Adaptive Page Migration Policy with Huge Pages in Tiered Memory Systems

Taekyung Heo*, Yang Wang†, Wei Cui†, Jaehyuk Huh*, Lintao Zhang†
*KAIST †Microsoft Research Asia
taekyung.heo@kaist.ac.kr, t-yangwa@microsoft.com, weicu@microsoft.com,
jhhuh@kaist.ac.kr, lintaoz@microsoft.com

Abstract— To accommodate the growing demand for memory capacity in a cost-effective way, multiple types of memory are incorporated in a single system. In such tiered memory systems consisting of small fast and large slow memory components, accurately identifying the performance importance of pages is critical to properly migrate hot pages to fast memory. Meanwhile, growing address translation cost due to the increasing memory footprints, helped adopting huge pages in common systems. Although such page hotness identification problems have existed for a long time, this paper revisits the problem in the new context of tiered memory systems and huge pages. This paper first investigates the memory locality behaviors of applications with three potential migration polices, least-recently-used (LRU), least-frequently-used (LFU), and random with huge pages. The evaluation shows that none of the three migration policies excel the others, as the effectiveness of each policy depends on application behaviors. In addition, the results show huge pages can be effective even with page migration, if a proper migration policy is used. Based on the observation, this paper proposes a novel dynamic policy selection mechanism, which identifies the best migration policy for a given workload. It allows multiple concurrently running workloads to adopt different policies. To find the optimal one for each workload, this study first identifies key features that must be inferred from limited approximate memory access information collected using accessed bits in page tables. In addition, it proposes a parallel emulation of alternative policies to assess the benefit of possible alternatives. The proposed dynamic policy selection can achieve 23.8% performance improvement compared to a prior approximate mechanism based on LRU lists in Linux systems.

Index Terms—Tiered Memory, Page Hotness, Page Migration, Huge Pages

1 Introduction

Memory systems are adopting multiple types of memory with different performance and capacity characteristics to increase the memory size in a cost-effective way. More expensive fast memory is backed by slower but higher capacity memory components, forming tiered memory systems. For capacity-optimized slow memory, non-volatile memories are already commercially available in the market, offering ∼300ns latency [1], [2]. In such tiered memory systems, accurately identifying the hotness of pages is critical to take advantage of the low-cost emerging memories while minimizing the performance loss.

Such page migration or replacement policies have been extensively studied for hardware caches, storage caches, and page management for virtual memory supports [3], [4], [5], [6], [7], [8], [9], [10], [11]. However, the emerging tiered memory systems provide new environments which are different from the prior work. The prior page replacement problems are restricted for choosing which data should be selected as victims to make room for new data. An access to the slow component triggers an immediate promotion of the data to the fast component (cache). Unlike such replacement problems, the page migration problem must find which data must reside in the fast memory at a given time, while the data in the slow memory are accessible not necessarily triggering promotions.

In addition, compared to the storage caching systems, memory access traces collected from accessed bits in page tables are quite incomplete and approximate by their nature. Compared to the traditional page swapping, the tiered memory systems have much smaller latency and bandwidth differences between fast and slow memories than those of the prior DRAM and storage devices. Migrations between the two memory types are much more frequent than the prior studies as the costs of migration are relatively low. The new environments raise the need for revisiting the policy space of page migration in the context of tiered memory systems.

In the mean time, increasing memory footprints caused excessive misses in translation lookaside buffers (TLBs) for address translation. To reduce the cost of address translation, 2MB huge pages have been adopted in x86 systems, which reduces TLB misses significantly for applications which otherwise suffer from the costs of TLB misses. While huge pages can drastically improve the address translation efficiency, their interaction with migration policies for tiered memory systems is yet to be investigated. Improving the accuracy of hotness measurement for huge pages and the efficiency of page migration for huge pages for tiered memory systems have been investigated by the prior studies [12], [13]. This paper explores the migration policy space with huge pages.

With such huge pages, this paper first investigates three widely accepted policies for page migration in tiered memory systems. It evaluates least-recently-used (LRU), least-frequently-used (LFU), and random policies for their behaviors on a range of applications. (i) Our investigation first shows that none of the policies excels the others for the range of applications, since each application has a different memory access behavior preferring a different migration policy. However, the page migration policies used by recent tiered memory studies are fixed to a single policy, such as a variant

of LRU lists in Linux systems [13]. (ii) If a proper migration policy for each application is used, our investigation shows that huge pages can be effectively used for migration in tiered memory systems, reaping the benefit of efficient address translation.

Based on the observation of the migration policy evaluation, this paper proposes a new dynamic policy selection technique tuned for huge pages in tiered memory systems, called *Adaptive Migration Policy (AMP)*. AMP constantly collects memory access information using accessed bits in page tables, and periodically selects the best policy out of the three potential policies. To identify the best policy for a given workload, the study first identifies which features of memory access behaviors are highly correlated to the policy selection.

The first feature is used to identify workloads favoring random page placements. When accessed pages exceed the fast memory capacity, the locality cannot be captured effectively with the tiered memory system. Therefore, the random policy, which does not migrate pages actively, results in the best performance without migration overheads. If the workload exhibits a certain level of locality, either LRU or LFU is selected. To select the best one out of the two policies, AMP maintains shadow page location states, which virtually mimics page migration. For example, if the current best policy is not LFU, the LFU policy is emulated with the shadow page location, and its effectiveness is tracked with the emulation. Based on the estimated effectiveness from the shadow state and the measured effectiveness from the current memory state, the better policy is selected for the next round.

Since the proposed technique can be applied for each memory control group (`memcgs`) in Linux, workloads in different memory control groups can select their own best policy. It allows the consolidated system to choose per-group optimal migration policies, allowing fine-tuning migration policies for co-running workloads.

We implement AMP in a Linux system, spanning from the kernel modification to the user-level components. The kernel is modified to track the recency and frequency of page accesses with accessed bits in page tables, and to provide policy options. The user-level components evaluate the features for each memory control group and apply the best policy for each group periodically.

We evaluate AMP in a Linux system with an emulated tiered memory system. The memory bandwidth of the slow memory node is throttled and saturated to emulate the slow memory. AMP can improve the performance of selected applications by 23.8% compared to the LRU lists adopted in the prior work [13]. Furthermore, AMP can achieve 10.9, 6.4, 17.6% higher performance than LRU, LFU, and Random, respectively.

The contributions of our study are as follows:

- We find that workloads have diverse preferences on page migration policies in tiered memory systems, and we analyze the reason behind the page migration policy preferences.
- Our investigation shows that huge pages can be effective with page migration in tiered memory systems, if a proper migration policy is used.
- We define several features that have a relationship with the performance of page migration policies: fast memory hit ratio, page migration stability, and accessed page ratio. After that, we analyze the correlation between the features and the performance.
- We propose AMP, which dynamically selects a page migration policy between LRU, LFU, and Random using the features.

The rest of the paper is organized as follows. Section 2 describes the background of tiered memory and page migration policies. Section 3 investigates the behaviors of different page migration policies in the tiered memory system. Section 4 analyzes the critical features for determining the best policy, and Section 5 presents the implementation. Section 6 presents the experimental results, and Section 7 discusses the remaining issues. Section 8 concludes the paper.

2 Background and Related Work

This section discusses the page migration problem in tiered memory systems, with its differences from the prior cache replacement along with the adoption of huge pages.

2.1 Tiered Memory Systems and Huge Pages

Tiered Memory Systems: A memory system composed of memories with various performance characteristics is called *a tiered memory system* [13]. The future memory systems are expected to be tiered memory systems due to the scaling limit of DRAMs. To increase the capacity of memory systems, memory systems are adopting non-volatile memories [1], [14], memory disaggregation [15], [16], [17], [18], and memory compression [19]. Usually, a tiered memory system is composed of fast memory and slow memory. Fast memory has a shorter latency and higher bandwidth compared to slow memory. A tiered memory system can be managed in hardware or in software. In this study, we assume a tiered memory system where an OS is responsible for managing data between two memory types. Data can be migrated at the page granularity, and the OS makes the page location and migration decisions.

Huge Pages for Efficient Address Translation: Future memory systems are expected to have TBs of memory with various latency and bandwidth [20]. In a memory system with a huge amount of memory capacity, address translations become a critical problem. Modern computer systems adopt virtual memory for efficient memory management. Therefore, virtual addresses should be translated to physical addresses to access data, and the mappings are maintained in a page table. The page table resides in main memory, and translation lookaside buffers (TLBs) cache page table entries to avoid costly memory accesses. The memory capacity that can be translated by a TLB is called *TLB reach*. The problem is that the TLB size is limited to thousands of entries to shorten the access latency to TLBs. This limits the TLB reach of TLBs. The TLB reach can be increased with huge pages [21], [22]. While a single TLB entry can cover a 4KB address space with 4KB base pages, a TLB entry can cover several MBs or GBs with huge pages.

Previously, huge pages had several performance issues such as the increased page fault latency, memory bloating, unfair huge page allocations, and losing the page sharing opportunities [23]. Thanks to the recent studies to mitigate the problems [23], [24], [25], huge pages are becoming a viable option. Moreover, a multi-threaded page migration mechanism for transparent huge pages [13] makes migrating pages at the huge page granularity feasible. Therefore, in this study, we use 2MB huge pages as a default page migration unit.

2.2 Page Migration Policies in Tiered Memory Systems

In this paper, we define *a page migration policy* as a policy that decides page locations in tiered memory systems, and it is

used interchangeably with page hotness selection. Page migration policies try to fill fast memory with performance-critical hot pages. The page migration problem differs from the traditional cache replacement or page swapping, since any access to the slow memory does not necessarily trigger the promotion of the page to the fast memory. In hardware caches, memory blocks are immediately inserted to the cache for handling misses. The page swapping also requires to move a swapped out page from the storage to the memory to resolve the page fault. Unlike the cache replacement problems, the page migration problem needs to address which part of memory should reside in the fast memory, while the data in the slow memory are still accessible without migrating to the fast memory.

In addition, the cost and performance characteristics of tiered memory systems should be considered in the design and implementation of page migration policies. For example, maintaining an LRU stack is costly for virtual memory and architectural caches. In virtual memory, CLOCK approximates LRU with a single reference bit [26]. In CPU caches, tree-based pseudo-LRU is used to lower the area overhead [27], [28]. Likewise, cache replacement policies should be adopted to tiered memory systems with the consideration of the cost and performance characteristics. In this section, we summarize the prior studies on page migration policies.

❶ ***Recency-based policies.*** Native Linux systems have LRU lists to reclaim pages under memory pressure. The native LRU lists approximate LRU with two LRU lists. One is the active list, and the other is the inactive list. The membership of pages is controlled by the heuristic implemented in the kernel. The kernel uses the accessed bits of pages and page types to update the membership of pages. Each page has an accessed bit in the page table entry (PTE). When a page is accessed, the corresponding accessed bit is set. As the goal of LRU lists is to reclaim pages under memory shortage, the native LRU lists do not update the membership of pages in non-memory pressure conditions.

❷ ***Frequency-based policies.*** Access frequency has been widely adopted in the page management for tiered memory systems. In modern processors, the exact access frequency of a page cannot be obtained due to the lack of hardware support. Instead, access frequency can be estimated using accessed bits. The accessed bit of a page is periodically checked and recorded to a per-page bit vector. An access frequency can be calculated by averaging the number of bits set in the bit vector. HeteroVisor [29] tracks the access frequency of pages to decide page locations between fast die-stacked DRAMs and slow off-chip DRAMs. If the access frequency of a page exceeds a predefined hot page threshold, the page is classified as hot and migrated to the fast die-stacked DRAM. On the other hand, Thermostat [12] finds the access frequency threshold using a user-specified maximum allowable slowdown.

❸ ***Limiting the promotion rate.*** In a tiered memory system where pages have to be migrated from slow memory to fast memory on every page access to slow memory, limiting the number of page migrations from slow memory to fast memory is a way to guarantee the performance slowdown [19]. The rate of pages migrated from slow memory to fast memory within a time window is defined as *the promotion rate*. Lagar-Cavilla et al. find the relationship between the page access recency and promotion rate, and the pages that are expected to show a low promotion rate are identified by measuring the access recency of a page.

2.3 Other Prior Work for Cache Replacement

In the prior cache replacement, unlike page migration, the promotion to the cache is immediately triggered while handling misses. Therefore, the hotness selection within the cache is focused on which data should be evicted as victims to make room for newly inserted data. For selecting the victims, there have been many different approaches with their own advantages and disadvantages. In this section, we summarize the prior studies on cache replacement policies.

❶ ***Recency-based policies.*** The most common cache replacement policy is the Least Recently Used (LRU) replacement policy [3]. LRU maintains an LRU stack, which is sorted by the access recency. The most recently accessed page is placed at the top of the LRU stack, and the least recently accessed page is placed at the bottom. Once a page is referenced, the page is removed from its position and moved to the top of the stack. On cache evictions, LRU replaces the least recently accessed page. Although LRU works well for most memory access patterns, LRU fails with scanning memory access patterns and loops. Scanning memory access patterns access pages only once and never use them again. Therefore, caching recently accessed pages wastes the cache space. Loops that access pages larger than the cache size evict pages that are accessed in the near future. Prior studies try to mitigate the limitations of LRU by identifying the anomalies [30], [31], [32].

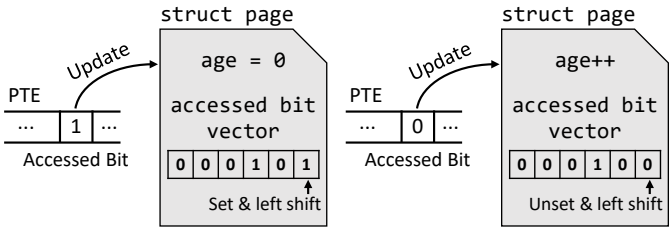
❷ ***Frequency-based policies.*** Least Frequently Used (LFU) tracks the access frequencies of pages and evicts a page with the least access frequency. Although LFU has an advantage that it can exploit the long-term access history, it has several disadvantages. LFU can make a wrong decision in the initial stage due to the lack of access history. Moreover, LFU may hold stale pages that are not hot anymore due to their previous access history.

❸ ***Adaptive policies.*** There have been many prior studies to improve the hit ratio of caches [33], [34], [4], [35], [7], [10]. Among the prior studies, we focus on the adaptive selection of cache replacement policies [36], [5], [6], [8], [9], [11], [37]. Abstractly, ARC [8] and CAR [9] partition a cache into two partitions for recently accessed pages and frequently accessed pages. The partition sizes are dynamically adjusted based on the workloads' behavior. A few prior studies adopt machine learning techniques to choose a cache replacement policy adaptively [6], [37]. In CPU caches, set dueling has been proposed to dynamically choose a policy between two competing policies [11]. Cache sets are divided into dedicated sets and follower sets. A small number of sets are allocated to dedicated sets, and the dedicated sets are managed with two competing policies to evaluate the performance of the policies. The policy that performs better is applied to the follower sets.

3 Migration Policies with Huge Pages

3.1 Migration Policies

To determine which pages should be placed in the fast memory, the page hotness selection mechanism consists of two components. First, it must record the access history of each huge page. The second component chooses hot pages based on the access histories of all pages. In this section, we investigate the page migration policies using huge pages (2MB). Improving the accuracy of hotness measurement for huge pages and facilitating the page migration for huge pages for tiered memory systems have been



(a) When accessed bit is set (b) When accessed bit is unset

Fig. 1: Tracking page age and access frequency using accessed bits

investigated by the prior studies [12], [13]. This paper explores the migration policy space with huge pages.

Although the main reason for using huge pages is to mitigate the cost of TLB misses, it also allows reducing the complexity of page managements for migration. We track the access information for each huge page to reduce the overhead of tracking the information. In addition, it also reduces the latency to select hot pages, as the number of candidate pages is significantly reduced by huge pages. A possible downside of using huge pages in tiered memory system is the potential waste of some fast memory, when only a small subset of huge pages are actively accessed. However, this paper is focused on page management based on huge pages, since huge pages were effective across all benchmark applications in our study, as shown in Section 3.3.

Tracking access history: To record the access history of each page, we use the accessed bit in each page table entry, which is set by the processor hardware. Figure 1 illustrates how the page history information of each page tracked. For each huge page, we record two types of access information, *age* and *history*. Every five seconds, the kernel checks the accessed bit of each huge page to update its age and history. Once the information is updated, the accessed bit in the page table entry is reset.

Age represents the recency of access for a page. As shown in the figure, if the accessed bit is set during the 5-second period, the page *age* is reset to 0. If the accessed bit is not set (no access for the last five seconds), its age is increased by 1. Therefore, the age of a page means when the last access for the page occurred at 5-second granularity.

History (accessed bit vector) represents the accessed bit vector for each page for the past n periods. Figure 1 shows that six bits are used for the access history. A new accessed bit is pushed to the tail of the vector. Although the accessed bit vector can be used for representing *age*, we use a separate age variable to record the age of longer periods without incurring a long bit vector for each page. The detailed accessed bit vector covers a shorter time range than the age variable. In this paper, we use 64 bits for each huge page for the accessed bit vector.

As will be discussed later, the proposed memory manager allows adjusting how much fast memory and slow memory can be used for each application group. If the number of application groups is set to one, all applications share the entire fast and slow memory.

Every five seconds, the access information of all huge pages is updated, and the set of pages which must be in the fast memory are determined. For the pages which are selected for the fast memory, but not already in the fast memory, the migration of the pages are initiated. Since we use huge pages, this step does not incur significant overheads. First, the number of huge pages for a given application footprint is 512 times smaller than that of base pages. Therefore, tracking and sorting huge pages cause only a small extra overhead during the five second period. Second, if a good migration policy is selected, only a small fraction of pages are migrated to the fast memory, since it is likely that many hot pages are already in the fast memory.

Migration policies: Among various page migration policies, we investigated the workloads' preference on policies of LRU, LFU, and Random. ❶ LRU tracks the access recency of pages with ages, and the most recently accessed pages are migrated to the fast memory. Using the age information for all pages, pages with the lowest ages are considered hot. ❷ LFU tracks the access frequency of pages, and the most frequently accessed pages are located in fast memory. The access frequency is obtained with the per-page accessed bit vectors, by averaging the number of bits set in the bit vector. ❸ Random simply fills the fast memory first, once the fast memory is filled, the slow memory is allocated. In Random, if a page located in the fast memory is freed, a random page from the slow memory is migrated to the fast memory.

3.2 Results

Methodology: We find the workloads' preferences on page migration policies by measuring the performance of workloads on an emulated tiered memory system. If a policy can accurately identify hot pages, the policy can present a higher performance compared to the other policies. We compose a tiered memory system on a NUMA system by throttling the memory bandwidth of one node. The bandwidth-throttled node becomes a slow memory node. 2MB transparent huge pages are used, and pages are migrated at the huge page granularity. In the rest of paper, *fast memory ratio* is the ratio of the fast memory over the entire memory footprint, which is set to 50% in the evaluation of this section. 4B is used to track the page age, and 8B is used to track the access frequency. We evaluate the performance of page migration policies with selected workloads from SPEC CPU 2017 [38], CloudSuite [39], NPB [40], and graph500 [41]. In the remaining part of this section, we describe the reason behind the page migration policy preferences.

Figure 2 presents the normalized performance of workloads with various page migration policies. We measure the execution time, and the performance is the reverse of execution time. The performance is normalized to the performance with the 100% fast memory ratio. Workloads can be classified into four groups: LRU-favor, LFU-favor, Random-favor, and neutral, as shown in the figure.

❶ ***LRU-favor.*** Workloads with strided memory access patterns favor LRU. Strided memory access patterns sequentially access pages with the same distance between memory accesses with low data reuse. Therefore, keeping frequently accessed pages in fast memory may degrade the performance. LRU can keep the recently accessed pages in fast memory, increasing the probability of accessing fast memory. `mcf`, `cactus`, and `cg.D.x` have strided memory access patterns. `mcf` has a pointer chasing in `price_out_impl` as shown in Code 1 [42]. The pointer chasing pattern of `mcf` is actually a strided memory access pattern because the data structures are sequentially located in a virtual address space [43], [42]. `cactus` strides over one dimension of a matrix while working on a multi-dimensional matrix [44]. `cg.D.x` calculates the eigenvalues of a sparse matrix using the conjugate gradient method. It operates on a large matrix, and the matrix is accessed sequentially with low data reuse [45].

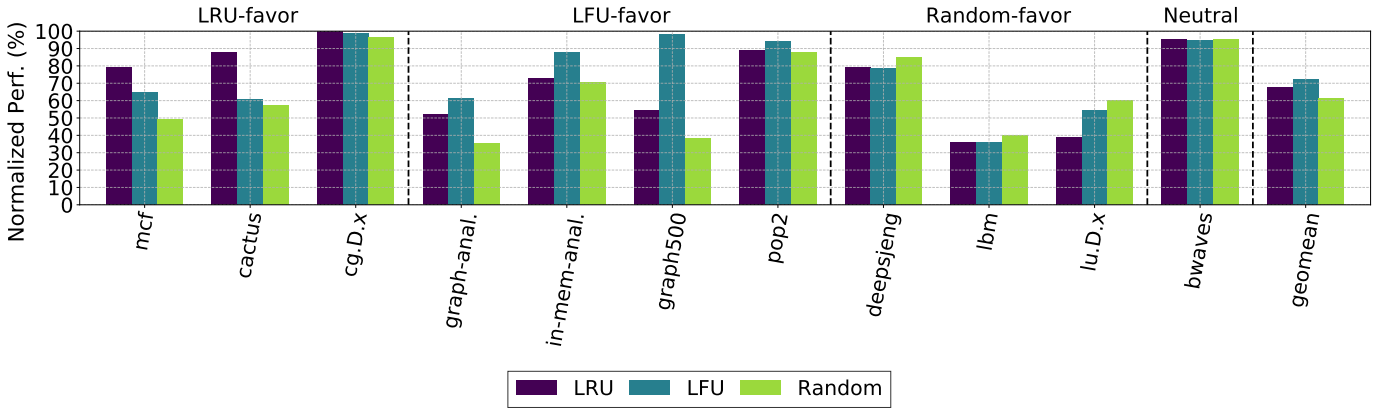


Fig. 2: Normalized performance of workloads with various page migration policies

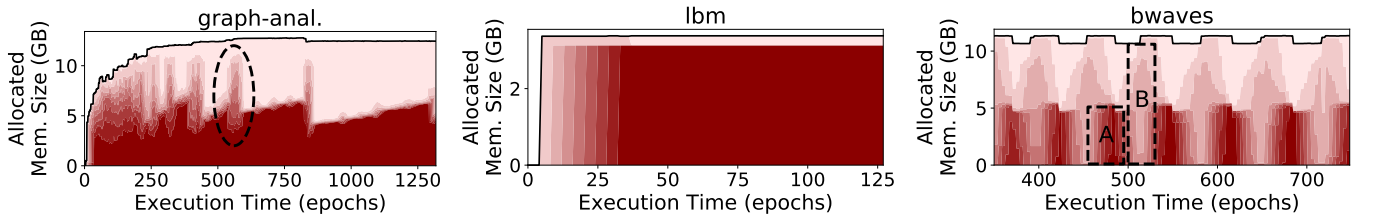


Fig. 3: Temporal change of access frequencies of pages. Pages are sorted by their access frequency, and the pages with higher access frequencies are drawn at the bottom with a darker color.

```
long price_out_impl(network_t *net)
{
...
    iterator = first_list_elem->next;
    while (iterator) {
        arcin = iterator->arc;
        tail = arcin->tail;
        ...
        iterator = iterator->next;
... } ...
}
```

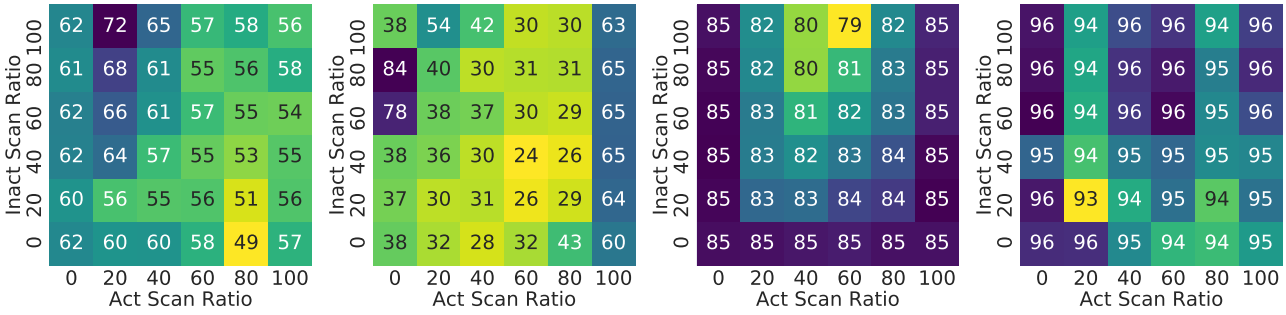
Code 1: A function that shows a strided memory access pattern in `mcf`

❷ ***LFU-favor.*** Workloads with frequently accessed data structures favor LFU. For the workloads, LFU can keep the frequently accessed hot data in fast memory, and it can protect the fast memory from being polluted by recently accessed cold data. `Graph-analytics` runs the PageRank algorithm, and it updates the ranks of neighbor vertices while walking on vertices. A vertex with more neighbors tends to be accessed more frequently. LFU can identify the vertices with more neighbors and keep them in fast memory. The first subfigure of Figure 3 shows the temporal change of access frequencies of pages of `graph-analytics`. The access frequencies of pages are tracked by checking the accessed bits of pages on every one epoch, whose length is four seconds. The size of the per-page accessed bit vector is 8-bit. The pages with higher access frequencies are drawn at the bottom, and the pages with lower access frequencies are illustrated at the top of the figure. `Graph-analytics` has several scanning memory access patterns. We emphasize one of the scanning patterns with an ellipse in the figure. LRU fails to keep frequently accessed hot pages in fast memory due to the scanning patterns.

`In-memory-analytics` runs the alternating least squares algorithm. It trains a model multiple times with various parameters to find the best parameters. On every training, the training dataset is loaded by scanning the dataset. LFU can keep the frequently accessed trained model in fast memory. `graph500` runs the breadth-first search (BFS) algorithm multiple times. On every BFS, `graph500` validates the result with the `validate_result()` function. Therefore, the graph is accessed frequently, and `graph500` favors LFU.

❸ ***Random-favor.*** Workloads with low locality favor Random. First, workloads that have pages with similar access recency or frequency prefer Random. If all pages are equally recently accessed or frequently accessed, migrating pages with LRU or LFU adds the performance overhead without benefit, and Random can eliminate the overhead. `lbm` and `lu.D.x` prefer Random because of this reason. `lbm` runs the Lattice Boltzmann Method to simulate fluids. `lbm` allocates grids that represent three dimensions. `lbm` sweeps the grids multiple times within a short time to simulate fluid collisions. As a result, the pages of `lbm` show similar access recency and frequency. The second subfigure of Figure 3 illustrates the temporal change of access frequencies of `lbm`, showing homogeneous page access frequencies. Second, workloads with random memory access patterns favor Random. LRU nor LFU cannot find hot pages from workloads with random memory access patterns. `deepsjeng` has a random memory access pattern. `deepsjeng` is a chess solver, and it has a hash table for the alpha-beta tree searching to find the next move. Memory accesses to the hash table show a random pattern.

❹ ***Neutral.*** Workloads with mixed memory access patterns show a neutral preference on page migration policies. `bwaves` has mixed memory access patterns. The last subfigure of Figure 3 shows the temporal change of access frequencies of `bwaves`. The execution of `bwaves` is composed of two phases. The first phase accesses the frequently accessed data continuously, favoring LFU (pattern A). The second phase accesses the remaining working set with a strided memory access pattern, favoring LRU (pattern B). As a result, `bwaves` does not have a large performance gap between policies.



(a) cactus (b) graph500 (c) deepsjeng (d) bwaves

Fig. 4: Normalized performance of workloads with various parameter combinations when the modified LRU lists are used

3.3 Huge Pages with the Prior Modified LRU Lists

A prior study proposed to reuse the LRU lists in the native Linux kernel to identify hot pages and migrate them to fast memory [13]. The study periodically scans the LRU lists to update the membership of pages. In the following sections, we call the LRU lists *the modified LRU lists*. The modified LRU lists classify pages in the active list as hot, and pages in the inactive list as cold. The modified LRU lists migrate the pages in the active list to fast memory and pages in the inactive list to slow memory. We use the modified LRU lists as the baseline migration policy to compare against the proposed technique in this paper.

In this subsection, we present that the performance of the modified LRU lists is parameter-sensitive. The modified LRU lists have two parameters: active list scanning ratio and inactive list scanning ratio. The (in)active list scanning ratio determines how many pages are scanned from the list on every scan. While scanning pages, the accessed bits of pages are checked, and the membership of pages is updated. If half of the pages in a list are scanned, the scanning ratio is 50%. By default, the modified LRU lists scan 50% of the active list and inactive list on every five seconds [46]. We evaluate the performance of workloads while varying the scanning ratios of lists from 0% to 100% with a 20% step. Experiments are run with 2MB transparent huge pages.

Figure 4 shows the normalized performance of selected workloads with various combinations of the active list scanning ratio and inactive list scanning ratio. We use the same experiment environment and definition for the performance that we use in Section 3.2. The cells with high performance are drawn with a darker color, and the cells with low performance are drawn with a lighter color. The parameter combination that shows the best performance is different for each workload. Note that the performance gap between the best and worst parameter combinations is 60% in `graph500`, showing the parameter-sensitivity of the modified LRU lists.

The problem originates from the inappropriate application of LRU lists to a tiered memory system. The original goal of LRU lists is to reclaim pages under memory pressure. Therefore, the pages in the inactive list become page reclamation *candidates*. It does not mean that the pages in the inactive list are cold. In the native LRU lists, pages in the inactive list are moved back to the active list again if the pages are accessed. However, in the modified LRU lists, pages in the inactive list are migrated to slow memory, assuming that all pages in the inactive list are cold. Although the cold-classified pages will be migrated back to fast memory when they are accessed, the performance degradation cannot be avoided. Therefore, the latest proposal, the modified LRU lists are not the best option for tiered memory systems. In the following sections, we run the modified LRU lists with the default parameters (50%, 50%) [46].

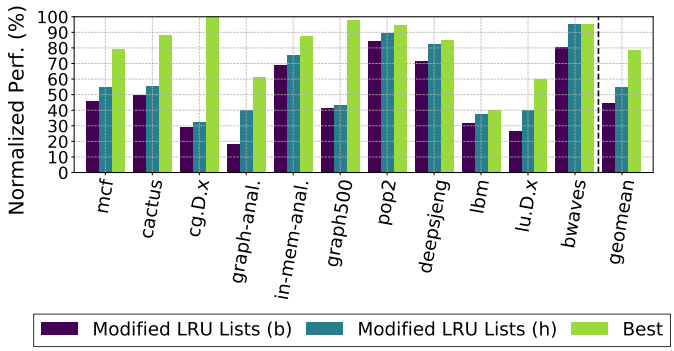


Fig. 5: Performance of the modified LRU lists with 4KB (base) and 2MB (huge) page sizes, compared to the potential best policy.

Figure 5 presents the performance of the modified LRU lists with the base page size (4KB), and huge page size (2MB). The performance is normalized to that with the ideal memory which consists of only fast memory. In addition to the performance of the modified LRU lists, it also shows the potential performance when the best replacement policy is selected among the aforementioned three policies. The results show that the huge page can improve the performance by 23.5% on average with the modified LRU lists. There are two potential advantages of huge pages. First, it reduces TLB misses by the increased translation capability. Second, page migration also uses huge pages, and thus when spatial locality exists, it can prefetch a large chunk of data from the slow to fast memory. However, the modified LRU lists do not provide good identification of page hotness, compared to the potential best policy. The best policy can excel the modified LRU lists by 33.71% and 23.5%, compared to the modified LRU list with the base and huge page sizes, respectively.

3.4 The Homogeneity of Huge Page Hotness

One of the key requirements of migrating pages at the huge page granularity is the homogeneity of hotness in a huge page. To show how much of a 2MB page is actually accessed, we present the results of the homogeneity of page hotness by measuring the number of accessed 4KB base pages within an accessed huge page. We define the ratio as the huge page access ratio. If a workload exhibits a high level of hotness homogeneity, the rest of the base pages within a huge page are likely to be accessed, when a base page in the huge page is accessed. The time interval is determined by the multiplication of the accessed bit check interval (4-second) and the length of the per-page bit vector (8-bit). As the accessed bits of base pages in a huge page cannot be tracked in the current

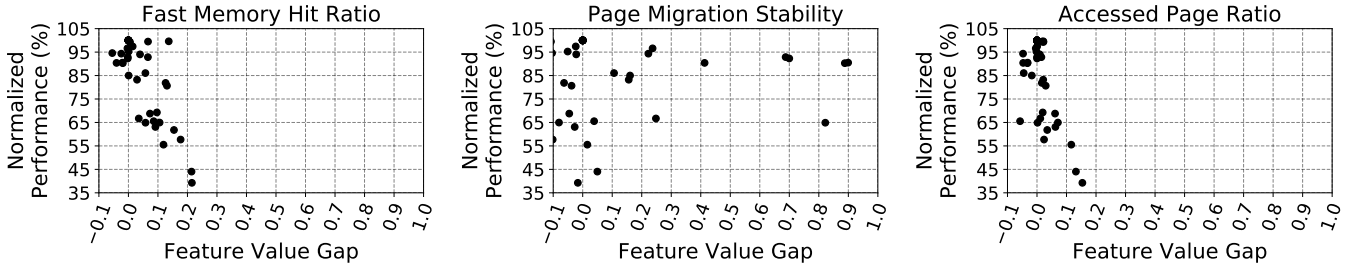


Fig. 6: Feature-performance scatter charts

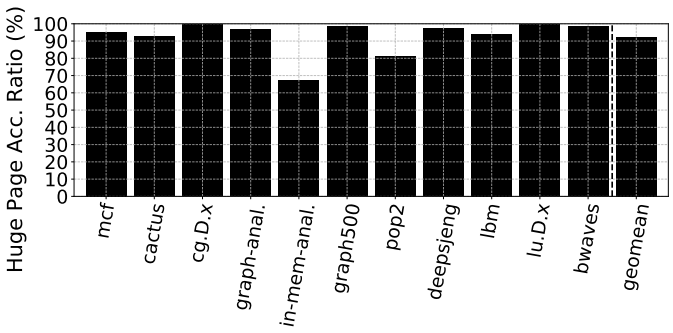


Fig. 7: Huge page access ratio of workloads

system, for this analysis, we use 4KB base pages to track the accessed bits of pages. Based on the access statistics on base pages, we infer how many base pages within a huge are accessed in each time interval. Figure 7 presents the huge page access ratio of workloads. On average, our workloads have huge page access ratios higher than 92%, justifying the huge-page-granular migrations.

4 Adaptive Page Migration Policy Selector

We present AMP, which adaptively selects a page migration policy preferred by a workload. AMP chooses a page migration policy between LRU, LFU, and Random. In this section, we define features and analyze the relationship between features and performance. At last, we present the main algorithm of AMP based on the feature analysis.

4.1 Feature Analysis

Correlation analysis. We define three features that are possibly related to the workloads' preferences on page migration policies: fast memory hit ratio, page migration stability, and accessed page ratio. ❶ ***The fast memory hit ratio*** is the number of accessed pages in fast memory divided by the total number of pages. We assume that a page migration policy that can identify hot pages better shows a higher fast memory hit ratio. ❷ ***The page migration stability*** is the number of stable pages divided by the number of total pages. The locations of pages are updated periodically in the baseline page migration policies. A page is regarded as *stable* if the page location has not been changed compared to the previous location. The page migration stability presents the page migration cost of a page migration policy. ❸ ***The accessed page ratio*** is the number of accessed pages divided by the total number of pages. The assumption behind this feature is that workloads touch pages as they progress. If a page migration policy has been successful in choosing hot pages, a workload can progress faster. Therefore, the page migration policy may present a higher accessed page ratio. These features are measured on every page migration, whose interval is five seconds.

	Correlation Coeff.	P-value
Fast Memory Hit Ratio	-0.8194	1.0505×10^{-11}
Migration Stability	0.0322	0.8355
Accessed Page Ratio	-0.4975	0.0006

TABLE 1: Pearson correlation coefficients between feature value gap and normalized performance

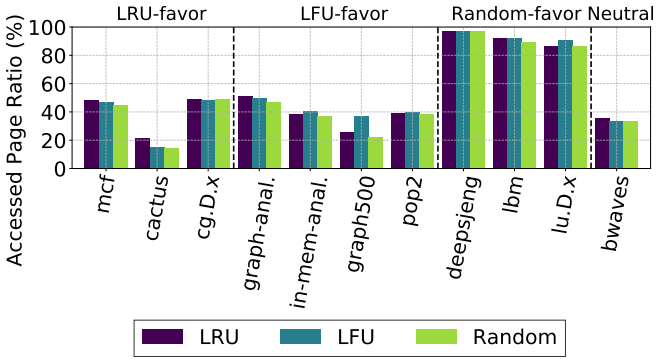


Fig. 8: Average accessed page ratio of workloads

We analyze the relationship between features and the normalized performance of workloads using the Pearson correlation coefficient. We define *a feature value gap*, which is the gap between the feature value of the page migration policy that performs the best and the feature value of a selected page migration policy. If the feature plays an important role in the performance, the lower the gap is, the closer the performance of the selected policy is to the performance of the best-performing policy. The performance is the reverse of the execution time, and it is normalized to the best-performing page migration policy. We calculate the Pearson correlation coefficient between the feature value gap and the normalized performance. The analysis is conducted on the data that we present in Section 3.1.

Figure 6 shows the scatter charts between the feature value gap and normalized performance. Table 1 presents the Pearson correlation coefficients. Additionally, it shows the p-values of the correlation coefficients. The absolute value of a correlation coefficient presents the strength of the correlation between the feature and performance. P-values show statistical significance. A feature is considered to have a statistically significant correlation if its p-value is lower than 0.01. Among the features that we have defined, the fast memory hit ratio shows the strongest correlation (-0.8194) and the smallest p-value (1.0505×10^{-11}). The accessed page ratio has a p-value lower than 0.01, showing that the feature has a statistically significant correlation. However, the absolute value of the correlation coefficient is smaller than the fast memory hit ratio's. On the other hand, the page migration stability does not have a statistically meaningful relationship with the performance (p-value=0.8355).

A feature for random-favor workloads. We find that having a high accessed page ratio is a hint for workloads to favor the

random migration policy. As we have presented in Section 3.2, random-favor workloads access memory with a low locality, and they have a huge memory footprint that exceeds the fast memory size. As a result, random-favor workloads have a higher average accessed page ratio compared to the other workloads. Figure 8 presents the average accessed page ratio of workloads. Random-favor workloads have average accessed page ratios higher than 80%.

4.2 Adaptive Page Migration Policy Selection

AMP adaptively selects a page migration policy between LRU, LFU, and Random using the features that we have analyzed in the previous subsection. Algorithm 1 describes the page migration policy decision of AMP. AMP classifies a workload as random-favor if the accessed page ratio of the workload exceeds the fast memory ratio significantly. The fast memory ratio is defined as the number of pages in fast memory divided by the number of total pages. The insight behind this heuristic is that a workload with a huge working set that exceeds the cache size may experience a thrashing. If the accessed page ratio exceeds the fast memory ratio by 20%, AMP chooses the random migration policy. The threshold is empirically set.

Otherwise, AMP selects a page migration policy between LRU and LFU. According to the feature analysis, the fast memory hit ratio and accessed page ratio have a strong relationship with the performance. Between the two features, we choose the fast memory hit ratio because its correlation is stronger than the other. AMP tracks the fast memory hit ratios of LRU and LFU simultaneously and chooses a policy that has a higher average fast memory hit ratio. The key challenge in tracking the fast memory hit ratios is that only one page migration policy can be applied to physical tiered memory. We overcome this problem by emulating a page migration policy. Figure 9 illustrates how AMP obtains the fast memory hit ratios of both policies simultaneously. AMP applies the page migration policy with the higher average fast memory hit ratio to physical tiered memory. At the same time, pages are migrated *virtually* with the other policy. For example, if LRU has a higher average fast memory hit ratio, LRU is applied to physical pages. At the same time, LFU is emulated virtually in the kernel without page migrations. As a result, the kernel can collect the fast memory hit ratios of both policies. If LFU turns out to have a higher average fast memory hit ratio, AMP applies LFU to the physical tiered memory in the next turn. AMP uses the moving average of fast memory hit ratios, and the window size is 36-epoch. We empirically find that the moving average can select a page migration policy accurately and stably.

Cost of Switching Policies. Switching the policy from one to another has a negligible cost, as it affects only which pages need to be in the fast memory. To choose the right policy, the proposed component tracks the hotness, fast memory hit ratio, and other information. However, this tracking computation is done in the background during each interval. An indirect cost is that some pages in the fast memory which were promoted by the old policy, may no longer be hot ones with the new policy. Therefore, gradually the pages are evicted, and new pages are migrated by the new policy.

Although the policy switching itself does not have any significant direct overheads, the extra information must be tracked and maintained to choose the right policy. It has some memory capacity overheads and computational costs. The spatial cost includes the per-page metadata to track hotness (4B age, 8B access history, 4B access frequency), per-page virtual page location (4B) to simulate page migrations, and per-policy features (fast memory hit ratio and accessed page ratio, 4B each). The majority of computational costs to simulate the other policy in the background without actual page migrations. This cost occurs when it sorts pages to find the relative hotness of pages. Note that the computation occurs in the background during each time interval, not during the policy switching.

Algorithm 1 Adaptive Page Migration Policy Selection

```
if accessed_page_ratio > (fast_memory_ratio + 20%) then
    repl_policy = Random
else
    if LRU_hit_ratio_avg > LFU_hit_ratio_avg then
        repl_policy = LRU
    else
        repl_policy = LFU
```

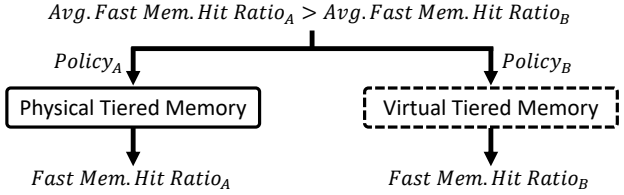


Fig. 9: Adaptive page migration policy selection for non-random-favor workloads

5 Implementation

We implement AMP in a Linux system. The implementation of AMP spans from the kernel to the user-level. The kernel tracks the age and access frequency of pages and provides several options for page migration policies. Additionally, features such as the fast memory hit ratio and accessed page ratio are collected in the kernel. AMP is built on the memory control groups in the kernel (`memcgs`). We assume that the processes in a `memcg` have the same preferences on page migration policies. Therefore, `memcg` is the basic unit of page migration policy decisions. The fast memory ratio can be set for each `memcg`.

The user-level controller periodically requests the scan of pages. The request is sent to the kernel by writing to a file under sysfs. On the request, the kernel scans all pages in the `memcg` and checks the accessed bits of the pages. Page age and access frequency are updated using the accessed bits. The accessed bits of pages are checked using the `page_is_idle` function [47]. After that, the accessed bit of the page is unset using the `set_page_idle` function to check further accesses to the page.

The user-level controller migrates pages between fast memory and slow memory by requesting to the kernel. The user-level controller sets the page migration policy, and the kernel applies the page migration policy. AMP chooses the random replacement policy if the accessed page ratio exceeds a predefined threshold. Otherwise, the policy with a higher average fast memory hit ratio is selected between LRU and LFU. The kernel reports the fast memory hit ratio of both policies to the user-level controller. The user-level controller collects the fast memory hit ratios and calculates the moving averages of fast memory hit ratios.

Page migration requires exchanges of pages between two nodes. In the native Linux, page exchanges involve redundant page (de)allocations, which causes the performance overhead. The

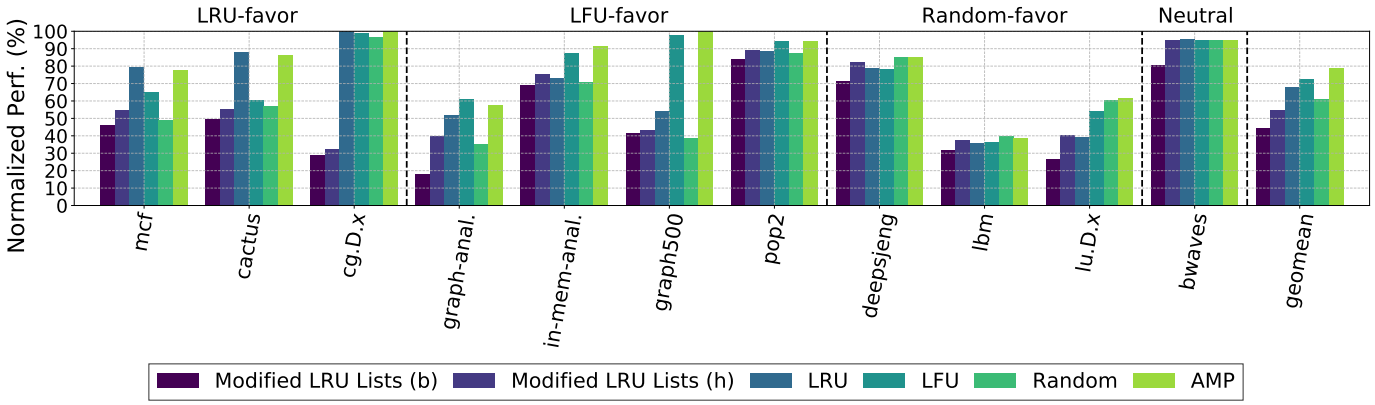
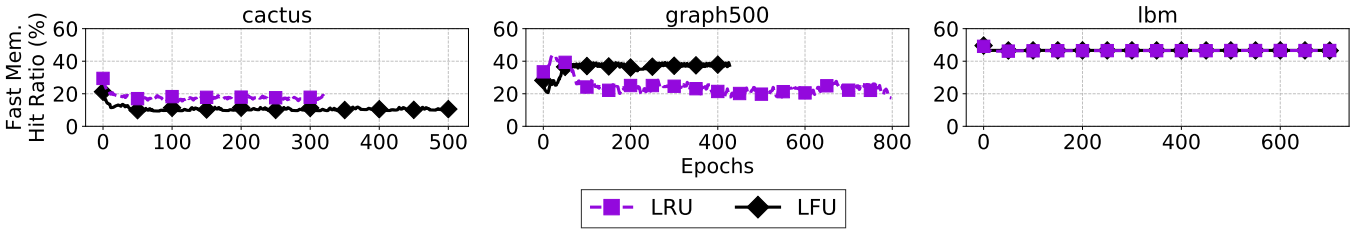
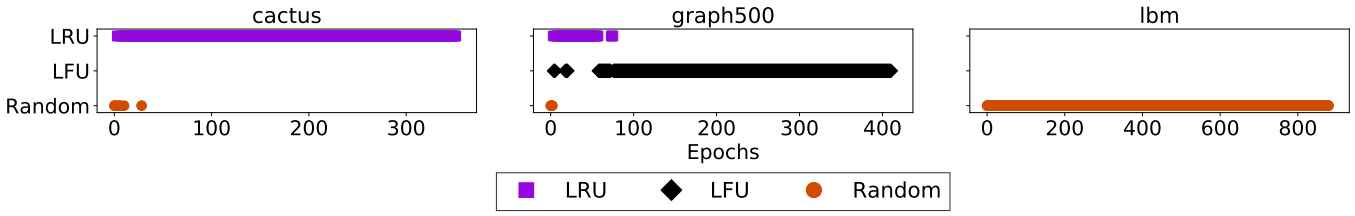


Fig. 10: Normalized performance of workloads with various page migration policies



(a) Moving average of fast memory hit ratios



(b) Page migration policy selection timeline

Fig. 11: Temporal change of fast memory hit ratios and page migration policy selection timeline

recently proposed optimization can eliminate the overhead [13]. The proposed optimization exchanges two pages by changing the mappings and exchanging the contents of pages without (de)allocating pages. We apply the kernel patch [46] to reduce the performance overhead of page migrations.

6 Evaluation

6.1 Experiment Setup

Intel Xeon Dual Socket System	
OS & Kernel	Ubuntu 18.04.2 - kernel 4.15.0
Processors	2-socket E5-2630 v4
Memory	DDR4 - 2133MHz
Fast Memory Latency	78ns
Fast Memory BW	32 GB/s
Slow Memory Latency (Emulated)	359ns
Slow Memory BW (Emulated)	5.8 GB/s

TABLE 2: System configurations

We evaluate AMP on a Linux system. The system runs as a two-socket QEMU virtual machine to emulate a tiered memory system. The system is composed of fast and slow memory nodes. The fast memory node has CPU cores, and its memory is allocated from the normal DRAM. The slow memory node does not have CPU cores, and its memory is allocated from a bandwidth-throttled DRAM. We throttle the memory bandwidth using power throttling [48], and we saturate the memory bandwidth using `membw` [49] to meet the reported latency (346ns) of Optane DC [2]. Table 2 describes the evaluation system configurations.

Workload Name	2MB Ratio	Workload Name	2MB Ratio
mcf	94%	pop2	77%
cactus	92%	deepsjeng	97%
cg.D.x	100%	lbm	94%
graph-analytics	97%	lu.D.x	99%
in-mem-analytics	96%	bwaves	98%
graph500	98%	**Geomean**	**95%**

TABLE 3: Huge page allocation ratio in anonymous pages

6.2 Performance of AMP

Figure 10 shows the normalized performance of workloads with various page migration policies. We compare the performance of workloads with the modified LRU lists, LRU, LFU, Random, and AMP. For the modified LRU lists, we run the experiments with 4KB base pages and 2MB transparent huge pages, respectively. The suffix in the legend shows the page size. `b` stands for base pages, and `h` means huge pages. For the other configurations, 2MB transparent huge pages are used. Table 3 shows the portion of huge page allocation ratio of each workload, presenting 95% on average. The performance is the reverse of the execution time, and the performance is normalized to the 100% fast memory ratio. In this experiment, we set the fast memory ratio to 50%. On average, AMP can achieve 10.9, 6.4, 17.6% higher performance compared to LRU, LFU, and Random, respectively.

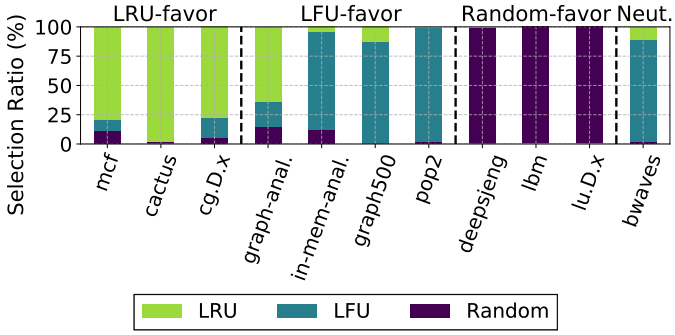


Fig. 12: Page migration policy selection ratio

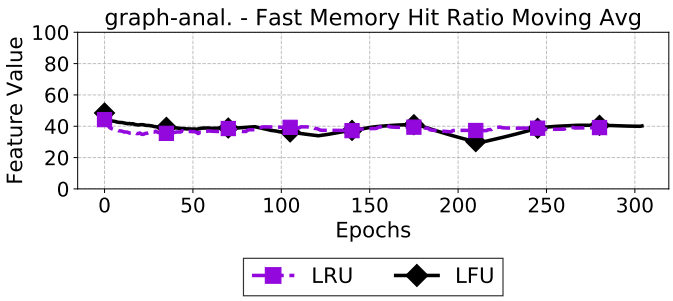


Fig. 13: Moving averages of fast memory hit ratios (`graph-analytics`)

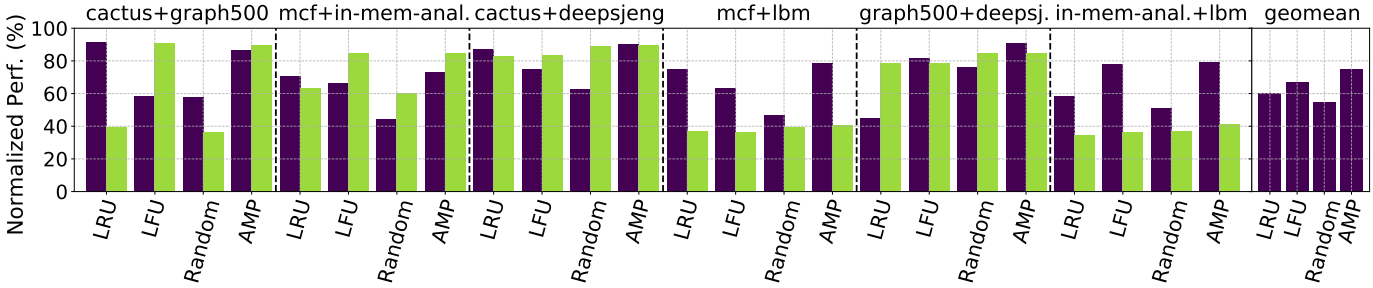


Fig. 14: Normalized performance of workload mixes with various page migration policies (consolidated)

Figure 11a shows the temporal change of average fast memory hit ratios, and Figure 11b presents the timeline of page migration policy selections. `cactus`, `graph500`, and `lbm` favor LRU, LFU, and Random, respectively. Overall, the preferred page migration policy has a higher average fast memory hit ratio during the execution time except for `lbm`. For `cactus`, LRU shows the higher average fast memory hit ratio. Therefore, `cactus` selects LRU except for the warming-up stage. For `graph500`, LRU has a higher average fast memory hit ratio in the initial stage. Therefore, `graph500` chooses LRU at first. As time goes by, the average fast memory hit ratio of LFU beats the LRU's. After the point, `graph500` chooses LFU. On the other hand, `lbm` does not show any difference between the fast memory hit ratios of LRU and LFU because it favors Random. Figure 11 shows that AMP can choose a page migration policy using the average fast memory hit ratios.

6.3 Page Migration Policy Selection Ratio

We measure the AMP's page migration policy selection ratio for each workload. Figure 12 presents the page migration policy selection ratio for each workload. Overall, AMP can choose a preferred page migration policy. For most workloads, the preferred page migration policy shows a selection ratio higher than 80%. However, `graph-analytics` presents the high selection ratio of LRU, although it prefers LFU when a policy is chosen statically. It is because the average fast memory hit ratios of LRU and LFU do not show a meaningful gap during most of the execution time. Figure 13 shows the temporal change of the average fast memory hit ratios of `graph-analytics`, and the figure presents the negligible gap between LRU and LFU. Therefore, choosing LRU is okay for `graph-analytics`. Another reason for a workload to choose a non-favored page migration policy is the change in page migration policy preference during the execution, as we have shown in `graph500` in the previous subsection.

LRU-favor	mcf, cactus, cg.D.x
LFU-favor	graph-anal., in-mem-anal., graph500, pop2
Random-favor	deepsjeng, lbm, lu.D.x
Neutral	bwaves

TABLE 4: Page migration policy preferences summary

6.4 Performance of AMP in Consolidated Environments

One of the advantages of AMP is that it can apply a different page migration policy for each `memcg`, simultaneously. Even though a set of workloads have different favors on page migration policies, AMP can offer the preferred page migration policy for each workload. In this subsection, we evaluate the performance of AMP when multiple workloads are running simultaneously in different `memcgs`. We run six workload mixes that are combinations of workloads with different page migration policy preferences. The fast memory ratio is set to 50% of the working sets.

Figure 14 shows the normalized performance of workload mixes. In the figure, there are seven groups of bar graphs. The first six groups represent workload mixes, and the corresponding workload mix is shown above the figure. The workloads' preferences on page migration policies are summarized in the Table 4. The first six groups are composed of grouped bar graphs. The first bar in a group shows the performance of the first workload within a mix, and the second bar presents the performance of the second workload in a mix. We use the same definition for the performance that we use in Section 6.2. The last group of bars shows the geomean performance of page migration policies. Each workload shows the best performance with the preferred page migration policy. Static policies such as LRU, LFU, and Random cannot offer the best performance for both workloads in a mix at the same time. AMP can offer the preferred policy for both workloads in a mix, achieving 14.7, 8.2, 20.4% higher geomean performance than LRU, LFU, and Random, respectively.

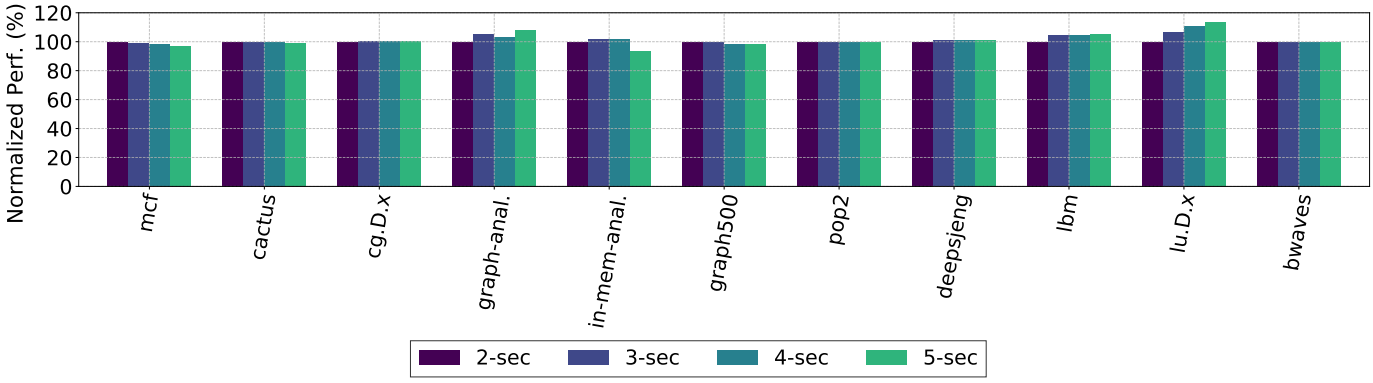


Fig. 15: Normalized performance of workloads with various page migration intervals

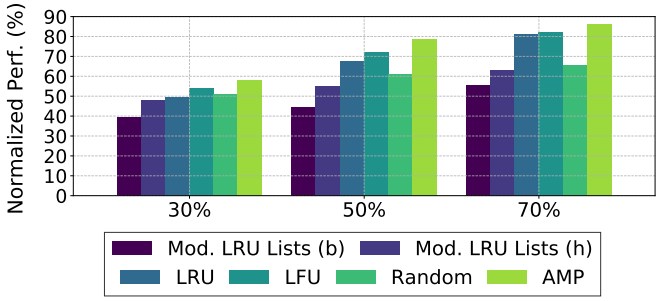


Fig. 16: Sensitivity to the fast memory ratio

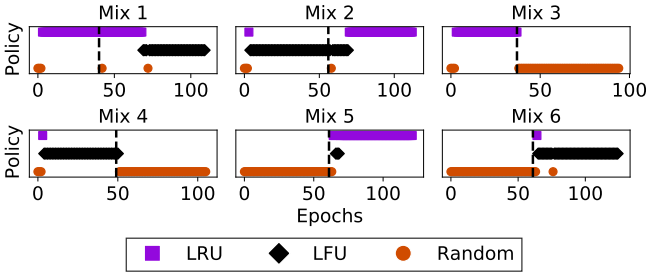


Fig. 17: Timeline of page migration policy selections

Mix 1	27-epoch	**Mix 2**	17-epoch	**Mix 3**	2-epoch
Mix 4	2-epoch	**Mix 5**	3-epoch	**Mix 6**	4-epoch

TABLE 6: The number of epochs until AMP learns the changes in the preference on page migration policy

6.5 Sensitivity to the Fast Memory Ratio

We evaluate the sensitivity of AMP to the fast memory ratio. In the previous experiments, we set the fast memory ratio to 50% of workloads' working set. In this section, we show that AMP performs better than the other page migration policies with various fast memory ratios. We measure the performance of workloads with 30, 50, and 70% fast memory ratios. We use the same workloads, page migration policies, and the same definition for the performance that we use in Section 6.2. Figure 16 presents the normalized performance of workloads with various page migration policies. We show the geomean performance of workloads. Overall, AMP performs better than the other page migration policies regardless of the fast memory ratio. AMP shows 4.0, 6.4, 4.1% higher performance than LFU at 30, 50, 70% fast memory ratio, respectively.

6.6 Responsiveness to the Phase Transitions

AMP works well for most macro-benchmarks with a gradual transition between phases. A phase is an interval of execution where the preference on page migration policies is the same. However, if a phase changes abruptly, AMP may experience a delay until updating the page migration policy selection because it uses the moving average of fast memory hit ratios. In this subsection, we measure the delay until AMP learns the changes in the page migration policy preference on abrupt phase transitions.

We use three types of synthetic benchmarks. ❶ ***LRU-favor*** has a strided memory access pattern with four same-sized working sets. Each working set is sequentially accessed for four seconds. ❷ ***LFU-favor*** has two same-sized working sets. One is a frequently-accessed hot working set, and the other is an infrequently-accessed cold working set. The infrequently-accessed cold working set is divided into 16 subsets, showing a strided memory access pattern with a low access frequency. ❸ ***Random-favor*** allocates memory, and it randomly accesses all pages. We compose six types of workload mixes with synthetic benchmarks. Table 5 presents the execution order of workload mixes.

Mix 1	LRU-favor → LFU-favor
Mix 2	LFU-favor → LRU-favor
Mix 3	LRU-favor → Random-favor
Mix 4	LFU-favor → Random-favor
Mix 5	Random-favor → LRU-favor
Mix 6	Random-favor → LFU-favor

TABLE 5: Synthetic benchmark mixes

Figure 17 illustrates the timeline of page migration policy selections for each workload mix. Each workload starts with Random in the warming-up stage, and it selects the preferred page migration policy after the warming up. The dashed vertical line shows the transition point, where the first workload exits and the second workload starts execution. Except for Mix 1, AMP can follow the preferred page migration policy within a relatively short delay. Table 6 summarizes the delay until AMP finds the favored page migration policy after a phase transition. AMP shows the low responsiveness in Mix 1 because of the stale fast memory hit ratios kept in moving average. LRU shows the high fast memory hit ratio in the first workload of Mix 1, and it takes time for AMP to learn the changes in policy preferences. Please note that this kind of abrupt phase transitions are not common in the real world.

6.7 Sensitivity to the Page Migration Interval

Page migration interval is one of the important parameters in migrating pages in tiered memory systems. There is a trade-off between responsive migration of hot pages to fast memory and page migration cost. In the previous subsections, the page migration interval is set to five seconds empirically. In this subsection,

we evaluate the sensitivity of AMP's performance to the page migration interval with shorter intervals than five seconds. Figure 15 presents the performance of workloads with AMP with various page migration intervals. The performance is normalized to the performance with the shortest page migration interval, 2-second. Most workloads show similar performance regardless of the page migration interval. `lu.D.x` presents 13.8% higher performance with 5-second page migration interval compared to 2-second interval's. This is because the memory access pattern of `lu.D.x` is random, and most pages are actively accessed. Therefore, frequent page migrations add mere performance overhead without increasing the fast memory hit ratio. `In-memory-analytics` presents the higher performance with the intervals shorter than 5-second, implying that the shorter page migration interval can capture the dynamic nature of hot working set of `in-memory-analytics`'s. To summarize, there are some workloads that are sensitive to page migration interval. However, there is a little gain in shortening page migration interval, on average.

7 Discussion

Low-overhead Hotness Tracking Mechanisms. Having a low-cost hotness tracking mechanism is important in identifying the hotness of pages. Checking the accessed bits of pages at a low frequency is one of the solutions to achieve the low overhead [19]. Although this is a viable solution to identify swap page candidates, it cannot be applied to migrating pages in tiered memory because of its low resolution. Alternatively, dynamically adjusting the unit of hotness tracking is a solution to reduce the performance overhead [50]. By identifying groups of pages that have similar page hotness, the number of accessed bit checks can be reduced. AMP achieves the low overhead in hotness tracking by checking the accessed bits at the granularity of huge pages.

Comparison with HW-based Migration Mechanisms. The target memory system of this study is a tiered memory system where software is responsible for managing page locations between memory tiers. Hardware data migration mechanisms [51], [52], [53] have an advantage in offering software-transparent fine-grained data migration. However, the hardware mechanisms require modifications to memory controllers, which need support from hardware vendors. The proposed software mechanism and policies can be applied without any support from hardware vendors, which goes well with the current data centers.

The hardware techniques [51], [52], [53] are intended for a hybrid memory system with a relatively small 3D stacked DRAM (fast memory) backed by the conventional DRAM (slow memory). Therefore, the fast memory capacity is smaller than what is used in general tiered memory where DRAM and NVM are combined. Thanks to the small fast memory capacity of 3D stacked DRAM, the HW approach maintains an extra layer of mapping between the fast and slow memory spaces, instead of using page tables. With the HW maintained mapping table, it is possible to migrate data at smaller granularity than pages in a nimble way. Those HW approaches access the mapping table slightly differently. Some approaches cache the HW-maintained mapping table in on-chip SRAM for fast access [51], while the other approaches look up the fast memory for mapping information whenever an LLC miss occurs [53]. Both of the mechanisms were possible since they are designed for the 3D stacked DRAM, which was assumed to be faster than DRAM for fast lookups of the mapping table, and to have a small capacity so that the mapping table can be efficiently cached in SRAM. However, the fine-grained HW mapping tables may not be scalable enough to cover the combined capacity of DRAM and NVM, and accessing DRAM first for mapping information for every LLC miss will slow down memory access times significantly. In addition, the HW mapping table has a limited associativity to reduce the size as much as possible. Therefore, the memory management is severely restricted, and there are only a few locations data can be stored either fast or slow memory.

Cost of TLB Shootdowns. When a page table entry is updated for a process, the Linux kernel sends Inter-Process Interrupts (IPIs) to the cores running threads of the same process for TLB shootdowns. Once an IPI arrives, the receiving core invokes the kernel to execute TLB invalidation. IPIs are sent to the cores running threads with the same address space, regardless of whether pages are actually shared or not [54], [55]. Note that the OS kernel does not track which pages are shared by what threads within a process. Therefore, even with a 4KB base page, an update of a PTE will send IPIs to the cores running the other threads of the same address space, even if the other threads do not access the page and the cores do not have the affected PTE in their TLBs. The majority of the shootdown cost is for initiating and responding to IPIs, regardless of TLB hits or misses during the TLB invalidation. Therefore, huge pages do not increase the occurrences of TLB shootdowns by data sharing, compared to base pages.

Instead, using huge pages can potentially reduce TLB shootdown occurrences. If the entire region of a huge page is accessed, a single shootdown can migrate a 2MB page. With 4KB base pages, 512 shootdowns can occur in the worst case for a migration of the same 2MB region. Note that shootdown IPIs will be sent to the cores running all threads in the same process, even if a PTE of any base or huge page is updated. In addition, using huge pages significantly reduces TLB misses for many workloads.

Partitioning Fast Memory. Although this study assumes a case where each workload uses fast memory exclusively, sharing fast memory can be preferred in a batch execution environment. Sharing fast memory between multiple workloads is similar to partitioning CPU caches between multiple processes. This problem has been studied for several decades in the computer architecture community. The insights from cache partitioning studies can be applied to sharing fast memory in tiered memory systems. Cache partitioning has been studied to allocate caches between multiple processes to minimize the miss rate and maximize throughput [56], [57], to guarantee the fairness between applications [58], [59], [56], [60], [61], [62], [63], and to protect the latency-sensitive jobs from batch jobs [64], [65], [66], [67]. Constructing miss rate curves or utility curves can give users a hint to allocate fast memory between multiple workloads [57], [68], [69], [70], [71], [72]. If users already know the utility curves of workloads, an auction can be used to allocate fast memory between workloads [73], [74].

8 Conclusion

Memory systems are adopting memories with different latency and bandwidth, comprising a tiered memory system. Page migration policies migrate pages to utilize fast memory with hot pages. We find that workloads have diverse preferences on page migration policies. We analyze the reason behind the various preferences on policies, and we find the relationship between features and performance of workloads. Based on the analysis, we propose AMP, which adaptively selects a page migration policy between

LRU, LFU, and Random. AMP can estimate the fast memory hit ratio of a page migration policy by emulating the policy without page migrations. AMP can achieve 10.9, 6.4, 17.6% higher performance than LRU, LFU, and Random, respectively. The source code is available at https://github.com/casys-kaist/AMP.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful feedbacks and comments. This work was supported by National Research Foundation of Korea (NRF-2019R1A2B5B01069816) and the Institute for Information & communications Technology Promotion (IITP-2017-0-00466).

REFERENCES

[1] "Intel Optane DC Persistent Memory." https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. [Online; accessed 13-January-2020].

[2] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, *et al.*, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," *arXiv preprint arXiv:1903.05714*, 2019.

[3] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[4] D. Shasha and T. Johnson, "2Q: A Low Overhead High Performance Buffer Management Replacement Algoritm," in *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, pp. 439–450, 1994.

[5] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," *IEEE Transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[6] I. Ari, A. Amer, R. B. Gramacy, E. L. Miller, S. A. Brandt, and D. D. Long, "ACME: Adaptive Caching Using Multiple Experts," in *Proceedings of the Workshop on Distributed Data and Structures (WDAS)*, pp. 143–158, 2002.

[7] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance," in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 31–42, 2002.

[8] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, pp. 115–130, 2003.

[9] S. Bansal and D. S. Modha, "CAR: Clock with Adaptive Replacement," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*, pp. 187–200, 2004.

[10] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement.," in *USENIX Annual Technical Conference (ATC)*, pp. 323–336, 2005.

[11] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, pp. 381–391, 2007.

[12] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent Page Management for Two-tiered Main Memory," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 631–644, 2017.

[13] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble Page Management for Tiered Memory Systems," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 331–345, 2019.

[14] "Intel Memory Drive Technology." https://www.intel.com/content/www/us/en/software/intel-memory-drive-technology.html. [Online; accessed 13-January-2020].

[15] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient Memory Disaggregation with Infiniswap," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 649–667, 2017.

[16] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, *et al.*, "Remote Regions: A Simple Abstraction for Remote Memory," in *USENIX Annual Technical Conference (ATC)*, pp. 775–787, 2018.

[17] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont, "Welcome to Zombieland: Practical and Energy-efficient Memory Disaggregation in a Datacenter," in *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, pp. 1–12, 2018.

[18] K. Koh, K. Kim, S. Jeon, and J. Huh, "Disaggregated Cloud Memory with Elastic Block Management," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 39–52, 2018.

[19] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, *et al.*, "Software-Defined Far Memory in Warehouse-Scale Computers," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 317–330, 2019.

[20] K. Keeton, "The Machine: An Architecture for Memory-centric Computing," in *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2015.

[21] "Transparent hugepages." https://lwn.net/Articles/359158/. [Online; accessed 13-January-2020].

[22] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, Transparent Operating System Support for Superpages," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 89–104, 2002.

[23] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and Efficient Huge Page Management with Ingens," in *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 705–721, 2016.

[24] A. Panwar, A. Prasad, and K. Gopinath, "Making Huge Pages Actually Useful," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 679–692, 2018.

[25] A. Panwar, S. Bansal, and K. Gopinath, "HawkEye: Efficient Fine-grained OS Support for Huge Pages," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 347–360, 2019.

[26] F. J. Corbato, "A Paging Experiment with the Multics System," tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[27] M. Kampe, P. Stenstrom, and M. Dubois, "Self-correcting LRU Replacement Policies," in *Proceedings of the 1st Conference on Computing Frontiers*, pp. 181–191, 2004.

[28] E. Teran, Y. Tian, Z. Wang, and D. A. Jiménez, "Minimal Disturbance Placement and Promotion," in *Proceedings of the 22nd International Symposium on High Performance Computer Architecture (HPCA)*, pp. 201–211, 2016.

[29] V. Gupta, M. Lee, and K. Schwan, "Heterovisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms," in *Proceedings of the 11th International Conference on Virtual Execution Environments (VEE)*, pp. 79–92, 2015.

[30] G. Glass and P. Cao, "Adaptive Page Replacement based on Memory Reference Behavior," in *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 115–126, 1997.

[31] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," in *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 122–133, 1999.

[32] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "A Low-overhead High-performance Unified Buffer Management Scheme that Exploits Sequential and Looping References," in *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[33] J. T. Robinson and M. V. Devarakonda, "Data Cache Management using Frequency-based Replacement," in *Proceedings of the 1990 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 134–142, 1990.

[34] E. J. O'neil, P. E. O'neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pp. 297–306, 1993.

[35] Y. Zhou, J. Philbin, and K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," in *USENIX Annual Technical Conference (ATC)*, pp. 91–104, 2001.

[36] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C.-S. Kim, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies," in *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 134–143, 1999.

[37] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan, "Driving Cache Replacement with ML-based LeCar," in *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[38] "SPEC CPU 2017." https://www.spec.org/cpu2017/. [Online; accessed 13-January-2020].

[39] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 37–48, 2012.

[40] "NAS Parallel Benchmarks." https://www.nas.nasa.gov/publications/npb.html. [Online; accessed 13-January-2020].

[41] "graph500." https://graph500.org/. [Online; accessed 13-January-2020].

[42] C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn, "Profile-guided Post-link Stride Prefetching," in *Proceedings of the 16th International Conference on Supercomputing (ICS)*, pp. 167–178, 2002.

[43] H. Al-Sukhni, J. Holt, D. A. Connors, M. Snyder, M. Smittle, and B. Grayson, "The Design of Cost-effective Stride-prefetching for Modern Processors,"

[44] R. Panda and L. K. John, "HALO: A Hierarchical Memory Access Locality Modeling Technique For Memory System Explorations," in *Proceedings of the 32nd International Conference on Supercomputing (ICS)*, pp. 118–128, 2018.

[45] C. Takahashi, M. Sato, D. Takahashi, T. Boku, H. Nakamura, M. Kondo, and M. Fujita, "Empirical Study for Optimization of Power-performance with On-chip Memory," in *High-Performance Computing*, pp. 466–479, 2005.

[46] "Nimble Page Management for Tiered Memory Systems - GitHub repository." https://github.com/ysarch-lab/nimble_page_management_asplos_2019/blob/5d503c456f1eceed24e4723ef758ce2c38db1ae0/mm/memory_manage.c#L777. [Online; accessed 13-January-2020].

[47] "Idle Page Tracking." https://www.kernel.org/doc/html/latest/admin-guide/mm/idle_page_tracking.html. [Online; accessed 13-January-2020].

[48] "Intel Xeon Processor E5 v4 Product Family." https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v4-datasheet-vol-2.pdf. [Online; accessed 13-January-2020].

[49] "intel-cmt-cat." https://github.com/intel/intel-cmt-cat. [Online; accessed 13-January-2020].

[50] S. Park, Y. Lee, and H. Y. Yeom, "Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality," in *Proceedings of the 20th International Middleware Conference Industrial Track*, pp. 1–7, 2019.

[51] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM as Part of Memory," in *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*, pp. 13–24, 2014.

[52] J. B. Kotra, H. Zhang, A. R. Alameldeen, C. Wilkerson, and M. T. Kandemir, "Chameleon: A Dynamically Reconfigurable Heterogeneous Memory System," in *Proceedings of the 51st International Symposium on Microarchitecture (MICRO)*, pp. 533–545, 2018.

[53] C. C. Chou, A. Jaleel, and M. K. Qureshi, "Cameo: A Two-level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-managed Cache," in *Proceedings of the 47th International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2014.

[54] N. Amit, A. Tai, and M. Wei, "Don't shoot down TLB shootdowns!," in *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, pp. 1–14, 2020.

[55] M. K. Kumar, S. Maass, S. Kashyap, J. Veselỳ, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna, "LATR: Lazy Translation Coherence," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 651–664, 2018.

[56] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource," in *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 13–22, 2006.

[57] M. K. Qureshi and Y. N. Patt, "Utility-based Cache Partitioning: A Low-overhead, High-performance, Runtime Mechanism to Partition Shared Caches," in *Proceedings of the 39th International Symposium on Microarchitecture (MICRO)*, pp. 423–432, 2006.

[58] S. Kim, D. Chandra, and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," in *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 111–122, 2004.

[59] T. Y. Yeh and G. Reinman, "Fast and Fair: Data-stream Quality of Service," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pp. 237–248, 2005.

[60] J. Chang and G. S. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors," in *Proceedings of the 25th International Conference on Supercomputing (ICS)*, pp. 402–412, 2007.

[61] X. Wang and J. F. Martínez, "ReBudget: Trading Off Efficiency vs. Fairness in Market-based Multicore Resource Allocation via Runtime Budget Reassignment," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 19–32, 2016.

[62] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology," in *Proceedings of the 26th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 194–205, 2017.

[63] J. Park, S. Park, and W. Baek, "CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-aware Workload Consolidation on Commodity Servers," in *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, pp. 1–14, 2019.

[64] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency while Preserving Responsiveness," in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pp. 308–319, 2013.

[65] H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-critical Workloads," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 729–742, 2014.

[66] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pp. 450–462, 2015.

[67] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for Latency-critical Tasks on Shared Multicore Systems," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 33–47, 2016.

[68] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 121–132, 2009.

[69] X. Zhang, S. Dwarkadas, and K. Shen, "Towards Practical Page Coloring-based Multicore Cache Management," in *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*, pp. 89–102, 2009.

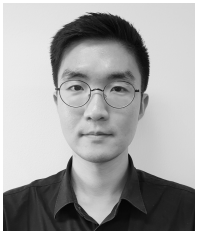
[70] N. Beckmann and D. Sanchez, "Jigsaw: Scalable Software-defined Caches," in *Proceedings of the 22nd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 213–224, 2013.

[71] X. Hu, X. Wang, Y. Li, Y. Luo, C. Ding, and Z. Wang, "Optimal Symbiosis and Fair Scheduling in Shared Cache," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1134–1148, 2016.

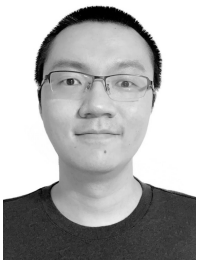
[72] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "DCAPS: Dynamic Cache Allocation with Partial Sharing," in *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, pp. 1–15, 2018.

[73] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem, "Ginseng: Market-driven Memory Allocation," in *Proceedings of the 10th International Conference on Virtual Execution Environments (VEE)*, pp. 41–52, 2014.

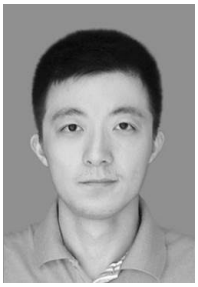
[74] L. Funaro, O. A. Ben-Yehuda, and A. Schuster, "Ginseng: Market-Driven LLC Allocation," in *USENIX Annual Technical Conference (ATC)*, pp. 295–308, 2016.



Taekyung Heo received the BS degree in computer engineering from Sungkyunkwan University, and the MS degree in computer science from KAIST. He is a PhD candidate of school of computing at KAIST. His research interests include memory systems, computer architecture, and accelerators.

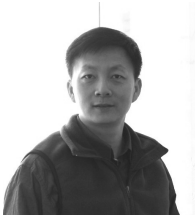


Yang Wang received the BS degree from University of Electronic Science and Technology of China. He is an internship at Microsoft Research Asia. His research interest includes computer architecture and general-purpose graphics processing unit architecture. He is a student member of IEEE.



Wei Cui received the BS degree in Nanjing University of Science and Technology, and the MS degree in Peking University, Beijing. He is an senior research SDE in Microsoft Research, Asia (Beijing). His research interests include computing accelerators, AI platform and system optimization.

Jaehyuk Huh received the BS degree in computer science from Seoul National University, and the MS and the PhD degrees in computer science from the University of Texas, Austin. He is a professor of school of computing at KAIST. His research interests include computer architecture, parallel computing, virtualization and system security. He is a member of IEEE.



Lintao Zhang received his BS degree in Physics from Peking University, and his PhD in Computer Engineering from Princeton University. He is currently a research manager in Microsoft Research Asia. His research interests are system issues in very large-scale, distributed systems. He is a Senior Member of IEEE.