ReStore: A Reinforcement Learning Approach for Data Migration in Multi-Tiered Storage

TIANRU ZHANG*, Uppsala University, Sweden
TARIKUL ISLAM PAPON†, University of Massachusetts Boston, USA
TEONA BAGASHVILI, Boston University, USA
SALMAN TOOR, Uppsala University, Sweden
MANOS ATHANASSOULIS, Boston University, USA

With the development of storage technologies, a wide variety of storage devices with different performance characteristics and cost profiles have emerged. Consequently, modern data systems are increasingly adopting multi-tiered storage solutions, where faster (but typically smaller) devices are placed in higher tiers, while lower tiers comprise slower, higher-capacity devices. A primary challenge in such systems is fine-grained data placement, i.e., determining how data blocks should be dynamically stored and migrated across different storage tiers to optimize overall performance. Effective data migration policies should be *lightweight* and *adaptive* to workload variations while considering *storage device characteristics*, notably the read/write asymmetry and parallelism of modern SSDs.

In this paper, we introduce ReStore, a lightweight page-level reinforcement learning (RL) approach for data migration in multi-tiered storage systems, addressing both the performance and fine-grained access challenges common in database systems. ReStore leverages RL to capture workload patterns (access frequency and recency) and device-specific characteristics (read/write asymmetry and internal parallelism). Each storage tier uses a different device and is associated with an RL agent that dynamically updates its parameters using temporal difference learning, ensuring continuous adaptability to changing workloads and system states. We experimentally show that ReStore achieves up to 6× lower runtime and up to 48× fewer migrations using industry-grade benchmarks, like TPC-C, TPC-E, and YCSB, real-life traces, like Google Thesios and MSR Cambridge, and a wide variety of synthetic workloads.

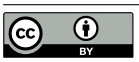
CCS Concepts: • **Information systems → Hierarchical storage management**; *Database management system engines*; • **Computing methodologies → Reinforcement learning**.

Additional Key Words and Phrases: Multi-Tiered Storage, Page Migration, SSD Concurrency and Asymmetry, Workload Adaptivity

ACM Reference Format:
Tianru Zhang, Tarikul Islam Papon, Teona Bagashvili, Salman Toor, and Manos Athanassoulis. 2026. ReStore: A Reinforcement Learning Approach for Data Migration in Multi-Tiered Storage. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 227 (June 2026), 28 pages. https://doi.org/10.1145/3802104

*Part of this work done while the author was a visiting graduate student at Boston University.
†Part of this work done while the author was a graduate student at Boston University.

Authors' Contact Information: Tianru Zhang, Uppsala University, Sweden, tianru.zhang@it.uu.se; Tarikul Islam Papon, University of Massachusetts Boston, USA, t.papon@umb.edu; Teona Bagashvili, Boston University, USA, teona@bu.edu; Salman Toor, Uppsala University, Sweden, salman.toor@it.uu.se; Manos Athanassoulis, Boston University, USA, mathan@bu.edu.



ACM 2836-6573/2026/6-ART227
https://doi.org/10.1145/3802104

1 Introduction

Rise of Big Data. Data generation is accelerating due to social media, digital transactions, sensors, and the growing Internet of Things (IoT) ecosystem [13, 76]. This surge presents core challenges in storage, retrieval, and maintenance – demanding scalable architectures that balance cost and performance.

Multi-Tiered Storage System for Big Data Management. To address these needs, scalable storage solutions have evolved both (i) horizontally, via distributed file systems like GFS [23], Cassandra [38], HDFS [68], Ceph [82]; and (ii) vertically, through multi-tiered systems like HP AutoRAID [83] and IBM Storage Tank [15, 48]. In a multi-tiered storage system, each tier consists of one storage device (or one class of devices that share similar performance characteristics). Different tiers (e.g., fast, medium, slow) are constructed by grouping devices according to their access latency. For example, modern SSDs span a wide performance spectrum, with read latencies ranging from $10\mu s$ (high-end NVMe SSDs) to $10ms$ (low-end SATA SSDs). These heterogeneous devices naturally form a multi-tiered storage hierarchy, similar to the classical memory hierarchy [4]. Generally, upper-tier devices are faster and smaller; lower tiers are slower and larger. Effective management is critical to optimize performance and resource use while controlling cost [88], typically by placing hot data blocks on faster devices and demoting colder data blocks to slow tiers. We now ask the question: *how do modern storage device properties affect the performance of a tiered storage system?*

Storage Devices and Their Properties. Modern multi-tiered storage systems typically integrate different storage devices, varying from Hard Disk Drives (HDDs) to Solid State Drives (SSDs). Since SSDs offer the highest performance among secondary storage technologies, multiple (or all) tiers within such systems often employ SSDs. Therefore, in this work, we primarily focus on better exploiting SSDs. Typically, SSDs exhibit a high degree of internal parallelism (termed *concurrency* in this paper) that can be harnessed to increase performance [54, 57]. On the other hand, for flash-based SSDs, the cost of reading is generally lower than the (amortized) cost of writing, leading to an SSD read/write asymmetry where writes can be up to one order of magnitude slower than reads [17]. These properties of SSDs are crucial for performance optimization: (i) proper use of SSD concurrency enables better device utilization, and (ii) special care of expensive writes can optimize overall workload execution [53, 55, 56]. Hence, faithful performance modeling of SSDs that captures (and exploits) device properties can help tiered storage systems improve resource utilization and performance.

Challenges of Tiered Storage Management. The quality of a tiered storage management approach depends on its setup (data types, access patterns, storage devices, and data temporal characteristics) and cost efficiency. We identify that common recency-based, frequency-based, or hybrid policies face three major challenges.

Challenge 1: Optimal Data Placement. Data placement is the core design decision in any multi-tiered storage system. Ideally, hot data should reside in fast tiers and cold data should be in slow tiers, but with dynamic workloads, data hotness can shift frequently. Rule-based strategies (e.g., LRU, LFU) struggle to adapt to workload drift and overlook SSD characteristics, leading to poor performance.

Challenge 2: Capturing Storage Device Properties. SSD *concurrency* and *asymmetry* have not been exploited in multi-tiered storage systems. For example, there is a 40× increase in the read bandwidth of the Intel P4510 SSD when using full concurrency vs. none, and 4KB writes are 3× slower than 4KB reads [54]. Without capturing these storage properties, the devices can remain vastly underutilized.

features	EXD	LRFU	tLRU	tLFU	Our Approach
access frequency	✓	✓	×	✓	✓
access recency	✓	✓	✓	×	✓
dynamic allocation	✓	✓	✓	✓	✓
workload drift	~	~	×	×	✓
device properties	×	×	×	×	✓

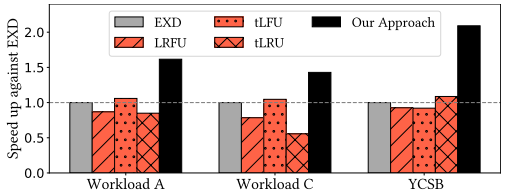


Fig. 1. Popular policies like EXD [22] and LRFU [39] capture workload features via utility function but ignore device properties and struggle with workload drift. tLRU and tLFU capture access statistics, but are slow to adapt to drift and do not consider device properties. Our approach captures workload features and device properties and can adapt to dynamic workloads. Among the three workloads, A is more skewed, and YCSB shows higher drift (details in §6.1). Our approach outperforms the state of the art by up to 2.2×.

Challenge 3: Finer Migration Granularity. Existing data migration policies for tiered storage are frequently designed for coarse-grained units, such as data files or large objects, as seen in systems like Ceph [82] and cloud-based lifecycle management like AmazonS3 [64]. While some enterprise solutions, such as IBM Easy Tier [21] and Linux bcache [41], operate at the block level, they typically rely on static heuristics or fixed observation windows. However, many performance-critical applications (e.g., DBMS, KV-stores) operate on small, fixed-size pages (e.g., 4KB). This creates the need for migration policies that can handle page-level granularity with the intelligence to distinguish between *hot* pages and *cold* pages at runtime. To work on such finer granularity, migration policies must have *low overhead* while capturing workload and device properties.

Reinforcement Learning to the Rescue. Reinforcement Learning (RL) employs intelligent *agents* to make decisions by *interacting* with an environment and receiving feedback in the form of rewards or penalties, with the objective of maximizing cumulative reward. The RL framework naturally models system states and decision processes, making it particularly well-suited for learning data placement or migration strategies in tiered storage. It enables dynamic adaptation to changing workloads and evolving device conditions.

Design Goals. We set the following design goals for an RL-based page migration policy for multi-tiered storage:

- **Faster Execution through Optimal Data Placement:** Leverage RL to learn optimal data placement via efficient migration policies to accelerate workload execution.
- **Model Heterogeneous Tier Properties:** Capture various properties per tier, including capacity, read/write asymmetry, and concurrency. To do this, we use a multi-agent architecture, assigning one agent to each tier.
- **Dynamic Tier Adjustability:** Support seamless addition or removal of tiers by the modular design – one RL agent per tier – to maintain system flexibility.
- **Lightweight Agent Design:** Ensure the RL agents are lightweight enough to handle frequent, fine-grained (page-level) placement decisions with minimal overhead. We achieve this by minimizing the number of parameters and employing simple value function approximations.
- **Adaptivity to access patterns and tier loads:** Continuously update agent parameters in response to changes in workload and the dynamic load conditions of each tier.

ReStore: An RL-Based Tiered Storage Migration Policy. To address the above challenges, we propose ReStore, an RL-based multi-tiered storage migration policy, which can be integrated into any multi-tiered storage system with **page-level** accesses. As part of the RL formulation, we define state variables that capture the workload features (recency and frequency) and device properties (concurrency and asymmetry), and use a reward function tied to system runtime. The policy is then optimized by maximizing the accumulated reward through value functions that quantify the

quality of the current system state, which in turn minimizes workload execution cost. To ensure adaptability, we update the value functions dynamically using *Temporal Difference* learning. A prime benefit of ReStore is that its RL-based page migration receives feedback from and adapts to system interaction. The table in Figure 1 contrasts current migration policies for multi-tiered storage. Compared to state-of-the-art approaches, ReStore responds well to dynamic workloads and is the only policy that considers device properties.

Figure 1 compares ReStore with rule-based and ML-based migration policies in the presence of workload drift (details in §6.1). The figure shows that ReStore achieves up to 2.2× speedup over other baselines for dynamic workloads, showcasing its adaptability.

Contributions. Our work offers the following contributions:

- We identify the key challenges of multi-tiered storage management: optimal data placement, page-level migration, and capturing storage device properties (§3).
- We propose ReStore, a multi-tiered storage management approach that uses *Reinforcement Learning* for *page-level migration*. By interacting with its environment and underlying storage, ReStore accurately captures the system status to make data placement and migration decisions. ReStore models the system based on (i) the *temperature* of each page, which captures the workload recency and frequency, and (ii) SSD properties (§5).
- We extensively evaluate ReStore against state-of-the-art approaches using a wide range of SSD configurations. Our evaluation leverages (i) the in-house multi-tiered storage framework, (ii) FEMU, a flash storage emulator that provides accurate and scalable SSD modeling [40], and (iii) cloud-based real storage. We demonstrate that ReStore can speed up execution by up to 6× while having up to 48× fewer page migrations (§6). We also make our code available for exploration and reproducibility at https://github.com/BU-DiSC/ReStore.

2 Background & Related Work

Multi-Tiered Storage Systems. Data migration has long been integral to storage systems [29, 30], especially in multi-tiered architectures, where it plays a key role in optimizing performance, scalability, and cost-efficiency while balancing resource utilization. For instance, AutoRAID [83] dynamically migrates frequently accessed data to faster storage tiers to improve access latency and overall system responsiveness.

Data Migration at File/Object Level. The process of moving data objects or files between storage layers is typically governed by predefined policies. For example, LMsT [74] applies rules and constraints to support dynamic workloads and varying service level agreements. Extent-Based Dynamic Tiering Management [27] uses a resource consumption model for cost-effective tiering. Migration cost models have also informed algorithms such as SPACE, TIME, and LMIN [63]. Further, temperature-based criteria are also widely adopted: MTM [62] employs an exponential moving average of data hotness, while other methods rely on extended temperature metadata [19]. ADLAM [87] leverages extent temperature for adaptive lookahead migration, and Siberia [20] focuses on managing cold data for main-memory optimized databases. All these policies operate at the file/object level, focusing on two tiers (memory and storage), whereas our objective focuses on page-level migration in a multi-tiered storage setup. Since no such policies for tiered storage exist formally in the literature, we take inspiration from in-memory caching techniques that support page-level granularity.

Page-level Caching and Eviction Policies. Caching is a well-established area in computer science [60]. The 5-minute rule [26] and its recent adaptations [6, 24, 25] highlight the importance of balancing between relative cost, access pattern, and performance. We categorize cache eviction policies into four classes: *(A) Recency-based policies* evict data based on how recently it is accessed. For example,

Table 1. Feature-based vs. ML/RL-based policies.

Category	Policy	Average time per request (μs)	Pre-Train needed
Feature-based	LFU	0.252	No
	LRU	1.128	No
	EXD	0.319	No
	LRFU	0.349	No
ML-based	Logistic Regr.	6.434	Yes
	XGBoost	6.865	Yes
RL-based	Sibyl [71]	27.356	Yes
	ReStore (ours)	0.801	No

Least Recently Used (LRU) [51] evicts the least-recently used data. Variations like 2Q [34], MQ [91], and CLOCK-Pro [31] either improve adaptability or reduce overhead. *(B) Frequency-based policies* focus on the access count. The Least Frequently Used (LFU) policy evicts the least-accessed data. Variants like LFU with Dynamic Aging [7] further penalize older data. We note that both recency and frequency-based approaches struggle with shifting workloads. *(C) Hybrid policies* combine recency and frequency to capture both short-term and long-term access trends. Adaptive Replacement Cache (ARC) [47], and Low Inter-reference Recency Set (LIRS) [32] are popular examples that dynamically adjust between LRU and LFU based on the workload. *(D) Function-based policies* use cost models or heuristics to decide when to move data between tiers. EXD [22] uses an exponential decay score that weighs recency and frequency. LRFU [39] combines the two via a parameterized exponential function.

ML-based Data Migration. In recent years, machine learning (ML) techniques like logistic regression, neural networks, and random forests have been applied to web caching [3, 11, 80]. In a tiered setup, the migration policy typically uses supervised learning, training predictive models on historical workload data to predict future access patterns. Approaches such as Support Vector Machines [66], Decision Trees [14], XGBoost [28], and popular neural networks, including MLP, LSTM, and CNN [61, 67, 75], have been applied. Unsupervised learning methods, including clustering [5, 52, 66] and evolutionary algorithms like Particle Swarm Optimization [2, 86] have also been explored to guide migration decisions based on inferred workloads. While adaptive, these approaches incur *heavy training overhead.*

Reinforcement learning (RL) has also been explored for efficiently managing data migration [36, 42, 43, 71, 81]. Among them, Sibyl [71] is the only one tailored for tiered storage, but its deep RL design introduces significant overhead, making it unsuitable for fine-grained, page-level migration. Moreover, existing ML/RL methods overlook device-specific characteristics. Table 1 compares prior works. Feature-based methods are fast but lack adaptability; while ML-based approaches adapt better but require pre-training and incur overhead. In contrast, our RL-based method is lightweight, page-level, device-aware, and learns online without pre-training.

SSD Asymmetry and Concurrency. We now discuss two key properties of SSDs – *asymmetry* and *concurrency*. SSDs exhibit read/write asymmetry due to (i) the *erase-before-write* design, (ii) the large erasure granularity, and (iii) the overhead of garbage collection [10, 17, 50]. In flash-based SSDs, once a page is written, it cannot be in-place updated [17, 58]. Instead, upon a page update request, the controller marks the old page as invalid and writes the new data to a different location [1]. Over time, invalidated pages accumulate, which triggers periodic *garbage collection*, which contributes to an increased *amortized write cost.*

Flash-based SSDs show parallelism at multiple levels [12, 46]. Typically, an SSD connects to the flash controller through multiple channels, each comprising a shared bus that links to several chips,

Table 2. Empirical values for SSD device properties.

Device	Read (μs)	Write (μs)	α	k_r	k_w
Intel Optane P4800X SSD	6.8	7.6	1.1	6	5
Intel PCIe DC-P4500 SSD	8.2	78.4	11.9	40	8
Intel SATA S4610 SSD	100	140	1.5	25	9
SATA ST2000NX0463 HDD	7000	7300	1.0	1	1

each divided into multiple dies, planes, and erase blocks, where the actual storage pages reside. This hierarchical architecture enables SSDs to handle concurrent accesses efficiently [54]. When multiple I/O requests are issued simultaneously, the flash controller distributes them across different parts of the device to maximize parallelism [46, 65], thus, peak bandwidth is achieved when multiple concurrent I/Os are in progress [8, 56]. The degree of *concurrency* depends on the device, access pattern, and block size [54]. Table 2 lists the read/write asymmetry (α), read concurrency (k_r), and write concurrency (k_w) of representative devices, where we see that both α and k vary for different devices and operations (read/write).

3 Tiered Storage Systems & Challenges

A multi-tiered storage system is a storage architecture that places data across multiple *tiers* of storage media (e.g., NVMe/Fast SSDs, slower SSDs, HDD, cloud object storage), each with different cost, latency, and capacity characteristics, while moving or replicating data between tiers so that *hot* (i.e., frequently accessed) data sits on faster tiers while *cold* (i.e., rarely accessed) data sits on cheaper, slower tiers. Note that each tier contains at least one device, and typically, devices across tiers have substantially different performance and cost characteristics (like in Figure 2). Traditional tiering systems manage data at the granularity of files or objects, however, database systems interact with storage at *page granularity*, where each page is placed in exactly one tier at any time. As shown in Figure 2, the workload is a sequence of page requests, where each request targets a page and specifies an operation type (read or write). For each request, the system first serves the page K from its *current* tier (i.e., the tier where the page resides at the time of the request). Serving requests from different tiers incurs different latency and throughput costs. As the workload changes, the system *migrates* pages across tiers to ensure that frequently accessed pages are delivered with low latency. A migration moves a page from tier T_i to a neighboring tier (typically adjacent), by either *promoting* K to a faster tier or *demoting* K to a slower tier. Migrations consume I/O bandwidth and generate additional writes, hence, they must be carefully controlled via a *migration policy*. Unlike classical buffer management, where a miss triggers a fetch into memory and eviction removes pages from the buffer pool, multi-tiered designs require decisions such as *where* a page should reside across tiers and *when* it is worth incurring migration costs. These differences introduce new challenges for page-level tiered storage management, which we now discuss.

Optimal Page Placement and Migration. Tiered systems use metadata of data objects to decide in which tier to store them and when to migrate them. The algorithms used to calculate optimal data placement for entire files do not apply to single pages. For example, unlike files, all pages have the same size. Crucially, the number of pages is much higher than the number of files, making complex ML-based algorithms hard to use. On the other hand, standard caching algorithms, like LRU and LFU, can be adapted to perform data migration (page promotion and demotion), however, they rely on simple heuristics and typically struggle under workload drift.

Storage Device Properties. Existing migration policies largely ignore device-specific characteristics. Modern SSDs typically exhibit a degree of read/write asymmetry and access parallelism. Devices can remain significantly underutilized if these properties are not considered, resulting in subpar performance. Existing migration policies do not faithfully model or exploit SSD characteristics.

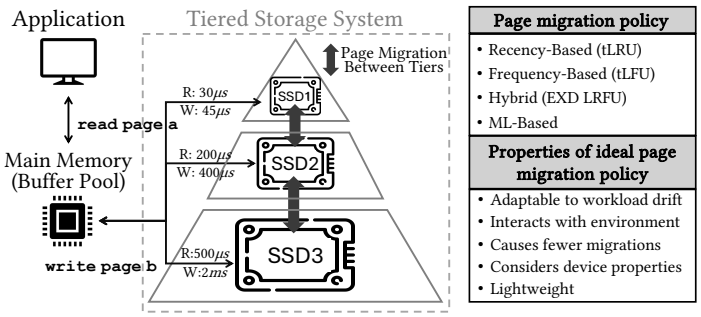


Fig. 2. High-level overview of a page-level multi-tiered storage system consisting of SSDs, where devices are tiered top-down based on their performance and capacity. The diagram illustrates the data path between the application layer, main memory, and the storage tiers.

Lightweight Migration Decisions. Following the first two challenges, we now see that page-level migration policies for multi-tiered storage systems must strike a fine balance between being, on the one hand, complex enough to be able to capture workload access properties and drift as well as device properties, and, on the other hand, lightweight enough to be able to make decisions at the page granularity, and as a result, at a much higher frequency.

Using Reinforcement Learning for Page Migration. To address these challenges, we propose using reinforcement learning (RL) to monitor the system's state and learn which migrations optimize performance. RL is a good fit in this context because it can model the multi-tiered storage system with sufficient detail, using state variables to capture both workload and device characteristics. Furthermore, it can evaluate the reward of state transitions using lightweight value functions, balancing complexity and efficiency. We next offer a brief primer on RL and our notation before discussing our RL-based design in detail.

4 A Primer on RL for Tiered Storage

RL is a widely used approach for decision-making based on statistical estimations and environmental variables [73]. Its primary goal is to develop intelligent agents capable of learning from environmental interactions to make optimal decisions, adapt to new circumstances, and efficiently achieve objectives. RL accomplishes these objectives by solving the Markov Decision Process (MDP), which is an environment formed by $< S, A, P, R, \gamma >$. Here S is states, A is actions, P is transition probabilities, R is rewards, and γ is the discount factor. RL solves MDP by learning the best policy ($\pi : S \rightarrow A$) that prescribes the optimal action based on the current state. This determination of the best action derives from optimizing both the *state-value function* and *action-value function* under the chosen policy and action. The executing entity or agent continuously updates its *value functions*, learning from the system transitions from one state to another, enabling it to perform the best action based on the current state and its refined parameters.

RL for Tiered Storage. In the context of page management in multi-tiered storage systems, RL can be effectively utilized by modeling the problem as an MDP. The Markov property holds because, at the granularity of page requests, the system's future behavior is determined by the current page placement state and the chosen migration action, rather than the full history of past accesses. In other words, individual page requests are independent at the granularity of consecutive accesses, i.e., the next page accessed does not depend on the identity of the current request. Accordingly, the MDP components are defined as follows: states (S) represent the current status of the storage tiers, actions (A) involve decisions on migrating pages between different tiers, transition probabilities

Table 3. Our notation for modeling tiered storage with RL.

Term	Definition
S / s / S_t	Set of states of an MDP / State / State at time t
A / A_t	Set of actions of an MDP / Action at time t
P	State transition probability matrix of an MDP
$P^a_{ss'}$	Transition probability from state s to s' by action a
R / R_t	Set of rewards of an MDP / Reward at time t
R^a_s	Future reward from state s taking action a
G_t	Return from time t
γ	Discount factor of an MDP
π	Policy, to determine action based on state
$v(s)$	State-value function
$q(s,a)$	Action-value function
C^i_j	Fuzzy categories of FRB function
$\mu^i_{C_j}(x_j)$	Membership function of category C^i_j
a_j, b_j	Hyperparameters of membership function
p^i	Trainable parameters of FRB function
λ	Trace decay parameter of TD(λ)
α^i_n	Learning rate of TD(λ) updates
$z_n(s)$	Eligibility trace

(P) capture the likelihood of moving from one state to another based on the actions, and rewards (R) are determined by the performance metrics such as reduced access latency or improved I/O efficiency. The migration problem is therefore translated to selecting appropriate migration actions (i.e., placing the right pages on fast tiers), which leads to better rewards (lower latency). By applying RL to this MDP framework, the system dynamically optimizes page placement, ensuring important pages reside in faster tiers while less important data is placed in slower, less costly storage, thus enhancing overall performance and cost-efficiency.

Basics and Nomenclature. RL trains intelligent agents to take *actions* in the MDP to achieve defined objectives. Formally, the MDP $< S, A, P, R, \gamma >$ is defined with S as the set of states with the Markov property (i.e., $\mathbb{P}(S_{t+1}|S_t) = \mathbb{P}(S_{t+1}|S_t, ..., S_1)$, t signifies time), A as the set of all actions, P as the state transition probability matrix with $P^a_{ss'} = \mathbb{P}(S_{t+1} = s'|S_t = s, A_t = a)$, R is the reward function with $R^a_s = \mathbb{E}(R_{t+1}|S_t = s, A_t = a)$, and γ is the discount factor used in defining the return G_t, which is the total discounted reward from time t: $G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{i=0}^{\infty} \gamma^i R_{t+1+i}$. A policy $\pi : S \rightarrow A$ is used to determine the next action based on the current state. A policy $\pi(a|s) = \mathbb{P}(A_t = a|S_t = s)$ is a distribution over actions given states, which defines the behavior of an agent. Therefore, the target of training an agent is to learn the best policy so that the agent takes correct action at each timestep, eventually attaining the objective. Table 3 shows in detail the nomenclature.

Value Functions. In RL, the objective is formulated as the return G_t, thus the training target of the agent becomes to maximize the return at each timestep. To mathematically express this target, state-value function $v_\pi(s)$ and action-value function $q_\pi(s,a)$ are defined, where $v_\pi(s) = \mathbb{E}_\pi(G_t|S_t = s)$ represents the expected return starting from state s then following policy π, and $q_\pi(s,a) = \mathbb{E}_\pi(G_t|S_t = s, A_t = a)$ is the expected return starting from state s, taking action a, and then following policy π. The optimal value function implies the best possible performance in the MDP [69], and an optimal policy can be found by maximizing over the optimal action-value function $q_*(s,a)$, that is, $\pi_*(a|s) = 1$ if $a = argmax_{a \in A} q_*(s,a)$.

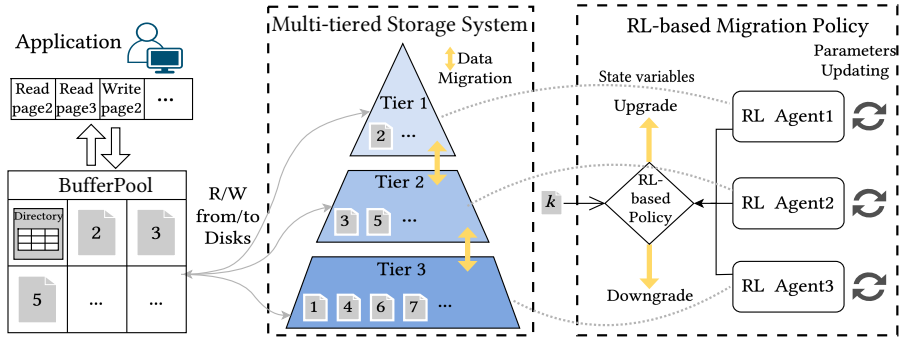


Fig. 3. Workflow of ReStore, where RL agents manage page migrations in a multi-tiered storage system. Upon an I/O request, the page is served from its current tier, while ReStore decides in the background whether migration is needed based on RL agents and state values.

5 ReStore Tiered Storage Management

We now build on the RL primer from Section 4 to describe in detail the core design of ReStore. This includes the specific formulation of our state-action space (§5.1), the construction of the reward function (§5.1) and value function (§5.2), the detailed mapping of page migration to the RL formulation (§5.3), the update process of the value function (§5.4), and the temperature model reflecting the page access patterns (§5.5).

5.1 States, Actions, and Rewards

We now formulate the multi-tiered storage data-migration problem using an RL-based framework. From a bird's-eye view, we design a cooperative multi-agent RL system in which each tier employs a single agent and all agents share a common reward objective. Figure 3 illustrates the workflow of our ReStore framework, in which data migration between neighboring tiers is coordinated by their respective agents toward a unified goal – minimizing overall system response time. This multi-agent design provides scalability, flexibility, and efficiency, with scalability arising from the ability to adapt to varying numbers of tiers by simply adding or removing RL agents, and efficiency stemming from the fact that only one agent needs to be updated when the status of a tier changes. To introduce our RL-based approach, we first configure the MDP environment:

- states: variables representing the current status of a tier;
- actions: decisions on which page transferred to where;
- rewards: cost signal as the negative of response time.

These components allow us to define an MDP for multi-tiered storage and solve it to find the best migration policy using RL.

States. Defining the states requires capturing key properties to accurately reflect both the workload and the underlying hardware status. In particular, the state variables should incorporate data access behavior, including access frequency and recency, as well as device-specific factors such as read/write asymmetry and concurrency. To capture these effects in a compact and tractable manner, we encode access frequency and recency through a temperature-based abstraction, where frequently and recently accessed pages naturally attain higher temperatures. Device characteristics and migration overheads are captured separately through a latency-related state variable. Based on these considerations, we define the following state variables:

s_1: the average temperature of all pages in the tier. It is calculated as $s_1 = \frac{1}{|K|} \sum_K temp(K)$, where $|K|$ is the total number of pages stored in the tier and $temp(K)$ is the temperature of page K,

representing a combined information about the access recency and frequency. The temperature model will be presented in detail later in Section 5.5. This variable indicates the percentage of hot/cold pages in the tier. A higher value of s_1 indicates a larger probability of pages in the tier being requested and, at the same time, a larger expected number of accesses.

s_2: the current accumulated latency in the tier. It is calculated as $s_2 = number\_of\_queued\_tasks \cdot avg\_process\_time$, where $avg\_process\_time = \frac{l_r}{k_r} + \frac{\alpha \cdot l_r}{k_w}$, l_r is the latency of read, α is the read/write asymmetry, and $k_r(k_w)$ is the read (write) concurrency. A higher value of s_2 denotes a longer waiting time in the tier, i.e., higher latency.

These state variables are used to represent the status of each tier at each timestep. They include page temperature, page access frequency, and concurrency potential, which are metrics that have a strong impact on storage systems. By including the specific temperature of the migration candidate page in the aggregate temperature of the storage tiers, the RL agent can make frequency– and recency–aware decisions. Note that this is a flexible framework, meaning that the state variables can also be defined in other ways depending on use cases [89, 90].

Actions. The actions model the page movements between tiers. Since each page movement changes the distribution of pages across tiers, we define actions for each page request. In more detail, for a request for page K residing in Tier i, there are two possible actions: $A_t = 0$ means no movement of the page, and $A_t = 1$ means upgrading the page to a faster tier j. In this definition, $A_t \in \{0, 1\}$ and $A = \{K, A_t\}$ where K is the page number.

Rewards. Reward $R_s^a = \mathbb{E}(R_{t+1}|S_t = s, A_t = a)$ is a function of state and action, representing the expectation of reward at the next timestep $t+1$. In our case, since the global objective is to minimize the overall response time, we define the reward as the negative of the estimated future system response time, so that the return $G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+1+i}$ represents the negative overall future system latency, and training the RL agent to maximize the return is equivalent to minimizing the system latency. More specifically,

$$
\begin{aligned}
R_s^a &= \mathbb{E}(R_{t+1}|S_t = s, A_t = a) \\
&= \begin{cases} \sum_{Tier} avg\_IO\_time \cdot \sum_{K \in Tier} \mathbb{E}(req_K), & \text{if } a = 1 \\ R_t, & \text{if } a = 0 \end{cases}
\end{aligned} \tag{1}
$$

where $\sum_{Tier}$ is the summation over all tiers, $avg\_IO\_time = w_r \cdot read\_time + w_w \cdot write\_time$ is the average I/O time of the tier, $\sum_{K \in Tier}$ is the summation over all pages in the tier, and $\mathbb{E}(req_K)$ is the expectation of future number of requests to the page. To approximate the future number of requests, we use the page temperature, i.e., $\mathbb{E}(req_K) \propto temp(K)$. By applying the law of large numbers and replacing average temperature of all pages in a tier $\frac{1}{|K|} \sum_K temp(K) = s_1$, the $\sum_{K \in Tier} \mathbb{E}(req_K)$ is then equal to $|K| \cdot exp(\frac{1+s_1}{2})$, where $|K|$ is the total number of pages in a tier and s_1 is the first state variable of that tier.

With the definitions of states, actions, and rewards, we formulate the data migration problem as an MDP. As mentioned earlier, RL solves MDPs by optimizing the value functions to provide the best action. We now discuss how value functions are used to solve the MDP. The state-value function $v_\pi(s)$ of an MDP is the expected return starting from states s following the policy π, $v_\pi(s) = \mathbb{E}_\pi(G_t|S_t = s)$. Since the reward is defined as the response time multiplied by -1, the expected return G_t represents the estimated future system response time. So, by maximizing the value function, we aim to find a policy that minimizes the future response time.

5.2 Value Function Representation

In simple reinforcement learning (RL) tasks with small discrete state spaces, value functions are often represented using lookup tables (e.g., Q-tables), where every state-action pair is explicitly stored and updated [73]. However, in real-world systems with large or continuous state spaces, such tabular representations become infeasible, necessitating the use of function approximation methods to generalize value estimates across states [37]. A range of function approximators exists, including linear models, kernel-based methods, decision trees, and neural networks [33, 44, 79]. Each alternative comes with its trade-off in terms of interpretability, training complexity, and sensitivity to parameter scaling. Neural networks and decision trees often require careful tuning, normalization, and regularization to remain stable and interpretable. We note that these factors can complicate deployment in resource-constrained or latency-sensitive environments.

To meet our design goal of having a lightweight, interpretable, and robust approximation model, we adopt the Fuzzy Rule-Based (FRB) function [9]. FRB offers an attractive balance: it is computationally efficient, supports continuous input spaces, and encodes domain knowledge through intuitive, human-readable rules. Importantly, it handles input scaling naturally through normalized membership functions and avoids overfitting thanks to its inherent smoothness and rule-based structure. The FRB function f maps the input vector $x \in \mathbb{R}^j$ to a scalar output y using a set of rules. A common form of a fuzzy rule is as follows:

$$\text{Rule } i\text{: IF } x_1 \subset C_1^i, x_2 \subset C_2^i, ..., x_j \subset C_j^i \text{ THEN } p^i,$$

where $x_1, ..., x_j$ are the components of input x, $C_1^i, ..., C_j^i$ are fuzzy categories, $x_j \subset C_j^i$ is a fuzzy rule, and p^i is the output parameter of this rule. The output of the rule-based function $f(x)$ is then a weighted average of p^i:

$$y = f(x) = \frac{\sum_{i=1}^{N} p^i w^i(x)}{\sum_{i=1}^{N} w^i(x)}, \tag{2}$$

where N is the number of rules, $w^i(x)$ is the weight of rule i computed by $w^i(x) = \prod_1^j \mu_{C_j^i}(x_j)$, and $\mu_{C_j^i}(x_j)$ is the membership function, which takes values in $[0, 1]$ and represent the measurement of an input variable x_j belonging to category C_j^i, i.e., a $\mu_{C_j^i}(x_j)$ value closer to 1 means x_j more likely belongs to C_j^i.

The FRB function is widely used for practical problems [18, 45]. It can model systems with varying complexity, ranging from a large number of complex rules to a few simple rules. In our case, the input x is the set of state variables (s_1, s_2) with different value ranges. To better characterize these state variables, we define two categories based on the relative magnitude of each variable: *Small* and *Large*, i.e., $C_j^i \in \{Small, Large\}$. Here, *Small* represents that the current value x_j is closer to the minimum of its range among all x_j, while *Large* indicates that x_j is closer to the maximum. As for the membership function, we adopt a sigmoid formulation:

$$\mu_L(x_j) = \frac{1}{1 + a_j e^{-b_j x_j}},\ \mu_S(x_j) = 1 - \mu_L(x_j). \tag{3}$$

where $\mu_{S/L}$ is the membership function of category *Small*/*Large*, and a_j, b_j are parameters. Figure 4 illustrates examples of the *Small* and *Large* membership functions under different values of a_j and b_j. To mitigate the influence of these parameters on biased membership values, we dynamically update them based on the statistical properties of the input variables. Specifically, we use the following formulas:

$$b_j = 10/range(x_j),\ a_j = e^{b_j avg(x_j)}, \tag{4}$$

where $range(x_j) = \max x_j - \min x_j$, $avg(x_j) = (\max x_j + \min x_j)/2$ is the average of x_j.

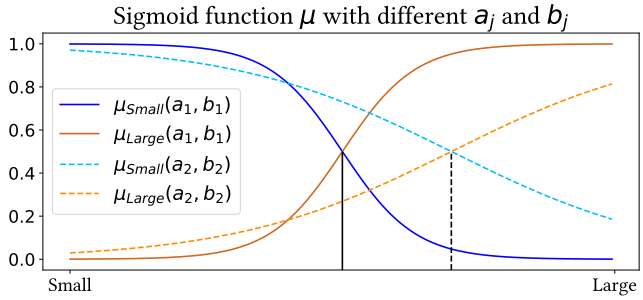


Fig. 4. Two examples of sigmoid functions (*Large* and *Small*). The function value indicates the degree to which an input belongs to each category. Varying the parameters a_j and b_j changes the exact shape and centroid of the functions.

The sigmoid formulation from Eq. (3) is well-suited for scenarios where input ranges lack constraints, and differences between outliers with very large values or very small values are considered less important than differences between centered data. The rationale for selecting this membership function lies in its ability to render the policy sensitive to minor system changes, such as moving a few pages, while remaining relatively robust against extensive, system-wide modifications at scale. Based on the two categories and the two state variables defined above, we form an FRB function with four rules as our value function. Note that the number of rules equals the number of categories to the power of the number of state variables, $2^2 = 4$ in our case. The value function is:

$$v(s) = \frac{\sum_{i=1}^{4} p^i w^i(s)}{\sum_{i=1}^{4} w^i(s)}, \tag{5}$$

where $s = (s_1, s_2)$ is the states variables, and $w^i(s) = \mu_{Small/Large}(s_1) \cdot \mu_{Small/Large}(s_2)$ is the multiplication of membership functions values at rule i.

5.3 Mapping the Page Migration problem

Having established the state variables and value function, we now use them to control page migration. For each page being requested, there are three scenarios:

- The page becomes hotter; it is upgraded to a faster tier.
- The page is not hot enough; it stays in the same tier.
- The page becomes hotter, but there is not enough space in the faster tier; the coldest page in the faster tier will be downgraded to make room for the new hotter page.

The data migration policy can be simplified as *deciding whether a page should be upgraded based on its properties and the current state of the involved tiers.* Accordingly, the action space A can be written as $\{0, 1\}$, where $A_t = 1$ indicates that the currently requested page should be upgraded to a faster tier, and $A_t = 0$ indicates that it should not. In this way, the action-value function $q(s, a)$ has two values for each s, which are $q(s, 0) = \mathbb{E}(G_t|S_t = s, A_t = 0) = \mathbb{E}(G_{t+1} = G_t|S_{t+1} = s) = v(s)$ ($A_t = 0$ means no migration and thus $S_{t+1} = S_t = s$), and $q(s, 1) = \mathbb{E}(G_t|S_t = s, A_t = 1) = \mathbb{E}(G_{t+1}|S_{t+1} = s')$ where s' is the next state due to the page migration decided by $A_t = 1$.

As indicated in Section 4, an optimal policy can be defined by maximizing over the optimal action-value function $q_*(s, a)$, where $q_*(s, a) = \max_{\pi} q_\pi(s, a)$. Therefore, finding the optimal policy is equivalent to maximizing the action-value function. Combining the expressions of the action-value function in the last paragraph, the RL-based migration policy can be formed in the following way:

For a requested page K currently in tier i, it is upgraded from tier i to tier $i + 1$ if

$$\begin{aligned} & q^i(s,1) + q^{i+1}(s,1) \geqslant q^i(s,0) + q^{i+1}(s,0) \\ \Leftrightarrow\ & v^i_{\text{up}}(s') + v^{i+1}_{\text{up}}(s') \geqslant v^i_{\text{not}}(s) + v^{i+1}_{\text{not}}(s) \end{aligned} \tag{6}$$

where $q^{i/i+1}(s, 1/0)$ denotes the action-value function of tier $i/i+1$ when taking action $A_t = 1/0$; $v^{i/i+1}_{\text{up}}$ is the value function of tier $i/i+1$ after page K is upgraded; s' represents the post-upgrade state; and $v^{i/i+1}_{\text{not}}$ and s correspond to the value function and state before upgrade, respectively. In this way, Eq. (6) indicates that taking the upgrading action $A_t = 1$ leads to an increase in the return G_t from tier i and $i+1$, implying a reduction in the overall system latency.

5.4 Updating Value Function

The policy described above is based on the inequality in Eq. (6), which depends on the state-value function $v(s)$. To ensure the policy makes effective decisions across changing states, the value function must be continuously updated as the system evolves. Our application falls into a specific category where only the state-value function needs to be learned, rather than the action-value function. This is because: (1) the same policy is used both for decision-making and for predicting future rewards, and (2) the action space is binary – either upgrade a page or not – making the action-value and state-value functions effectively equivalent (derived in the previous subsection 5.3). In RL, standard approaches for learning the state-value function include Monte Carlo methods and Temporal Difference (TD) learning [72]. Given the characteristics of our case, we adopt the on-policy TD approach TD(λ), which provides a flexible trade-off between Monte Carlo and one-step TD updates parameterized by λ, and enables efficient and stable learning of the value function during system transitions.

TD(λ) is a procedure for iteratively approximating the value function under a given policy. It updates the value function approximation $\hat{v}(s)$ as follows:

$$\begin{aligned} \hat{v}(s) &= \hat{v}(s) + \alpha_n(R_n + \gamma\hat{v}(s_{n+1}) - \hat{v}(s_n))z_n(s) \\ \gamma &= e^{-\beta\tau_n}, z_n(s) = \lambda e^{-\beta\tau_n} z_{n-1}(s) + \mathbb{1}(s = s_n) \end{aligned} \tag{7}$$

where $\hat{v}(s)$ is the approximation of $v(s)$, α_n is the learning rate at the n^{th} state, R_n is the rewards at state s_n, γ is the discounting factor (depending on the hyperparameter β), τ_n is the time the agent spends in state s_n, and z_n is the eligibility trace, which is initialized to 0 and updated by the formula above, and λ is the trace-decay parameter of TD(λ). Recalling Eq. (5), the cost function is actually a linear combination of basis functions with parameters p^i. Thus, we can rewrite the $v(s)$ in the form of $v(s,p) = \sum_{i=1}^{4} p^i\phi^i(s)$, where $\phi^i(s) = \frac{w^i(s)}{\sum_{i=1}^{4} w^i(s)}$. So, the updating of $v(s)$ in Eq. (7) using TD(λ) is equivalent to updating the parameters p^i. The updating formula can be rewritten for p^i as follows:

$$\begin{aligned} p^i_{n+1} &= p^i_n + \alpha^i_n(R_n + \gamma\hat{v}(s_{n+1}) - \hat{v}(s_n))z^i_n(s) \\ \alpha^i_n &= 0.1/(1 + 100\sum_{t=0}^{n-1}\phi^i(s_t)) \\ R_n &= (1/X_n)\sum_{i=1}^{X_n} r_i e^{-\beta(t_{n,i} - t_n)} \\ z^i_n(s) &= \lambda\gamma z^i_{n-1}(s) + \phi^i(s_n), \end{aligned} \tag{8}$$

where the learning rate α^i_n is initialized as 0.1 and decreases according to the accumulated basis function ϕ. The eligibility trace z^i_n is initialized as 0 and updated by Eq. (8). The reward R_n is given by the cost signal, where X_n is the total number of requests in state s_n, r_i is the response time of

Algorithm 1 RL-based Page Migration

Require: RL agents for each tier: $agent1, agent2, agent3$,
temperature changing model $temp\_change$
Input: Workload w={ <page number K, $Request$> }
for $K, Request$ in w **do**
Locate page K in Tier 1/2/3
if K in Tier1 **then**
Execute $Request$ on page K in Tier1
// No need to check for migration since page already in fastest tier.
else if K in Tier2/3 **then**
First execute $Request$ on page K in current Tier
// Meanwhile use RL-based policy to determine whether to upgrade page K to upper Tier
Calculate state variables of upper and current Tier before/after page upgrade:
$states\_t1\_be, states\_t2\_be,$
$states\_t1\_af, states\_t2\_af$.
Use RL agents to calculate cost signal:
$cost\_t1(2)\_be = agent1(2).cost\_phi(state\_vrs\_t1(2)\_be)$
$cost\_t1(2)\_af = agent1(2).cost\_phi(state\_vrs\_t1(2)\_af)$
$left = cost\_t1\_be + cost\_t2\_be, right = cost\_t1\_af + cost\_t2\_af$
if $left \leqslant right$ **then**
Upgrade page K to upper Tier
while Tier1 is full **do**
// Downgrade lowest temperature page in Tier1 to Tier2
end while
end if
// Update RL agents' parameters
$agent1.learn(), agent2.learn()$
end if
// Change temperatures of pages according to the temperature model
$temp\_change.step()$
end for

each request, $t_{n,i}$ is the arrival time of request i, and t_n is the time arrived at state s_n. This iteration process is proved to converge when the basis functions $\phi^i(s)$ are linearly independent [79].

5.5 Temperature Model

To capture frequency and recency information from the workload, we define a *temperature* model for each page to represent its hotness. It plays an essential role in both the definition of state variables and the action decisions given by Eq. (6). Specifically, we define a *hot-cold* function to represent these temperature changes, including an increasing process and a decreasing process, which are as follows:

- *Temperature increasing*: The temperature of a page rises as it receives more access requests. We describe this process using an exponential formula: $temp = 1 - \frac{0.5}{e^{\delta \cdot req}}$, where req is the total number of requests to the page in a time window, and δ is a tunable scaling parameter. The initial temperature of a page is then 0.5 when $req = 0$.
- *Temperature decreasing*: When a page is not accessed for a period of time, its temperature gradually decreases, reflecting its reduced importance and priority. We define the temperature decay rule as follows: *After a certain number of timesteps without any request, the page temperature decreases by a fixed amount* Δ_temp_drop *until it reaches a minimum threshold of* 0.01. The frequency threshold at which the temperature decreases, and the decay magnitude Δ_temp_drop, are determined by the workload duration and access distribution. Once a new request arrives, the temperature begins to rise again according to the temperature-increasing rule above.

This mechanism models the dynamic behavior of page temperature in response to access patterns. Figure 5 illustrates an example of this process, showing the temperature evolution of a page that experiences frequent requests during the intervals [10, 70] and [500, 700], with occasional accesses at timesteps 220 and 400, over a total of 1000 timesteps. The figure demonstrates that the temperature

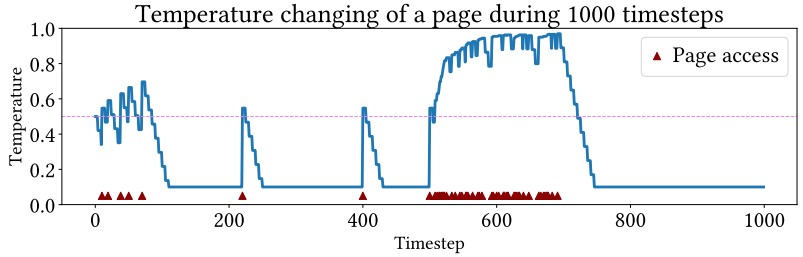


Fig. 5. Page temperature evolution when it receives frequent accesses during the intervals [10, 70] and [500, 700] over a total of 1000 timesteps.

Table 4. Upgrade/downgrade decision of different policies.

Policy	Upgrade Criteria	Downgrade
tLFU	Frequency	the LFU page
tLRU	Recency	the LRU page
LRFU	CRF value	the lowest CRF value page
EXD	EXD score	the lowest EXD score page
XGB	XGB prediction	the LRU page
Logi	Logi prediction	the LRU page
TEMP	Temperature	the lowest temperature page
ReStore	RL condition (Eq.6)	the lowest temperature page

model effectively captures the page access characteristics, maintaining higher temperatures during high-frequency access periods and dropping to lower values when the page becomes cold.

While the temperature model parameters can be calibrated to specific workloads, we find that ReStore is robust to a wide range of values. For instance, in our experiments, the temperature increment parameter δ is typically set between 0.05 and 0.2 depending on the average page access frequency, ensuring that the temperature increases gradually, which filters out noise from occasional, "one-hit" accesses while amplifying the signal for truly hot pages. For the cooling mechanism, we set the decay frequency to $\frac{1}{10}$ of the total page count and the decay magnitude to 0.02. This configuration provides a moderate cooling rate that prevents the state space from saturating while ensuring that the temperature of inactive pages quickly drops to 0. We observed that minor variations in these parameters do not significantly impact the RL agent's convergence, as the agent learns to adapt its policy to the resulting temperature distribution. Algorithm 1 presents the pseudo-code of our method.

6 Experimental Evaluation

This section evaluates ReStore and baseline policies, showing that ReStore outperforms traditional methods in performance, migration overhead, adaptability, and scalability.

6.1 Experimental Methodology

We design a comprehensive set of experiments with both classical and advanced data migration policies to thoroughly evaluate the effectiveness of ReStore. The performance of each policy is evaluated in experiments across both synthetic workloads and workloads adapted from real traces, with different characteristics, including access skew, read/write imbalance, and changing patterns. These workloads consist of "post-cache" requests, representing the I/Os that reach storage devices after being filtered by the originating system's buffer pool. We also test on different device properties, such as different capacity, latency, concurrency, and asymmetry.

Table 5. FEMU SSD configurations. Each configuration group defines different read/write latencies and concurrency levels to emulate a range of device classes.

Tier1/2/3	**default**	**config1**	**config2**	**config3**
Read latency (μs)	30/200/500	7/200/500	30/200/500	7/200/500
R/W asymmetry	1.5/2/4	1.5/2/4	1.5/2/4	1.5/2/4
Concurrency	8/4/2	8/8/2	8/8/2	8/4/2

Data Migration Policies Tested. To evaluate ReStore, we compare it with eight baseline policies: (i) tLFU (tiered LFU), (ii) tLRU (tiered LRU), (iii) EXD [22], (iv) LRFU [39], (v) XGBoost [28], (vi) Logistic Regression (Logi), (vii) TEMP (migration policy based on the *temperature* model in Section 5.5), and (viii) Sibyl [71]. XGBoost and Logi are ML-based methods adapted from prior work for page-level operations. Both approaches track the five most recent accesses to each page, using access intervals as input features. The class label is binary: 1 if the page is accessed again within a fixed future window, and 0 otherwise. Their models are trained on the first 5% of the workload trace, chosen for training-time and memory feasibility, and then used as migration policies to predict future accesses. TEMP is a feature-based policy similar to LRFU and EXD but solely based on our temperature model. It serves as an ablation of ReStore considering only the temperature-featured s_1. By comparing TEMP with ReStore, we isolate the performance gain provided by the device-aware state variable s_2. Sibyl is an RL-based method for block-level tiering using neural network-based value function approximation. We use its public implementation [70] under the same storage configuration. For static workloads, we also define an *Oracle* policy as the optimal baseline when the workload pattern is known in advance. Note that the Oracle policy uses future-known static access frequencies, making a single initially optimal placement but performing no migrations afterward. For instance, in skewed workloads, *Oracle* places frequently accessed pages in faster tiers (Tier 1, or partially in Tier 2) and the rest in slower tiers (Tier 2 and 3). The upgrade/downgrade mechanisms of all tested policies are listed in Table 4.

Tiered Storage System Simulation. To emulate different multi-tier storage configurations, we develop a C++-based framework that supports varying the number of tiers, device characteristics, and migration policies. Each tier maintains information about the tier's maximum capacity, read/write latency, and concurrency. A hash table of `<page_id, page_metadata, location>` tracks page distribution, where `page_metadata` includes features such as recency, frequency, and temperature used by each policy. On each access, the policy's upgrade/downgrade criteria (Table 4) determine migrations between tiers. Each tier maintains a min-heap to identify pages for downgrading based on the policy.

Tier Properties. To evaluate ReStore across diverse hardware environments, we use three setups: our multi-tiered storage emulator, FEMU, and cloud-based storage instances. All setups follow a three-tier storage system, where Tier 1 has 1%/3%/6%/12% of total capacity, Tier 2 has 4%/8%/16%, and Tier 3 covers the full 100% capacity. *The storage emulation framework* uses the default configuration shown in Table 5. This setup simulates the fast, medium, and slow tiers based on real-world SSD performances. Tier 1 has read latency 30μs and read/write asymmetry 1.5 (i.e., write latency 45μs); Tier 2 is configured with read latency 200μs and asymmetry 2.0; and Tier 3 with read latency 500μs and asymmetry 4.0. The concurrency of fast, medium, and slow tiers is set to 8, 4, and 2, respectively. We also experiment with varying levels of concurrency and asymmetry, as well as without asymmetry (to simulate HDDs) for completeness. *FEMU* [40] is used to evaluate ReStore across a wide range of SSD architectures. It supports fine-grained configuration of SSD geometry (i.e., channels, chips, dies, and planes) as well as page-level read, program, and erase latencies. From the host perspective, FEMU devices appear as standard NVMe SSDs, allowing applications to interact with emulated devices as if they are physical drives [40]. We define four configuration groups

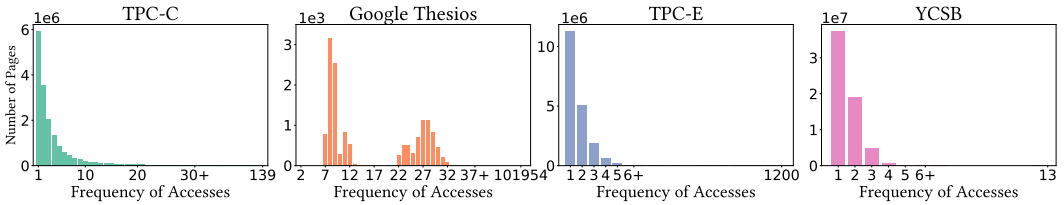


Fig. 6. Distribution of page access frequencies in standard benchmarks and real-world workload traces.

to assess robustness, covering a large variety of SSDs. Detailed settings are provided in Table 5. Additionally, we run experiments on cloud-based volumes to validate ReStore on real storage devices. We use three attached storage volumes configured with different read/write speeds via OpenStack QoS (`read_iops_sec` and `write_iops_sec`) parameters, serving as the fast, medium, and slow tiers. Specifically, for fast tier we set `read_iops_sec` to 33,333 and `write_iops_sec` to 22,222, to approximate the read/write latencies used in emulation (30μs / 45μs). For the medium and slow tiers, `read_iops_sec` and `write_iops_sec` are set to (5000, 2500) and (2000, 500), respectively.

Static Workloads. We test three static workloads with varying skew: *5hf90*, *10hf90*, and *10hf80*. The notation *xhfy* represents that *y%* of the accesses are performed on *x%* high-frequent (*hf*) pages. *5hf90* and *10hf90* differ in access concentration, while *10hf80* has lower skew. To isolate the effects of access skew, different high-frequency pages are used for *5hf90* and *10hf90*, but the same pages are used for *10hf90* and *10hf80*. Unless otherwise mentioned, these workloads have 50% reads and 50% writes, and they consist of 1,000,000 operations and 10,000 unique pages. We also test system scalability with increased page counts and number of operations.

Dynamic Workloads. To evaluate the performance of different policies under workload drift, we introduce four dynamic workloads: A (*5hf90 + 10hf80*), B (*5hf90+10hf80+20hf80*), C (*5hf90*10*), and D (*5hf90*50*). These workloads simulate shifting access patterns by combining multiple phases. For example, Workload A consists of a first phase with the *5hf90* pattern, followed by a second phase with the *10hf80* pattern. Further, *5hf90*n* refers to a workload with *n* phases, where each phase has a *5hf90* skew on a different 5% of the pages. All synthetic workloads are summarized in Table 6.

Table 6. Our synthetic workloads.

Name	Type	Description
5hf90	static	Skewed 90% access on 5% pages
10hf90	static	Skewed 90% access on 10% pages
10hf80	static	Skewed 80% access on 10% pages
A	dynamic	*5hf90 + 10hf80*
B	dynamic	*5hf90 + 10hf80 + 20hf80*
C	dynamic	10 phases of *5hf90*
D	dynamic	50 phases of *5hf90*

Table 7. Standard benchmarks and real-world traces used.

Trace Name	#unique_pages	#operations	Read/Write (%)
TPC-C	16842331	62925641	77.5%/22.5%
TPC-E	19274163	31884418	87.9%/12.0%
YCSB	61960292	92728323	99.6%/0.3%
Google Thesios	16417	95970071	59.7%/40.3%
MSR_hm_1	51733	2308560	4.7%/95.3%
MSR_wdev_0	79677	695995	71.8%/28.2%

Real-world Workloads. To evaluate the robustness of each policy, we use real-world I/O traces from well-known benchmarks, such as including TPC-C (8 traces from 14000 warehouses and 300 users) [35, 77], TPC-E (6 traces from 200,000 customers) [35, 78], and YCSB [84, 85]. The YCSB trace is generated using Workload A on a 250M-record RocksDB instance with 1B operations, collected on a 28-core server with 128 GB RAM and two NVMe SSDs using *blktrace*. We also include Google Thesios [59] and two MSR traces (hm_1, wdev_0) [49]. The Thesios traces are synthesized from Google production HDD workloads using the sampling-and-combination method, which reconstructs representative full-resolution traces by combining sampled I/O from similar entities. The MSR Cambridge traces include raw I/O activity from 36 volumes across 13 enterprise servers. These traces are replayed as collected and converted into page-level workloads, assuming

a 4KB page size. Each trace contains various access patterns, page sizes, and levels of skew. Table 7 provides a high level description, while Figure 6 visualizes the distribution of access frequencies.

6.2 Experimental Analysis

We now present a detailed experimental analysis of ReStore's performance across various scenarios. Unless stated otherwise, the experiments use our multi-tiered emulation framework by default.

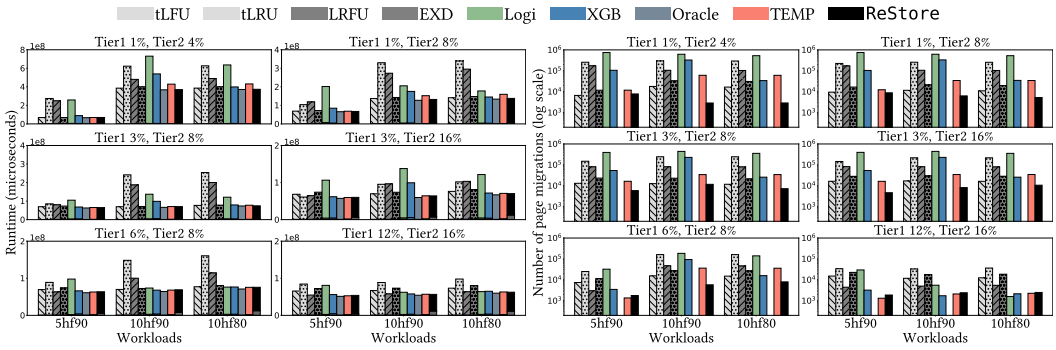
ReStore Outperforms All Baselines in Static Workloads. In our first set of experiments, we evaluate the performance of each policy for the static workloads (*5hf90*, *10hf90*, *10hf80*) under different tier capacity configurations. Figure 7a presents the overall latency of each workload for each policy as tier capacities change. We observe that latency decreases as the size of the faster tier increases, i.e., the top row of Figure 7a (small fast tier) has the highest latency, while the bottom row (large fast tier) has the lowest. The figure also shows that ReStore achieves the overall best performance, for instance, ReStore achieves 5%, 40%, 23%, 7%, 49%, 31%, and 13% lower runtime compared to tLFU, tLRU, LRFU, EXD, Logi, XGB, and TEMP for *10hf90* workload with 1% Tier 1 and 4% Tier 2 capacity (middle group of bars in the top left sub-figure of Figure 7a). ReStore demonstrates stable performance trends similar to the optimal *Oracle* policy as tier capacity increases.

The figure also shows that tLFU performs well under such skewed workloads, as its frequency-based decisions enable it to effectively capture long-term access patterns in the workloads. We also observe that TEMP works particularly well when the faster tiers have higher capacities, whereas EXD's performance does not depend on the tier capacity too much. This suggests that while the *temperature* model is highly efficient at identifying *hf* pages, it lacks resource optimization constraints and is more sensitive to workload variations than the EXD score. The ML-based policies, Logi and XGB, perform very well when fast tier capacities are large, indicating that ML models are highly accurate in predicting *hf* pages – Logistic Regression provides solid accuracy, while XGBoost performs even better. However, due to their rigid prediction-driven mechanism, these policies trigger a significantly higher number of page migrations when fast tier capacity is limited (as shown in Figure 7b). We further notice that tLRU and LRFU have relatively worse performance in terms of runtime in these experiments, which is because recency-based policies incur a lot of data migrations in such skewed *hf* workloads that adversely affect the overall runtime.

The latency bars of the figure also include CPU time (lower part of each bar, detailed numbers can also be found in Table 8), stacked with data access time. While ReStore has a higher CPU time compared to other feature-based policies because of the high computation cost of RL updating, *ReStore still outperforms other baselines thanks to its optimal data placement decision based on both workload and device properties*. This, in turn, causes fewer migrations compared to other policies.

ReStore Improves Performance through Fewer Migrations. Migration is a costly process involving both an upgrade and a downgrade operation. Figure 7b presents the total number of page migrations (on a logarithmic scale for better visualization) for each policy from the previous experiment. Each bar reflects the combined number of migrations from Tier 3 to Tier 2 and from Tier 2 to Tier 1. We observe that tLRU and LRFU incur the highest number of migrations, indicating their inability to capture workload skewness. Logi and XGB result in even more migrations under capacity-limited scenarios, highlighting a core limitation of ML-based policies in such cases. In contrast, ReStore consistently leads to significantly fewer migrations than all other policies – ReStore reduces migrations by up to 8×, 48×, 19×, 7×, 53×, 27× and 21× compared to tLFU, tLRU, LRFU, EXD, Logi, XGB, and TEMP, respectively. This demonstrates that ReStore improves the overall performance via minimizing migration overhead through optimal data placement.

ReStore can Adapt to Workload Drift. In this set of experiments, we evaluate ReStore and other policies under dynamic workloads where the access patterns drift recurrently. Figure 8



(a) Overall runtime (CPU time + data access time) under different tier capacity configurations and static workloads.

(b) Number of total page migrations (in logarithm scale), both from Tier 3 to Tier 2 and from Tier 2 to Tier 1.

Fig. 7. (a) In general, ReStore outperforms the other baseline policies. Frequency-based policy like tLFU performs well for this type of skewed workload. Recency-based policy like tLRU struggles to capture the long-term *hf* workload. (b) For such workloads, tLRU, LRFU, and ML policies cause a very high number of migrations, whereas ReStore causes the least.

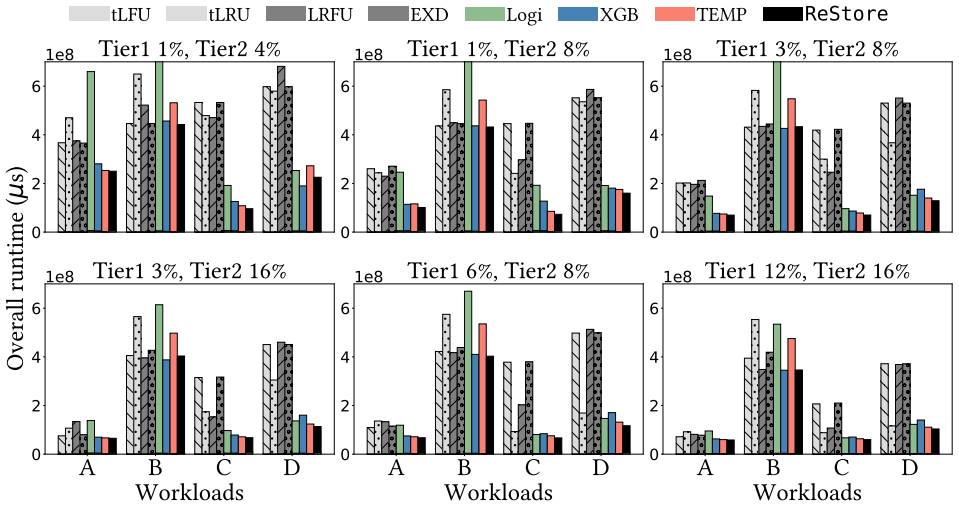


Fig. 8. Total runtime under dynamic workloads. ReStore significantly outperforms other baselines with the greatest benefit in high drift scenarios (Workloads C & D).

presents the performance evaluation, which shows that ReStore consistently outperforms other policies, particularly in scenarios where access patterns change more frequently. For example, for 3% Tier 1 and 8% Tier 2 (mid-left subfigure of Figure 8), ReStore achieves up to 2.81×, 1.03×, 3.51×, 4.28× speedup compared to LRFU (the next best rule-based policy) for workload A, B, C, and D, respectively. The performance trend remains similar for different tier capacities across different workloads. We observe that ReStore has the highest performance improvement for workloads C and D, which includes the highest amount of workload drift, showcasing ReStore's adaptability. Overall, ReStore achieves up to 6.1×, 5.3×, 4.9×, 6.0×, 2.1×, 1.6×, and 1.4× speedup compared to

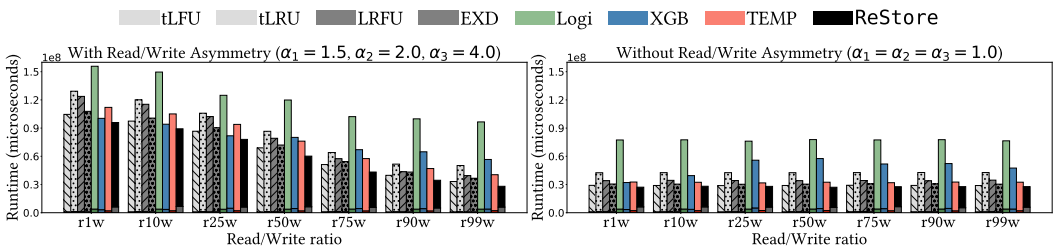


Fig. 9. Performance under varying read/write ratio with asymmetry (top) and without asymmetry (bottom) for *5hf90* workload with 3% Tier 1 and 8% Tier 2 capacity. ReStore delivers stable performance for any read/write ratios, with performance improvement independent of asymmetry.

tLFU, tLRU, LRFU, EXD, Logi, XGB, and TEMP, respectively; and 1.19×–1.88× on average. We also observe that recency-based policies, such as tLRU, show a marked improvement compared to their performance on static workloads, reflecting their adaptability to short-term changes. On the other hand, tLFU, which performs well on static workloads, struggles with workload drift but remains competitive. TEMP and EXD exhibit higher robustness than LFU, with TEMP showing its strength for frequently changing workloads (workloads C and D). XGB and Logi show improved performance compared to static workloads, especially under high-drift conditions, which highlights the generalizability of ML models. However, they incur over 10× more page migrations (on average 13.2× for XGB, and 20.7× for Logi) compared to ReStore to attain similar performances.

ReStore is Robust for Different Read/Write Ratios. We now evaluate ReStore and other policies as we vary the read/write ratio of the *5hf90* workload under both symmetric and asymmetric device configurations. Figure 9 presents the performance of different policies where the x-axis r1w-r99w, represent workloads with 1%-99% reads. The top figure shows the total runtime on devices with read/write asymmetry (e.g., SSDs), while the bottom shows performance on devices without asymmetry (e.g., HDDs). As expected, Figure 9 shows that in an asymmetric environment, latency increases with more writes. ReStore outperforms other baselines for all read/write ratios, demonstrating its stability. A careful look at both figures also reveals ReStore's resiliency to device asymmetry, maintaining stable performance across different read/write ratios, unlike other policies that exhibit higher latency in asymmetric conditions. This is because ReStore considers the asymmetry in its decision-making, which is crucial in heterogeneous storage environments.

ReStore Scales with Data Size. This set of experiments demonstrates the scalability of each policy as we increase the size of the *5hf90* workload by orders of magnitude. Specifically, we increase the number of pages from 10,000 to 10 million (a three-order magnitude increase) and the number of operations from 1 million to 100 million (a two-order magnitude increase). Figure 10 shows the normalized latency of each policy relative to the *Oracle* policy. We observe that ReStore scales effectively with data size. Specifically, its normalized latency increases slightly, from 1.03 to 1.12, as both the number of pages and operations grow significantly (first group of bars and fourth group of bars). In contrast, LFU experiences an increase in normalized latency from 1.1 to 1.45, LRU from 1.35 to 1.55, LRFU from 1.04 to 1.27, EXD from 1.16 to 1.53, XGB from 1.08 to 1.44, and TEMP from 1.04 to 1.28. Meanwhile, the performance of the Logistic Regression-based policy degrades significantly, primarily due to an explosively increasing number of page migrations. These results highlight ReStore's ability to handle large-scale workloads while maintaining predictable performance.

ReStore is Robust across Different Devices. We further test robustness across hardware configurations by varying device speeds. Specifically, we set the read latencies for Tiers 1, 2, 3 to (30, 200, 500), (30, 200, 2000), and (30, 2000, 5000) μs to simulate different SSD/HDD combinations, and re-run the *5hf90* workload under the same capacity setups as in Figure 7a. We observe consistent

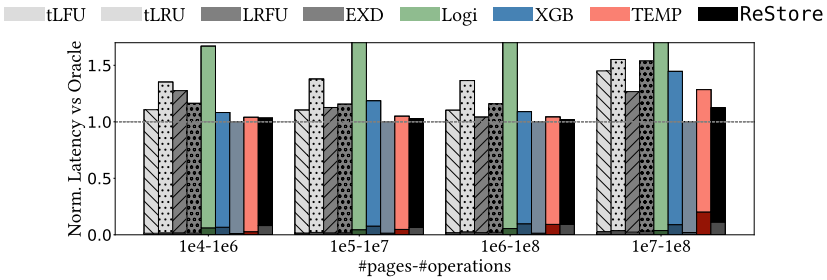


Fig. 10. Normalized latency relative to *Oracle* of each policy as #pages and #operations increases for *5hf90* workload with 3% Tier 1 and 8% Tier 2 capacity. ReStore scales effectively with the data size.

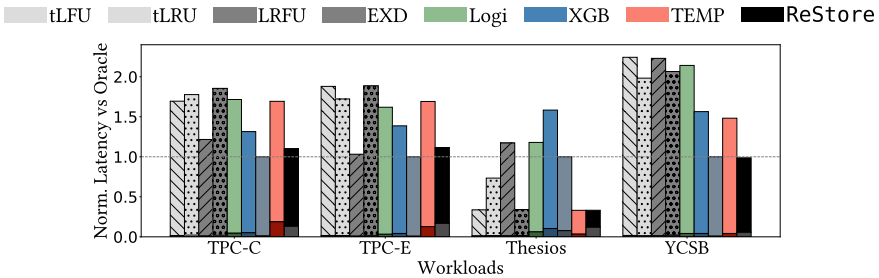


Fig. 11. ReStore achieves the lowest normalized latency (relative to *Oracle*) under standard benchmarks and real-world traces, consistently outperforming all other baselines.

trends – for example, ReStore is 1.4× faster than tLRU and 1.1× faster than XGB – indicating the stable performance of ReStore regardless of device characteristics. We exclude the figure for brevity.

ReStore Excels for Real-World Workloads. We evaluate ReStore and other policies using standard benchmarks like TPC-C [77], TPC-E [78], YCSB [16, 84], and Thesios [59], whose workload characteristics are detailed in Section 6.1. Given the varying number of pages in these benchmarks, we use different tier capacity settings: for TPC-C and TPC-E, the maximal capacity of Tier 1, Tier 2 and Tier 3 is 2M, 8M, and 20M pages; for Thesios, capacity of Tier 1, Tier 2 and Tier 3 is 1K, 5K, and 20K pages; for YCSB, Tier 1, Tier 2 and Tier 3 capacity is 10M, 20M, and 70M pages. Since the number of unique pages and the number of requests differ, the runtimes vary significantly across workloads. Therefore, we present in Figure 11 the normalized latency of each policy relative to the *Oracle*. The figure shows that ReStore consistently outperforms other policies across all workloads. Notably, ReStore can surpass the static *Oracle* in high-drift scenarios because its continuous adaptation outweighs the benefits of a fixed, albeit initially optimal placement, demonstrating ReStore's effectiveness for real-world scenarios. More precisely, ReStore achieves 2.3×, 2×, 2.2×, 2.1×, 2.2×, 1.6× and 1.5× lower runtime compared to tLFU, tLRU, LRFU, EXD, Logi, XGB, and TEMP for the YCSB workload, and similarly maintains superior performance across other workloads. We further observe that LFU performs well for Thesios, which closely resembles our static skewed workloads. While the performance differences between recency- and frequency-based policies depend on specific traces, ReStore maintains a clear advantage by dynamically adapting to diverse workload patterns.

Comparison with Sibyl. For completeness, we compare against Sibyl [71], the only RL-based data placement approach for multi-tiered storage. While all policies in our study operate at the page-level granularity using our multi-tiered storage emulator, Sibyl migrates groups of pages. Figure 12 compares Sibyl with ReStore and the baseline policies using MSR_hm_1 and MSR_wdev_0 traces (originally used in the evaluation of Sibyl). We run experiments on FEMU [40] with the

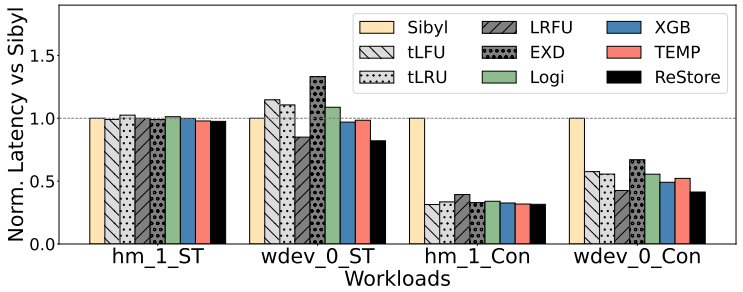


Fig. 12. Normalized latency of each policy relative to Sibyl on MSR workloads. Sibyl, which uses a single thread, underperforms compared to other approaches that exploit parallelism. ReStore consistently achieves the lowest latency.

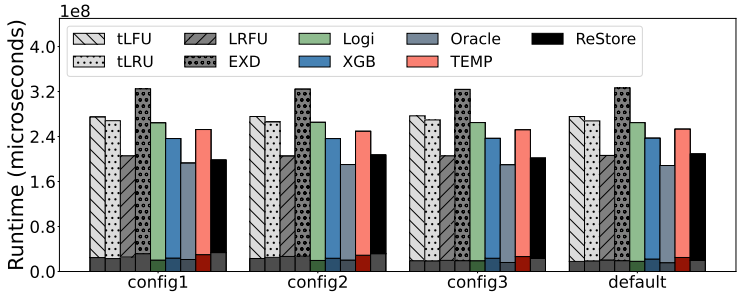


Fig. 13. Performance of various policies with different storage setups (details in Table 5). Performance is stable for all policies, with ReStore consistently achieving the best results.

default configuration as specified in Table 5. Each group of bars represents the normalized latencies against Sibyl. To match Sibyl's single-threaded setup, we first enforce all policies to use a single thread. The results under this condition are shown in the first two bar groups: hm_1_ST (single thread) and wdev_0_ST. We then allow requests to be grouped for ideal concurrency and re-run the same workloads, with the corresponding results reported in the latter two bar groups (denoted with the '_Con' suffix). These results show that Sibyl can reduce I/O latency through improved data placement using its RL agent, outperforming some baseline policies. However, its lack of concurrency awareness leads to significant under-utilization of the storage devices. Furthermore, as reported in Table 1, Sibyl requires an extensive pre-training phase – over 10× longer than the training times for XGB and Logi – to achieve satisfactory results. In contrast, ReStore consistently outperforms all other policies across both workloads, demonstrating both adaptability and efficiency.

ReStore is Robust across Diverse FEMU SSDs. Figure 13 presents the overall response time of multiple policies under different FEMU setups when running the MSR_wdev_0 workload. The performance trends of all policies remain consistent with those observed in the default configuration, demonstrating the robustness of these methods to varying device characteristics. Notably, ReStore consistently achieves the lowest response time across all configurations.

Extensibility to Different Tier Numbers. To demonstrate that ReStore is naturally extensible to various storage architectures beyond a 3-tiered setup, we tested a 2-tiered setup using the default Tier 1 configuration (30μs read, 45μs write, 1%–12% capacity) and the previous Tier 3 as the new archival Tier 2. Figure 14 shows that ReStore consistently performs the best across both static and dynamic workloads, mirroring the trends of the 3-tiered experiments. This confirms that ReStore's agent-per-tier architecture is robust and scales seamlessly across different storage hierarchies.

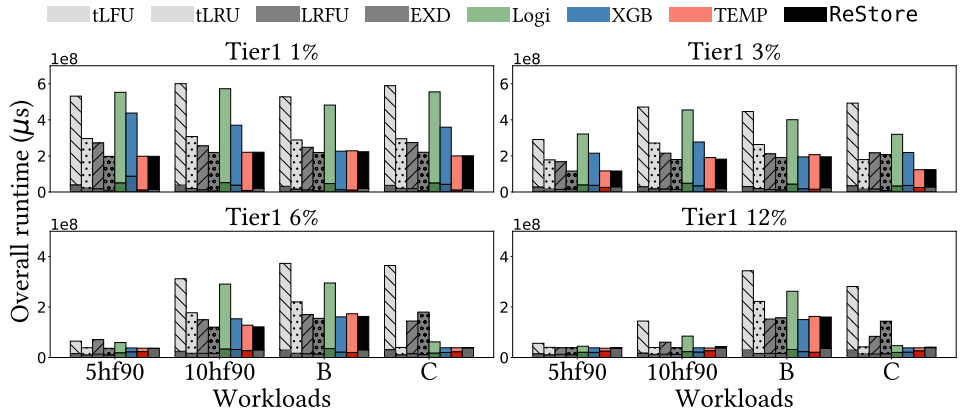


Fig. 14. Performance comparison in a 2-tiered configuration using both static and dynamic workloads. The results demonstrate ReStore's extensibility, showing that its performance advantages and the trends relative to other baselines remain consistent across different tier counts.

Experiment on Real Cloud-based Storage. To further evaluate the robustness of ReStore and baseline policies, we conduct experiments on the cloud server. We run selected workloads: *10hf90*, MSR_hm_1, and Google Thesios on the three configured volumes. Figure 15 shows that ReStore outperforms the baseline approaches, demonstrating its robustness for cloud-based execution. Notably, the ML-based policies (XGB and Logi) perform significantly worse, primarily due to their excessive migration operations – where the additional cost of data transfer between cloud volumes, amplified by network bandwidth limitations, leads to increased latency.

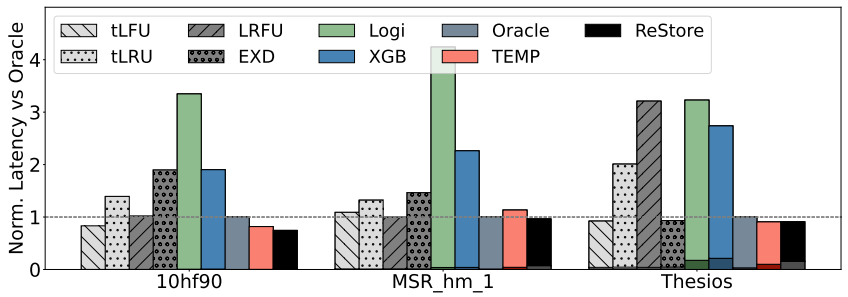


Fig. 15. ReStore outperforms baseline policies on real devices in a cloud server with three QoS-configured storage tiers, showing trends consistent with the emulator results.

CPU and Memory Overhead. While ReStore achieves good runtime improvements, it comes with an increased CPU cost. Compared to the *Oracle* policy, LFU and LRU require extra memory to track page frequency or recency and maintain a heap for the least-used pages. LRFU and EXD also use heaps to record their CRF values or EXD scores. TEMP keeps a temperature model per page (as described in §5) with a heap to locate the coldest page. ReStore adds slightly more overhead than TEMP due to its multiple RL agents. Table 8 reports the average CPU time per access and memory usage for each policy on the *5hf90* workload. XGB shows the highest overhead because of its complex model, while ReStore also has moderately high costs. However, as seen in Figures 7-13, ReStore's performance gains far outweigh these extra CPU and memory requirements, making the overhead negligible.

Table 8. CPU time and memory overhead of each policy.

	Oracle	tLFU	tLRU	LRFU	EXD	Logi	XGB	TEMP	ReStore
CPU (μs)	0.701	0.734	0.899	0.798	0.802	2.117	4.013	1.860	4.358
Mem. (MB)	1.919	2.249	2.280	2.303	2.344	2.755	2.907	2.136	2.349

7 Conclusion

Data migration remains a core challenge in multi-tiered storage systems. Existing methods either do not balance performance gain and migration cost or fail to consider device-specific characteristics. Leveraging reinforcement learning's strengths in flexible state definitions and dynamic policy adjustment, we present ReStore, an efficient data migration solution for multi-tiered storage. Through carefully designed state variables, ReStore considers both workload and storage device properties, including read/write asymmetry and concurrency. While our temperature-only baseline (TEMP) offers a lightweight solution for simpler settings, the full RL formulation in ReStore bridges the gap between workload patterns and hardware constraints, as well as provides the necessary adaptivity to changes in the environment, resulting in more stable improvements under a wide range of configurations. We demonstrate ReStore's effectiveness through experiments on workloads of varying sizes, access patterns, and different device configurations. ReStore outperforms other baselines, including the temperature-only approach, under skewed workloads, workload drift, and varying read/write ratios, while also scaling effectively, demonstrating its adaptivity and robustness.

Acknowledgments

This work is funded by the Swedish Foundation for Strategic Research under Grant No. BD15-0008, the Kjell and Märta Beijer Foundation, the US National Science Foundation under Grants No. IIS-2144547 and CCF-2403012, a Facebook Faculty Research Award, and a Meta Gift. We would also like to acknowledge the National Academic Infrastructure for Supercomputing in Sweden for providing computing resources, which is partially funded by the Swedish Research Council through grant agreement No. 2022-06725.

References

[1] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 147–160. http://dl.acm.org/citation.cfm?id=1376616.1376634

[2] Rizik Al-Sayyed, Hussam N. Fakhouri, Ali Rodan, and Colin Pattinson. 2017. Polar Particle Swarm Algorithm for Solving Cloud Data Migration Optimization Problem. *Modern Applied Science* 11, 8 (2017), 20. https://www.ccsenet.org/journal/index.php/mas/article/view/68060

[3] Waleed Ali, Sarina Sulaiman, and Norbahiah Ahmad. 2014. Performance improvement of least-recently-used policy in web proxy cache replacement using supervised machine learning. *International Journal of Advances in Soft Computing and Its Applications* 6, 1 (2014).

[4] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. 1994. The Uniform Memory Hierarchy Model of Computation. *Algorithmica* 12, 2/3 (1994), 72–109. https://doi.org/10.1007/BF01185206

[5] Malak Alshawabkeh, Alma Riska, Adnan Sahin, and Motasem Awwad. 2012. Automated Storage Tiering Using Markov Chain Correlation Based Clustering. In *Proceedings of the International Conference on Machine Learning and Applications (ICMLA)*. 392–397. https://doi.org/10.1109/ICMLA.2012.71

[6] Raja Appuswamy, Renata Borovica-Gajic, Goetz Graefe, and Anastasia Ailamaki. 2017. The Five minute Rule Thirty Years Later and its Impact on the Storage Hierarchy. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*.

[7] Martin F. Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. 2000. Evaluating content management techniques for Web proxy caches. *SIGMETRICS Performance Evaluation Review* 27, 4 (2000), 3–11. https://doi.org/10.1145/346000.346003

[8] Teona Bagashvili, Tarikul Islam Papon, and Manos Athanassoulis. 2025. ACE-in-Action: A Smart DBMS Bufferpool for SSDs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 23–26. https://doi.org/10.

1145/3722212.3725077

[9] Hamid Reza Berenji. 1994. Fuzzy Q-learning: a new approach for fuzzy dynamic programming. In *Proceedings of IEEE International Fuzzy Systems Conference*. 486–491. https://ieeexplore.ieee.org/document/343737

[10] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, and Julian Shun. 2016. Efficient Algorithms with Asymmetric Read and Write Costs. In *Proceedings of the Annual European Symposium on Algorithms (ESA)*. 14:1–14:18.

[11] Zheng Chang, Lei Lei, Zhenyu Zhou, Shiwen Mao, and Tapani Ristaniemi. 2018. Learn to Cache: Machine Learning for Network Edge Caching in the Big Data Era. *IEEE Wireless Communications* 25, 3 (2018), 28–35. https://doi.org/10.1109/MWC.2018.1700317

[12] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Transactions on Storage (TOS)* 12, 3 (2016), 13:1–13:39.

[13] Jinchuan Chen, Yueguo Chen, Xiaoyong Du, Cuiping Li, Jiaheng Lu, Suyun Zhao, and Xuan Zhou. 2013. Big data challenge: a data management perspective. *Frontiers of computer Science* 7 (2013), 157–164.

[14] Peng Cheng, Yutong Lu, Yunfei Du, Zhiguang Chen, and Yang Liu. 2019. Optimizing Data Placement on Hierarchical Storage Architecture via Machine Learning. In *Proceedings of the IFIP International Conference on Network and Parallel Computing (NPC)*. 289–302. https://doi.org/10.1007/978-3-030-30709-7%5C_23

[15] Edward I Cohen, Gary M King, and James T Brady. 1989. Storage Hierarchies. *IBM Systems Journal* 28, 1 (1989), 62–76. https://doi.org/10.1147/sj.281.0062

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

[17] Michael Cornwell. 2012. Anatomy of a Solid-State Drive. *Communications of the ACM (CACM)* 55, 12 (2012), 59–63.

[18] Robert Czabanski, Michal Jezewski, and Jacek M. Leski. 2017. Introduction to Fuzzy Systems. In *Theory and Applications of Ordered Fuzzy Numbers - A Tribute to Professor Witold Kosiński*. 23–43. https://link.springer.com/chapter/10.1007/978-3-319-59614-3_2

[19] Dominic Davies-Tagg, Ashiq Anjum, Ali Zahir, Lu Liu, Muhammad Usman Yaseen, and Nick Antonopoulos. 2024. Data Temperature Informed Streaming for Optimising Large-Scale Multi-Tiered Storage. *Big Data Mining and Analytics* 7, 2 (2024), 371–398. https://ieeexplore.ieee.org/document/10506769

[20] Ahmed Eldawy, Justin J Levandoski, and Per-Åke Larson. 2014. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *Proceedings of the VLDB Endowment* 7, 11 (2014), 931–942. http://www.vldb.org/pvldb/vol7/p931-eldawy.pdf

[21] IBM Storage FlashSystem C family. 2025. Easy Tier. https://www.ibm.com/docs/en/flashsystem-c200/9.1.1?topic=pools-easy-tier.

[22] Avrilia Floratou, Nimrod Megiddo, Navneet Potti, Fatma Özcan, Uday Kale, and Jan Schmitz-Hermes. 2016. Adaptive Caching in Big SQL using the HDFS Cache. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 321–333. https://doi.org/10.1145/2987550.2987553

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 29–43. https://doi.org/10.1145/945445.945450

[24] Goetz Graefe. 2007. The five-minute rule twenty years later. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 6. http://doi.acm.org/10.1145/1363189.1363198

[25] Jim Gray and Goetz Graefe. 1997. The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *ACM SIGMOD Record* 26, 4 (1997), 63–68. http://doi.acm.org/10.1145/271074.271094

[26] Jim Gray and Franco Putzolu. 1986. The 5 Minute Rule for Trading Memory for Disc Accesses and the 5 Byte Rule for Trading Memory for CPU Time. *Tandem Computers - Technical Report* (1986). http://db.cs.berkeley.edu/cs286/papers/fiveminute-tr1986.pdf

[27] Jorge Guerra, Himabindu Pucha, Joseph S. Glider, Wendy Belluomini, and Raju Rangaswami. 2011. Cost Effective Storage using Extent Based Dynamic Tiering. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 273–286. http://www.usenix.org/events/fast11/tech/techAbstracts.html#Guerra

[28] Herodotos Herodotou and Elena Kakoulli. 2019. Automating Distributed Tiered Storage Management in Cluster Computing. *Proceedings of the VLDB Endowment* 13, 1 (2019), 43–56. http://www.vldb.org/pvldb/vol13/p43-herodotou.pdf

[29] HP. Accessed: 2024-10-14. Adaptive Optimization for HPE 3PAR StoreServ Storage. https://www.hpe.com/psnow/doc/4aa4-0867enw.

[30] IBM. Accessed: 2024-09-12. IBM Integrated Analytics System - Tiered storage. https://www.ibm.com/docs/en/ias?topic=storage-tiered.

[31] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 323–336. http://www.usenix.org/events/usenix05/tech/general/jiang.html

[32] Song Jiang and Xiaodong Zhang. 2002. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*. 31–42. https://doi.org/10.1145/511334.511340

[33] Chi Jin, Zhuoran Yang, Zhaoran Wang, and Michael I. Jordan. 2020. Provably efficient reinforcement learning with linear function approximation. In *Proceedings of the Conference on Learning Theory (COLT)*. 2137–2143. http://proceedings.mlr.press/v125/jin20a.html

[34] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 439–450. http://dl.acm.org/citation.cfm?id=645920.672996

[35] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. 2008. Characterization of storage workload traces from production Windows Servers. In *2008 IEEE International Symposium on Workload Characterization*. 119–128. https://doi.org/10.1109/IISWC.2008.4636097

[36] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. 2020. RL-Cache: Learning-Based Cache Admission for Content Delivery. *IEEE Journal of Selected Areas in Communications* 38, 10 (2020), 2372–2385. https://doi.org/10.1109/JSAC.2020.3000415

[37] Michail G. Lagoudakis. 2010. Value Function Approximation. In *Encyclopedia of Machine Learning*. https://doi.org/10.1007/978-0-387-30164-8_870

[38] Avinash Lakshman and Prashant Malik. 2010. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40. http://dl.acm.org/citation.cfm?id=1773912.1773922

[39] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong-Sang Kim. 2001. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.* 50, 12 (2001), 1352–1361. https://doi.org/10.1109/TC.2001.970573

[40] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. 2018. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 83–90. https://www.usenix.org/conference/fast18/presentation/li

[41] Section 1.2. Linux kernel 3.10. 2013. bcache, a block layer cache for SSD caching. https://kernelnewbies.org/Linux_3.10#Bcache.2C_a_block_layer_cache_for_SSD_caching.

[42] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. 2002. Aqueduct: Online Data Migration with Performance Guarantees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 219–230. http://www.usenix.org/publications/library/proceedings/fast02/lu.html

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

[44] Hamid Reza Maei, Csaba Szepesvári, Shalabh Bhatnagar, Doina Precup, David Silver, and Richard S. Sutton. 2009. Convergent Temporal-Difference Learning with Arbitrary Smooth Function Approximation. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*. 1204–1212. https://proceedings.neurips.cc/paper/2009/hash/3a15c7d0bbe60300a39f76f8a5ba6896-Abstract.html

[45] Luis Magdalena. 2015. Fuzzy Rule-Based Systems. In *Springer Handbook of Computational Intelligence*. https://doi.org/10.1007/978-3-662-43505-2_13

[46] Bo Mao, Suzhen Wu, and Lide Duan. 2018. Improving the SSD Performance by Exploiting Request Characteristics and Internal Parallelism. *IEEE Trans. on CAD of Integrated Circuits and Systems* 37, 2 (2018), 472–484.

[47] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 115–130. http://www.usenix.org/event/fast03/tech/full_papers/megiddo/megiddo.pdf

[48] Jai Menon, David A Pease, Robert M Rees, Linda Duyanovich, and Bruce Light Hillsberg. 2003. IBM Storage Tank - A heterogeneous scalable SAN file system. *IBM Systems Journal* 42, 2 (2003), 250–267. https://doi.org/10.1147/sj.422.0250

[49] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage* 4, 3 (2008), 10:1–10:23. https://doi.org/10.1145/1416944.1416949

[50] Suman Nath and Phillip B. Gibbons. 2008. Online Maintenance of Very Large Random Samples on Flash Storage. *Proceedings of the VLDB Endowment* 1, 1 (2008), 970–983. http://www.vldb.org/pvldb/1/1453961.pdf

[51] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 297–306. http://dl.acm.org/citation.cfm?id=170035.170081

[52] Ratko Orlandic. 2003. Effective Management of Hierarchical Storage Using Two Levels of Data Clustering. In *Proceedings of the IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies (MSS)*. 270–279. https://doi.org/10.1109/MASS.2003.1194863

[53] Tarikul Islam Papon. 2024. Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 5644–5648. https://doi.org/10.1109/ICDE60146.2024.00454

[54] Tarikul Islam Papon and Manos Athanassoulis. 2021. A Parametric I/O Model for Modern Storage Devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 2:1–2:11. https://doi.org/10.1145/3465998.3466003

[55] Tarikul Islam Papon and Manos Athanassoulis. 2021. The Need for a New I/O Model. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[56] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1326–1339.

[57] Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. 2024. CAVE: Concurrency-Aware Graph Processing on SSDs. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 3 (2024), 125:1–125:26. https://doi.org/10.1145/3654928

[58] Stan Park and Kai Shen. 2009. A performance evaluation of scientific I/O workloads on Flash-based SSDs. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 1–5.

[59] Phitchaya Mangpo Phothilimthana, Saurabh Kadekodi, Soroush Ghodrati, Selene Moon, and Martin Maas. 2024. Thesios: Synthesizing Accurate Counterfactual I/O Traces from I/O Samples. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1016–1032. https://doi.org/10.1145/3620666.3651337

[60] Stefan Podlipnig and László Böszörményi. 2003. A survey of Web cache replacement strategies. *Comput. Surveys* 35, 4 (2003), 374–398. https://doi.org/10.1145/954339.954341

[61] Jinting Ren, Xianzhang Chen, Duo Liu, Yujuan Tan, Moming Duan, Ruolan Li, and Liang Liang. 2021. A machine learning assisted data placement mechanism for hybrid storage systems. *Journal of Systems Architecture* 120 (2021), 102295. https://doi.org/10.1016/j.sysarc.2021.102295

[62] 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 European Conference on Computer Systems (EuroSys)*. 803–817. https://doi.org/10.1145/3627703.3650075

[63] Beomjoo Seo and Roger Zimmermann. 2005. Efficient disk replacement and data migration algorithms for large disk subsystems. *ACM Transactions on Storage* 1, 3 (2005), 316–345. https://doi.org/10.1145/1084779.1084781

[64] Amazon Web Services. 2025. Managing your storage lifecycle. https://docs.aws.amazon.com/AmazonS3/latest/userguide/object-lifecycle-mgmt.html.

[65] Kai Shen and Stan Park. 2013. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 67–78.

[66] Milan M. Shetti, Bingzhe Li, and David Du. 2022. Machine Learning-based Adaptive Migration Algorithm for Hybrid Storage Systems. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage (NAS)*. 1–8. https://doi.org/10.1109/NAS55553.2022.9925545

[67] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 413–425. https://doi.org/10.1145/3352460.3358319

[68] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[69] David Silver. 2015. Lectures on Reinforcement Learning. URL: https://www.davidsilver.uk/teaching/.

[70] Gagandeep Singh and Rakesh Nadig. 2022. Sibyl. https://github.com/CMU-SAFARI/Sibyl.git. Last accessed on October 8, 2024.

[71] 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 *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. 320–336. https://doi.org/10.1145/3470496.3527442

[72] Richard S. Sutton. 1988. Learning to Predict by the Methods of Temporal Differences. *Machine Learning* 3 (1988), 9–44. https://doi.org/10.1007/BF00115009

[73] Richard S Sutton and Andrew G Barto. 1998. *Introduction to Reinforcement Learning*. MIT Press. http://incompleteideas.net/book/RLbook2020.pdf

[74] Jianzhe Tai, Bo Sheng, Yi Yao, and Ningfang Mi. 2014. Live Data Migration for Reducing SLA Violations in Multi-tiered Storage Systems. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*. 361–366. https://doi.org/10.1109/IC2E.2014.8

[75] Luis Thomas, Sebastien Gougeaud, Stéphane Rubini, Philippe Deniel, and Jalil Boukhobza. 2021. Predicting file lifetimes for data placement in multi-tiered storage systems for HPC. *ACM SIGOPS: Operating Systems Review* 55, 1 (2021), 99–107. https://doi.org/10.1145/3469379.3469392

[76] Quoc-Cuong To, Juan Soto, and Volker Markl. 2017. A Survey of State Management in Big Data Processing Systems. *CoRR* abs/1702.0 (2017).

[77] Transaction Processing Performance Council. 2024. TPC-C Benchmark. http://www.tpc.org/tpcc/. Last accessed on October 8, 2024.

[78] Transaction Processing Performance Council. 2024. TPC-E Benchmark. http://www.tpc.org/tpce/. Last accessed on October 8, 2024.

[79] John N. Tsitsiklis and Benjamin Van Roy. 1997. An analysis of temporal-difference learning with function approximation. *IEEE Trans. Automat. Control* 42, 5 (1997), 674–690. https://doi.org/10.1109/9.580874

[80] P Venketesh and R Venkatesan. 2009. A Survey on Applications of Neural Networks and Evolutionary Techniques in Web Caching. *IETE Technical Review* 26, 3 (2009), 171–180. https://www.tandfonline.com/doi/abs/10.4103/0256-4602.50701

[81] Shakthi Weerasinghe, Arkady B. Zaslavsky, Seng W. Loke, Alexey Medvedev, Amin Abken, Alireza Hassani, and Guang-Li Huang. 2024. Reinforcement Learning Based Approaches to Adaptive Context Caching in Distributed Context Management Systems. *ACM Transactions on Internet of Things* 5, 2 (2024), 12:1–12:32. https://doi.org/10.1145/3648571

[82] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell D E Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 307–320. http://www.usenix.org/events/osdi06/tech/weil.html

[83] John Wilkes, Richard A Golding, Carl Staelin, and Tim Sullivan. 1996. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems (TOCS)* 14, 1 (1996), 108–136. https://doi.org/10.1145/225535.225539

[84] Gala Yadgar, Moshe Gabel, Shehbaz Jaffer, and Bianca Schroeder. 2021. SSD-based Workload Characteristics and Their Performance Implications. *ACM Transactions on Storage* 17, 1 (2021), 8:1–8:26. https://doi.org/10.1145/3423137

[85] Gala Yadgar, Moshe Gabel, Shehbaz Jaffer, and Bianca Schroeder. 2021. YCSB RocksDB SSD Traces. https://iotta.snia.org/traces/block-io/28568. Last accessed on June 17, 2025.

[86] Geng Yushui and Yuan Jiaheng. 2015. Cloud Data Migration Method Based on PSO Algorithm. In *Proceedings of the International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*. 143–146. https://ieeexplore.ieee.org/document/7429577

[87] Gong Zhang, Lawrence Chiu, and Ling Liu. 2010. Adaptive Data Migration in Multi-tiered Storage Based Cloud Environment. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*. 148–155. https://doi.org/10.1109/CLOUD.2010.60

[88] Tianru Zhang. 2024. Autonomous Hierarchical Storage Management via Reinforcement Learning. *Proceedings of the VLDB Endowment. ISSN* 2150 (2024), 8097.

[89] Tianru Zhang, Ankit Gupta, María Rodríguez, Ola Spjuth, Andreas Hellander, and Salman Toor. 2024. Data management of scientific applications in a reinforcement learning-based hierarchical storage system. *Expert Systems with Applications* 237, Part B (2024), 121443. https://doi.org/10.1016/j.eswa.2023.121443

[90] Tianru Zhang, Andreas Hellander, and Salman Toor. 2023. Efficient Hierarchical Storage Management Empowered by Reinforcement Learning. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 35, 6 (2023), 5780–5793. https://doi.org/10.1109/TKDE.2022.3176753

[91] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 91–104. http://www.usenix.org/publications/library/proceedings/usenix01/zhou.html

Received October 2025; revised January 2026; accepted February 2026