing DRAM capacity.
- We develop a real-system prototype implementation of MULTI-CLOCK using Linux version 5.3.1 by extending the kernel's page reclamation algorithm to include its dynamic page migration logic.
- We evaluate the performance of our prototype using diverse workloads including graph analytics and key-value stores to compare MULTI-CLOCK with existing solutions.

We evaluate MULTI-CLOCK against Nimble [11], AutoTiering [12], and Memory-mode [7]. Our evaluation with YCSB workloads [13] using a Memcached [3] backend and with GAPBS [14], a graph processing benchmark, shows that MULTI-CLOCK provides up to 132% higher performance compared with static tiering and up to 352% compared with other state-of-art solutions such as Nimble [11], AutoTiering [12], and Memory-mode [7]. From these experiments, we find that the page selection mechanism in dynamic tiered memory systems is of critical importance. We also demonstrate that

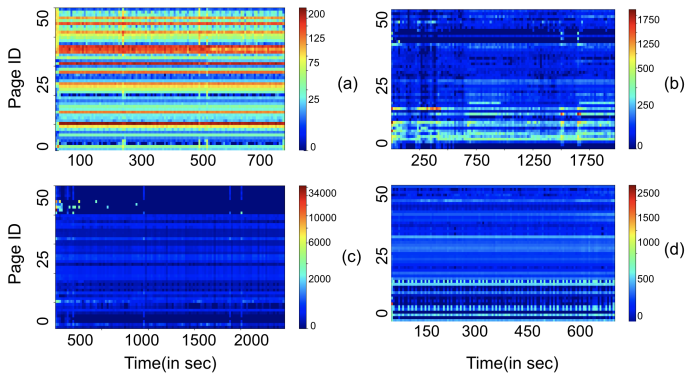

Fig. 1. **Heat-map of pages access frequencies** depicts access frequencies of the sampled pages in (a) RUBiS OLTP, (b) SPECpower (OLTP), (c) Dacapo *xalan* and (d) Dacapo *lusearch*.

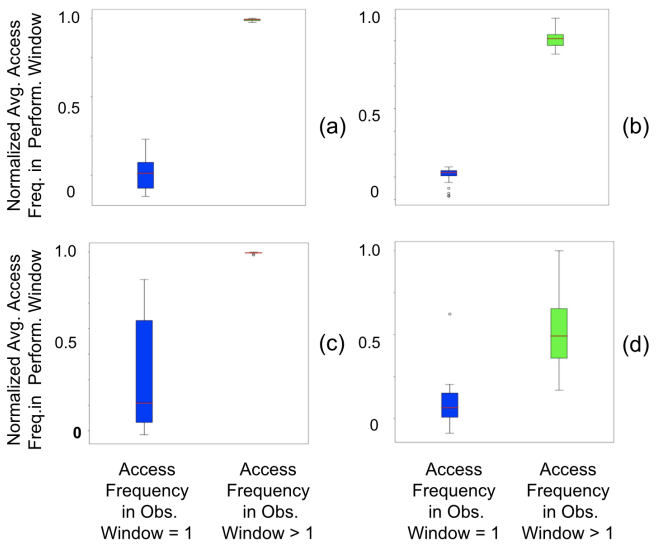

Fig. 2. **Distribution of access frequencies for different page types** depicts the distribution of access frequencies in the performance windows for the two types of pages: pages that were accessed only once during the observation window and pages that were accessed multiple times in the observation window. Workloads for the experiment: (a) RUBiS OLTP, (b) SPECpower (OLTP), (c) Dacapo *xalan* and (d) Dacapo *lusearch*.

the state-of-the-art page selection mechanisms do not consider page access frequency distributions for identifying page migration candidates, and we demonstrate that doing so is vital for optimizing performance in a tiered memory system.

II. Motivation

Integrating persistent memory (PM) devices into existing systems force a rethink of the system architecture. Due to the relatively high read and write latency and low bandwidth, using only PM as the main memory is not ideal. On the other hand, a hybrid memory system with DRAM and PM can deliver both high throughput combined and increased capacity. However, designing a memory hierarchy with PM to improve the performance of applications is non-trivial.

One promising approach of utilizing PM is to configure both DRAM and PM as separate tiers in a multi-tier memory system. With tiering, data residing in the byte-addressable PM is treated as resident in the main memory and directly addressable by the CPU. Tiers represent disjoint sets of memory frames. The operating system identifies which frames belong to each memory type and assigns them to their proper tier. Tiers can be arranged in a specific order, following the characteristics of the different types of memory from Higher Tier - *high performance - low capacity* to Lower Tier - *low performance - high capacity*), to service memory allocations.

In this section, we discuss the diversity in the access patterns of pages across applications. We also discuss why the careful selection of candidate pages for specific tiers based on both the frequency and recency is pivotal for performance. We also discuss the existing solutions for the DRAM-PM tiered memory systems and their limitations.

A. Diversity in Page Access Patterns

Let us consider a simple tiered memory system wherein pages are first allocated (or get "born in") in the DRAM tier. When the system starts running low on free space in DRAM, the system starts demoting less frequently accessed pages to the PM tier to free up DRAM space for new allocations. Without an available promotion mechanism, a demoted page would reside in the PM tier for the rest of its lifetime. If a significant number of demoted pages get frequently accessed post-demotion, a complementary promotion mechanism that allows demoted pages to move back to the DRAM tier may result in better system performance. However, a tiered system with the facility of promoting pages from PM to DRAM can improve performance only if promoted pages are accessed relatively more frequently for a reasonable amount of time afterward. To evaluate the potential for a promotion mechanism in improving workload performance, we recorded the access patterns of pages in memory over time within applications. To keep the overhead tractable, we randomly sampled pages from memory, assigned them unique identifiers, and traced the accesses to these sampled pages.

In Figure 1, the heatmap depicts the frequency of accesses of the sampled pages for the execution periods of four workloads from different benchmarks: (a) RUBiS OLTP benchmark [15], (b) SPECpower (OLTP) [16] running at 80% of the maximum throughput, (c) Dacapo *xalan*(XML to HTML conversation) and (d) Dacapo *lusearch*(searching keyword over a corpus of data using lucene) [17]. On the Y axis, 50 sampled pages are sorted in ascending identifier order. The x axis represents execution time. Each block of the heatmap shows the intensity of the access frequency for a particular page for a particular time segment. The heat maps indicate fairly diverse access patterns for the sampled pages. Some pages show frequent accesses throughout the execution period. We denote these pages by *DRAM friendly pages* which should always reside in DRAM. Other pages have very infrequent accesses over the entire execution time. The total number of accesses from

Tiering Technique	Page Access Tracking	Page Selection		NUMA Aware	Space Overhead	Generality	Evaluation	Usability Limitation	Key Insight
		Promotion	Demotion						
Static-Tiering	N/A	N/A	N/A	Yes	N/A	All	PM	None	Straight forward
Thermostat	Software Page Fault	N/A	Frequency	No	Yes	Huge Page	Emulator (KVM)	Not Open Source	Poisoning huge pages
AutoNUMA-Tiering	Software Page Fault	Recency	N/A	Yes	Yes	All	PM	Config. NUMA Paths	NUMA balancing
AutoTiering	Software Page Fault	Recency	Frequency	Yes	Yes	All	PM	Config. NUMA Paths	Maintain N-bit history for demotion
Nimble	Reference Bit	Recency	Recency	No	No	All	Emulator	Config. Launcher	Optimize huge page migrations
AMP	Reference Bit	Recency+ Frequency+ Random	Recency	No	Yes	Huge Page	Emulator (QEMU)	No KMEM DAX Support	Hybrid page selection
MULTI-CLOCK	Reference Bit	Recency+ Frequency	Recency	Yes	No	All	PM	None	Low overhead Recency/Frequency

TABLE I
Comparison of existing memory tiering techniques.

these pages is very low compared to the total access count during the execution. Thus, the tier residence of these pages does not significantly impact the overall performance. Apart from these two types of pages, we see that certain pages can significantly benefit a tiered memory system. These *Tier friendly pages* show bimodal access behavior whereby for some time segments they get accessed at a much higher rate than other time segments. If these pages can be identified by analyzing their access patterns and moved to the DRAM tier when they start to get accessed at a higher rate, the overall application performance can potentially be improved. Thus, our core motivation for a dynamic tiering system is driven by two main observations: (a) the importance of pages changes over time, and (b) at any given time, the importance of different pages in the system can vary significantly.

Next, we investigate the importance of frequency of accesses along with the recency for identifying *Tier friendly pages*. Recent works such as Nimble [11] select pages only based on the recency since capturing frequency on the real system with minimal overhead is challenging. To understand the access frequency of pages, we divide the whole execution period of the workloads that were used in the experiment in Figure 1 into multiple sets of *observation windows* followed by *performance windows*. We divide sampled pages that were accessed into two defined categories: pages that were accessed only once during that particular observation window and pages that were accessed multiple times. Finally, we measure their accesses in the next performance window, and we follow the same procedure for all (observation window, performance window) pairs. In the frequency distribution shown in Figure 2, we can notice that pages that were accessed multiple times in the observation windows are accessed with a much higher frequency on average in the performance windows compared to the pages that were accessed only once. This suggests that pages with higher frequency in some observation windows have a higher probability of getting accessed in the next performance window.

B. Persistent Memory in Memory-mode

Persistent memory in Memory-mode is a natively system-supported solution for using PM as memory. It is implemented in recent memory controllers that support PM and by recent operating systems that support PM DIMMs [7]. In Memory-mode, DRAM is directly mapped as the cache for data stored in PM and used as the last level cache in addition to the L1/L2/L3 caches. The system recognizes only the PM as memory. In a multi-socket system, DRAM can only act as a cache for the PM DIMMs on the same socket [18]. The primary limitation with using PM in Memory-mode is that the available DRAM capacity is unusable by the operating system and thus applications as well.

C. Memory Caching and Tiering

The classical caching problem when used with memory hierarchies in computer systems is distinct from the dynamic memory tiering problem. With caching, every item needs to be fetched from the higher-performing tier (i.e., DRAM) before accessing it. With tiering, in addition to the high performance (DRAM) memory tier, there's a second (lower-performing) memory tier that is also directly accessible. Due to the small performance gap between the high-performing and the lower-performing tiers, items can be directly fetched from the lower-performing tier without significant performance loss. Thus, the core problem to address here is placing the right data in the right memory tier, online.

Caching-aware applications (e.g., compilers) can organize prefetching and increase memory access efficiency during execution. In the future, if tiers of memory get individually exposed to applications, it is conceivable that applications can achieve prefetching of data from PM to DRAM via OS hints. MULTI-CLOCK provides a currently usable method where the kernel can automatically identify the hot items and can serve them from the higher memory tier. This technique is entirely oblivious to applications. Furthermore, dynamic migration implemented in systems such as MULTI-CLOCK is complementary to prefetching-based techniques and can also

be effective in systems where prefetching is not feasible or accurate.

D. Existing Tiered Memory Systems and Their Limitations

Table I shows the comparison of the existing and MULTI-CLOCK tiering system. A straightforward way to tier is static tiering, whereby a memory page, once mapped to a tier, may not get reassigned to a different tier during its lifetime. However, this is inefficient; when an application wins the race to allocate memory from a higher tier, and such space is exhausted, future allocations will be downgraded to use lower tiers during their entire lifetime, regardless of how the *importance* of the contained data changes over time.

[Software Page Fault Based Page Access Tracking.] Thermostat [19] focused on tracking huge pages by poisoning the page table entry (PTE) and triggering a software page fault, and migrating cold pages to the lower memory tier. AutoNUMA-tiering [20] and AutoTiering [12] are based on AutoNUMA [21]. Similar to Thermostat, these solutions use a software page fault technique called hint page fault to track the page access and use recency to identify hot pages for promotion. Although the software page fault techniques can provide high accuracy in page access tracking, it is costly to track all the pages as every page fault has to be handled before accessing the page. Moreover, these techniques also require additional memory to store each page's individual scan time on which its page hotness classification depends. AutoTiering designs a conservative approach (AutoTiering-CPM) to migrate pages to the best NUMA node. In addition, AutoTiering maintains an n-bit vector for each page to determine the page coldness and designs a progressive approach (AutoTiering-OPM) to demote cold pages to lower tier. We could not evaluate Thermostat as its source code was not available. We evaluate both AutoTiering-CPM and AutoTiering-OPM to compare the performance with MULTI-CLOCK. AutoTiering-CPM is designed using AutoNUMA-tiering, and thus we did not explicitly compare with AutoNUMA-tiering.

[Reference Bit Based Page Access Tracking.] Nimble [11] focuses on transparent huge page (THP) migration, enables multi-threaded concurrent migration, and two-sided page exchange to improve the overall page migration performance. However, Nimble uses the existing page profiling technique of the Linux kernel to exchange the top most recently accessed pages in the lower tier with the least recently accessed pages in the upper tier. Nimble is evaluated on an emulator, and applications need to run through Nimble's *launcher* to utilize its page migration techniques. As Nimble mainly focuses on the optimization of the overall page migration process, we separated its hot/cold page identification technique and implemented a single threaded Nimble page selection mechanism in a real system for the singular purpose of comparing against MULTI-CLOCK's page selection mechanism. MULTI-CLOCK itself is implemented as a built-in kernel feature, and hence, applications do not require to follow any purpose-built launcher mechanism for using MULTI-CLOCK.

AMP [22] proposes a tiered memory system that focuses on page selection mechanisms based on the popular cache replacement algorithms, including least-recently-used (LRU), least-frequently used (LFU), and random selection. AMP is designed, implemented, and evaluated using an emulator. AMP uses one node, only for DRAM allocations, and the other node only for PM allocations, which is unrealistic in a two socket NUMA machine wherein each node typically has its own DRAM, PM, and CPUs [7]. Moreover, AMP is implemented on Linux kernel version 4.15, which does not support the required KMEM DAX driver (available from kernel v5.1) to use PM as the main memory in a tiered system. Furthermore, the core design principle of AMP requires it to scan and profile all the memory pages from both DRAM and PM tier, which is impractical in the kernel on a real system as the number of in-memory pages can grow to hundreds of millions for the workloads we evaluated. Hence, for multiple practical reasons, we could not deploy AMP on a real system for evaluation.

Identifying the hot/cold data in virtual memory management may cause a high overhead. For low overhead, efficient tracking, the Linux kernel implements CLOCK, which is the approximation of the popular LRU cache replacement policy. As tracking every in-memory page access is not feasible, LFU is considered impractical for general virtual memory management. The CLOCK algorithm does not consider the frequency of the access. In a tiered memory system, as we have shown in Section II-A, it is important to capture both recency and frequency for hot/cold page identification. Hence, in this paper, we try to solve the following two novel research questions for tiered memory systems:

- *RQ1: How to identify hot pages for promotion based on recency and frequency?*
- *RQ2: How to design a simple and low overhead yet efficient system in the kernel?*

III. MULTI-CLOCK

A fundamental problem with static tiering is the mismatch of page access performance requirements with tier performance capabilities. Dynamic memory tiering mechanisms address this problem with a solution that dynamically migrates important pages to higher tiers and less important pages to lower tiers. The principal hypothesis of designing MULTI-CLOCK is that the pages that are recently accessed more than once are more likely to be accessed in the *near* future. MULTI-CLOCK determines the relative importance of pages within and across tiers by running a modified version of Linux's Page Frame Reclamation Algorithm (PFRA) (which is based on the CLOCK algorithm) to each memory tier separately.

MULTI-CLOCK is implemented based on the well-known CLOCK because of its low overhead and effectiveness. However, MULTI-CLOCK does not use CLOCK exactly as it is. The CLOCK algorithm approximates LRU by checking for references when scanning the list of pages and moving any referenced page to the head of the list. MULTI-CLOCK uses a new approach to identify important pages in the lower tier. In addition to the active and inactive lists, MULTI-CLOCK

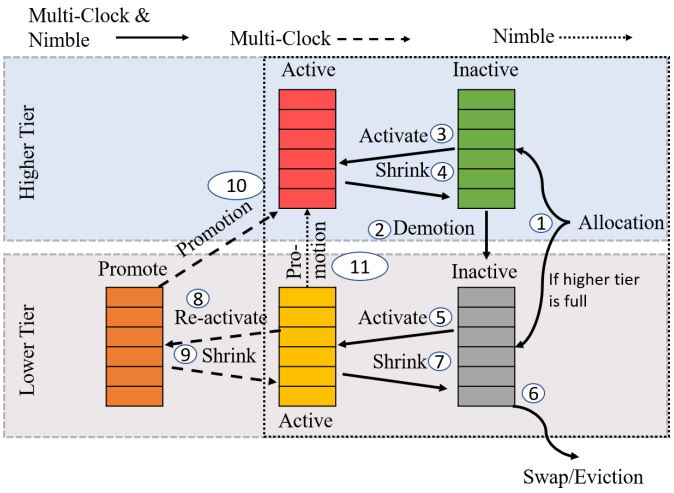

Fig. 3. **MULTI-CLOCK architecture.** The new data structures that we add to each tier and the interaction between these data structures. The arrows show the movement of pages across different page lists. The solid arrow represents both MULTI-CLOCK and Nimble, the dashed arrow is for MULTI-CLOCK only, and the dotted arrow is for Nimble only. The label numbers do not represent any particular order of the operations.

introduces a new *promote* list to select the candidate pages for promotion. MULTI-CLOCK completely reworks how pages are moved across the lists in both tiers, thus introducing a new lifecycle of pages.

New pages are allocated from the DRAM. Once the DRAM is full, pages are allocated from the PM tier. Every page in the system is arranged in one of its tier's three lists according to their degree of hotness/coldness of accesses. The DRAM tier in the system does not use a promote list since there is no higher-performing tier to migrate pages to. With these changes, MULTI-CLOCK is able to capture both recency and frequency. Hence, although MULTI-CLOCK is based on the CLOCK algorithm, it is different from CLOCK and has significant algorithmic contributions and unique implementation challenges that we elaborate next in this section and in Section IV.

A. Life Cycle of a Page

Figure 3 depicts the overall arrangement of lists in the two tiers on the system and the possible movement of pages within and across these tiers for both MULTI-CLOCK and Nimble. Every list is scanned at various points in time to make decisions regarding migrations. In MULTI-CLOCK, a recently allocated page starts in the inactive list as shown in the Figure 3(1). The inactive list of a higher-performing tier maintains candidate pages for *demotion*, i.e., migration to a lower-performing tier. A page is said to be referenced if any type of access (i.e., read or write) occurs to the page. Both inactive and active lists make a differentiation between pages that were referenced and those that were not referenced since the last scan.

During a scan, if a page has been marked referenced since the previous scan is encountered, it is then marked as not referenced and moved to the head of the list. On the other hand, if the page was not referenced, it is moved according to which list it belongs to: *(a)* if it belongs to the active list, it is moved to the inactive list as shown in Figure 3 (4) and (7), and *(b)* if it belongs to the inactive list is then migrated to its lower tier, and if none exists, evicted out of memory (Figure 3 (6)). This movement of pages out of a list is referred as the shrink of the source list. At the same time, when access occurs to a page in the inactive list and that page was marked as referenced, this page is *activated* by being moved to the active list's head, where it starts out by being marked as not referenced (Figure 3 (3) and (5)). A similar process is followed when a page is *re-activated* and is moved from the active list to the promote list's head (Figure 3(8)), where it becomes a candidate for *promotion* (i.e., migration to a higher-performing tier) as shown in Figure 3(10).

With this arrangement, the system is able to classify pages into three categories: *hot*, *warm*, and *cold*. Hot pages navigate the lists within a tier and eventually reach the promote list where they become candidate pages to migrate to the higher-performing tier. On the other extreme, cold pages remain in the inactive list where they become candidates for migration to a lower-performing tier when the tier experiences memory pressure. Thus, MULTI-CLOCK makes decisions on how to place each page within an appropriate tier and within an appropriate list according to their access frequency and recency.

The key difference in the architecture of MULTI-CLOCK and Nimble is shown in Figure 3. The life cycle in Nimble involves the page only residing in the dotted box on the right side of the Figure 3. Nimble does not have any promote list, and thus it cannot differentiate between pages accessed exactly once and those accessed more than once. Nimble selects a fixed number of the top pages in the lower tier's active list to promote to the bottom of the higher tier's active list as shown in Figure 3(11). The number of pages that get selected by MULTI-CLOCK is not fixed as it qualitatively chooses pages from the lower tier's active list based on recent re-accesses to the pages.

One of the key challenges in designing MULTI-CLOCK is keeping track of accesses and updating the reference status of pages in a timely matter. This is addressed differently depending on the type of page access used by applications. Applications can access memory pages in two ways: *supervised*, using the operating system's (OS) file system call interface, and *unsupervised*, by memory mapping pages into their address-space.

1) Supervised Access: This type of access is typically used for file-backed pages, and it gives the OS control at the moment of the access to perform the necessary book-keeping. When applications use supervised access to memory pages, the operating system is able to mark these pages referenced (for e.g., in Linux, via `mark_page_accessed()`) and, if necessary, to move between lists (activate or re-activate) before even processing the requested data access.

2) Unsupervised Access: Accesses to anonymous or file-backed memory that is directly mapped into the application's virtual address space via `mmap` are more difficult to monitor.

This type of access is entirely unsupervised, and the OS is not able to mark such pages as referenced. To handle unsupervised access, MULTI-CLOCK relies on the page reference bit set by the CPU in the process' page table entry. During each scan, as described earlier, before making any decision regarding a specific page, MULTI-CLOCK checks within every process' page table that maps it for a set referenced bit. If a referenced bit is found set, MULTI-CLOCK updates the page status and takes care of the necessary movement between lists (i.e., mark as referenced, activate, or re-activate the page).

B. Promotion Mechanism

We design a new system daemon, `kpromoted`, that is woken up periodically to scan the lists, update them, and migrate any pages from the promote list to a higher tier due to recent unsupervised accesses. Every time `kpromoted` runs, it first selects the candidate pages for promotion and promotes all the pages it selected. Thus, once a page is selected for promotion, the page gets promoted to the DRAM in the same `kpromoted` run. As `kpromoted` promotes all the pages it selects, the number of promotions depends on the running application. If the application frequently accesses a large number of pages from the PM tier, the number of promotions will increase. On the other hand, if the application does not frequently access pages from the PM tier, `kpromoted` will promote fewer pages or no pages at all.

Implicitly, MULTI-CLOCK relies on the periodicity of `kpromoted` waking up to ensure that hot pages in lower tiers are migrated to higher tiers in a timely manner. The frequency of `kpromoted`'s execution defines the capacity of the system to react quickly to workload changes. If scheduled too frequently, excessive context switches to accommodate its execution could also affect application performance. Careful tuning of `kpromoted`'s execution schedule is necessary to ensure that applications benefit from the promotion mechanism in MULTI-CLOCK. In the prototype system we built, we chose the `kpromoted` execution schedule to be every 1 second as discussed in Section V-E and this worked fairly well for the workloads we evaluated the system with. It resulted in sufficient responsiveness in promoting hot pages without imposing high CPU overheads due to unnecessary scanning of every page in the LRU lists.

C. Demotion Mechanism

Demotion allows moving cold pages from a higher-performing tier to a lower-performing tier when these pages are no longer sufficiently important. MULTI-CLOCK's design of this mechanism is based on the page eviction design in today's virtual memory systems. To avoid running out of memory on a given tier, a tier is marked under memory pressure proactively when it reaches specific watermark levels. These levels are calculated by the system according to the amount of memory in the tier vs. the total amount of memory in the system.

If any tier is marked as being under memory pressure, each list is scanned with the objective of freeing up memory. Any page in the promote list is first attempted to be migrated to a higher-performing tier, and if that is not possible — for instance, the page is locked — then it is moved to the active list. If the higher-performing tier is also under memory pressure, promotions from the lower tier result in immediate page demotions from the higher tier. Next, if the ratio of pages in the active list with respect to the inactive list exceeds a tunable threshold (inherited from PFRA and typically $\sqrt{10 * n} : 1$, where n is the amount of memory in GB available in the tier), pages not marked as referenced in the active list are moved to the inactive list. Finally, the inactive list is scanned in search of pages not marked as referenced to be migrated to a lower tier. Migration may not be possible, specifically because the candidate pages belong to the lowest tier in the system. In this case, these pages are written back to block storage (i.e., file-backed pages to file system and anonymous pages to the swap area if available) before triggering the out-of-memory (OOM) killer as the last option.

IV. Implementation

The existing Linux mechanism to describe physical memory relies on the definition of nodes. In NUMA architectures, each bank of memory is represented by a single NUMA node. On the other hand, for UMA architectures, Linux uses a single NUMA node to represent all physical memory in the system. The data structure used to represent nodes is called `pglist_data`. Each node is then divided into memory ranges called *memory zones*, and Linux uses the data structure `zone` to represent them in memory. Zones are of different types, and each type is suitable for a different usage (i.e., `ZONE_DMA` gathers physical addresses that can be accessed by legacy hardware through DMA). We implemented a prototype of MULTI-CLOCK for NUMA architectures for Linux kernel v5.3.1. Our prototype evaluates a hybrid two-tiered memory system: one tier of DRAM and another of persistent memory. In comparison with Nimble, which requires an additional launcher to run any workload on the kernel, our implemented prototype of MULTI-CLOCK can directly run any workload without any additional configuration setup or prior knowledge. We used the Intel Optane DC Persistent Memory on a real platform as the persistent memory tier (discussed in Section V-A). Upon creating a new namespace in *devdax* mode using the `ndctl` tool [23], we can hot-plug the namespace as system memory with the `DAX-KMEM` driver. `DAX-KMEM` driver is available in the kernel from v5.1 and onwards. The `DAX-KMEM` driver separates newly added PM from the DRAM by hot-plugging PM as a new node. We modified the `DAX-KMEM` driver to tag the newly hot-plugged node as a PM node, so that MULTI-CLOCK can recognize it by adding a new flag in the `pglist_data` structure. Although PM is hot-plugged as a new node, this node id is different from the physical node of the PM, i.e., the socket where it is physically installed. We define all the DRAM nodes as the DRAM tier and all the PM nodes in the system as the PM tier.

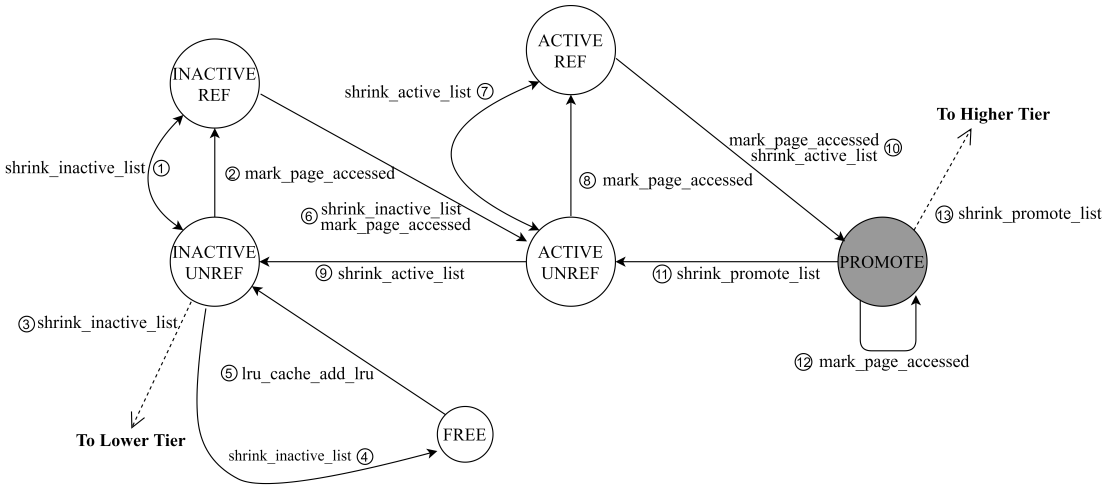

Fig. 4. **Page state diagram depicting the Linux implementation of MULTI-CLOCK** Each vertex represents a page state; white vertices are original PFRA page states while the gray vertex is a new page state introduced by MULTI-CLOCK. Solid edges represent Linux procedures that change page state; dashed edges represent page migration to a different tier. Counterparts to shrink_list methods are implicit on page allocations that cause lists to expand. The numeric edge numbers do not represent any particular order of operations.

The main design principle of MULTI-CLOCK is to migrate cold data from the DRAM tier to the PM tier and move hot data from the PM tier to the DRAM tier. We rely on the existing Linux migration mechanisms already in place for the hot-plug/hot-remove of memory. Linux's page migration mechanism (`migrate_pages()`) is in charge of allocating new memory pages given an allocation routine, copying the memory contents from origin pages to the newly allocated destination pages, and fixing any memory mappings that refer to the migrated pages.

Originally, each memory node maintains its own set of LRU lists: anonymous inactive, anonymous active, file inactive, file active, and unevictable. We added two lists: *anonymous promote* and *file promote*. Unevictable pages belong to the unevictable list and are pages in the system that are locked into memory (typically via `mlock()`) and cannot be evicted nor migrated. Every evictable page in the system, depending on being file-backed or anonymous, will belong to one set of LRU lists (anonymous lists or file lists), and it will traverse these by transitioning through different states as depicted in Figure 4. We also extended the `struct page` flags which maintain the status of a page during its existence to add a new flag: `PagePromote`. This new flag is used by the OS to mark that the page in question, which is to be added to the LRU lists, belongs to the promote list. The memory overhead of these modifications is negligible since we reused the list pointer on the `struct page` to index the pages in the promote lists; we also reused the space allocated for the page flags to maintain the newly defined flag.

We implemented the system daemon discussed in Section III-B as a new kernel thread, `kpromoted`, which is woken up periodically to execute the migration of any pages sitting in the promote list to a higher tier. This thread's design follows those of PFRA for the `kswapd` eviction daemon: one kernel thread per NUMA node. This design aims to avoid lock contention on critical per-node data structures.

Source File	New Lines	Modified Lines
drivers/base/node.c	4	1
drivers/dax/kmem.c	10	0
include/linux/gfp.h	7	1
include/linux/mm.h	6	0
include/linux/mm_inline.h	8	1
include/linux/mmzone.h	52	1
include/linux/nodemask.h	6	1
include/linux/page-flags.h	19	6
include/trace/events/mmflags.h	7	1
mm/Kconfig	3	0
mm/memcontrol.c	8	2
mm/migrate.c	1	0
mm/page_alloc.c	43	2
mm/swap.c	59	6
mm/vmscan.c	364	7
mm/vmstat.c	16	0

TABLE II
Linux source code modifications measured as number of lines modified.

Our implementation of the MULTI-CLOCK algorithm is encapsulated mostly within `mm/vmscan.c` and `mm/swap.c`. Table II presents how much new code was added for MULTI-CLOCK and which files were modified in the Linux's source code. In total, MULTI-CLOCK inserted 673 new lines and modified 30 existing lines of code. We extended `mark_page_accessed()` to check for pages that are already referenced and marked as active and are being referenced again to mark such pages with the `PagePromote` flag and to move them from their corresponding active list to the promote list (see transition 10 in Figure 4). We created a new `shrink_promote_list()` method that complements the existing `shrink_active_list()` and `shrink_inactive_list()` methods to handle movements of pages out of the promote list. Migrations to the upper tier are handled via `shrink_promote_list()` and

migration to the lower tier (or evictions) are handled via `shrink_inactive_list()`. Both methods result in a physical frame in the tier being freed after the successful migration of its contents.

Figure 4 depicts all the states and transitions of the pages. New pages start with the *inactive unreferenced* state. Depending on whether the page was accessed since the last scan or not, it can move to the *inactive referenced* state via transition (1), and (2), can get demoted to the lower tier via (3), or can be freed via (4). Pages not in LRU can also get added to *inactive unreferenced* state via (5). Pages in the *inactive referenced* state can either move to *inactive unreferenced* via (1) or move to the active list via (6). Pages in the active list with the *active unreferenced* state move to the *active referenced* state using (7) or (8) if they get accessed. Furthermore, if the page is not accessed for a long time, the page state changes to *inactive unreferenced* via (9). From *active referenced* state, a page moves to the promote list if it gets accessed via transition (10). If pages in the promotion list do not get accessed, they move to the *active unreferenced* state again via (11). If they get accessed in this state, then pages remain in the same state, as shown by (12). Lastly, `kpromoted` uses (13) to promote all the pages found in this state.

V. Evaluation

In this section, we evaluate the performance of our MULTI-CLOCK implementation. The goal of our evaluation is to determine if, when, and how the MULTI-CLOCK is able to improve the performance of application workloads. We evaluate using diverse workloads such as high memory-consuming graph applications and key-value stores. We compare MULTI-CLOCK performance with static tiering, Nimble, AutoTiering, and Memory-mode. As Nimble uses Linux's CLOCK (an approximation of LRU) based default page profiling mechanism, we do not compare MULTI-CLOCK again with CLOCK or LRU. We also avoid comparing MULTI-CLOCK with the *Least Frequently Used* (LFU) policy as it requires tracking every memory access, which is impractical. Additional reasons for not comparing MULTI-CLOCK with other memory tiering techniques such as AMP [22], Thermostat [19], and AutoNUMA-Tiering [20], are discussed in Section II-D. Finally, we conduct an in-depth sensitivity analysis to better understand the impact of each component of MULTI-CLOCK.

A. Experimental Platform

We used a dual-socket Intel Xeon Gold 5218 Processor with 16 cores per socket for evaluating and comparing the performance of static tiering, Nimble, AutoTiering-CPM (AT-CPM), and AutoTiering-OPM (AT-OPM) with MULTI-CLOCK. This machine has 12 DDR4 (2666 MT/s) DIMMs of 16GB in capacity each and 4 Intel Optane DC Persistent Memory (DCPM) of 128GB in capacity each. In total, the available memory space is 192GB DRAM and 512GB persistent memory. We used another platform to compare the performance of static tiering, Memory-mode, and MULTI-CLOCK. This machine runs a dual-socket Intel Xeon Processor with 24 cores per socket. The system is equipped with 12 DDR4s (2666 MT/s), each 32GB in capacity and another 12 Intel Optane DCPM with 128GB capacity per DIMM. Hence, the total DRAM capacity is 376GB, and PM capacity is 1.5TB. The only reason for using two separate machines is to expedite the evaluation process.

B. Workloads

We evaluate MULTI-CLOCK using diverse workloads. Here, we discuss our results using six different workloads from Yahoo! Cloud Serving Benchmark (YCSB) [13] and six workloads from the GAP Benchmark Suite (GAPBS) [14]. YCSB workloads are divided into two phases: a *load* phase and an *execution* phase. The load phase is in charge of populating the back-end key-value store with the required number of records. On the other hand, the execution phase carries out diverse types of operations over the back-end. These workloads are named Workload A, B, C, D, E, and F. Workload A is a mix of 50% reads, and 50% writes. Workload B is 95% reads, and only 5% writes. Workload C is 100% read. None of these workloads inserts new records except workload D, where new items are added and read. Workload E issues short ranges queries on the records. And in workload F, a record is read, modified, and then written back. We also created a new workload W, which issues 100% writes. For our evaluation, we used Memcached [3], an in-memory cache service that uses a large amount of main memory to maintain its data, as the key-value store back-end of YCSB. One thing to note is that YCSB's workload E makes use of SCAN operations that may or may not be implemented by the different back-end key-value stores. Memcached does not implement SCAN operations, making workload E non-operational. Further, the load phase is the same for all workloads, and most workloads (all but D and E) do not change the number of records in the back-end. For all our experiments, we follow the prescribed execution sequence [24] for the YCSB workloads. Since workload D changes the number of records in the back-end, the order of execution is arranged in the following manner: Load Phase, Workload A, Workload B, Workload C, Workload F, Workload W, and Workload D. We report the performance of the six Workloads, excluding the data load phase.

GAPBS is a framework for graph analytics capable of running a wide variety of graph processing algorithms. It has six workloads: Breadth-First Search (BFS), Single-Source Shortest Paths (SSSP), PageRank (PR), Connected Components (CC), Betweenness Centrality (BC), and Triangle Counting (TC). For each of the six workloads, GAP first loads the graph in memory and then executes multiple trials of the workload. We report the average execution time taken per trial for the workloads. During the execution phase, the actual algorithm is executed over the already memory-resident graph representation of the data.

C. Evaluation Result

To evaluate the overall performance of MULTI-CLOCK, we first compare MULTI-CLOCK against systems using PM in

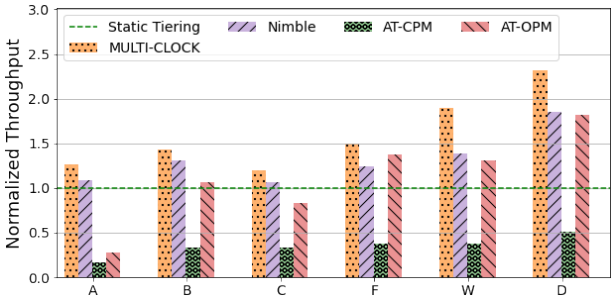

Fig. 5. **MULTI-CLOCK, Nimble, AutoTiering-CPM(AT-CPM), and AutoTiering-CPM(AT-OPM) throughput comparison for YCSB workloads.** Y axis presents the throughput normalized to static tiering (higher is better).

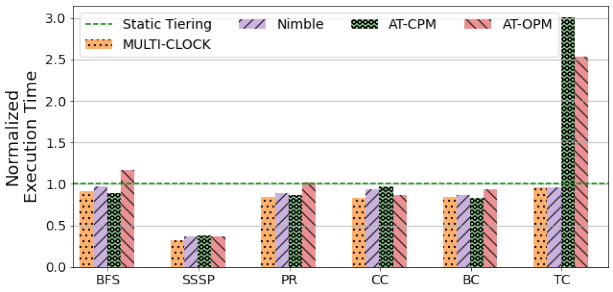

Fig. 6. **Performance comparison of GAPBS workloads.** Y axis presents the normalized execution time to the static tiering (lower is better).

static tiering, Nimble, AutoTiering-CPM, and AutoTiering-OPM. Then we compare the performance of MULTI-CLOCK with Memory-mode. We configure workloads for all the systems such that their memory footprints are larger than the DRAM size and consume enough persistent memory. For both MULTI-CLOCK and Nimble, we set the number of page scan to 1024. The scanning interval of MULTI-CLOCK and Nimble is set to one second as discussed in Section V-E.

1) Comparison With Tiered Memory Systems: We compare the performance of static tiering, MULTI-CLOCK, Nimble, AutoTering-CPM (AT-CPM), and AutoTiering-OPM (AT-OPM) using YCSB and GAPBS workloads. Figure 5 shows the performance for YCSB workloads. In Figure 5, the Y-axis presents the throughput (operations per second) normalized to static tiering; all the workloads are on the X-axis. MULTI-CLOCK outperforms static tiering, Nimble, AT-CPM, and AT-OPM for all the workloads.

For the YCSB workloads, MULTI-CLOCK outperforms static tiering by 20-132%. MULTI-CLOCK achieves the maximum throughput gain in Workload D as this workload inserts new records and modifies the most recent records multiple times. As MULTI-CLOCK selects the pages that are recently accessed multiple times for promotion, Workload D and other workloads with a similar property would get the most benefit from MULTI-CLOCK. In comparison with Nimble, MULTI-CLOCK achieves 9-36% better performance as MULTI-CLOCK promotes pages more selectively than Nimble. The selective promotion of MULTI-CLOCK helps to reduce the migration overhead incurred for promoting less qualified pages. When compared to AT-CPM, MULTI-CLOCK outperforms by 260-677%. Finally, MULTI-CLOCK achieved 10-352% better performance than AT-OPM. In comparison with MULTI-CLOCK, AT-CPM and AT-OPM perform worse due to costly software page fault-based page access tracking as well as the high overhead of tracking the page history bits for identifying cold pages.

Figure 6 presents the results of executing different GAPBS's workloads normalized to static tiering. The Y-axis shows the normalized execution time; the X-axis presents all the workloads. As we can see, MULTI-CLOCK outperforms static tiering by 4-68% for the GAPBS workloads. When compared to Nimble, MULTI-CLOCK improved the execution time by 1-16%. In both comparisons, MULTI-CLOCK reduces the execution time of the SSSP workload the most. Similar to the YCSB workloads, in GAPBS, MULTI-CLOCK benefits from the better page selection mechanism for promotions.

In comparison with AT-CPM, MULTI-CLOCK reduces the execution time by 3-68% for SSP, PR, CC, and TC workloads. However, AT-CPM shows 3% and 1% better performance than MULTI-CLOCK for BFS and BC workloads. As AT-CPM tries to find the best location of the pages, its performance thus highly depends on the initial placement of the pages. If pages are already placed in the best locations, AT-CPM needs to migrate fewer pages. We think the slight performance gain for BFS and BC workloads might be due to this reason. On the other hand, MULTI-CLOCK shows better performance than AT-OPM by 4-62%. Compared to AT-CPM, AT-OPM induces additional overhead of identifying cold pages and page demotions, which is the reason for the observed performance.

From Figure 5 and Figure 6 we observe that the MULTI-CLOCK achieved better performance gain for the YCSB workloads than the GAPBS's workloads. The performance of the graph processing algorithms can depend on the locality of the data [25]. We assume that the GAPBS workloads first allocate memory that would be accessed the most as graph processing workloads are known to exhibit substantial locality [26]. As static tiering, MULTI-CLOCK, Nimble, AT-CPM, and AT-OPM fill the DRAM first, DRAM contains most of the highly accessed pages. Hence, the performance of the MULTI-CLOCK, Nimble, AT-CPM, and AT-OPM is close to the static tiering for most of the GAPBS workloads. However, by selectively promoting the hot pages from PM to DRAM, MULTI-CLOCK achieves a slightly better performance on average than other tiering mechanisms across different workloads.

In Section II-A, we analyzed the workloads from various benchmarks to show the existence of DRAM-friendly and Tier-friendly pages. The goal of MULTI-CLOCK is to identify these pages and place the frequently accessed pages in the DRAM tier. Workloads with weak locality will not have such a division of pages and would not benefit from MULTI-CLOCK. On the other hand, workloads with strong locality will have many DRAM and Tier friendly pages and can reap benefits from the dynamic tiering capabilities of MULTI-CLOCK. Among the YCSB workloads, workload D inserts new data in PM (as

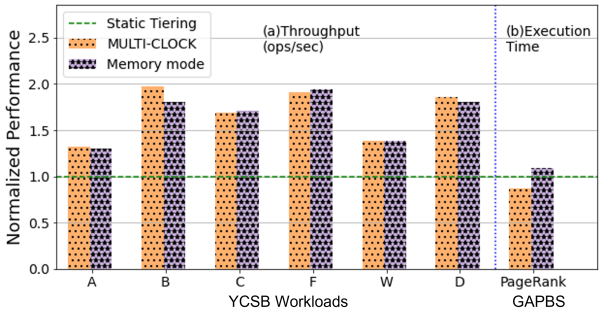

Fig. 7. **Performance comparison of MULTI-CLOCK with Memory-mode.** Y axis presents the normalized performance in (a) throughput (higher is better) and (b) execution time (lower is better).

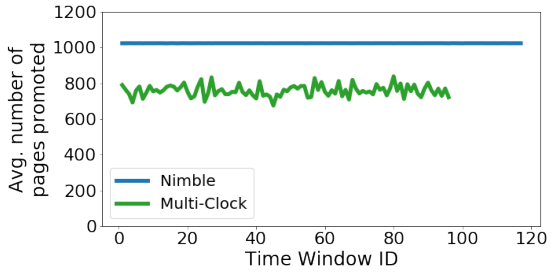

Fig. 8. **The average amount of pages promoted in each scan over time.** Y-axis is the average number of pages that are promoted in 20 seconds window. X-axis presents the time window IDs.

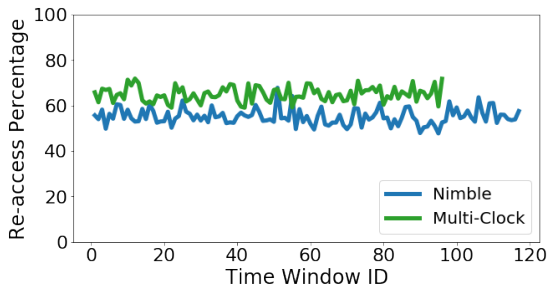

Fig. 9. **The average re-access percentage of the promoted pages in each scan.** Y-axis is the average number of promoted pages that got re-accessed. The average is calculated based on a time window of 20 seconds. X-axis presents the time window IDs.

DRAM is already full) and frequently accesses the recently inserted data, thereby exhibiting a stronger locality than the other workloads. In comparison with static tiering, MULTI-CLOCK obtains the greatest performance gain (132%) for this workload.

2) Comparison With Memory-mode:: Finally, we compare the performance of MULTI-CLOCK with Memory-mode. As Memory-mode uses all of the DRAM capacity for caching, to allow for a competitive comparison with MULTI-CLOCK, we set the workload size to be 4x of the available DRAM capacity.

In Figure 7, the Y-axis shows performance normalized to that of static tiering. Figure 7(a) shows the normalized throughput for the YCSB workloads and Figure 7(b) shows the normalized execution time for the GAPBS's PageRank algorithm. For the YCSB workloads, MULTI-CLOCK outperforms Memory-mode by as much as 9% and operates within 2% of Memory-mode's performance. For PageRank, MULTI-CLOCK outperforms Memory-mode by 21%. To improve application performance, Memory-mode uses all the available DRAM as cache, thus hiding the available DRAM capacity from the applications; it achieves as much as 2% better performance than MULTI-CLOCK. On the other hand, MULTI-CLOCK exposes all the available DRAM and PM capacity to the application and provides performance that is either better or very similar to Memory-mode.

D. Performance Analysis

To understand the reason behind MULTI-CLOCK's better performance outcomes, we first analyze the number of pages promoted by MULTI-CLOCK and Nimble. Then we see how many of these promoted pages are getting re-accessed again from the DRAM tier. This discussion helps us understand MULTI-CLOCK in more detail.

1) Number of page promotions: In Figure 8, we report the number of pages being promoted across tiers for both MULTI-CLOCK and Nimble. In Figure 8, the Y-axis shows the average number of pages promoted in a time window. We chose the time window as twenty seconds. The X-axis represents the time window ID. As we can see from the figure, the average number of pages Nimble promotes is always 1024. This is because Nimble always selects a fixed number of pages for promotion, and we used 1024 as the fixed value. On the other hand, MULTI-CLOCK promotes 758 pages on average per scan. Similar to Nimble, MULTI-CLOCK scans a maximum of 1024 pages, but unlike Nimble, MULTI-CLOCK selects the pages that have been recently accessed multiple times. If pages that do not get re-accessed again in the future get promoted to DRAM, then the overhead to promote such pages can hurt system performance.

2) Percentage of Pages Re-accessed: Now, we analyze the number of pages that have been promoted in the last scan, get re-referenced again from the DRAM. In Figure 9, the Y-axis shows the re-access percentage, which represents the average percentage of the recently promoted pages which have been re-accessed. The average percentage is calculated for 20 second time window. The time window IDs are shown on the X-axis. From Figure 9, we can see that pages promoted by MULTI-CLOCK have 15% higher re-access percentage than Nimble. In combination with Figure 8, we come to an interesting observation. Nimble promotes more pages than MULTI-CLOCK, but a lower percentage of the promoted pages are re-accessed again. This explains the improved performance results that we observed with YCSB and GAPBS workloads.

E. Scanning Interval Sensitivity

As described in Section IV, the `kpromoted` daemon wakes up after a specific time interval. `kpromoted` is responsible for moving pages from the inactive list to the active list, from the active list to the promote list, and from the promote list to the DRAM tier. Varying this time interval in MULTI-CLOCK is expected to affect the performance of the application. We set

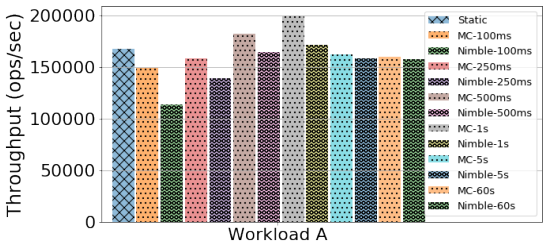

Fig. 10. **Throughput comparison of Static tiering, MULTI-CLOCK, and Nimble with different scan intervals for YCSB Workload A.** Y axis presents throughput (higher is better).

the time interval to 100ms, 250ms, 500ms, 1s, 5s, and 60s and run the workload A from YCSB with each of these MULTI-CLOCK versions. Nimble uses a similar daemon thread to promote pages periodically. Similar to MULTI-CLOCK, we also evaluated Nimble with different time intervals. From Figure 10 we see that overall MULTI-CLOCK performs better when compared to Nimble. For larger scan intervals above 5s, we do not observe much difference due to the lag in the reaction time. The one-second scan interval was found to be the best performing for various workloads, but in Figure 10, we only show the results for YCSB workload A as a representative. Hence, we chose the one-second scanning interval for all the other evaluations for MULTI-CLOCK and Nimble.

F. Overhead

Mainly the overhead of MULTI-CLOCK includes the overhead for promotion and demotion of the pages across different tiers. While running memory access intensive applications, the overhead depends on which tier the pages are being accessed from. First, if DRAM pages are heavily accessed, then there will be no overhead due to no migration being incurred. Second, if pages from the PM tier are heavily accessed, then to reduce access latency, MULTI-CLOCK will identify these pages and promote them to DRAM, incurring promotion overheads as well as demotion overheads if the DRAM is full. However, if the application is memory intensive, then the promoted pages would be accessed repeatedly from the DRAM tier, which can benefit the application due to DRAM's lower access latency. Thus, for memory-intensive workloads, MULTI-CLOCK's benefit will surpass the migration overhead.

VI. Related Work

Emerging persistent memory technologies show promise in three distinct areas: non-volatility, very large capacity (as compared to DRAM), and performance suitable for direct load/store access by the CPU. Most studies on persistent memory, far too many to list here, focus on the non-volatility, using it to replace or extend block storage, implement persistent caches, or explore the persistent execution of processes that can survive power failures [27]–[31]. In contrast, our work focuses on the large capacity characteristic of persistent memory and the ability to directly read, write, and execute data residing in persistent memory.

There have been many studies that explore the use of different types of memory for the building of *hybrid memory systems*. Such systems make use of the different characteristics of the available memory types to combine them into a hybrid solution. Most hybrid memory systems do not establish any specific hierarchy between the different memory types as tiered memory systems do. As discussed in Section II-D, Thermostat [19], Nimble [11], AMP [22], AutoNUMA-Tiering [20], and AutoTiering [12] are the recent works on dynamic tiered memory system.

Yang [32] proposes a design to use persistent memory as a NUMA node efficiently. This tiered design is aware of both DRAM and PM nodes and handles promotion/demotion for anonymous pages only via NUMA balancing. In contrast, MULTI-CLOCK selects pages for promotion more carefully by scanning pages periodically and moving them across inactive, active, and newly added promote list depending on page access. Moreover, MULTI-CLOCK is capable of managing all types of pages, anonymous and file-backed pages, making MULTI-CLOCK a complete solution.

Qureshi et al. [33], Dhiman et al. [34], Ramos et al. [35], and Lee et al. [36] propose hybrid memory systems, where DRAM is used as buffer cache, PM is used as the DRAM's extension, and a hardware-based solution is used to find best page replacement policy. In contrast to these works, we provide a page selection mechanism that can be used to improve the performance of a dynamic tiered memory system without any hardware modification, where DRAM and PM both co-exist as system main memory.

Many replacement algorithms have been studied in the past in the context of caching [37]–[43]. Our solution is orthogonal to these efforts and builds upon existing memory replacement mechanisms, and presents a modified page migration and replacement algorithm for tiered memory.

Liu *et al.* [44] provide object-level memory allocation and migration in hybrid memory systems. Data placement and migration at the object granularity requires modification of the existing application to use the new APIs. In contrast, MULTI-CLOCK operates seamlessly at the kernel level, and existing applications can be run as-is without any modification.

VII. Discussion

MULTI-CLOCK relies on the page reference bit for classifying pages according to their frequency of accesses and characterizing the *importance* of a page. In the current version, MULTI-CLOCK does not differentiate between the data read and write. One possible improvement to this approach is to also include the dirtiness information for memory pages in a weighted formula to compute the *importance* of a page. By including this extra information, we could weigh the different types of accesses for a page (read or write) in the decision of page placement. This additional information becomes particularly relevant when the underlying memory hardware exhibits non-uniform latency for the different types of accesses. For instance, some PM devices, e.g., Intel Optane PM, are known to have asymmetric read and write latencies.

The scanning interval for MULTI-CLOCK is 1s as we discussed in Section V-E. We compared the performance of MULTI-CLOCK across multiple scanning intervals and chose the 1s scan interval. However, it could be valuable to dynamically adjust the scanning interval for `kpromoted` by analyzing the characteristics of the running workload. Additionally, it will also be interesting to see the performance of MULTI-CLOCK with varying DRAM and PM ratios.

VIII. Conclusions

Byte-addressable, high capacity memory such as PM opens up a new space for optimization of the memory system design and implementation. In this work we design and develop MULTI-CLOCK, a dynamic memory tiering system that is designed to ensure that the right data is in the right tier at the right time. Unlike some other recent approaches for tiered systems, MULTI-CLOCK uses both access recency and frequency to identify potential pages for migration without adding significant system overhead. We deployed MULTI-CLOCK in a real system by developing a prototype that runs CentOS 7 (Linux kernel 5.3.1) and evaluated our prototype using graph processing and key-value store workloads. Our results demonstrate that MULTI-CLOCK is able to significantly improve the performance of these workloads compared to the state-of-the-art techniques without compromising the amount of usable main memory made available to these workloads. MULTI-CLOCK sources can be downloaded at https://doi.org/10.5281/zenodo.5790897

Acknowledgments

We would like to thank the reviewers of this paper for insightful feedback that helped improve the content and presentation of this paper substantially. This work was supported in part by NSF grants CCF-1718335, CNS-1956229, and CNS-2008324.

References

[1] "Apache Spark," http://spark.apache.org, January 2012.

[2] "Sap hana," http://hana.sap.com, January 2020.

[3] B. Fitzpatrick, "Distributed caching with memcached," in *Linux Journal*, 2004.

[4] Salvatore Sanfilippo and Pieter Noordhuis, "Redis," http://redis.io, January 2020.

[5] A. Makarov, V. Sverdlov, and S. Selberherr, "Modeling emerging non-volatile memories: Current trends and challenges," in *Proceedings of the International Conference on Solid State Devices and Materials Science*, ser. SSDM '12, April 2012.

[6] S. Mittal, "Energy saving techniques for phase change memory (PCM)," in *arXiv:1309.3785*, 2013.

[7] "Intel Optane DC Persistent Memory," 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, July 2021.

[8] I. Peng, M. Gokhale, and E. Green, "System evaluation of the intel optane byte-addressable nvm," in *arXiv:1908.06503*, 2019.

[9] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. Soh, Z. Wang, Y. Xu, S. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane dc persistent memory module," in *arXiv:1903.05714*, 2019.

[10] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies*, ser. MSST '15, June 2015.

[11] Z. Yan, D. Lustig, D. Nellans, and A. 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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 331–345. [Online]. Available: https://doi.org/10.1145/3297858.3304024

[12] J. Kim, W. Choe, and J. Ahn, "Exploring the design space of page management for multi-tiered memory systems," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 715–728. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the ACM symposium on Cloud computing*, ser. SoCC '10, June 2010.

[14] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," in *arXiv:1508.03619*, 2015.

[15] OW2 Consortium, "RUBiS: Rice University Bidding System," http://rubis.ow2.org/.

[16] SPECpower_ssj2008, "https://www.spec.org/power/."

[17] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[18] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, "Single machine graph analytics on massive datasets using intel optane dc persistent memory," in *arXiv:1904.07162*, 2019.

[19] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," *SIGPLAN Not.*, vol. 52, no. 4, p. 631–644, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3093336.3037706

[20] H. Ying, "tiering-0.6," https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/commit/?h=tiering-0.6, 2020.

[21] R. van Riel and V. Chegu, "Automatic numa balancing, 2014," https://www.redhat.com/files/summit/2014/summit2014_riel_chegu_w_0340_automatic_numa_balancing.pdf, 2014.

[22] T. Heo, Y. Wang, W. Cui, J. Huh, and L. Zhang, "Adaptive page migration policy with huge pages in tiered memory systems," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[23] , "Utility library for managing the libnvdimm," https://github.com/pmem/ndctl, January 2020.

[24] B. Cooper and N. Bailey, "Ycsb core workloads," https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads, October 2010.

[25] Y.-Y. Jo, J. Hong, M.-H. Jang, J.-G. Bang, and S.-W. Kim, "Data locality in graph engines: Implications and preliminary experimental results," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1885–1888. [Online]. Available: https://doi.org/10.1145/2983323.2983865

[26] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 56–65.

[27] J. Coburn, A. Caulfield, A. Akel, L. Grupp, R. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011.

[28] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the ACM Symposium on Operating Systems Principles*, ser. SOSP '09, 2009.

[29] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the European Conference on Computer Systems*, ser. EuroSys '14, 2014.

[30] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, ser. FAST '14, February 2014.

[31] J. Yang, Q. Wei, C. Chen, C. Wang, and K. L. Yong, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *Proceedings of the USENIX Conference on File and Storage Technologies*, ser. FAST '15, February 2015.

[32] Y. Shi, "Another approach to use pmem as numa node," https://lore.kernel.org/linux-mm/1553316275-21985-1-git-send-email-yang.shi@linux.alibaba.com/, March 2019.

[33] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high-performance main memory system using phase-change memory technology," in *Proceedings of the International Symposium on Computer Architecture*, ser. ISCA '09, 2009.

[34] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: a hybrid pram and dram main memory system," in *Proceedings of the Annual Design Automation Conference*, 2009.

[35] L. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the 25th International Conference on Supercomputing*, ser. ICS '11, 2011.

[36] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency dram: A low latency and low cost dram architecture," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, ser. HPCA '13, 2013.

[37] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *USENIX File and Storage Systems (FAST)*, 2008.

[38] N. Megiddo and D. S. Modha, "Outperforming lru with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, 2004.

[39] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "Cflru: a replacement algorithm for flash memory," in *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, October 2006.

[40] T. M. Wong and J. Wilkes, "My cache or yours? making storage more exclusive," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC '02, June 2002.

[41] Y. Zhou, J. F. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the 2001 USENIX Annual Technical Conference*, ser. USENIX ATC '01, June 2001.

[42] S. Jiang, F. Chen, and X. Zhang, "Clock-pro: An effective improvement of the clock replacement," in *Proc. of the USENIX Annual Technical Conference*, 2005.

[43] S. Bansal and D. Modha, "Car: Clock with adaptive replacement," in *Proc. of the Third USENIX Conference on File and Storage Technologies*, 2004.

[44] H. Liu, R. Liu, X. Liao, H. Jin, B. He, and Y. Zhang, "Object-level memory allocation and migration in hybrid memory systems," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1401–1413, 2020.