IDT: Intelligent Data Placement for Multi-tiered Main Memory with Reinforcement Learning

Juneseo Chang
Seoul National University
South Korea
jschang0215@snu.ac.kr

Wanju Doh
Seoul National University
South Korea
wj.doh@scale.snu.ac.kr

Yaebin Moon
Samsung Electronics
South Korea
yaebin.moon@samsung.com

Eojin Lee
Inha University
South Korea
ejlee@inha.ac.kr

Jung Ho Ahn
Seoul National University
South Korea
gajh@snu.ac.kr

ABSTRACT

To address the limitation of a DRAM-based single-tier in satisfying the comprehensive demands of main memory, multi-tiered memory systems are gaining widespread adoption. To support these systems, operating-system-level solutions that analyze the application's memory access patterns and ensure data placement in the appropriate memory tier have been vastly explored.

In this paper, we identify reinforcement learning (RL) as an effective solution for tiered memory management, and its policy can be formulated in a solvable form using RL. We also demonstrate that an effective region-granularity memory access monitoring method is necessary to provide an accurate environment state to the RL model. Thus, we propose **IDT**, an **i**ntelligent **d**ata placement for multi-**t**iered main memory. IDT incorporates an RL-based demotion policy autotuning and a mechanism that efficiently demotes cold pages to lower-tier memory. IDT also promotes hot pages to upper-tier memory to minimize access on slow memory, featuring a lightweight machine learning algorithm. IDT employs region-granularity memory access monitoring with statistical-testing-based adjacent region merge and split to improve precision and mitigate ambiguity observed in prior works. Experiments on an actual four-tiered memory system show that IDT achieves an average 2.08× speedup over the default Linux kernel and 11.2% performance improvement compared to the state-of-the-art solution.

CCS CONCEPTS

• **Software and its engineering → Memory management**; • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Computing methodologies → Reinforcement learning**.

KEYWORDS

Memory Tiering, Emerging Memory Technologies, Memory Management, Reinforcement Learning

ACM Reference Format:
Juneseo Chang, Wanju Doh, Yaebin Moon, Eojin Lee, and Jung Ho Ahn. 2024. IDT: Intelligent Data Placement for Multi-tiered Main Memory with Reinforcement Learning. In *The 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24), June 3–7, 2024, Pisa, Italy.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3625549.3658659

1 INTRODUCTION

The growing demand for memory-intensive workloads, such as high-performance computing, graph analytics, and in-memory databases, is highlighting the scaling limitations of a DRAM-based single-tier main memory [39]. To tackle this issue, a variety of memory types with diverse performance characteristics have been adopted to compose tiered memory systems. Recently, the rising interest in memories attached to Compute Express Link (CXL-Memory [9]) underscores that the future lies in multi-tiered memory systems by integrating various heterogeneous memories with a main-memory-like appearance to a system [36]. Cloud vendors, such as Amazon and Google, already serve large memory cloud instances based on multi-tiered memory systems [20, 33].

Tiered memory management requires a keen insight into an application's memory usage and placing the data in the proper memory tier according to its hotness. Thus, a number of prior studies have proposed operating-system-(OS)-level solutions to improve application performance by attentively exploiting the tiered memory system [2, 12, 15, 19, 23, 27, 36, 38, 48, 51, 55]. These OS-level tiered memory solutions typically consist of *data placement* to fully leverage diverse memory types and *memory access monitoring* to gather information for guiding data placement.

Data placement. Infrequently accessed pages in tiered memory should be *demoted* to lower-tier slow memory for efficient utilization of upper-tier fast memory. Moreover, to complement demotion, hot pages trapped in slow memory should be identified and *promoted* to upper-tier memory. Several prior studies have utilized the Linux kernel's 2Q LRU [19, 21, 35, 36, 56, 57] or multi-generational LRU (MGLRU) [58] to determine demotion candidates. However, the data hotness identified by 2Q LRU or MGLRU often fails to reflect the actual data hotness of the application. Therefore, precisely tracking both access frequency and recency for each page, and establishing a demotion policy with solid criteria would be more effective. Yet, implementing this method presents the challenge of adjusting the demotion criteria for the current system behavior. Moreover, in a multi-tiered memory system, the demotion criteria should be configured to meet the different access patterns in each memory tier (§2.2).

Memory access monitoring. To address the high overhead of *page-granularity* memory access monitoring, recent studies have focused on *region-granularity* monitoring that groups pages with similar access patterns [40, 45, 48]. They periodically merge or split adjacent regions to reconfigure the region distribution as the access patterns of memory objects continuously change. While existing approaches have been successfully employed in data access-aware memory management, their merging and splitting mechanisms lack logical justifications (§2.4).

Key idea. These motivate us to present **IDT** (**I**ntelligent **D**ata Placement for Multi-**t**iered Main Memory). IDT employs reinforcement learning (RL) to adjust demotion criteria for the current system behavior. IDT also provides a promotion mechanism to minimize access on slow memory, featuring a lightweight machine learning (ML) algorithm for predictive promotion. Furthermore, IDT proposes a statistical testing-based region reconfiguration technique to effectively merge and split adjacent regions with a logical basis.

Challenges. Prior works have employed ML to determine parameters for demotion or predict future access patterns of each page for data placement [12, 15, 27, 34]. However, due to the inherent overhead of ML models, they face excessive increases in memory usage and execution time or require substantial computing resources for training. IDT seeks to overcome these issues and demonstrate performance improvements in actual multi-tiered memory systems. We focus on implementing a system that outperforms existing state-of-the-art studies against proper evaluation metrics.

The major contributions of this paper are as follows:

- We identify the need for an RL-based demotion policy by analyzing prior demotion methods (§2.2, §2.3), and formulate demotion policy in a problem solvable with RL (§3). We also state the need for region-granularity memory access monitoring with statistical testing-based region reconfiguration (§2.4).
- We propose IDT, which employs (1) a region-granularity memory access monitoring (§4.1), (2) a *demotion* with RL-based policy autotuning (§4.2), (3) a *promotion* to minimize potential slow memory access with a lightweight ML algorithm (§4.3), and (4) a *region reconfiguration* that merges or splits adjacent regions by a statistical method (§4.4).
- We evaluate IDT on an actual four-tiered memory system with diverse benchmarks. We confirm that IDT achieves an average 2.08× speedup over the default Linux kernel, and 11.2% performance improvement over the state-of-the-art solution (§5.2). IDT's source code is available on Github.[1]

2 BACKGROUND AND MOTIVATION

2.1 OS Support for Tiered Memory Systems

A *tiered memory system* comprises main memory with memory nodes of distinct performance characteristics. Such systems have emerged due to the limitation of DRAM-based single-tier memory systems in satisfying the ever-growing capacity, bandwidth, and

[1]https://github.com/scale-snu/IDT

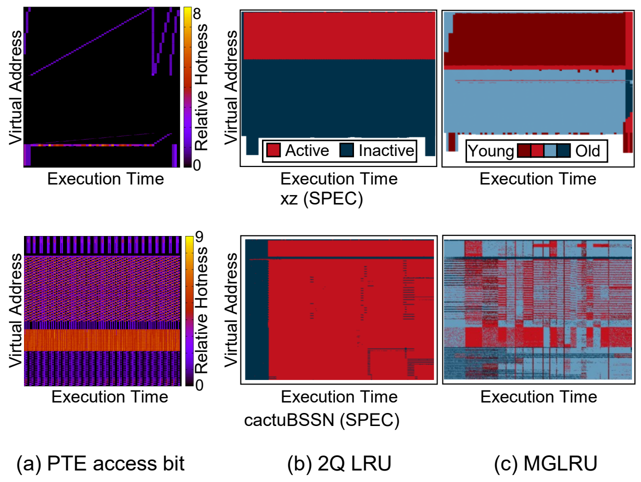


Figure 1: Comparison of data hotness derived from the Linux kernel's (a) PTE access bit, (b) 2Q LRU (in)active lists, and (c) MGLRU generations. The accurate data hotness was profiled using PTE access bit, set by hardware and cleared by the OS.

latency demands for main memory. The rising interest in CXL-Memory [9] highlights the industry's recognition of these challenges. It facilitates the integration of tiered memory and allows a system to recognize them similarly to CPU-less NUMA nodes [36]. Additionally, High-Bandwidth Memory (HBM)-enabled processors are becoming available [41], offering an additional tier in the memory hierarchy. Further, multi-socket NUMA architecture is now the standard for modern server systems [48]. Thus, the future lies in *multi-tiered* memory systems with multiple memory nodes of different performance characteristics.

Among various approaches to manage tiered memory, we focus on *OS-level solutions*, where the OS kernel is responsible for data placement across various types of memory. This approach offers the advantage of reflecting the system's runtime behavior without requiring hardware or application modifications. They *demote* seldomly accessed pages to a lower-tier memory for efficient use of deficient fast memory and *promote* hot pages trapped in lower-tier memory. They also track the memory access patterns of individual workloads and place pages in proper memory tiers to maximize workload performance.

However, a majority of prior OS-level solutions only support memory systems with two tiers and lack supporting more memory tiers [19, 21, 28, 35, 36, 56]. Extending them to support multi-tiered memory is not straightforward, as some solutions lack support for migration across multiple memory tiers. Therefore, multi-tiered solutions have been proposed to leverage the entire memory system [23, 48]. However, they often miss the faster memory tier in the fallback path of demotion and show subpar performance compared to two-tiered solutions due to the increased complexity of managing multiple memory tiers.

2.2 Challenges in Designing a Demotion Policy

An effective demotion scheme is especially crucial because the scarce fast-memory capacity limits fast memory usage and the operation of the promotion scheme. Further, incorrectly identifying

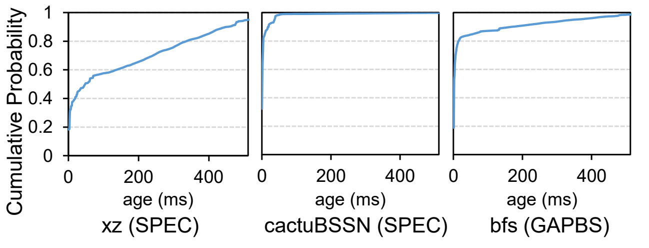


Figure 2: Cumulative probability distribution of accessed pages' recency to their last access (*age*).

demotion targets causes ping-pong of demotion and promotion. Therefore, an effective demotion scheme should accurately identify data hotness and establish proper criteria for determining the demotion candidates.

Utilizing the Linux kernel's LRU mechanism to determine demotion candidates has limitations in accurately identifying data hotness. Many prior works [19, 21, 35, 36, 56, 57] have utilized the Linux kernel's 2Q LRU for their demotion mechanisms. Also, the Linux kernel community recently introduced MGLRU [58], which maintains multiple generations of LRU lists for a more fine-grained page eviction policy. However, the data hotness discerned from 2Q LRU and MGLRU often deviates from the actual data hotness, as shown in Figure 1, motivating a more accurate methodology for accurately identifying data hotness.

A demotion strategy tracking each page's access frequency (*freq*) and recency (*age*) to determine demotion candidates enables a more accurate data hotness identification. Meanwhile, establishing appropriate criteria to trigger demotion is challenging. Prior works have chosen pages with small *freq* and large *age* as demotion candidates [19, 21, 27, 28, 35, 36, 54, 56, 57]. While the criterion of small *freq* is apparent due to the prevalence of cold pages in memory-intensive workloads [27, 36], the degree of a large *age* requires a more precise standard. Figure 2 shows the cumulative probability distribution of accessed pages' *age* in different workloads. The convex distributions confirm higher access probabilities for pages with lower *age*. Thus, an effective demotion strategy would demote pages with *age* above a threshold (`age_thres`) to keep frequently accessed pages in the fast memory. Meanwhile, the distinct steepness motivates adapting `age_thres` to the current system behavior.

Furthermore, a multi-tiered memory system should determine proper `age_thres` for each memory tier. Because the characteristics of pages residing in each memory tier are distinct, applying unified demotion criteria across all memory tiers is not suitable. This motivates the need for a system that autonomously adjusts the demotion criteria online for each memory tier.

2.3 ML in Tiered Memory Management

Adjusting a policy based on current information is a task at which ML excels [11, 37]. Because the OS lacks prior knowledge of each workload's memory access patterns, it must infer appropriate demotion policy from runtime information. ML can handle this by adapting the policy towards the optimization objectives, such as minimizing the execution time. For example, an ML model can be trained to determine the appropriate `age_thres` to minimize performance penalty from slow memory accesses.

Prior studies have employed ML for tiered memory management. Kleio [12] utilizes a per-page online training LSTM model that predicts the future access pattern for each page. It then allocates a page to an appropriate memory tier according to its predicted future coldness. While Kleio demonstrates the potential of applying ML to tiered memory management, it has substantial per-page training time (2 hours) and memory usage (tens of GBs). Coeus [13] improves Kleio's overhead up to 3× by training the LSTM model on a group of pages sharing the same access pattern. However, Coeus still exhibits substantial overhead over non-ML solutions, as it needs to train and infer numerous models for page placement. Another study that has been employed in Google's warehouse-scale computing (WSC) [27] optimizes demotion parameters using a Gaussian Process Bandit model based on WSC's memory traces. However, searching for the optimal value reportedly takes a day, making it feasible only at the warehouse scale.

The common limitations of these ML-based approaches are extensive training times for accurate predictions and maintaining large training datasets for supervised learning. To address this, we explore the use of RL, which has been successfully adapted to data placement policy in hybrid storage systems [32, 43, 50]. RL is an ML algorithm where an agent learns by interacting with the environment [52]. At each time step, the agent observes the environment state and selects an action based on its policy. After applying the action, the agent receives a reward that serves as feedback to the policy. RL operates in a lightweight manner that's highly analogous to system management strategies. Furthermore, RL facilitates online training and adaptability compared to other ML algorithms, thereby allowing the system to respond to dynamic system behavior.

Takeaway 1: RL can effectively provide demotion criteria by adjusting it to the current system behavior.

RL also offers extensibility, empowering it to support multiple memory tiers efficiently. By running inference of an RL model for each memory tier, the system can adaptively adjust demotion criteria for each memory tier at runtime.

Takeaway 2: RL's extensibility can empower dynamic adjustment of demotion criteria for each memory tier.

To apply RL, obtaining the current system environment state is imperative. However, capturing the state by page-granularity monitoring would result in excessive overhead and RL model complexity. Moreover, it can lead to a proportional increase in monitoring overhead as the number of pages grows. Thus, it is particularly unsuitable for memory-intensive workloads. This underscores the need for an effective memory access monitoring method to gather current environment information with low overhead.

2.4 Region-granularity Memory Access Monitoring

To address the limitations of page-granularity monitoring, region-granularity monitoring, which monitors groups of pages with similar characteristics, has been recently gaining popularity. DAMON [44], the Linux kernel data access monitoring framework, demonstrates that its region-granularity monitoring scheme accurately discerns workload access patterns. Furthermore, MTM [48] shows that the multi-tiered memory can be effectively managed using information from region-granularity monitoring.

Region-granularity monitoring involves merging and splitting adjacent regions to group pages with analogous access patterns. Considering that the access patterns of memory objects may change as the workload executes, region merge and split have to operate periodically. Thus, the key point of region-granularity monitoring lies in the proficiency of merge and split operations.

Despite the importance of region merge and split, prior works lack foundation in their method. For instance, DAMON merges adjacent regions if their access frequency difference is less than 10% of the maximum *freq* across all regions. Additionally, DAMON periodically splits each region into random sizes. MTM merges adjacent regions if their access counts differ by less than one-third of the scan count. For split, it employs two sampling pages from a region and splits the region when their access count difference exceeds two-thirds of the scan count. While the mechanism and thresholds used for merging and splitting are undoubtedly crucial, they lack a logical basis for their values and mechanism.

To address these limitations in prior works, we revisit the essence of region-granularity monitoring. Region-granularity monitoring assumes that similar pages are grouped into the same region. Under this assumption, the similarity between adjacent regions is assessed based on the sampling pages. Thus, two key aspects are considered as the core of region management: (1) the region groups similar pages together, and (2) the sampling pages' information determines the identicality of adjacent regions.

For the first aspect, a straightforward approach involves sampling multiple pages within a region and comparing their status. For the second aspect, this is a classic statistical testing problem of inferring population similarity based on samples. In other words, determining region merge can be effectively addressed with a statistical foundation.

Takeaway 3: Statistical testing can introduce a foundation in region-granularity monitoring methods.

3 RL FORMULATION

We formulate the demotion policy in a proper form to apply RL, using its mathematical framework. Then, we propose an approximation of the formulated problem to ensure its practicality for implementation in an actual system.

3.1 Problem Formulation

An effective demotion policy aims to maximize workload performance with a proper degree of fast memory reservation. Demoting pages that will be infrequently accessed in the future ensures minimal access to the lower-tier memory. Moreover, this can prevent the unnecessary presence of cold pages in fast memory, reserving fast memory for future allocation requests. Thus, we can express the optimization goal for deriving optimal demotion policy π^* as:

$$\pi^* = \arg\max_{\pi} \frac{1}{T} \sum_{t=1}^{T} \left((1-\beta) Perf(t) + \frac{\beta}{Mem(t)} \right) \tag{1}$$

where $Perf(t)$ and $Mem(t)$ represent the performance and fast memory usage at time t. The parameter β signifies the balance between performance and fast memory reservation. A small β value indicates an objective to maximize the application's performance [23, 36, 48], whereas a large one represents prioritizing fast memory reservation to reduce the total cost of ownership [15, 27, 38].

Through the established mathematical framework of RL [52], we can formulate our goal into a solvable form using RL. The optimization goal in Equation 1 can be converted to Markov Decision Process (MDP) $M = (S, A, P, R)$. In this context, the demotion policy π maps the system state S to demotion action A. The transition probability $P(s'|s,a)$ represents the probability of transitioning to state s' given s and action a. The immediate reward when transitioning from s to s' by taking a is $R(s,a,s') = (1-\beta)Perf(t) + \beta/Mem(t)$.

From MDP, the value function $V^{\pi^*}(s)$, representing the maximum cumulative reward starting from s and following optimal policy π^*, can be denoted by the Bellman equation:

$$V^{\pi^*}(s) = \arg\max_{a\in A} \left[\sum_{s'\in S} P(s'|s,a) \left(R(s,a,s') + \gamma V^{\pi^*}(s') \right) \right] \tag{2}$$

Thus, the optimization goal is to find π^* that satisfies Equation 2. However, solving Equation 2 to find π^* is computationally expensive. Hence, by representing the value function $V^{\pi}(s)$ with the Q-function $Q^{\pi}(s,a)$, which denotes the cumulative reward when choosing a from s and following π, π^* can be expressed as:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi} \left[Q^{\pi}(s,a) \right]$$

When parameterizing π against θ and defining the objective function $J(\theta) = \mathbb{E}_{\pi_\theta}\left[Q^{\pi_\theta}(s,a)\right]$, π^* can be found using the Policy Gradient (PG) algorithm:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)]$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

However, PG can sometimes lead to large policy updates that make learning θ unstable. To address this, Proximal Policy Optimization (PPO) algorithm constrains the policy updates, ensuring that the updated policy does not deviate significantly from the previous one [49]. In summary, a proper demotion policy can be derived using RL, based on its mathematical framework.

3.2 Approximating Goal, State, and Action

3.2.1 Optimization Goal. Our primary objective is to maximize performance improvement by fully leveraging tiered memory. However, we should not overlook the significance of reserving fast memory capacity. Over-utilizing fast memory can lead to performance degradation since every subsequent allocation would necessitate demotion to free up available space in deficient fast memory. Therefore, we intend to proactively initiate demotion when the available space in the fast memory goes below `demote_wmark` and set its value higher than the watermark of prior studies [19, 21, 23, 35, 36, 47, 48, 57]. Thus, `demote_wmark` enables us to omit the term related to $Mem(t)$ in Equation 1:

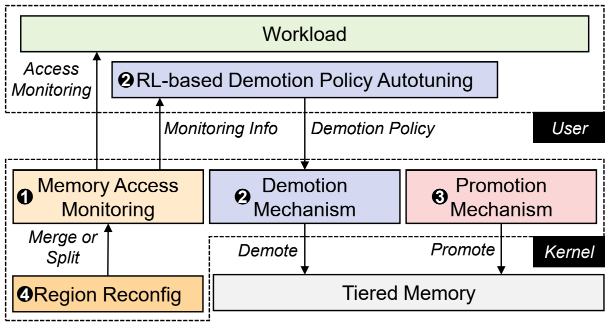


Figure 3: The overall architecture of IDT.

$$\pi^* = \arg\max_{\pi} \frac{1}{T} \sum_{t=1}^{T} Perf(t)$$

To capture $Perf(t)$ more pragmatically, we formulate the performance in terms of memory access. $Perf(t)$ is inversely proportional to the lower-tier memory access frequency (`slow_hit`) due to performance degradation when accessing slow memory. However, denoting $Perf(t)$ simply as `1/slow_hit(t)` would lead π^* to avoid any page demotion to minimize `slow_hit`. Meanwhile, we have demonstrated that reserving fast memory is crucial for future memory allocation requests. Thus, accounting reserved fast memory in $Perf(t)$ can lead π^* to minimize `slow_hit` while appropriately demoting pages.

3.2.2 State. The RL model's state should be a simplified representation that accurately reflects the current environment information. The readily available information is *freq* and *age* for each region. Given that a significant portion (30-40%) of pages allocated by data center workloads are not accessed for dozens of seconds [27, 36], we can only consider pages with minimum access frequency as demotion candidates. Thus, restricting our interest to pages with minimum *freq*, the state can be simplified as a function of *age*.

3.2.3 Action. As discussed in §2.2, prior works showed the effectiveness of demoting the least recently used cold pages. We also stated that the goal of our demotion policy is to provide a clear *age* threshold, `age_thres`. This allows us to simplify the action in RL by determining the appropriate `age_thres` value at each time point.

4 IDT: DESIGN

Based on these motivations, we design **IDT**, **I**ntelligent **D**ata placement for multi-**T**iered main memory. IDT consists of four components, as shown in Figure 3:

(1) *Memory Access Monitoring* gathers region-granularity memory access information.
(2) *Demotion* policy and mechanism provide RL-based `age_thres` autotuning and effective support for multi-tiered memory.
(3) *Promotion* mechanism promotes hot pages in slow memory to minimize slow memory access.
(4) *Region Reconfiguration* merges and splits adjacent regions based on statistical methods.

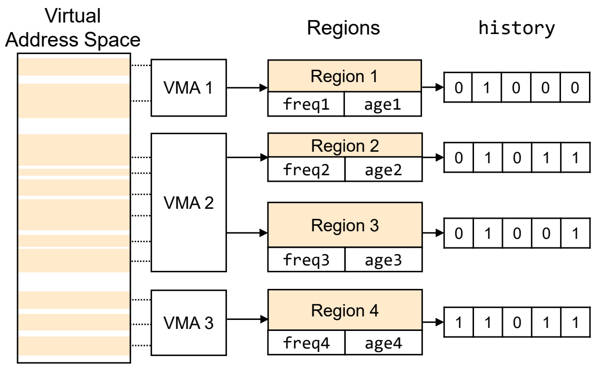


Figure 4: IDT's memory access monitoring and associated `freq`, `age`, and `history` variables for each region.

4.1 Memory Access Monitoring

4.1.1 Region-granularity Monitoring. IDT partitions the Virtual Memory Area (VMA) into multiple regions. VMA represents a contiguous range of virtual memory that shares common attributes. Each region is sampled for access at every `sample_interval`. Then, the regions are dynamically reconfigured by merging or splitting at every `aggregate_interval`. The region reconfiguration method is further explained in §4.4. The `sample_interval` and `aggregate_interval` are set to 10ms and 1,000ms. The sensitivity study for the interval values is provided in §5.8.

4.1.2 Page Sampling. IDT maintains `freq`, `age`, and `history` variables for each region, as shown in Figure 4. Access for each region is tracked utilizing the PTE access bit. IDT designates one page from the first half and another page from the remaining half of the region as *sampling pages*. The `freq` of the region is incremented in each `sample_interval` if any of the sampling pages has its PTE access bit set. The `age` is managed using DAMON's aging algorithm [45] by utilizing `freq` and region size changes. The `history` is a 1024-bit vector that records region access status at each `sample_interval`.

For a region split, the new region inherits the original values. For a region merge, the new region's variables are set based on the size-weighted average of the merged regions.

IDT also records `demoted_pages` and `slow_hit` for each memory node. `demoted_pages` represents the total number of demoted pages from the memory node. `slow_hit` represents the aggregate size (in a unit of page) of regions that were demoted from the memory node but reaccessed in the lower tier memory.

4.2 Demotion

To address **Takeaway 1** and **Takeaway 2**, IDT employs an RL-based policy autotuning in the user space and a demotion mechanism within the OS kernel.

4.2.1 RL-based Demotion Policy. IDT's RL model operates as shown in Figure 5: The state is derived from the moving average of the aggregated age distribution's `min`, `q1` (25th percentile), `q2` (50th percentile), `q3` (75th percentile), and `max` values. Further, we apply principal component analysis (PCA) to remove extreme outliers when calculating the moving average ❶. IDT then feeds

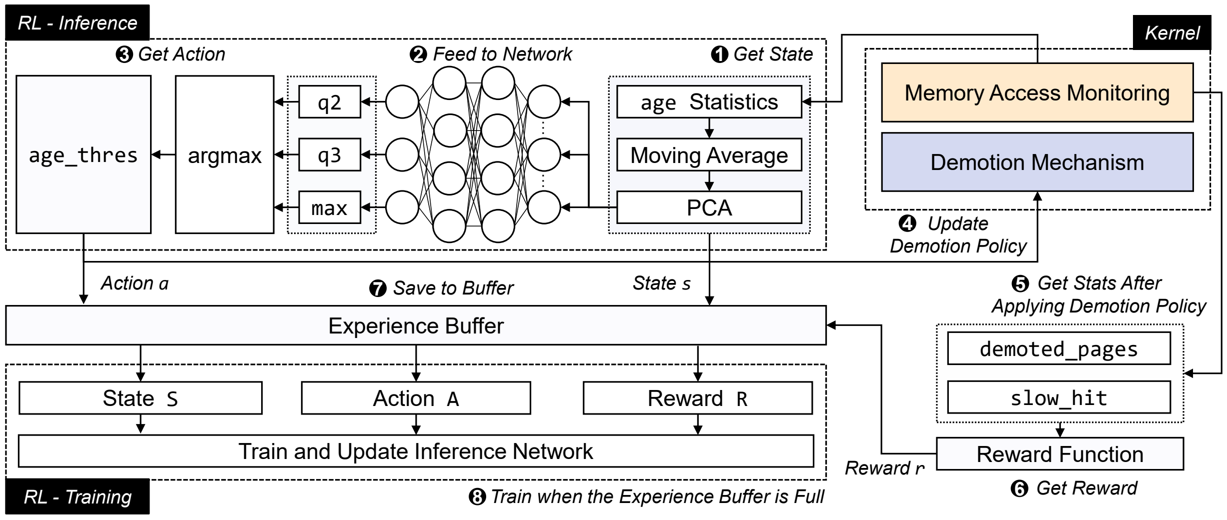


Figure 5: The overall architecture of IDT's RL model for optimizing age_thres.

the gathered state to the inference network ❷ and determines the new age_thres value from the inference network output ❸. Using the new age_thres, IDT updates the demotion policy ❹. After applying age_thres, the system waits for an action_interval and retrieves the number of demoted_pages and slow_hit during the interval ❺. The reward is then calculated as follows ❻.

$$reward \triangleq log\left(\frac{\texttt{demoted\_pages} + \epsilon}{\texttt{slow\_hit} + \epsilon}\right) \tag{3}$$

The state, action, and reward are stored in the experience buffer ❼. When it is full, IDT trains and updates the inference network ❽.

Detailed design. The input layer of the RL model consists of five nodes representing each moving average of age statistics. To access region statistics of each memory tier, we developed a kernel driver to expose them to the /proc filesystem. IDT utilizes the moving average of five representative values to encapsulate the essential characteristics of the age distribution into a lower dimension.

The inference network, which is for the parameterization of demotion policy (i.e., θ in §3.1), comprises two fully connected layers with 16 and 32 nodes. The output layer consists of three nodes, each representing the q2, q3, and max age statistics values of the regions with appropriate freq criteria. The age statistics value, signified by the node with the maximum value in the output layer, is chosen as the new age_thres. That is, the RL model chooses the age_thres among the age distribution's q2, q3, and max values to control the aggressiveness of demotion.

The reward function defined in Equation 3 guides the RL model toward an action that reserves fast memory while minimizing performance degradation. Given that demoted_pages and slow_hit are readily available from §4.1, Equation 3 is a feasible reward function. ϵ prevents the equation from diverging when the numerator and denominator are of small value.

The network is trained using the PPO algorithm [49]. For hyperparameters, we set the discount factor to 0.9, the learning rate to 0.01, and the exploration rate to 0.05. The batch size was aligned to the experience buffer size. We conducted the sensitivity study for RL hyperparameters in §5.3. The RL model was implemented using RLlib framework [29] based on PyTorch [46].

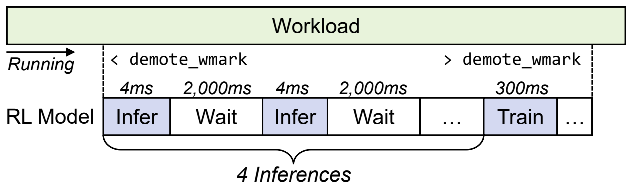


Figure 6: The execution phase of IDT's RL model.

Execution phase. We have demonstrated in §2.3 that prior studies on ML-based tiered memory management exhibit high overhead due to training time and memory usage of their ML models. To address this, IDT presents a novel execution phase (see Figure 6) that significantly mitigates the RL model's overhead. The RL model is triggered when the remaining space of a memory node reaches demote_wmark. Following inference, it undergoes an action_interval delay, contributing to a further overhead reduction. On a single CPU core, the inference model takes 2-4ms, and the training, which occurs every four inferences, takes 200-300ms. We set the action_interval=2s and experience buffer size to 4 to ensure that the RL's maximum theoretical overhead is below 5%. Given these values, the maximum overhead of RL can be calculated as $(4ms \times 4 + 300ms)/(2s \times 4) = 3.95\%$ of a single core.

Furthermore, we design the RL model to utilize the lowest-tier memory to minimize the proportion of memory used by RL in the overall system. We provide a detailed analysis of the RL model's overhead and memory usage in §5.4.

Multi-tier support. As described in §2.2, each memory tier should manage its own age_thres. Thus, individual inferences are executed for each memory tier. Meanwhile, training is done by gathering all experiences from each memory tier. The trained network is shared by all inferences for each memory tier.

Transfer learning. Transfer learning can yield the effect of fine-tuning for online learning models, for which the RL model is

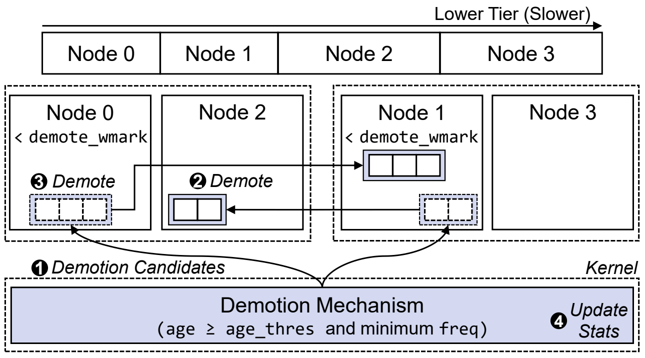


Figure 7: Demotion mechanism in IDT.

pre-trained using the giga update operations per second (GUPS) microbenchmark [18]. The GUPS microbenchmark used for pre-training encompasses three distinct access patterns that were also used for evaluation in [47]: *Uniform random access*, *Hot set* and *Dynamic hot set*. *Uniform random access* performs random access over the working set. *Hot set* performs 90% of access on hot objects and the remaining uniform randomly. *Dynamic hot set* changes consecutive 4GB hot objects every 150-second intervals. The GUPS microbenchmark was configured to use a single CPU core with a working set size of 100GB.

4.2.2 *Demotion Mechanism.* Figure 7 shows the overall demotion mechanism implemented in the OS kernel. Demotion is triggered when the memory node's available space falls below `demote_wmark`. A higher `demote_wmark` value encourages a proactive demotion for fast memory preservation, whereas a smaller value maximizes the fast memory utilization to enhance performance.

IDT identifies regions with the *minimum* `freq` *and* `age` *larger than* `age_thres` as demotion candidates ❶. Starting from the lowest memory tier with remaining space below `demote_wmark`, IDT migrates the demotion candidates to the next lower-tier memory by `migrate_pages()` [30] of the Linux kernel ❷. The `demoted` flag of a region is set following a successful demotion. Subsequently, IDT repeats the demotion across higher tiers ❸. Upon completion, the `demoted_pages` variable is updated in each memory node to reflect the total number of demoted pages ❹.

Aggressive demotion. When a workload's memory usage intensifies to severely depleting memory space, demotion must be more aggressive. Thus, IDT tightens the demotion candidates criteria when a memory node's remaining space falls below `critical_wmark`. To tighten the demotion criteria, the `freq` criterion is adjusted to the *average of the minimum and maximum* `freq` values across all regions. Further, when the demotion candidates are actively used by an application during page migration, demotion via `migrate_pages()` fails. In such a case, IDT attempts to demote all regions if no pages are demoted during aggressive demotion. In the cases of extremely deficient fast memory, `kswapd` is activated when the available space drops below the Linux kernel's watermark (0.1%).

In summary, IDT's demotion works as Algorithm 1. We set `demote_wmark` and `critical_wmark` to 10% and 1%, respectively. The generous watermark values can supplement the potential slow

Algorithm 1 Demotion in memory node *nid*

```
1:  $max\_freq \leftarrow max\{\texttt{freq}(r) \mid r \in regions(nid)\}$
2:  $min\_freq \leftarrow min\{\texttt{freq}(r) \mid r \in regions(nid)\}$
3:  if capacity($nid$) < critical_wmark then
4:      $freq\_thres \leftarrow (min\_freq + max\_freq)/2$
5:      Demote regions with:
            age ≥ age_thres and freq ≤ $freq\_thres$
6:      if no demoted pages then
7:          Try to demote all regions
8:      end if
9:  else if capacity($nid$) < demote_wmark then
10:     Demote regions with:
            age ≥ age_thres and freq = $min\_freq$
11: end if
```

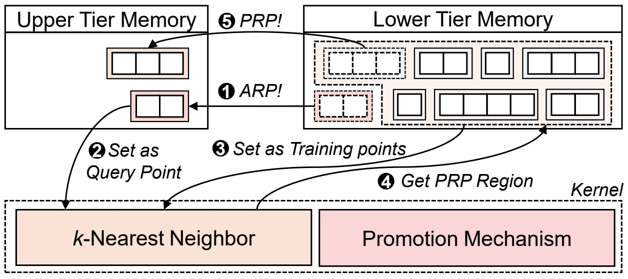


Figure 8: Predictive Region Promotion (PRP) for predictive promotion using the *k*-Nearest Neighbor algorithm.

migration speed of `migrate_pages()`. The sensitivity study for the watermark values is conducted in §5.8.

4.3 Promotion

Hot regions trapped in the lower-tier memory should be promoted to minimize performance degradation due to slow memory access. To address this, IDT incorporates a promotion mechanism consisting of three components: (1) *Accessed Region Promotion (ARP)* for promoting regions upon access, (2) *Predictive Region Promotion (PRP)* for predictive promotion, and (3) *Misplaced Region Promotion* for relocating regions in a suboptimal memory tier.

4.3.1 *Accessed Region Promotion.* ARP is triggered when IDT's memory access monitoring detects access to a region in a lower-tier memory. This immediate response is motivated by the principle of temporal locality, indicating that recently accessed regions are likely to be needed again soon. Then, the uppermost memory tier is set to the destination node for promotion. However, if the remaining space in the uppermost memory tier falls below `critcal_wmark`, IDT promotes the region to the next highest tier. This prevents IDT from thrashing when the fast memory's available space is deficient. Once the destination node is determined, all pages within the region are migrated. Upon completion, the `slow_hit` variable is updated in each memory tier to reflect the total number of promoted pages.

4.3.2 *Predictive Region Promotion.* ARP's limitation is that it does not initiate promotion until access to the region's sampling pages is observed. If regions that are *similar* to the ones being promoted by ARP are preemptively promoted, we can reduce slow-memory access that is undetected by our region-granularity monitoring method. Thus, IDT provides predictive region promotion (PRP) by proactively promoting *similar* regions to the one that triggered ARP (see Figure 8).

PRP is initiated immediately when ARP is triggered ❶. To identify *similar* regions, we employ k-Nearest Neighbor (k-NN), a simple ML algorithm that finds the k closest training points for a given set of query points [26]. In the context of PRP, IDT sets the query point to the ARP region ❷, and the training points to regions in the memory tier where the region promoted with ARP was originally at ❸. Then, IDT selects a region with a minimum distance to the query point by k-NN. If the selected region has a distance less or equal to the so-far minimum distance, it is selected as a PRP region ❹. We formulate the distance function as the sum of the normalized virtual address distance and the access history distance. For regions r_1 and r_2, in which r_1's end address is lower than r_2's start address, the distance $d(r_1, r_2)$ is defined as:

$$\text{Let } \ d_{\text{VA}}(r_1, r_2) \triangleq r_2.\texttt{start\_addr} - r_1.\texttt{end\_addr}$$

$$\text{Let } \ d_{\text{H}}(r_1, r_2) \triangleq \texttt{Hamming}(r_1.\texttt{history}, r_2.\texttt{history})$$

$$d(r_1, r_2) \triangleq \frac{d_{\text{VA}}(r_1, r_2) - \overline{d_{\text{VA}}}}{\sigma_{d_{\text{VA}}}} + \frac{d_{\text{H}}(r_1, r_2) - \overline{d_{\text{H}}}}{\sigma_{d_{\text{H}}}} \tag{4}$$

where each term in Equation 4 represents spatial and temporal locality. Finally, the PRP region is promoted following ARP's 3, 4, and 5 steps ❺.

4.3.3 Misplaced Regions Promotion. When IDT 's promotion mechanism places a region in a suboptimal tier due to the full uppermost memory tier, a fair re-promotion is needed if there's available space in the uppermost memory tier. Moreover, when a workload suddenly allocates more pages than the available space in the fast memory, the Linux kernel's `kswapd` may demote pages to a lower tier memory before IDT can handle them. We refer to these regions as *misplaced regions*. Misplaced regions are identified during the page sampling, as regions residing outside the uppermost memory tier without a `demoted` flag set. As soon as a misplaced region is identified, IDT promotes it to the highest memory tier with remaining space above `critcal_wmark`. For the workloads used in our evaluation, this promotion component corresponds to an average of 9.71% of the total promotion.

4.4 Region Reconfiguration

4.4.1 Region Merge. To address **Takeaway 3**, IDT employs a statistical technique to estimate whether adjacent regions can be classified into the same category. We define the *similarity* as *regions having similar access frequencies at each time interval.*

IDT applies a sliding window of size n over the `history` vector to verify the similarity of the access ratio. Within this window, IDT checks the access counts for two regions, a_1 and a_2. Then, IDT statistically validates the similarity of adjacent regions by Fisher's exact test. The null hypothesis of the test is that the regions have different access ratios:

$$\text{pvalue} = \binom{n}{a_1} \cdot \binom{n}{a_2} \Big/ \binom{2n}{a_1 + a_2} \tag{5}$$

If the test result rejects the null hypothesis at a 90% significance level across every window on the `history` vector, we consider the two regions to be similar and merge them. We choose a relatively low significance level from commonly used values. This allows us to merge windows with roughly similar distributions and later verify if regions are appropriately grouped in the splitting process.

The sliding window enables IDT to assess whether two regions have similar access patterns over each temporal interval. We also utilize Fisher's exact test for its simplicity of implementation in the kernel space, efficient computation, and no prior assumption for the population to follow a normal distribution. The sliding window's stride was set to one, while the actual computational overhead was kept low due to repeated values of a_1 and a_2. We select $n = 16$ to avoid complex overflows that arise in the calculation of Equation 5.

4.4.2 Region Split. In contrast, when pages within a region exhibit different access patterns, the region should be split. If a region grows excessively large, the potential penalty for an improper demotion increases. To address these issues, IDT divides a region into two halves of equal size when the access status of the sampling pages differ at `sample_interval`. This strategy is based on the principle that a region should group pages with *similar* access patterns.

5 EVALUATION

5.1 Experimental Setup

System configuration. We developed IDT on top of the Linux kernel v6.0.19. IDT's memory access monitoring was implemented based on DAMON [44]. To build multi-tiered memory systems with publicly available systems, we configured the evaluation system with a four-tiered memory hierarchy based on the latency of each memory [48], as shown in Figure 10. The evaluation system features two 24-core Intel Xeon Platinum 8260 processors with 32GB DDR4 DRAM (fast memory) and 256GB Intel Optane DCPMM (slow memory) for each socket. The DCPMMs were configured to App-direct mode to expose them as kernel NUMA nodes.

Methodology. We evaluated IDT against Graph500 [4], SPECspeed 2017 (SPEC [8]), and GAP Benchmark Suite (GAPBS [5]) with the twitter graph. From SPEC, we selected benchmarks with large resident set size (RSS): `deepsjeng`, `imagick`, `bwaves`, `xz`, `roms`, and `cactuBSSN`. For GAPBS, we utilized Breadth-First Search (`bfs`) and PageRank (`pr`). We bound processes to the CPU cores of socket 0 using the `numactl -c0` option so that our system is configured with four memory tiers. We configured benchmarks to have RSS between 96GB and 110GB, which facilitates utilizing three memory tiers, two fast and one slow memory. The RSS was set by tuning graph properties in Graph500 and running multiple copies of benchmarks for SPEC and GAPBS.

We compared IDT with four state-of-the-art OS-level tiered memory management solutions: *Intel Tiering* 0.8 [21], *TPP* [36], *AutoTiering* OPMX [23], and *AutoNUMA Tiering (MGLRU)* [57, 58]. *Intel Tiering* and *TPP* are designed for two-tiered memory, whereas *AutoTiering* and *AutoNUMA Tiering (MGLRU)* support multi-tiered memory. We configured *AutoNUMA Tiering (MGLRU)* by setting the demotion path the same as IDT, and enabling MGLRU in the Linux kernel v6.1.0. Given the substantial overhead associated with prior

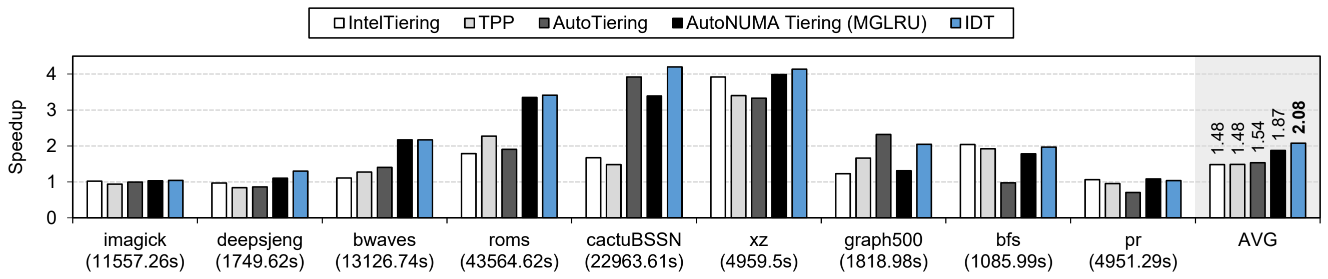


Figure 9: Speedup (higher is better) normalized to the baseline, AutoNUMA Balancing. The numbers below each benchmark indicate the raw execution time of the baseline.

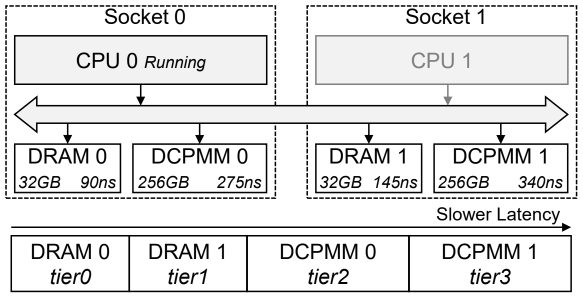


Figure 10: Multi-tiered memory system with four memory tiers used for evaluation.

ML-based solutions [12, 13, 27], we decided that directly comparing their performance with IDT and state-of-the-art solutions was not suitable. To ensure a precise experiment, we disabled Hyper-Threading, DVFS, Intel Turbo-boost, and prefetching during evaluation. We used the reciprocal of execution time as the performance metric, and the experimental results were normalized against the default Linux scheme, *AutoNUMA Balancing* [16] in the vanilla Linux kernel v6.0.19.

5.2 Performance of IDT

IDT improves performance on each benchmark through its adaptability to the current system behavior and effective use of multi-tiered memory. Figure 9 shows the speedup of IDT and its counterparts, normalized to *AutoNUMA Balancing*. IDT achieves an average speedup of **2.08**×, which is better than 1.48× of *Intel Tiering* and *TPP*, 1.56× of *AutoTiering*, and 1.87× of *AutoNUMA Tiering (MGLRU)*. That is, IDT outperforms the best-performing state-of-the-art solution by 11.2%.

IDT shows substantial speedups in `bwaves`, `roms`, and `cactuBSSN`, which exhibit clear distinction of hot/cold regions and high memory access locality. IDT benefits from these workloads by adapting the demotion policy through the RL model according to the workload's regular access patterns. Further, high locality benefits PRP by effectively promoting regions that would be accessed in the future (§5.6).

For `graph500`, *AutoTiering* performs better than IDT. `graph500` exhibits low memory access locality and random access patterns during the graph search phase. Random access patterns pose challenges in the online training RL model and effective predictive

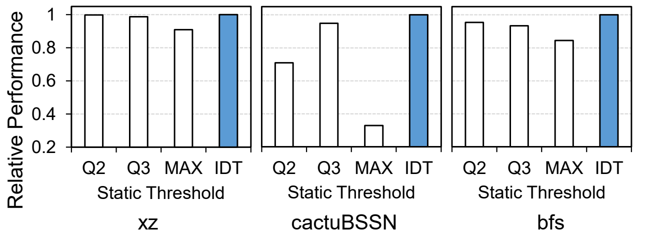


Figure 11: Performance comparison when applying *static* age_thres (higher is better).

promotion by PRP. Also, low locality increases the number of regions, increasing the monitoring overhead. Still, IDT outperforms the second-best performing solution (*Intel Tiering*) by 23.5%.

All solutions show modest speedup in `imagick`. `imagick` is predominantly cold, which indicates that the performance improvement from tiering systems is marginal. Further, `imagick` shows similar performance even when the fast memory is expanded to fit all working sets (§5.7). Overhead is a more significant concern in such cases, and the modest performance improvement of IDT over other solutions highlights its minimal overhead.

5.3 RL Effectiveness Analysis

To validate the effectiveness of IDT's RL model, we compared the overall performance against applying *static* `age_thres` instead of value obtained from the RL model. The *static* `age_thres` were set among the three possible RL model's actions: `q2`, `q3`, and `max` of the age distribution. IDT's RL model continuously adapts `age_thres` among these three configurations to the workload's current execution context. As shown in Figure 11, the best-performing *static* `age_thres` varied across each workload. Meanwhile, IDT outperforms all *static* configurations, implying that the RL model adapts `age_thres` value appropriately for each workload.

We further analyzed the RL model's effectiveness by observing performance variation on `cactuBSSN` according to three critical RL hyperparameters: learning rate (α), exploration rate (ϵ), and discount factor (γ). IDT's RL model utilizes $\alpha = 0.01$, $\epsilon = 0.05$, and $\gamma = 0.9$.

Online training efficacy. The learning rate determines the rate of updating network weight during online training. Figure 12(a) shows that IDT improves performance over a non-online learning

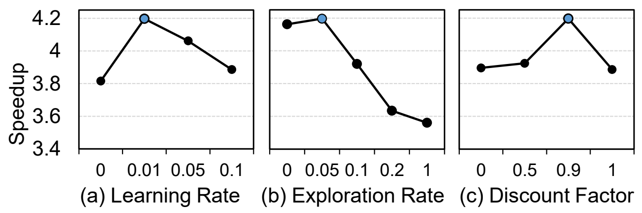


Figure 12: Performance variations of IDT to the RL model's (a) learning rate, (b) exploration rate, and (c) discount factor.

baseline ($\alpha = 0$), which underscores the effectiveness of online training.

Exploration rate. The exploration rate determines the favor of taking random actions over learned policy. Figure 12(b) shows a sharp performance drop as exploration rates increase, which indicates that IDT is more effective than the random policy. Meanwhile, performance was slightly degraded without exploration ($\epsilon = 0$). This highlights the exploration's role in preventing the RL agent from being stuck in a suboptimal policy.

Pre-training efficacy. The substantial performance difference between a static pre-trained network (i.e., $\alpha = 0$) and a complete random policy (i.e., $\epsilon = 1$) highlights the efficacy of pre-training the RL model on the GUPS microbenchmark.

Discount factor. The discount factor progressively weighs recent rewards when calculating the cumulative reward. Figure 12(c) shows that IDT performs better than the case of only accounting for the immediate reward (i.e., $\gamma = 0$).

5.4 RL Overhead Analysis

To measure the overhead of IDT's RL model, we profiled the CPU and memory usage of RL during the evaluation of xz in §5.2.

The average CPU usage of IDT's RL model is *1.35% of a single core*, with peak usage reaching 3.75% during training phases. The peak memory usage is 4,776MB, and by assigning the RL to the lowest memory tier (*tier3* in Figure 10), this only constitutes *0.83% of the total memory*. The RL model's low CPU and memory overhead makes IDT a *feasible* solution for real-world systems, compared to the prior ML-based approaches.

5.5 Memory Access Monitoring Effectiveness

To validate the effectiveness of IDT's memory access monitoring, we compared the overall performance against applying DAMON-based monitoring [44]. Hence, we implemented DAMON-based region reconfiguration (*IDT-DAMON*) by replacing IDT's region reconfiguration method with DAMON's algorithm. As shown in Figure 13, IDT's region reconfiguration method achieves a modest performance improvement over DAMON.

To better understand the precision of region management, we compared the characteristics of regions merged by DAMON and IDT's region reconfiguration. Specifically, we compared the `history` vector's hamming distance of the merged regions, which indicates a difference in their access patterns. DAMON's average hamming distance was 8.13, while IDT was 5.15. This implies that DAMON often merges more regions with dissimilar behaviors than

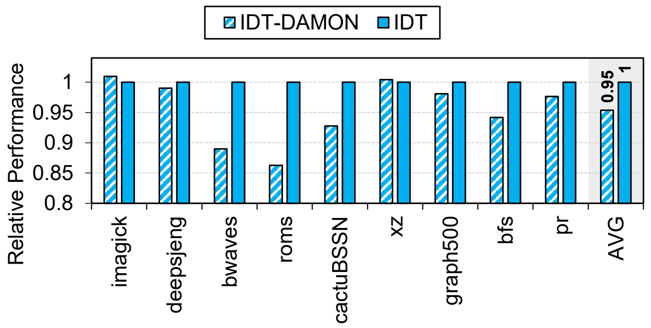


Figure 13: Performance comparison when applying DAMON-based memory access monitoring (higher is better).

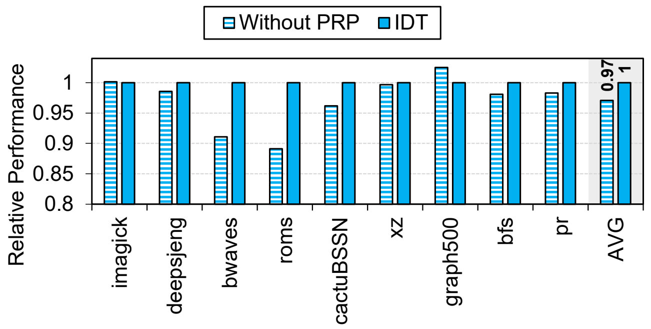


Figure 14: Performance comparison when PRP is not used (higher is better).

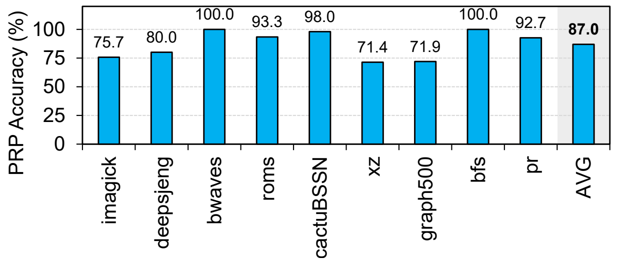


Figure 15: Accuracy of PRP (ratio of region accessed that was promoted by PRP).

IDT. Thus, we suggest that IDT's region reconfiguration method addresses limitations observed in prior works, while better capturing similar regions.

5.6 Predictive Region Promotion Accuracy

To validate the effectiveness of PRP, we compared the overall performance when PRP is not used in IDT. Figure 14 confirms that PRP contributes to the performance improvement of IDT.

To better understand the efficacy of PRP, we evaluated its accuracy. We define *accuracy* as the ratio of regions promoted by PRP that were subsequently accessed again, detected by the memory access monitoring. Only regions that were promoted and not subsequently demoted were considered. Figure 15 shows the high

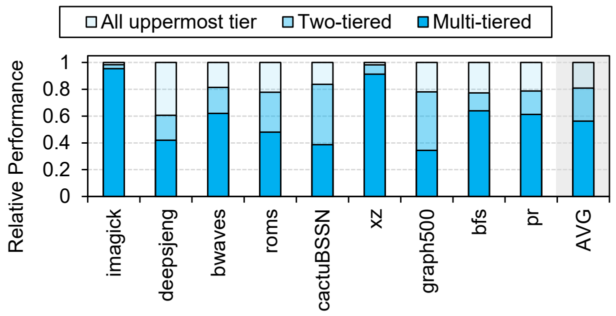


Figure 16: Performance variations under different memory configurations with fixed RSS: uppermost tier expanded to fit all RSS (*All uppermost tier*), upper two tiers expanded to fit all RSS (*Two-tiered*), and multi-tiered setting used in previous evaluation sections (*Multi-tiered*).

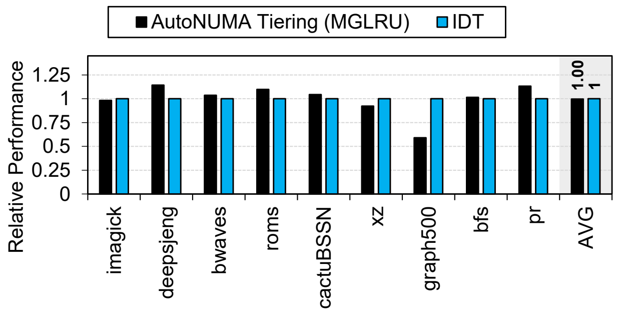
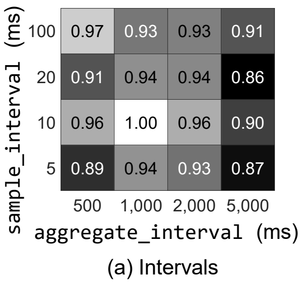


Figure 17: Performance comparison in *Two-tiered* memory configuration (higher is better).

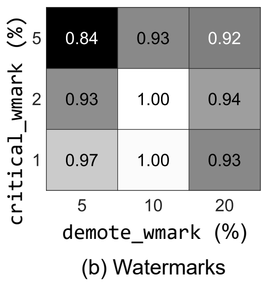
accuracy of PRP, 87% on average. The accuracy of PRP is particularly high in workloads with high memory access locality. Further, workloads with higher accuracy showed more performance improvement in Figure 14. Thus, we can confirm that PRP is a simple yet effective optimization technique to complement accessed-based promotion.

5.7 Performance Variation to Tiered Memory Configurations

We observed the performance of IDT while varying the tiered memory configuration with the workload's RSS fixed to our evaluation setting. We tested on three configurations: the uppermost tier memory's capacity expanded to entirely fit all RSS (*All uppermost tier*), a two-tiered memory setting where the capacity of *tier0* and *tier1* memory in Figure 10 were expanded to 64GB (*Two-tiered*), and a multi-tiered memory setting used in previous evaluation sections (*Multi-tiered*). Specifically, *Two-tiered* configuration allows each half of the RSS to be distributed across the upper two tiers. The *Multi-tiered* setting seeks 1/4 of the RSS to be distributed across the top two tiers and the remainder in *tier2*. Performance was normalized to the case of *All uppermost tier*. Figure 16 shows that in *Two-tiered* configuration, IDT achieves an average 81.3% performance of *All uppermost tier*.



sample_interval (ms) \ aggregate_interval (ms)	500	1,000	2,000	5,000
100	0.97	0.93	0.93	0.91
20	0.91	0.94	0.94	0.86
10	0.96	1.00	0.96	0.90
5	0.89	0.94	0.93	0.87



critical_wmark (%) \ demote_wmark (%)	5	10	20
5	0.84	0.93	0.92
2	0.93	1.00	0.94
1	0.97	1.00	0.93

Figure 18: Relative performance according to IDT 's (a) interval and (b) watermark values.

We also compared the overall performance of IDT and *AutoNUMA Tiering (MGLRU)* in *Two-tiered* memory configuration. Figure 17 shows that IDT achieves similar performance to *AutoNUMA Tiering (MGLRU)* on average. Thus, we can confirm that IDT shows comparable performance with the state-of-the-art solution even in the two-tiered memory, while outperforming it in the multi-tiered memory configuration as shown earlier.

5.8 Sensitivity Study

To investigate the sensitivity related to interval and watermark values, we measured the average performance across three benchmarks (`bwaves`, `cactuBSSN`, and `pr`) according to (a) 16 different combinations of `sample_interval` and `aggregate_interval`, (b) 9 different combinations of `demote_wmark` and `critical_wmark`. Smaller interval values allow finer sampling and more responsive demotion/promotion, but at the cost of increased overhead. Larger watermark values reserve fast memory for potential allocation requests, but may not fully leverage the performance benefits of fast memory. Figure 18 shows the relative performance for each combination against the best-performing configuration, which is used as IDT's default setting.

6 DISCUSSION

6.1 Reward Function Design

The performance of an RL model is significantly influenced by the design of its reward function. A suitable reward function is one that reflects the objectives of tiered memory management, while the cost of deriving its value is inexpensive. IDT's reward function reflects an appropriate demotion policy. Also, it uses data only obtained from IDT's memory access monitoring, thus reducing the cost of deriving the reward value. However, before coming up with such a function, we tried diverse designs based on promising intuitions.

Utilizing precise slow memory hit rate. IDT uses `slow_hit`, which is the number of pages promoted by ARP, as a representation of slow memory access. Since `slow_hit` is an approximate value, performance improvement may be expected when using an accurately measured lower-tier memory access frequency. However, the precise slow memory access information is not readily available in the default Linux kernel. To obtain such statistics, tools like `perf` [1] should be used, which introduces additional overhead to tiered memory management. Moreover, transferring `perf` measurements to the RL model with low overhead at runtime [42] is technically challenging to implement.

Maximizing demotion to promotion interval. A well-demoted region should have a long demotion to promotion interval (`DtoP`). Therefore, an ideal demotion should maximize the `DtoP` of the promoted regions. Obtaining `DtoP` is relatively straightforward by extending IDT's memory access monitoring. However, this method requires waiting for the demoted regions to be promoted again. This results in a long-delayed reward, which is even more vulnerable to a non-stationary environment [52] like memory management.

6.2 Limitations and Future Works

IDT's limitation is its *black-box* nature of the RL model. While it's evident that the RL model enhances overall performance, the inherent opacity of deep RL models makes it difficult to explain the exact reasons for such improvement.

Another limitation is that IDT's RL-based demotion policy operates in the *user space*, whereas most kernel's policies and mechanisms both reside within the *kernel space*. Meanwhile, there are some recent studies to provide an ML framework for integrating various ML algorithms in the kernel space [3]. Leveraging these innovations, there is a potential for IDT's RL model to be seamlessly integrated inside the kernel.

A recent study [28] proposed selectively applying huge pages according to data hotness for performance improvement. IDT is orthogonal to this approach, thus performance improvement can be expected from their methods. Additionally, future research may explore other RL algorithms than PPO, or devise a sophisticated feedback control mechanism for more effective demotion policy autotuning.

7 RELATED WORK

To the best of our knowledge, IDT is the first work to employ RL in tiered memory management. IDT also distinguishes from prior works that have attempted to integrate ML in this domain, by boosting performance on an actual system. In this section, we compare IDT with prior works.

OS-level tiered memory management. Numerous OS-level solutions have been proposed to adaptively place hot pages in faster memory and cold pages in slower memory [2, 7, 12, 15, 16, 19, 21–23, 25, 27, 28, 28, 36, 38, 47, 48, 51, 55–57]. *AutoNUMA Balancing* [16], which is the default Linux kernel scheme, dynamically adjusts data placement on NUMA systems by profiling NUMA fault statistics. *AutoNUMA Tiering* [57] is an extension to *AutoNUMA Balancing* by also considering the tier of the accessed memory. *AutoNUMA Tiering* can be extended to utilize MGLRU from the Linux kernel v6.1 [58]. *Intel Tiering* [21] enhances *AutoNUMA Tiering* by employing NUMA hinting fault statistics in the promotion mechanism. *TPP* [36], a solution from Meta, further improves *AutoNUMA Tiering* by adding hysteresis to promotion to prevent unnecessary traffic. However, because the NUMA hinting fault is on the critical path of memory access, these *AutoNUMA*-based schemes have high hotness identification overhead [15, 28].

Intel Tiering and *TPP* were initially proposed for two-tiered memory, and extending them in the context of multi-tiered memory is not straightforward. Before *AutoNUMA Tiering* effectively supported multi-tiered memory, *AutoTiering* [23] demonstrated effective demotion, promotion, and migration schemes between multiple memory nodes. However, it often underperforms compared to two-tiered solutions, as shown in §5.2. We believe this is due to the increased complexity of managing multiple memory tiers.

In contrast, IDT not only supports multi-tiered memory but also autonomously adjusts demotion criteria to effectively place data across multiple memory tiers. This strategic placement results in performance improvements compared to previous works.

ML-based data placement in tiered memory. Prior works that utilized ML for tiered memory management employed online-trained LSTM to analyze individual [12, 14, 34] or groups [13] of pages' access patterns for data placement. Others used Gaussian Process bandits [15, 27] to optimize demotion parameters. However, these studies often suffer from excessive increases in execution time and memory usage [12–14, 34], or require substantial resources for model training [15, 27]. In contrast, IDT employs RL with minimal overhead, enhancing performance on an actual system.

RL-based system management. In storage systems, where latency is less critical, numerous studies have employed RL for data placement across multiple storage devices [32, 43, 50]. Furthermore, RL has been explored for cache management [31], prefetching [6], execution time estimation [10], power management [24], scheduling [17], and congestion control [53]. Alongside these RL-based system optimization works, IDT further showcases that tiered memory can also be adeptly managed using RL.

8 CONCLUSION

We present IDT, an RL/ML-based multi-tiered memory management solution. We identified that RL can be effective at constructing demotion policies, and how demotion can be formulated in a solvable form using RL. We also identified the need for a logical basis in region-granularity memory access monitoring to provide an accurate environment state to the RL model. Based on these analyses, we developed memory access monitoring, demotion with RL-driven policy autotuning, promotion to minimize slow memory access, and statistical testing-based region reconfiguration. Experiments on a real-world multi-tiered memory system show that IDT consistently outperforms the default Linux kernel by 2.08× on average, and the state-of-the-art solution by 11.2%. We hope that IDT not only offers an efficient solution for multi-tiered memory systems, but also validates the feasibility of ML-based approaches in the realm of tiered memory management.

ACKNOWLEDGMENTS

This paper was supported by SNU-SK hynix Solution Research Center (S3RC), Korea Institute for Advancement of Technology (KIAT) grant funded by the Korea Government (Ministry of Education) (P0025681-G02P22450002201-10054408, Semiconductor-Specialized University), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2022R1F1A1062826), and an Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2021-0-01343 and IITP-2023-RS-2023-00256081). This work was done when Yaebin Moon was at Seoul National University (SNU). Jung Ho Ahn, the corresponding author, is with the Department of Intelligence and Information and the Interdisciplinary Program in Artificial Intelligence, SNU.

REFERENCES

[1] 2023. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System Programming Guide, Part* (2023).

[2] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-Transparent Page Management for Two-tiered Main Memory. In *ASPLOS* (Xi'an, China). Association for Computing Machinery, New York, NY, USA, 631–644. https://doi.org/10.1145/3037697.3037706

[3] Ibrahim Umit Akgun, Ali Selman Aydin, Aadil Shaikh, Lukas Velikov, and Erez Zadok. 2021. A Machine Learning Framework to Improve Storage System Performance. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems* (Virtual, USA) *(HotStorage '21)*. Association for Computing Machinery, New York, NY, USA, 94–102. https://doi.org/10.1145/3465332.3470875

[4] James Alfred Ang, Brian W Barrett, Kyle Bruce Wheeler, and Richard C Murphy. 2010. Introducing the Graph 500. (2010).

[5] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[6] 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* (Virtual Event, Greece). Association for Computing Machinery, New York, NY, USA, 1121–1137. https://doi.org/10.1145/3466752.3480114

[7] Shai Bergman, Priyank Faldu, Boris Grot, Lluís Vilanova, and Mark Silberstein. 2022. Reconsidering OS Memory Optimizations in the Presence of Disaggregated Memory. In *ISMM* (San Diego, CA, USA). Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3520263.3534650

[8] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 41–42. https://doi.org/10.1145/3185768.3185771

[9] CXL 3.0 2022. Compute Express Link. https://www.computeexpresslink.org.

[10] Nicolas Denoyelle, Swann Perarnau, Kamil Iskra, and Balazs Gerofi. 2022. Rapid Execution Time Estimation for Heterogeneous Memory Systems Through Differential Tracing. In *High Performance Computing: 37th International Conference, ISC High Performance 2022, Hamburg, Germany, May 29 – June 2, 2022, Proceedings* (Hamburg, Germany). Springer-Verlag, Berlin, Heidelberg, 256–274. https://doi.org/10.1007/978-3-031-07312-0_13

[11] Tonmoy Dey, Kento Sato, Bogdan Nicolae, Jian Guo, Jens Domke, Weikuan Yu, Franck Cappello, and Kathryn Mohror. 2020. Optimizing Asynchronous Multi-Level Checkpoint/Restart Configurations with Machine Learning. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1036–1043. https://doi.org/10.1109/IPDPSW50202.2020.00174

[12] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. 2019. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/3307681.3325398

[13] Thaleia Dimitra Doudali and Ada Gavrilovska. 2022. Coeus: Clustering (A)like Patterns for Practical Machine Intelligent Hybrid Memory Management. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing* (Taormina, Italy) *(CCGrid '22)*. 615–624. https://doi.org/10.1109/CCGrid54584.2022.00071

[14] Thaleia Dimitra Doudali, Daniel Zahka, and Ada Gavrilovska. 2021. Cori: Dancing to the Right Beat of Periodic Data Movements over Hybrid Memory Systems. In *2021 IEEE International Parallel and Distributed Processing Symposium* (Portland, OR, USA) *(IPDPS '21)*. 350–359. https://doi.org/10.1109/IPDPS49936.2021.00043

[15] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *ASPLOS* (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 727–741. https://doi.org/10.1145/3582016.3582031

[16] Mel Gorman. 2012. *Foundation for Automatic NUMA Balancing.* https://lwn.net/Articles/523065/.

[17] Nathan Grinsztajn, Olivier Beaumont, Emmanuel Jeannot, and Philippe Preux. 2021. READYS: A Reinforcement Learning Based Strategy for Heterogeneous Dynamic Scheduling. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 70–81. https://doi.org/10.1109/Cluster48925.2021.00031

[18] GUPS 2021. *GUPS (Giga Updates Per Second).* https://icl.utk.edu/projectsfiles/hpcc/RandomAccess/.

[19] 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

[20] Amazon Inc. [n. d.]. *Amazon EC2 High Memory Instances.* https://aws.amazon.com/ec2/instance-types/high-memory/.

[21] Intel. 2022. Tiering-0.8. https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/.

[22] M. Jorda, S. Rai, E. Ayguade, J. Labarta, and A. J. Pena. 2022. ecoHMEM: Improving Object Placement Methodology for Hybrid Memory Systems in HPC. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, Los Alamitos, CA, USA, 278–288. https://doi.org/10.1109/CLUSTER51413.2022.00040

[23] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *USENIC ATC* (Virtual Event). USENIX Association, 715–728. https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

[24] Seyeon Kim, Kyungmin Bin, Sangtae Ha, Kyunghan Lee, and Song Chong. 2021. zTT: Learning-Based DVFS with Zero Thermal Throttling for Mobile Devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services* (Virtual Event, Wisconsin) *(MobiSys '21)*. Association for Computing Machinery, New York, NY, USA, 41–53. https://doi.org/10.1145/3458864.3468161

[25] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2018. Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona) *(HPDC '18)*. Association for Computing Machinery, New York, NY, USA, 219–230. https://doi.org/10.1145/3208040.3208059

[26] Oliver Kramer. 2013. *K-Nearest Neighbors.* Springer Berlin Heidelberg, Berlin, Heidelberg, 13–23.

[27] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *ASPLOS* (Providence, RI, USA). Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/3297858.3304053

[28] 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 *SOSP* (Koblenz, Germany). Association for Computing Machinery, New York, NY, USA, 17–34. https://doi.org/10.1145/3600006.3613167

[29] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 3053–3062. https://proceedings.mlr.press/v80/liang18b.html

[30] Linux community. 2023. *migrate_pages() Function of Linux Kernel.* https://elixir.bootlin.com/linux/v6.0.19/source/mm/migrate.c#L1395.

[31] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An Imitation Learning Approach for Cache Replacement. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 6237–6247. https://proceedings.mlr.press/v119/liu20f.html

[32] Kaiyang Liu, Jun Peng, Jingrong Wang, Boyang Yu, Zhuofan Liao, Zhiwu Huang, and Jianping Pan. 2022. A Learning-Based Data Placement Framework for Low Latency in Data Center Networks. *IEEE Transactions on Cloud Computing* 10, 1 (2022), 146–157. https://doi.org/10.1109/TCC.2019.2940953

[33] Google LLC. [n. d.]. *Memory-optimized machine family for Compute Engine.* https://cloud.google.com/compute/docs/memory-optimized-machines/.

[34] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-Based Memory Allocation for C++ Server Workloads. In *ASPLOS* (Lausanne, Switzerland). Association for Computing Machinery, New York, NY, USA, 541–556. https://doi.org/10.1145/3373376.3378525

[35] A. Maruf, A. Ghosh, J. Bhimani, D. Campello, A. Rudoff, and R. Rangaswami. 2022. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 925–937. https://doi.org/10.1109/HPCA53966.2022.00072

[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 *ASPLOS* (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 742–755. https://doi.org/10.1145/3582016.3582063

[37] H. Menon, A. Bhatele, and T. Gamblin. 2020. Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 831–840. https://doi.org/10.1109/IPDPS47924.2020.00090

[38] Yaebin Moon, Wanju Doh, Kwanhee Kyung, Eojin Lee, and Jung Ho Ahn. 2023. ADT: Aggressive Demotion and Promotion for Tiered Memory. *IEEE Computer Architecture Letters* 22, 1 (2023), 21–24. https://doi.org/10.1109/LCA.2023.3236685

[39] Onur Mutlu and Lavanya Subramanian. 2014. Research Problems and Opportunities in Memory Systems. *Supercomputing Frontiers and Innovations: An*

International Journal 1, 3 (oct 2014), 19–55. https://doi.org/10.14529/jsfi140302

[40] Alan Nair, Sandeep Kumar, Aravinda Prasad, Andy Rudoff, and Sreenivas Subramoney. 2023. Telescope: Telemetry at Terabyte Scale. arXiv:2311.10275 [cs.OS]

[41] Nevine Nassif, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. 2022. Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 44–46. https://doi.org/10.1109/ISSCC42614.2022.9731107

[42] Aleix Roca Nonell, Balazs Gerofi, Leonardo Bautista-Gomez, Dominique Martinet, Vicenç Beltran Querol, and Yutaka Ishikawa. 2018. On the Applicability of PEBS based Online Memory Access Tracking for Heterogeneous Memory Management at Scale. In *Proceedings of the Workshop on Memory Centric High Performance Computing* (Dallas, TX, USA) *(MCHPC'18)*. Association for Computing Machinery, New York, NY, USA, 50–57. https://doi.org/10.1145/3286475.3286477

[43] Lu Pang, Anis Alazzawe, Madhurima Ray, Krishna Kant, and Jeremy Swift. 2023. Adaptive Intelligent Tiering for modern storage systems. *Performance Evaluation* 160 (2023), 102332. https://doi.org/10.1016/j.peva.2023.102332

[44] SeongJae Park. 2020. *DAMON: Data Access Monitor.* https://docs.kernel.org/mm/damon/index.html.

[45] 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* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 4–15. https://doi.org/10.1145/3502181.3531466

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library.* Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[47] 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 *SOSP* (Virtual Event, Germany). Association for Computing Machinery, New York, NY, USA, 392–407. https://doi.org/10.1145/3477132.3483550

[48] Jie Ren, Dong Xu, Ivy Peng, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2023. Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory Systems. arXiv:2302.09468

[49] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347

[50] Gagandeep Singh, Rakesh Nadig, Jisung Park, Rahul Bera, Nastaran Hajinazar, David Novo, Juan Gómez-Luna, Sander Stuijk, Henk Corporaal, and Onur Mutlu. 2022. Sibyl: Adaptive and Extensible Data Placement in Hybrid Storage Systems Using Online Reinforcement Learning. In *ISCA* (New York, New York). Association for Computing Machinery, New York, NY, USA, 320–336. https://doi.org/10.1145/3470496.3527442

[51] Kevin Song, Jiacheng Yang, Sihang Liu, and Gennady Pekhimenko. 2023. Lightweight Frequency-Based Tiering for CXL Memory Systems. arXiv:2312.04789 [cs.DC]

[52] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement Learning: An Introduction.* MIT press.

[53] Chen Tessler, Yuval Shpigelman, Gal Dalal, Amit Mandelbaum, Doron Haritan Kazakov, Benjamin Fuhrer, Gal Chechik, and Shie Mannor. 2022. Reinforcement Learning for Datacenter Congestion Control. *SIGMETRICS Performance Evaluation Review* 49, 2 (jan 2022), 43–46. https://doi.org/10.1145/3512798.3512815

[54] 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* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. https://doi.org/10.1145/3503222.3507731

[55] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. MATRYOSHKA: Non-Exclusive Memory Tiering via Transactional Page Migration. *arXiv preprint arXiv:2401.13154* (2024).

[56] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *ASPLOS* (Providence, RI, USA). Association for Computing Machinery, New York, NY, USA, 331–345. https://doi.org/10.1145/3297858.3304024

[57] Huang Ying. 2020. *AutoNUMA: Optimize Memory Placement for Memory Tiering System.* https://lwn.net/Articles/835402/.

[58] Yu Zhao. 2022. *Multigenerational LRU Framework.* https://lwn.net/Articles/880393/.