ArtMem: Adaptive Migration in Reinforcement Learning-Enabled Tiered Memory

Xinyue Yi*
Xiamen University
Xiamen, China
nvmyi@stu.xmu.edu.cn

Hongchao Du*
City University of Hong Kong
Hong Kong, Hong Kong
hongcdu2-c@my.cityu.edu.hk

Yu Wang
Xiamen University
Xiamen, China
orwangyu@stu.xmu.edu.cn

Jie Zhang
Peking University
Beijing, China
jiez@pku.edu.cn

Qiao Li
Mohamed bin Zayed University of Artificial Intelligence
Abu Dhabi, United Arab Emirates
qiaoli045@gmail.com

Chun Jason Xue
Mohamed bin Zayed University of Artificial Intelligence
Abu Dhabi, United Arab Emirates
jason.xue@mbzuai.ac.ae

Abstract

With the increasing memory demands of emerging applications, tiered memory has become a viable solution for reducing data center hardware costs. Given the low performance of the capacity tiers in tiered memory systems, optimizing memory management is crucial in improving overall system performance. This paper identifies three key limitations in existing tiered memory solutions. First, existing solutions often perform differently across different workloads, leading to suboptimal performance in some workloads. Second, they often fail to adjust migration strategies in response to low fast memory tier access rates, resulting in ineffective data placement. Third, they often miss the opportunity to dynamically tune the memory migration scope based on workload patterns, leading to unnecessary page migrations and under-utilization of tiered memory potential. This paper proposes ArtMem, a reinforcement learning (RL)-driven framework that dynamically manages tiered memory systems and adapts to workload evolution to address these limitations. ArtMem enables better placement of memory pages, enhancing system performance while reducing unnecessary migrations. Experimental evaluations show that ArtMem outperforms state-of-the-art tiering systems, achieving 35% - 172% performance improvements over diverse workloads.

CCS Concepts

• **Computer systems organization → Heterogeneous (hybrid) systems**.

Keywords

Operating system, Tiered memory management, Virtual memory, Reinforcement learning

*Both authors contributed equally to this work.

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731001

ACM Reference Format:
Xinyue Yi, Hongchao Du, Yu Wang, Jie Zhang, Qiao Li, and Chun Jason Xue. 2025. ArtMem: Adaptive Migration in Reinforcement Learning-Enabled Tiered Memory. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731001

1 Introduction

With the growth of memory-intensive applications [39, 45, 47], such as graph processing, machine learning, and in-memory databases, DRAM-based main memory systems experience significant drawbacks of high costs and insufficient capacity. Data from several companies [30, 36, 64] indicate that memory accounts for approximately 40% of server costs. Furthermore, DRAM density cannot scale due to physical limitations [32, 49], implying that using DRAM solely as main memory is insufficient for memory-intensive applications. Byte-addressable persistent memory (PM) and Compute Express Link (CXL)-enabled tiered memory systems present new opportunities for designing cost-effective, large-capacity [63, 66] memory architectures. However, the latency of these slow memories, such as PM or CXL-attached memory, can be up to twice as high as DRAM [23]. To deal with this, emerging tiered memory systems consisting of DRAM and slow memory tiers dynamically migrate pages between tiers, leveraging the performance benefits of fast memory while maintaining the capacity advantages of high-latency tiers.

Existing tiered memory systems have proposed various designs to manage data placement across memory tiers. AutoNUMA [58] and AutoTiering [27] track memory accesses using page faults and promote pages based on access frequency. Nimble [65] scans page tables to identify hot pages and migrates them to the fast memory tier. HeMem [50] leverages hardware-based sampling to monitor memory accesses and makes migration decisions based on a precomputed hotness threshold. Thermostat [9] and Kleio [18] propose to migrate data at the granularity of segments or objects. While each excels for a subset of workloads, none covers a full spectrum of workloads to adapt to all access patterns. The static hotness threshold and heuristics used in these systems struggle to capture the dynamic memory behaviors of different workloads, leading to suboptimal performance. In addition, the limited scope of page

migration either hampers the efficient utilization of fast memory tiers or incurs unnecessary migrations with significant overhead.

To address these issues, a tiered memory system should employ adaptive and workload-aware migration policies, derive robust metrics considering the overall access distributions, and dynamically determine the scope of page migration between memory tiers, which is the goal of this paper.

This paper introduces ArtMem, a reinforcement learning-enabled tiered memory system that addresses the aforementioned issues through adaptive migrations. The proposed design consists of three key components. First, we introduce an efficient RL-enabled tiered memory architecture that learns to adaptively migrate pages between tiers to increase the access ratio to the fast memory tier. An RL agent continuously optimizes its migration policy through interactions with the environment, effectively capturing the dynamic memory behavior of various workloads. Second, the proposed approach dynamically adjusts the migration scope to avoid unnecessary page migrations while maximizing performance benefits. This is achieved by incorporating the exponential moving average of access frequency (EMA) and recency information while dynamically adjusting the hotness threshold. Third, to integrate RL into the tiered memory system without impacting application performance, we introduce dedicated background threads that perform necessary RL computations, sampling, and migration operations asynchronously. These background threads operate transparently to the applications, ensuring that the critical path of memory access remains unaffected. Through extensive evaluation across a wide range of memory-intensive workloads, we demonstrate that ArtMem outperforms state-of-the-art tiered memory systems in almost all cases, achieving superior performance while minimizing migration overheads. Compared to state-of-the-art solutions, ArtMem improves performance by 114% on average in all scenarios.

The main contributions of this paper are:

- We analyzed the behavior of recent tiered memory systems under different memory access patterns and found that their performance is workload-dependent;
- We propose a tiered memory management framework, ArtMem, which dynamically places pages at the appropriate tiers. This framework includes several components to achieve efficient and effective dynamic page placement: hardware sampling of page access frequency, page ordering based on recency, and migration scope-aware page migration;
- We integrated reinforcement learning (RL) to enable adaptability. By dynamically adjusting migration scope, ArtMem consistently improves performance, surpassing static or heuristic-based methods.

2 Background and Related Work

2.1 Access Monitoring in Tiered Memory

One rule in tiered memory is to place hot data in DRAM. Memory access monitoring is adopted to track the access frequency of pages. Typically, there are three methods for monitoring [15]: page table entry scanning [19, 25, 35, 51, 65], page fault capturing [9, 27, 36, 58, 59], and hardware event sampling [19, 30, 50, 51]. The Linux kernel tracks memory access by periodically checking and clearing the accessed/dirty bits. DAMON [42, 43] and MTM [51] also collect page access information by periodically scanning page tables, and control overhead and accuracy by splitting and merging sampling regions. The scan interval and frequency of page tables affect CPU overhead and result accuracy, requiring careful trade-offs. Autotiering [27] migrates or collects data upon page faults. However, since page faults occur on the program's critical execution path, performing other operations at this time can introduce significant latency. For example, Thermostat experiences a several-fold slowdown when tracking all pages [9, 54]. Common hardware event sampling is based on the Performance Monitoring Unit (PMU) and Precise Event-Based Sampling (PEBS)[3]. The PMU records the occurrence count of specific events [7]. As an extension of PMU, PEBS can accurately capture contextual information of triggering events, such as instruction addresses and register states. The Linux kernel provides a system call to configure the sampling events, sampling period, and other parameters of hardware event sampling. The advantage of using hardware events is that they can provide page access information written to the PEBS buffer and flexibly set the events of interest. However, if the hardware changes, the register addresses of the event might change, requiring corresponding kernel modifications.

2.2 Page Migration Metrics

The frequency and recency of page accesses are two common migration metrics in literature [27, 30, 35, 36, 50, 51, 58, 65]. The setting of migration metrics directly impacts the migration efficiency of tiered memory. When using frequency as a migration metric, pages with higher total access counts are more likely to be migrated to the fast memory tier. However, when frequently accessed regions are no longer accessed over time, the accumulated access counts become ineffective. When using recency as a migration metric, recently accessed pages are more likely to be migrated to the fast memory tier. This approach can quickly react to memory regions exhibiting temporal locality. However, it lacks the accumulation of access data, which can lead to overlooking moderately hot regions and causing jitter. The Linux kernel uses LRU lists to organize pages with different reclamation priorities. Some works extend the LRU list to multi-tier memory systems [27, 29, 65]. For instance, Multi-clock [35] adds a third LRU list as a candidate list in the lower memory tier, significantly enhancing the throughput of tiered memory systems. In addition to using LRU to organize pages, the structure of the pages is also modified [30] to track the number of accesses to individual pages.

2.3 Reinforcement Learning for System

Reinforcement learning (RL) can learn optimal behavior strategies through trial and error in an environment by maximizing cumulative rewards [53]. The first step of RL is to structure the problem as a Markov Decision Process (MDP), which has five components (agent, environment, state, action, and reward) that work together in a well-defined way. An agent receives observations s from the environment and selects actions a to gain rewards or avoid penalties. Q-learning [52] and Sarsa [56] are two widely used model-free RL methods. Both methods update the action-value function $Q(s, a)$ to estimate the expected return of a particular action in a given state. The Q value reflects the expected cumulative reward from

the current state by taking a specific action and following a certain policy stored in a Q table. The update of Q values involves three important hyperparameters. The learning rate α represents how much new experiences influence the Q values. The discount factor γ affects the weight of long-term returns. The parameter ϵ is used to balance experience and exploration.

Reinforcement learning algorithms possess strong generalization capabilities and real-time adaptability, enabling agents to adjust their behavior based on immediate environmental feedback. Many existing works leverage these advantages of reinforcement learning to address issues in the architecture domain. For instance, online reinforcement learning has been used for cache decision-making [33], SSD I/O request and load management [40], predicting 3D NAND flash failures [61, 62], and memory data prefetching [14, 26, 46, 67]. These works demonstrate the potential of reinforcement learning in solving such problems.

3 Motivation and Observations

Real-world workloads often have large data sizes and complex access patterns, making it difficult to capture their behavior accurately [30]. Additionally, the complexity of real workloads complicates the control of variables during analysis. To explore the performance of different page migration strategies under varied memory access patterns, we construct four workloads based on insights from previous tiered memory studies [18, 35, 43, 51] that visualize memory access patterns of real-world applications. These synthetic access patterns are simple but sufficient to illustrate the observed issues. Figure 1 illustrates the constructed workloads (referred to as S_k, where k indicates the pattern index). These workloads are run on MASIM [41], a simulator for dense memory access that allows users to specify data access patterns through configuration files. The total memory usage is 32GB, with 16GB of memory usage coming from DRAM and the remaining half from PM. The tiered memory systems used for evaluation and detailed hardware specifications can be found in Section 6. The latency and bandwidth of the system can be found in Table 2. We evaluate seven state-of-the-art tiered memory systems and use the normalized execution time as the performance metric. We made several observations that led to the design of ArtMem.

3.1 Performance Divergence by Access Pattern

Figure 2 shows the performance comparison of different strategies on the synthetic access patterns. For each memory access pattern, we explain the reasons for the performance behaviors in tiered memory systems.

Pattern S_1 has a high access locality, with over 90% of accesses occurring in two 500MB hot regions. AutoTiering can quickly distinguish hot pages by sorting pages based on NUMA fault counts. Multi-clock, similar to AutoNUMA and TPP, uses changes in access bits to adjust page placement. These approaches are slower in recognizing hot regions when the hot areas are small but intensely accessed. MEMTIS performs well but marks all pages as hot for this workload. This happens because MEMTIS determines the hotness threshold based on the size of the DRAM tier. As a result, all pages in Pattern S_1 can meet this hotness threshold. MEMTIS actively migrates over 15GB of data and keeps these hot pages in the fast memory tier.

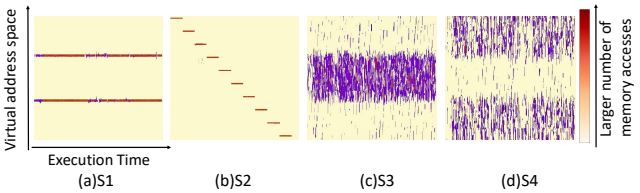


Figure 1: Four manually-generated access patterns.

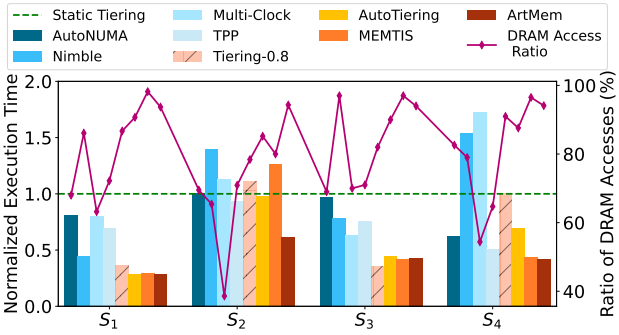


Figure 2: Performance comparison of seven tiered memory systems under diverse memory access patterns. Results are normalized to the static tiered memory without migrations.

Pattern S_2 is designed to simulate scenarios where a region is frequently accessed during a specific period but not thereafter. AutoNUMA, TPP, and AutoTiering exhibit less than a 10% improvement compared to the static baseline. When unable to adapt to such access patterns, current designs may even have negative impacts. MEMTIS uses an exponential moving average of access frequency to guide page migration, which makes its decisions unreliable for workloads with clear recency characteristics. MEMTIS and Nimble have the worst performance. Both fail to migrate hot pages accurately and make incorrect migrations.

Pattern S_3 consists of a 12GB hot region, differing from Pattern S_1 in that the performance improvement relies more on how quickly these regions are identified and migrated to the fast memory tier. Multi-clock's performance gap with other systems narrows in this scenario because, with the increase in hot pages, identifying and migrating all of them to the upper tier takes a longer time. Meanwhile, Nimble's disadvantage of slow page hotness differentiation is further amplified.

Pattern S_4 consists of a 20GB hot region but with half the heat of Pattern S_3. Since the DRAM tier cannot accommodate all hot pages, the tiered memory must avoid thrashing. AutoNUMA identifies the cold pages in the DRAM node and migrates them to the PMEM node. TPP designs a lightweight demotion. Compared to the other three access patterns, the designs of AutoNUMA and TPP give them an advantage in Pattern S_4. In this scenario, Multi-clock fails to migrate 82% of the pages, leading to performance loss. Nimble's performance loss is similar to that in Pattern S_2.

As analyzed, we summarize the advantage and disadvantage patterns of the seven tiered memory systems in Table 1 and make the first observation. **Observation 1: Existing methods may perform well on certain workloads but poorly on others**.

Table 1: Comparison of existing tiered memory designs.

Strategy	Key Design	Advantage Pattern	Disadvantage Pattern
MEMTIS[30]	Exploiting the exponential moving average of access frequency	High spatial locality	Random access
AutoTiering[27]	Opportunistic promotion and migration	Easily distinguished hot and cold data	Warm data
TPP[36]	Lightweight demotion, decoupled allocation and reclamation paths	Stable pattern	Unstable pattern with burst of hot pages
AutoNUMA[58]	Mostly frequently accessed	Stable pattern	Unstable pattern with burst of hot pages
Multi-clock[35]	Candidate LRU lists	Easily distinguished hot and cold data	Warm data
Nimble[65]	Batch migration	High spatial locality	Random access
Tiering-0.8[59]	Reset hotness threshold once workload change	High spatial locality	Random access

3.2 The Impact of Fast Memory Access Ratios

In tiered memory systems, due to performance differences between the device media, it will be beneficial if the majority of memory accesses occur in the fast memory tier. We used the *perf* [1] tool to sample the addresses where memory accesses occurred periodically. We collect the average DRAM access ratio throughout the program run time for each memory system and present the results in Figure 2. The DRAM access ratio significantly decreases for a tiered memory system when faced with access patterns that it cannot adapt to. Furthermore, we analyzed the correlation between performance and DRAM access ratio for different tiered systems, as shown in Figure 3. There exists a strong correlation between the two. **Observation 2: A low access ratio in the fast memory tier indicates that the current page migration mechanism is ineffective under such conditions.** This metric can serve as a real-time indicator of memory utilization efficiency in a tiered memory system. By leveraging this information, we can make more informed and targeted page placement decisions. Unlike heuristic-based methods, this approach enables adaptive learning and decision-making through real-time system feedback and exploration.

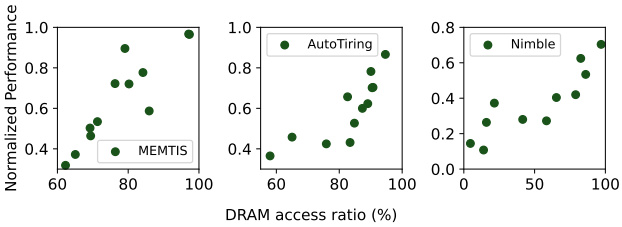
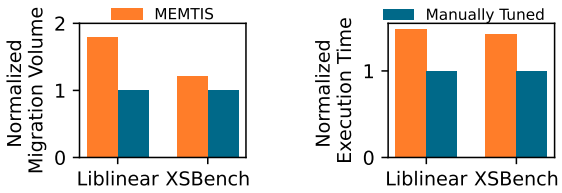


Figure 3: The correlation between performance and DRAM access ratio. The y-axis indicates performance normalized to DRAM-only execution, and the x-axis is the DRAM access ratio. Each point corresponds to a workload. Each subplot represents a recent tiered memory system, where the Pearson correlation coefficients are 0.89, 0.81, and 0.87, respectively.



(a) Migration volume. **(b) Performance.**

Figure 4: MEMTIS default hotness threshold determined by DRAM capacity vs. manually tuning MEMTIS hotness threshold for different applications. The lower the value, the better.

3.3 The Impact of Memory Migration Scope

Migration scope represents a set of pages that are eligible for migration. The size of the migration scope is controlled by the migration number. Existing work defines the migration scope by setting a hotness threshold as a migration metric [27, 35, 36, 50, 51, 58, 59, 65]. However, Memory access patterns change dynamically during program execution, making the static hotness threshold less effective. Heuristic hotness threshold adjustment methods based on empirical rules can achieve better results [30]. However, such a design may experience many unnecessary page migrations and an inaccurate hotness threshold in some workloads. For example, in Pattern S_1, MEMTIS identifies all pages as hot pages and migrates 15GB of data because the DRAM capacity is large enough, while in reality, only up to 1GB migration is needed. Using frequency alone as the migration metric has limitations. When hot regions are not repeatedly accessed, as in Pattern S_2, the slow cooling operation cannot remove the old hot data and migrate the new hot data in time. Also, for warm pages about to become hot pages, MEMTIS may incorrectly classify them as cold pages and migrate them to slow memory [64]. In Pattern S_4, pages with the same access frequency amount to 20GB, exceeding the DRAM capacity. In such a case, MEMTIS migrates 47GB of pages, causing memory thrashing.

Adjusting the hotness threshold according to the program's access patterns can improve the aforementioned issues. We ran Liblinear [31] and XSBench [57] as examples, manually reducing the hotness bins of MEMTIS and counting them into warm bins. As shown in Figure 4a, the page migrations for Liblinear are significantly reduced. Figure 4b shows that an empirical setting can improve the performance by 47% and 42% for Liblinear and XSBench, respectively. **Observation 3: The memory migration scope should be tuned dynamically for different workload patterns to fully exploit the tiered memory's potential.**

3.4 The Need for Adaptive Learning Strategies

Due to the complexity of the memory access patterns, it is difficult, if not impossible, to manually adjust the hotness threshold for different access patterns in real applications. We need to enable tiered memory to adapt to changes in access patterns and select the appropriate migration scope based on the characteristics of the current access pattern. Adopting a strategy with learning capabilities to provide guidance becomes an appealing choice.

Training models to predict memory access patterns is a possible solution, but it faces two key challenges. First, the model must exhibit adaptability to handle diverse and dynamic memory access patterns, requiring the training benchmarks to encompass a wide range of behaviors. Second, tiered memory systems allow highly flexible combinations of memory media, necessitating that the model observes benchmark sets under various configurations. Given these challenges, reinforcement learning (RL), which learns through interaction with the environment, is better suited for this problem than supervised or unsupervised learning methods.

4 ArtMem Design

4.1 Overview

In this section, we propose an efficient and effective tiered memory system based on reinforcement learning, which enables dynamic and adaptive page migration among memory tiers with low overheads. Figure 5 illustrates the overall architecture of the proposed system, which is composed of three main components. At the top layer lies ArtMem's reinforcement learning (RL) framework, responsible for training and decision-making. The middle and bottom layers comprise a range of software and hardware functions that support ArtMem's operation. The dark green lines in the figure represent the interactions and dependencies between these functional modules, the RL framework above, and the underlying hardware components.

In the following, we first introduce the design of an efficient reinforcement learning framework for tiered memory (§4.2) and then introduce how to properly utilize the information in the page migration system to make accurate migration scope decisions (§4.3). Section 4.4 introduces how to seamlessly integrate the RL framework into tiered memory to achieve dynamic and adaptive page migration. Finally, Section 4.5 summarizes the ArtMem workflow.

4.2 Efficient RL Framework for Page Migration

ArtMem employs a Q-learning–based, model-free reinforcement learning (RL) approach with constrained state and action spaces to minimize computational overhead. In memory systems, the vast address space, often encompassing millions of pages, makes per-page RL decision-making computationally impractical, particularly in large-scale tiered memory architectures. To address this challenge, ArtMem defines states and actions at the system-wide level rather than at the granularity of individual pages.

In Q-learning, both states and actions must be discretized to enable efficient indexing and updates within the Q-table. However, defining an excessively large state or action space can significantly slow down the learning process by increasing Q-table overhead

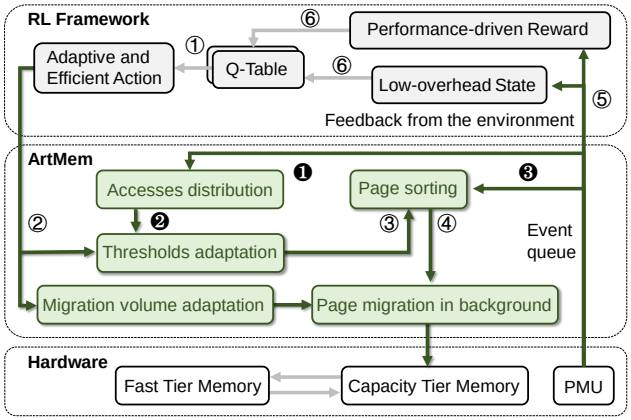


Figure 5: ArtMem architecture, which includes three main layers. Arrows represent the data flow between different functions.

and delaying convergence. Therefore, it is critical to design a compact yet expressive set of discrete states and actions that balances memory and computational efficiency with effective reward-driven learning.

State design. The state must reflect how well the tiered memory adapts to the current program's memory access patterns and perceive changes in access patterns. Based on observation 2, ArtMem uses the access ratio of fast memory to represent the state. The raw access ratio is a continuous value between 0 and 1. We round it to discrete k+1 states $[0, \ldots, k]$ as follows:

$$\tau = \lfloor \frac{DRAM_{access} \times k}{DRAM_{access} + PM_{access}} \rfloor \quad (1)$$

where $\tau = k$ means all sampled accesses happen in the DRAM with $PM_{access} = 0$. It is important to note that a state value of 0 might indicate all accesses occur in the slow memory tier or no events were sampled ($DRAM_{access} = PM_{access} = 0$, e.g., most of the accesses hit in the CPU cache or the event of concern did not occur), we set a separate state ($k + 1$) for the latter.

Action design. Based on observation 3, ArtMem adopts migration scope as the action target, which depends on two key parameters: the migration number and the hotness threshold. Two separate Q-tables and independent action sets are designed to adjust these parameters dynamically and collaboratively. Each Q-table is designed with an appropriate number of actions to balance system overhead and performance, e.g., set a different migration number and ±4 for the hotness threshold. Details are provided in Section 5.

Reward design. Rewards serve as the primary feedback mechanism through which ArtMem learns from experience, and their definition directly shapes the desired learning behavior and impacts overall learning effectiveness. In ArtMem, the optimal behavior (i.e., the most highly rewarded) is to migrate an appropriate number of pages in a timely manner in response to changes in memory access patterns, thereby maintaining a high access ratio in the fast memory tier. In contrast, unnecessary migrations that degrade DRAM utilization should be penalized.

ArtMem's objective is to maximize both immediate rewards and cumulative return over time. If the reward were defined solely as the

increase in the fast memory tier access ratio between consecutive phases (i.e., $\tau_i - \tau_{i-1}$), the agent might exploit this by inducing large access ratio fluctuations to gain more rewards, potentially destabilizing system performance. To mitigate this, we introduce an additional reward term based on the deviation from a target access ratio, guiding the learning process toward maintaining the access ratio near an ideal value. The full definition of reward is as follows:

$$\tau_i - \beta + \lambda(\tau_i - \tau_{i-1}) \tag{2}$$

where τ_i is the fast tier access ratio defined in Equation 1 for period i, and β represents the desired fast memory tier access ratio. If no migration occurred in the previous period, we would not impose penalties or rewards for changes in access ratios because it is only related to the access characteristics of the workload in this case. Therefore, we apply λ to indicate whether a migration occurred in the last period to reward only the first item when there is no migration ($\lambda = 0$).

The combined reward structure encourages ArtMem to perform migrations that improve the fast tier access ratio and discourages those that do not improve or even worsen it.

4.3 Dynamic Adjustment of Migration Scope

One key challenge in fully leveraging tiered memory systems lies in accurately and dynamically determining the appropriate migration scope—i.e., how many and which pages should be moved between memory tiers. This subsection describes how ArtMem addresses this challenge using exponential moving averages (EMA), page sorting mechanisms, and reinforcement learning. Figure 6 provides an overview of how these components interact within the system.

EMA captures the distribution of memory accesses. Given that memory access frequencies typically follow a Zipfian or Pareto distribution [8, 10], EMA is employed to track access patterns over time in a lightweight, adaptive manner. ArtMem records the number of memory accesses each page receives, grouping them into exponential bins with base 2 to compactly represent the access distribution. Upon each access sampling, the corresponding page is checked for bin reassignment, and the bin counters are updated accordingly. Additionally, a cooling operation is triggered every two million samples, during which all bin counts and per-page access records are halved to gradually discount stale information.

Dynamic adjustment of hot page threshold. Based on the memory access distribution captured by EMA, a simple approach is to define the hot page threshold according to the DRAM capacity. However, this method can lead to excessive and unnecessary data migrations in certain workloads, as we discussed in Section 3.3. Prior work [64] has also shown that such static threshold definitions may fail to effectively distinguish truly hot pages.

To address this, ArtMem initially sets the hot page threshold based on DRAM capacity and resets it following each cooling operation. Between cooling phases, the threshold is dynamically refined using knowledge learned by the reinforcement learning agent, allowing for adaptive migration decisions aligned with evolving access patterns.

Page sorting. The above design is mainly based on page access frequency but fails to utilize recency information well. NUMA fault-based methods can assist EMA in detecting new hot pages [64]

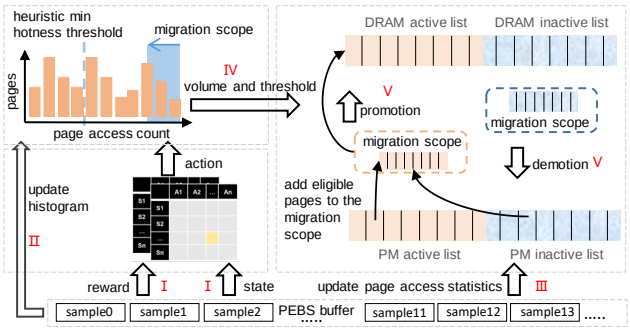


Figure 6: The overview of ArtMem. I: The bottom sampling data provides the basis for updating rewards and states in the RL algorithm. At the same time, the sampling data are used to II update the distribution of memory accesses and III maintain the access statistic of each sampled page The RL algorithm IV determines a suitable migration scope based on the given action and then V performs migration between the sorted LRU lists at different tiers.

but may introduce high overheads. Therefore, this paper proposes an LRU list-based page sorting method to represent recency information in ArtMem. In Linux, both the fast memory tier and the capacity tier maintain separate active and inactive page lists [22]. ArtMem incorporates recency information by preferentially selecting migration candidates from the inactive list of the fast tier for demotion and the active list of the capacity tier for promotion.

Existing inter-layer page migration strategies [30, 58] adopt a conservative strategy: when a page is migrated, it retains its activity status from the source memory tier. For instance, a PM page promoted from the inactive list will be placed on DRAM's inactive list. This approach can cause premature demotion, leading to performance jitter. To address this issue, ArtMem adopts a more aggressive policy: regardless of its current status, a migrated page is always inserted at the head of the active list in the fast memory tier. This design ensures that recently accessed pages are promptly recognized and prioritized. As a result, ArtMem performs lightweight sorting of memory pages by considering both recency and access frequency.

Dynamic adjustment of the number of pages to migrate. Analyzing the recency and frequency and adjusting the hotness threshold can rank the hotness of pages. The next question is how to set the number of pages to be migrated. If a program's memory access is inherently uniform, relying solely on threshold-based strategies can result in numerous unnecessary migrations, as seen in Pattern S_1. Since it is difficult to determine the appropriate number of migrated pages, we use RL to automatically learn the appropriate number based on real-time feedback (Section 4.2).

4.4 Integrating RL in Tiered Memory

To function effectively, RL requires the ability to continuously observe environmental states and take corresponding actions—in this case, monitoring memory behavior and issuing page migrations. However, as noted in prior work [30, 36, 59, 65], conventional tiered memory architectures are not designed to support such structured

interactions between a learning agent and the memory subsystem. This mismatch necessitates a redesign of the migration mechanism to better accommodate adaptive, learning-based control. In this section, we describe how ArtMem introduces background threads to support the RL algorithm, enabling it to operate efficiently within the tiered memory system.

Sampling threads. The sampling threads are crucial for ArtMem to obtain environmental information, providing the necessary data for subsequent state and Q-value updates. ArtMem focuses on the memory addresses and memory load events of the target PID during periodic sampling. The sampling thread acquires this information through PEBS (Precise Event-Based Sampling) and places the sampled data into a ring buffer. ArtMem then processes the page information in the ring buffer sequentially, sorts pages ❸, updates the memory access distribution bins ❶, performs cooling ❷, and updates the fast tier access ratio ⑤.

Migration thread. The migration thread is periodically awakened by ArtMem to perform inter-tier migration operations. When the migration thread receives a migration instruction from RL ②, it first checks how much free space is available in the fast memory tier. If the free capacity is less than the amount to be promoted, the thread first performs a demotion operation. Demotion starts from the tail of the inactive list in the fast memory tier. When the free capacity of the fast memory tier can accommodate the number of pages to be migrated, pages are migrated from the head of the active list in the capacity tier to the head of the fast memory tier list in order ④.

4.5 ArtMem Workflow

Model-free reinforcement learning iterates continuously to reach an optimal solution. ArtMem interacts with the environment over several cycles, each repeating the same process. This subsection presents how ArtMem operates by the iteration sequence of model-free reinforcement learning.

When the program starts, ArtMem sets the initial state to k, which means the DRAM access ratio is 100%, referring to Equation 1. This setting is reasonable as the program loads from DRAM. In the Q-Table for migration for the state of k, ArtMem assigns a high Q-value of 1 to action 0. Migration is unnecessary because the program is already fully utilizing the advantages of the fast memory tier. All other Q-values in this Q-Table and the Q-Table for the hotness threshold are initialized to 0 (Algorithm 1 line 1). This is because it is not possible to determine which action is better for each of these other states.

ArtMem selects an action from the Q-Table based on the current state ①. With a probability of ε, ArtMem randomly chooses an action to explore whether the action can make the tiered memory system more efficient. Conversely, with a probability of $1-\varepsilon$, ArtMem selects the action with the highest Q-value, leveraging the learned actions (Algorithm 1 line 4).

Actions cause changes in the system environment, which are captured by the sampling thread ⑤. ArtMem obtains the DRAM access ratio changes through fixed interface access and updates the Q-values accordingly ⑥. It then receives the new state and repeats the previous step. Algorithm 1 shows the breakdown of the key steps in this process.

Algorithm 1: ArtMem Workflow.

1 Initialize $Q(k,0)$ to 1 and the rest Q table to 0;
2 Initialize τ_{i-1} to k;
3 **while** *Programs not finish running* **do**
4 Choose an action from τ_{i-1} using policy derived from Q (ϵ-greedy);
5 Migration thread migrates pages based on updated parameters;
6 Observe state τ_i from sampling data;
7 $r \leftarrow \tau_i - \beta + \lambda(\tau_i - \tau_{i-1})$;
8 $Q(\tau_{i-1}, a) \leftarrow Q(\tau_{i-1}, a) + \alpha[r + \gamma max_{a'} Q(\tau_i, a') - Q(\tau_{i-1}, a)]$;
9 $\tau_{i-1} \leftarrow \tau_i$;
10 **end**

5 Implementation

We implemented a prototype of ArtMem in the Linux kernel v5.15.19, which is open-sourced at https://github.com/Yitrus/ArtMem. This section presents the details of the implementation.

Tracking Page Access Information. We use 2MB huge pages as the default page migration unit to reduce address translation overhead. We can store huge page access data without additional memory overhead by leveraging an unused struct page within the *compound_page*.

Adding Kernel Threads. ArtMem implements a background *kmigrated* thread to handle page migrations. Each CPU core has a sampling thread *ksampled*. ArtMem ensures the consistency of statistical data through atomic operations.

Adding Interaction Channels for Environment and Agent Information. Two mount points are added under the memory control group directory. ArtMem uses a pseudo-file system to access and control kernel memory resources. The *memory.hit_ratio_show* file is used to retrieve information collected by the sampling threads, while the *memory.action_show* and *memory.threshold_show* files are used to pass actions to the migration thread. Establishing these interaction channels allows the reinforcement learning algorithm to be implemented in user space, facilitating algorithm parameter adjustments and comparative experiments.

Heuristic Minimum Hotness Threshold. RL may set the hotness threshold too low during the exploration phase, leading to numerous unnecessary migrations. Although ArtMem can adjust the hotness threshold later based on reward and state, these pages will likely be downgraded to the capacity memory tier shortly, causing thrashing. Thus, it is crucial to set a minimum hotness threshold for threshold adjustment. In our experiments, we empirically set the minimum hotness threshold to 16 accesses for each page.

Q-table Setting. Given our minimum hotness threshold is 16, smaller adjustment steps do not adequately reflect differences in page access frequencies. We choose adjustment ranges of ±8, ±4, and 0 for the Q-table that dynamically adjusts the threshold. For example, the previous hotness threshold was x, and +4 indicates that ArtMem has decided to increase the hotness threshold to $x+4$. For the Q-table controlling migration number, we define nine possible migration sizes. The minimum migration size is set to 16MB (i.e.,

Table 2: Hardware overview of the experimental system.

Memory Tier	Latency	Bandwidth
Fast Memory	92ns	81 GB/s
Slow Memory	323ns	26 GB/s

eight 2MB pages), and each subsequent action doubles the size up to a maximum of 2048MB (1024 2MB pages). An additional action representing no migration (0MB) is also included, enabling the agent to decide against migrating when necessary. For states, we set $k = 10$, so there are 12 different states in total.

6 Evaluation

This section presents the evaluations of ArtMem. The goal is to examine the adaptability of ArtMem under different access patterns by various applications (§6.2). We also discuss how it improves workload performance (§6.3) and analyze its overhead (§6.4).

6.1 Experimental Setup

ArtMem can be applied to any tiered memory system with fast and slow memory, such as PM-based, CXL-based, and NUMA-based systems. We show the results of ArtMem on a real system with DRAM and PM, but other tiered memory systems can also benefit from our design.

Hardware setup. We used an Intel(R) Xeon(R) Gold 6330 CPU with 28 cores per socket. Each socket has 4 DDR4 modules, each with a capacity of 16GB, and 4 Intel persistent memory modules, each with 128GB. We used one socket during the evaluation, with 64GB DRAM as the fast memory tier and 512GB PM as the slow memory tier. PM are configured as remote memory nodes with the ndctl [2] tool. We measured the latency and bandwidth of the system in Table 2 with the Intel Memory Latency Checker [5].

Comparison targets and Workloads. We compared seven state-of-the-art tiered memory systems as shown in Table 1. For each system, the page size adheres to the original design. By doing so, we ensure that the evaluated system can fully leverage its advantages. Notably, AutoNUMA uses version 5.18.0 of the kernel.

We selected 8 representative memory-intensive applications. The details are shown in Table 3. These workloads are widely used to evaluate tiered memory systems [30, 35] and meet the need to demonstrate different memory access patterns.

Memory Ratio Configuration. ArtMem uses the cgroup [4] to control the memory usage of each tier during program execution. Following previous work [30, 60], we set the memory ratios to 2:1, 1:1, 1:2, 1:4, 1:8, and 1:16, respectively. MEMTIS [30], and Nimble [65] also utilized the cgroup interface. For AutoNUMA [58], AutoTiering [27], Tiering-0.8 [59], Multi-clock [35], and TPP [36], we modified the kernel boot parameters using the *memmap* GRUB option [48] to limit DRAM size to achieve the corresponding fast memory ratio. For example, we configured Memcache, with a 32G footprint, to have 16GB DRAM available under the 1:1 memory ratio.

6.2 Performance of Different Situations

ArtMem runs the Liblinear program several times to initialize the RL algorithm, primarily to obtain a Q-table with learning experiences.

Table 3: Workloads for evaluation.

Workloads	Memory footprint	Descriptions
YCSB [16]	32G	In-Memory Database
CC [11, 55]	69G	Connected Components
SSSP [34, 37]	64G	Single Source Shortest Path
PR [13, 28]	25G	PageRank
XSBench [57]	69G	HPC Workloads
DLRM [24, 38]	72G	Deep Learning Recommendation Model
Btree [6]	24G	In-Memory Index Lookup
Liblinear [31]	68G	Machine Learning

Throughout the evaluation, the hyperparameters of RL remain fixed, and the sensitivity study is shown in §6.3. In all experiments, ArtMem first places pages in fast memory before overflowing to the slower tier. We show the runtime for each comparison, normalized to AutoNUMA at 1:16 ratio, where a lower value indicates better performance. The results are shown in Figure 7, and analysis is provided below.

Performance Results Summary. Under different memory configurations (DRAM:PM ratios), ArtMem achieves an average performance improvement of 132%, 124%, 104%, 91%, 72%, and 67% compared to the seven baseline systems, respectively. Although ArtMem performs similarly to the best baseline in some workloads, ArtMem shows better adaptability under different workloads and memory ratios, with an average improvement of 10.4%-43.65% compared to other baseline systems.

In-memory database. We ran YCSB workloads A, B, C, D, and F in Memcached [21], executing them sequentially in the order of ABCFD. YCSB E was omitted because the Memcached we used does not support the scan operation. Workload A is updated frequently, workload B mainly performs read operations with only 5% writes, and workload C is entirely dedicated to read operations. Workload D performs 95% read operations, and 5% insert operations. It inserts new data into the database, and users primarily focus on reading recently inserted records. Workload F involves reading, modifying, and then writing back records. ArtMem achieves selective page promotion similar to Multi-clock. This reduces the migration overhead caused by promoting ineligible pages. As a result, it outperforms other methods by 65% on average.

Graph. GAP [12] is a graph analytics framework that runs various graph processing algorithms. Here, we present the evaluation results for SSSP, PR and CC, with the input graphs derived from previous research: Web [17], Twitter [28] and Urand [20]. GAP first loads the graph into memory and then performs multiple workload trials. During the execution phase, the algorithm operates on the data graph already in memory. Due to the inherent randomness of graph algorithms, we recorded the average execution time of 20 running trials to obtain stable performance data. ArtMem shows 12% - 509% performance improvements over other works. This significant improvement is mainly attributed to learning the graph access characteristic, as the performance of graph processing algorithms largely depends on data locality. ArtMem, aiming to

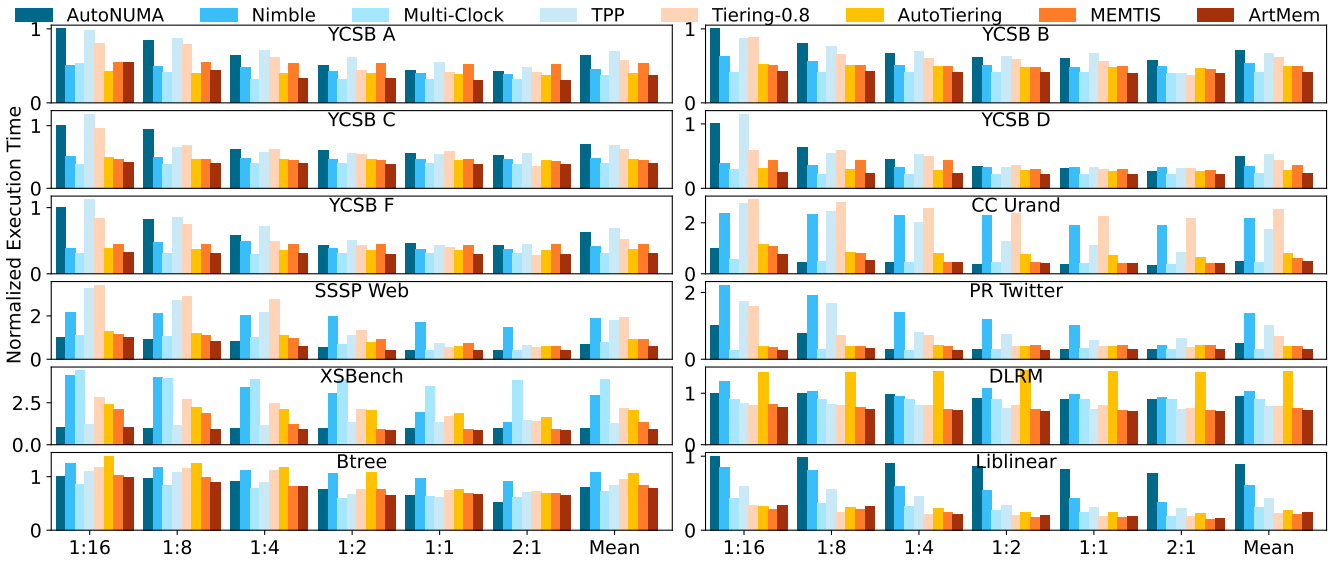


Figure 7: The performance comparison of ArtMem against other systems under various tiering settings (fast tier vs. capacity tier = 2:1, 1:1, 1:2, 1:8, 1:16. For example, the DLRM 2:1 configuration is equipped with 48GB DRAM and 24GB PM.). The execution time of AutoNUMA (1:16) is the baseline, with others normalized to that value. The lower the value, the better.

increase the access ratio of the fast memory tier, is better adapted to the locality inherent in graph algorithms. When the fast memory tier is reduced, ArtMem dynamically adjusts the migration priority, maintaining a comparable overall runtime.

HPC. XSBench is a large-scale simulation program for advanced reactors. ArtMem promptly places the hot regions in the fast memory tier. Throughout the program's execution, ArtMem benefited by migrating the most recently and frequently accessed pages to the fast memory tier. Consequently, compared to other workloads, the performance improvement ranged from 16% to 311% on average.

DLRM. We evaluated ArtMem's effectiveness in the context of recommendation systems by running the Deep Learning Recommendation Model (DLRM) training phase. The model indexes the corresponding embedding vectors based on the input sparse features during each training iteration. The embedding tables occupy the majority of the DLRM model's memory. The random access patterns of these tables are not friendly to tiered memory systems. However, the dense features in DLRM are accessed sequentially during the forward and backward propagation processes. This access pattern is something ArtMem can learn and leverage effectively. Compared to other works, ArtMem demonstrated performance improvements ranging from 5% to 110%.

In-Memory Index Lookup. We use Btree [6] to measure the lookup performance of an in-memory index. We populated the Btree with 300 million key-value pairs and performed 8 billion random lookup operations. Except for multiclock, ArtMem outperforms other works by 4% to 36%. ArtMem's performance is close to that of the best-performing multiclock (within 7%), while multiclock performs poorly on other workloads like XSBench. This also demonstrates our design's adaptability to different access patterns.

Machine Learning. We ran Liblinear using the KDD12 dataset. In the Liblinear workload, ArtMem's performance is 9% lower than MEMTIS, but it still shows a 76% average performance improvement over all designs. We found that during the initial gradient descent phase of Liblinear, ArtMem's access ratio drops sharply to nearly 0% and then takes some time to recover to 70%. In contrast, MEMTIS' access ratio directly drops to 70%. This is mainly because Liblinear's memory access is relatively uniform in the early phase, with no extremely hot pages surpassing ArtMem's migration threshold, so ArtMem performs minimal migrations. MEMTIS, in contrast, uses DRAM capacity as a migration threshold, migrating pages with access counts between 8 and 16, which are subsequently accessed more frequently in the next phase.

Further analysis reveals that the rate at which ArtMem increases the fast memory tier access ratio—from 0% to 70%—is a critical factor influencing its performance on the Liblinear workload. This ramp-up speed is strongly correlated with the accuracy of the sampled access ratios. By increasing the sampling frequency, at the cost of an additional 5.91% overhead for getting DRAM access ratio, ArtMem achieves a further 17.11% performance improvement on Liblinear.

6.3 Understanding ArtMem Performance

This chapter analyzes the factors that affect performance and other features in ArtMem.

6.3.1 Ablation Study. Figure 8 illustrates the performance contributions of ArtMem's three key components. The RL algorithm delivers the most significant improvements. As the proportion of DRAM allocation decreases, the performance gains from RL become more significant. Although the page sorting component shows less noticeable average improvement across workloads, it achieves more than 10% performance gains for specific workloads such as PR and XSBench. Heuristic-based page migration methods perform reasonably well when there is more DRAM.

6.3.2 Fast Memory Access Ratio. We collected the DRAM tier access ratio using the Linux perf tool. To evaluate the necessity of

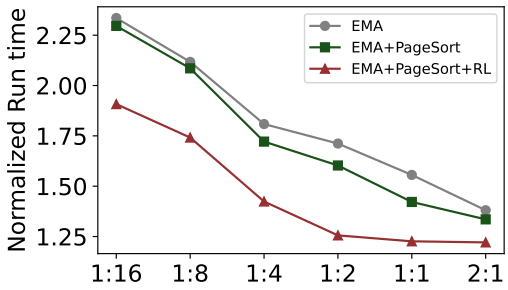
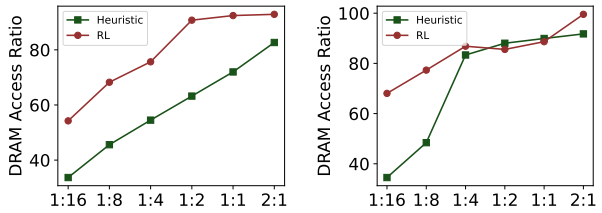


Figure 8: Ablation Study for ArtMem to show the effectiveness of individual ArtMem components and highlight the performance gap between ArtMem and DRAM-only configurations.



(a) SSSP. **(b) CC.**

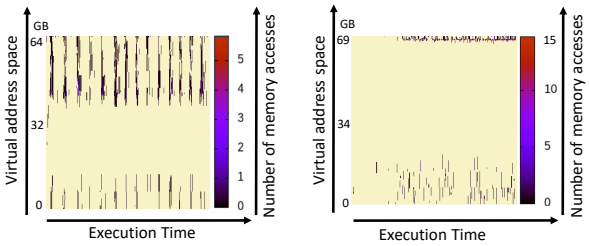
Figure 9: The changes in DRAM access ratio when using heuristic adjustment strategies compared to those incorporating the RL algorithm. Workloads with different memory access patterns exhibit varying trends.

adaptability under different scenarios as emphasized in our design, we analyzed two workloads, SSSP and CC, to observe variations in DRAM access ratio. Figure 9a shows that the DRAM access ratio increases as the PM allocation decreases. Across all configurations, the RL-based method consistently outperforms the heuristic-based approach. In Figure 9b, when the DRAM-to-PM ratio reaches 1:4, both methods converge to similar DRAM access ratios, with no further significant changes. This plateau is attributed to workload-specific memory access characteristics. The access footprints presented in Figure 10 further illustrate these differences, helping to explain the trends observed in DRAM access behavior.

In Figure 10b, CC exhibits a more distinct separation between hot and cold data, with hot data concentrated in smaller regions. This makes it easier to identify hot data, and when DRAM capacity is sufficient to accommodate these regions, migration becomes straightforward and effective. As a result, in Figure 9b, both methods achieve similarly high DRAM access ratios at higher DRAM allocations.

Conversely, in Figure 10a, SSSP has a broader distribution of hot regions with minor differences in access frequency between hot and cold areas. In this case, increasing DRAM capacity gradually improves the DRAM access ratio for both methods. The RL-based method is more proactive in selecting operations that maximize rewards and can identify and migrate pages more effectively, leading to a higher DRAM access ratio.

6.3.3 Number of Migration. To demonstrate the excessive page migrations introduced by heuristic-based strategies in certain scenarios, we present the migration counts for the CC and DLRM



(a) SSSP. **(b) CC.**

Figure 10: The memory access patterns of two workloads, as measured by DAMON[42–44] tool, explain the differing trends in DRAM access ratio in Figure 9.

workloads in Figure 11. MEMTIS exhibits significant fluctuations in migration volume due to its reliance on a hotness threshold defined by DRAM capacity. Across all evaluated workloads, MEMTIS incurs approximately 10× higher average CPU overhead from migration threads compared to ArtMem.

ArtMem and AutoNUMA maintain relatively low migration volumes in every scenario. This stability in ArtMem stems from two design principles: first, the RL agent is penalized for unnecessary migrations through the reward function; second, page migrations are prioritized via access-based sorting, improving overall migration efficiency. ArtMem migrated 30,000 times fewer pages in the DLRM workload compared to CC. This is because DLRM's memory access is largely unskewed, with only a few hot memory regions. ArtMem can adapt to such access patterns through learning, reducing unnecessary migrations.

6.3.4 Exploring Customizability: Using Latency as Reward. The RL agent interacts with ArtMem through a well-defined interface, enabling optimizations tailored to specific deployment environments. Besides fast memory access ratio, memory access latency is another direct factor highly correlated with overall performance. In this section, we examine the impact of using a latency-based reward function within the RL framework. Due to the lack of hardware events that directly report the latency of individual memory accesses, we approximate memory latency by monitoring the number of pending memory access requests per cycle. Experimental results show that this latency-based strategy yields, on average, 3.4% lower performance compared to the DRAM access ratio-based reward. Additionally, it introduces extra runtime overhead due to the additional data collection required. As shown in Figure 12, the latency-based approach exhibits delayed adjustments in page migration decisions in the XSBench workload, which may account for the observed performance degradation.

6.3.5 Exploring Customizability: RL Algorithm Comparison. We compared the performance of Q-learning and SARSA algorithms in tiered memory management. We ran a workload under four scenarios with six memory ratios, normalizing the performance improvements and taking the average. The experimental results in Figure 13 show that both algorithms perform similarly across workloads and access patterns. This indicates that within the current design of the ArtMem framework, both Q-learning and SARSA can effectively adapt to changes in memory access patterns and

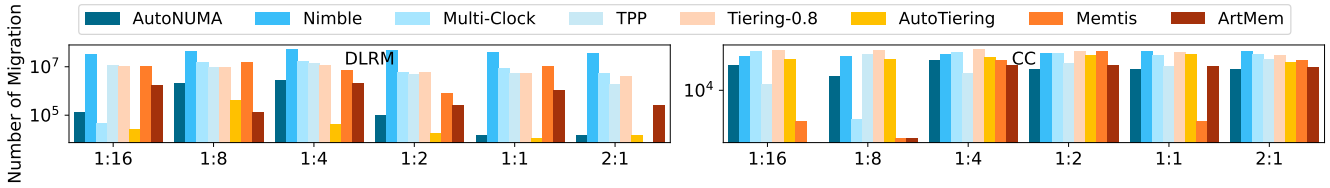


Figure 11: Page migration volume.

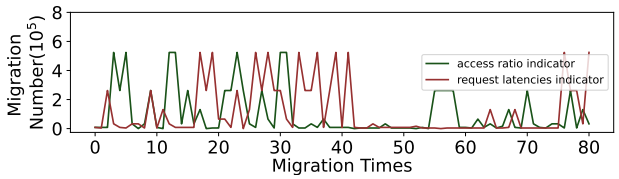


Figure 12: The change in the number of migrations over time when using latency and DRAM access ratio as RL rewards in ArtMem while running XSBench. It also reflects ArtMem's adaptability.

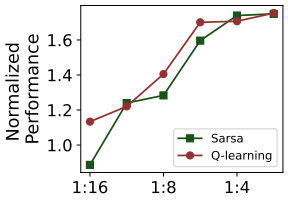


Figure 13: RL algorithm comparison.

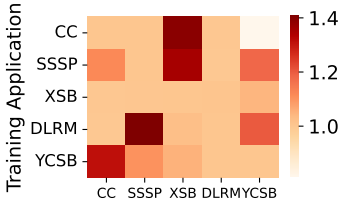


Figure 14: Sensitivity to initial applications.

provide consistent performance improvements. This demonstrates the broad adaptability and effectiveness of the proposed approach.

6.3.6 Robustness. We evaluate the sensitivity of the RL model to training data by training it on different applications until it reaches a stable and optimal performance state. The resulting Q-table from this convergence point is then reused to run other workloads, and the resulting runtimes are compared against those obtained using workload-specific training, as shown in Figure 14. A higher value in the figure indicates greater performance degradation. Among the 25 (5×5) evaluated combinations, only 7 cases exhibit a performance drop exceeding 10%, suggesting that the initial state of the Q-table—determined by the training workload—can influence performance, but the model retains a reasonable degree of generalization across workloads.

Ideally, the Q-table should be trained on the target application or a workload with similar memory access patterns. To assess the retraining overhead when initializing with a suboptimal Q-table, we also measure the number of iterations required to reach at least 95% of the best achievable performance for each application. Across all tested workloads, convergence required between 1 and 6 iterations, with an average of 3, indicating moderate retraining costs under mismatched initialization.

6.3.7 Hyperparameter Sensitivity. Figure 15 shows the overall improvement achieved by ArtMem under six different memory tier ratios, evaluated across various RL and system parameters. Our

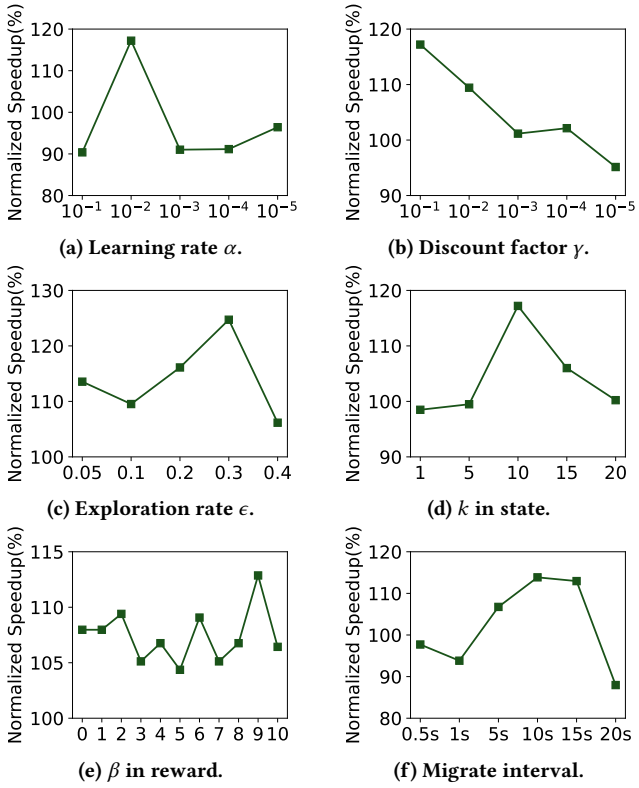
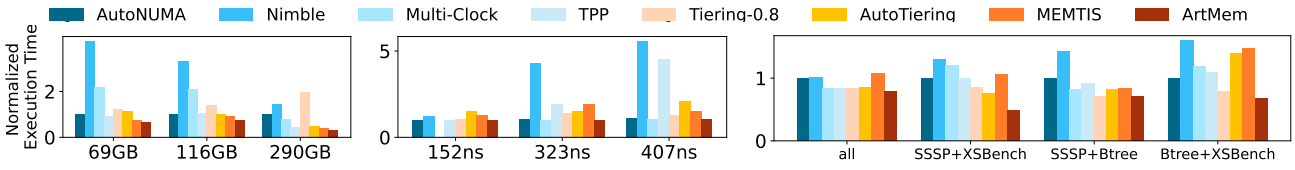


Figure 15: Sensitivity study for hyperparameters.

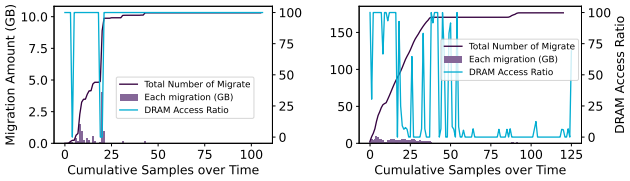
experiments achieved optimal performance with a learning rate of $\alpha = e^{-2}$, emphasizing the balance between past experiences and new knowledge. Figure 15b shows that the best discount factor was $\gamma = e^{-1}$, highlighting the importance of balancing immediate rewards and long-term gains. Figure 15c shows that both excessive and insufficient exploration lead to suboptimal learning, so we choose $\epsilon = 0.3$ for optimal performance. Figure 15e demonstrates that setting the reward parameter, β, between 8-10 can guide RL to maintain a high fast memory tier access ratio, which helps improve performance. Figure 15f shows that too long migration intervals lead to delayed page movement and performance degradation. Based on these findings, we configure the migration interval to 10 seconds, which falls within the empirically optimal range of 5–15 seconds.

6.3.8 Memory Size Scalability. ArtMem is evaluated by increasing the memory footprint of CC from 69GB to 290GB (the maximum size of the graph) by changing the input graph size. In all experiments, the size of the fast tier is fixed at 54GB. As shown in Figure 16a, with the increase in memory footprint, ArtMem's performance improves



(a) Scalability with memory sizes. **(b) Sensitivity to latency.** **(c) Adaptability for mixed workloads**

Figure 16: Performance comparison by (a). varying memory sizes, (b). varying relative latency, and (c). dynamic and complex access patterns.



(a) ArtMem. **(b) TPP.**

Figure 17: The variation in migration operations and DRAM access ratio across different systems when running a mixed workload.

by at least 6%. These results demonstrate that ArtMem can scale to larger memory sizes.

6.3.9 Sensitivity to Relative Latency. We use the remote socket DRAM, local PM, and remote PM as slow memory tiers (with their respective latencies shown in Figure 16b) to investigate ArtMem's sensitivity to slow memory latency. Multi-Clock does not support the remote socket DRAM node as the slow memory, resulting in a missing value. In all experiments, we control the available memory size of the local DRAM node to 32GB and run the same SSSP program. The running time of AutoNUMA, with a latency of 152ns for the capacity tier, is used as the baseline, and all data are normalized accordingly. Figure 16b shows that as the latency gap between memory tiers increases, the performance gap between the comparisons increases. ArtMem maintains stable performance, outperforms other baselines, and demonstrates strong adaptability.

6.3.10 Adaptability to Highly Irregular Workloads. We simulate a scenario with dynamic and complex access patterns by running multiple workloads concurrently. The three selected workloads come from different application domains. When all three workloads run together, the DRAM size is set to 64GB; when two workloads run together, the DRAM size is set to 32GB. Figure 16c shows the normalized results for the mixed runtime. Compared to the second-best method, ArtMem shows an average performance improvement of 11%. These advantages come from ArtMem's ability to classify pages accurately.

As shown in Figure 17a, ArtMem performs some exploratory migrations at the start. Later, the RL algorithm stabilizes. When the access ratio remains at 100%, the Q-Table is more likely to choose action = 0. This prevents unnecessary migrations. The mixed workload of SSSP and XSBench includes both frequent hot data accesses and many temporary cold page accesses. Traditional heuristic methods struggle with this pattern. They may mistakenly evict hot pages. In Figure 17b, TPP achieves a good access ratio after early migrations but keeps migrating pages, 17.5 times more than ArtMem. Later, when the access ratio drops, it fails to respond effectively.

6.4 Overheads

Sampling Overhead. The sampling period is initialized to 200, meaning hardware events are recorded once every 200 occurrences. We dynamically adjust the sampling period to control the sampling overhead. The sampling thread collects data every 2ms and updates the corresponding data structures and counters. We observed that the sampling overhead is at most 3% of the CPU in all benchmarks.
Q-Table Computation Overhead. Updating the Q-table involves calculating new Q-values based on observed rewards and the maximum future rewards. Given that the reward calculations we set are simple and that model-free reinforcement learning has inherently low computational overhead, we observed a maximum computation overhead of 0.07% of the CPU.
Q-Tables Memory Overhead. ArtMem includes two Q-tables. The Q-tables require memory to store state-action pairs and their corresponding Q-values. With our configuration, where actions and rewards are limited, the Q-table sizes are very small and do not require significant overhead. In our experiments, the Q-tables occupy less than 10KB of memory.

7 Conclusion

This paper introduces ArtMem, a tiered memory system leveraging reinforcement learning for adaptive memory management. By dynamically adjusting based on real-time fast memory usage, ArtMem optimizes fast memory utilization while reducing unnecessary migrations. The evaluation shows ArtMem outperforms state-of-the-art systems, achieving an average performance improvement of 114% with minimal overhead.

Acknowledgments

We sincerely thank the anonymous reviewers for their insightful comments and constructive feedback, which greatly helped improve the quality of this paper.

References

[1] 2020. Linux profiler perf. https://www.brendangregg.com/perf.html
[2] 2022. Utility library for managing the libnvdimm (non-volatile memory device) sub-system in the Linux kernel. https://github.com/pmem/ndctl
[3] 2023. Intel's Precise Event-Based Sampling. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/precise-events.html
[4] 2024. cgroups(7) — Linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html
[5] 2024. Intel® Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html

[6] Reto Achermann and Ashish Panwar. 2019. Mitosis workload BTree. https://github.com/mitosis-project/mitosis-workload-btree Jul 15, 2020.

[7] Reto Achermann and Ashish Panwar. 2024. Perfmon Metrics. https://github.com/intel/perfmon/

[8] Lada A. Adamic and Bernardo A. Huberman. 2002. Zipf's law and the Internet. *Glottometrics* 3 (2002), 143–150. https://api.semanticscholar.org/CorpusID:3289089

[9] Neha Agarwal and Thomas F Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 631–644.

[10] B.C. Arnold. 2015. *Pareto Distributions*. CRC Press. https://books.google.com/books?id=Ti_OBgAAQBAJ

[11] David A. Bader, Guojing Cong, and John Feo. 2005. On the Architectural Requirements for Efficient Execution of Graph Algorithms. In *Proceedings of the 2005 International Conference on Parallel Processing (ICPP '05)*. IEEE Computer Society, 547–556. https://doi.org/10.1109/ICPP.2005.55

[12] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[13] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. arXiv:1508.03619 [cs] http://arxiv.org/abs/1508.03619

[14] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. Association for Computing Machinery, 1121–1137.

[15] Jinyoung Choi, Sergey Blagodurov, and Hung-Wei Tseng. 2021. Dancing in the Dark: Profiling for Tiered Memory. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 13–22.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. Association for Computing Machinery, 143–154. https://doi.org/10.1145/1807128.1807152

[17] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[18] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. 2019. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence *(HPDC '19)*. Association for Computing Machinery, 37–48. https://doi.org/10.1145/3307681.3325398

[19] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 727–741.

[20] Pál Erdős and Alfréd Rényi. 1959. On Random Graphs I. *Publicationes Mathematicae* 4 (1959), 3286–3291.

[21] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux J.* 2004, 124 (aug 2004), 5.

[22] The Linux Foundation. 2022. Linux Kernel Version 5.15.19. https://www.kernel.org/

[23] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the idiosyncrasies of real persistent memory. *Proc. VLDB Endow.* 14, 4 (dec 2020), 626–639. https://doi.org/10.14778/3436905.3436921

[24] Udit Gupta, Xiaodong Wang, Maxim Naumov, Carole-Jean Wu, Brandon Reagen, David Brooks, Bradford Cottel, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Andrey Malevich, Dheevatsa Mudigere, Mikhail Smelyanskiy, Liang Xiong, and Xuan Zhang. 2019. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. *CoRR* abs/1906.03109 (2019). https://arxiv.org/abs/1906.03109

[25] Taekyung Heo, Yang Wang, Wei Cui, Jaehyuk Huh, and Lintao Zhang. 2022. Adaptive Page Migration Policy With Huge Pages in Tiered Memory Systems. *IEEE Trans. Comput.* 71, 1 (2022), 53–68. https://doi.org/10.1109/TC.2020.3036686

[26] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *2008 International Symposium on Computer Architecture*. 39–50. https://doi.org/10.1109/ISCA.2008.21

[27] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for {Multi-Tiered} Memory Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 715–728.

[28] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. Association for Computing Machinery, 591–600. https://doi.org/10.1145/1772690.1772751

[29] Soyoon Lee, Hyokyung Bahn, and Sam H Noh. 2013. CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures. *IEEE Trans. Comput.* 63, 9 (2013), 2187–2200.

[30] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 17–34.

[31] Chih-Jen Lin. 2021. LIBLINEAR– A Library for Large Linear Classification v2.43. https://www.csie.ntu.edu.tw/~cjlin/liblinear/

[32] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In *2008 International Symposium on Computer Architecture*. 453–464. https://doi.org/10.1109/ISCA.2008.15

[33] Xiaoyang Lu, Hamed Najafi, Jason Liu, and Xian-He Sun. 2024. CHROME: Concurrency-Aware Holistic Cache Management Framework with Online Reinforcement Learning. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1154–1167. https://doi.org/10.1109/HPCA57654.2024.00090

[34] Kamesh Madduri, David A. Bader, Jonathan W. Berry, and Joseph R. Crobak. 2007. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, 23–35.

[35] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. 2022. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems.. In *HPCA*. 925–937.

[36] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. 742–755.

[37] Ulrich Meyer and Peter Sanders. 1998. Delta-Stepping: A Parallel Single Source Shortest Path Algorithm. In *Proceedings of the 6th Annual European Symposium on Algorithms (ESA '98)*. Springer-Verlag, 393–404.

[38] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[39] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2010. The case for RAMClouds: scalable high-performance storage entirely in DRAM. 43, 4 (jan 2010), 92–105. https://doi.org/10.1145/1713254.1713276

[40] Yuqian Pan, Haichun Zhang, Runze Yu, Zhaojun Lu, Haoming Zhang, and Zhenglin Liu. 2023. LightWarner: Predicting Failure of 3D NAND Flash Memory Using Reinforcement Learning. *IEEE Trans. Comput.* 72, 3 (2023), 853–867. https://doi.org/10.1109/TC.2022.3184270

[41] SeongJae Park. 2022. masim: memory access workload simulator. https://github.com/sjp38/masim AuG 22, 2022.

[42] SeongJae Park. 2024. DAMON: Data Access Monitor. https://sjp38.github.io/post/damon/ Jun 14, 2024.

[43] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. 2022. DAOS: Data Access-aware Operating System. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22)*. Association for Computing Machinery, 4–15.

[44] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference Industrial Track (Middleware '19)*. Association for Computing Machinery, 1–7.

[45] Andy Patrizio. 2018. Facebook and Amazon are causing a memory shortage. https://www.networkworld.com/article/965006/facebook-and-amazon-are-causing-a-memory-shortage.html

[46] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 285–297. https://doi.org/10.1145/2749469.2749473

[47] Chris Petersen. 2021. Software Defined Memory: A Meta perspective. https://www.opencompute.org/events/past-events/2021-ocp-global-summit

[48] pmem.io. 2019. Using the memmap Kernel Option. https://docs.pmem.io/persistent-memory/getting-started-guide/creating-development-environments/linux-environments/linux-memmap.

[49] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*. Association for Computing Machinery, New York, NY, USA, 24–33. https://doi.org/10.1145/1555754.1555760

[50] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. Hemem: Scalable tiered memory management for big data applications and real

nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 392–407.

[51] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, 803–817.

[52] G. A. Rummery and M. Niranjan. 1994. On-Line Q-Learning Using Connectionist Systems. *Technical Report* (1994).

[53] Mohit Sewak. 2019. *Mathematical and Algorithmic Understanding of Reinforcement Learning*. Springer Singapore, Singapore, 19–27. https://doi.org/10.1007/978-981-13-8285-7_2

[54] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2023. vTMM: Tiered Memory Management for Virtual Machines. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery, 283–297. https://doi.org/10.1145/3552326.3587449

[55] Yossi Shiloach and Uzi Vishkin. 1982. An O(logn) parallel connectivity algorithm. *Journal of Algorithms* 3, 1 (1982), 57–67. https://doi.org/10.1016/0196-6774(82)90008-6

[56] R. S. Sutton and A. G. Barto. 2018. *Reinforcement Learning: An Introduction* (2 ed.). MIT Press, Cambridge, MA, USA. 129–131 pages.

[57] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XS-Bench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. Kyoto. https://www.mcs.anl.gov/papers/P5064-0114.pdf

[58] Rik van Riel and Vinod Chegu. 2014. Automatic NUMA balancing. https://events.static.linuxfound.org/sites/events/files/slides/summit2014_riel_chegu_w_0340_automatic_numa_balancing_0.pdf

[59] Vishal Verma. 2022. Tiering-0.8. https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/log/?h=tiering-0.8

[60] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, 609–621. https://doi.org/10.1145/3503222.3507731

[61] Chao Wu, Yufei Cui, Cheng Ji, Tei-Wei Kuo, and Chun Jason Xue. 2020. Pruning Deep Reinforcement Learning for Dual User Experience and Storage Lifetime Improvement on Mobile Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3993–4005. https://doi.org/10.1109/TCAD.2020.3012804

[62] Chao Wu, Cheng Ji, Qiao Li, Congming Gao, Riwei Pan, Chenchen Fu, Liang Shi, and Chun Jason Xue. 2020. Maximizing I/O Throughput and Minimizing Performance Variation via Reinforcement Learning Based I/O Merging for SSDs. *IEEE Trans. Comput.* 69, 1 (2020), 72–86. https://doi.org/10.1109/TC.2019.2938956

[63] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of intel optane persistent memory: a close look at its on-DIMM buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, 488–505. https://doi.org/10.1145/3492321.3519556

[64] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. 2024. FlexMem: Adaptive Page Profiling and Migration for Tiered Memory. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, Saurabh Bagchi and Yiying Zhang 0005 (Eds.). USENIX Association, 817–833. https://www.usenix.org/conference/atc24/presentation/xu-dong

[65] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 331–345.

[66] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. USENIX Association, 169–182.

[67] Pengmiao Zhang, Rajgopal Kannan, Ajitesh Srivastava, Anant V. Nori, and Viktor K. Prasanna. 2022. ReSemble: reinforced ensemble framework for data prefetching. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '22)*. IEEE Press, Article 81, 14 pages.