Sibyl: Adaptive and Extensible Data Placement in Hybrid Storage Systems Using Online Reinforcement Learning

Gagandeep Singh[1] Rakesh Nadig[1] Jisung Park[1] Rahul Bera[1] Nastaran Hajinazar[1]

David Novo[3] Juan Gómez-Luna[1] Sander Stuijk[2] Henk Corporaal[2] Onur Mutlu[1]

[1]ETH Zürich   [2]Eindhoven University of Technology   [3]LIRMM, Univ. Montpellier, CNRS

ABSTRACT

Hybrid storage systems (HSS) use multiple different storage devices to provide high and scalable storage capacity at high performance. Data placement across different devices is critical to maximize the benefits of such a hybrid system. Recent research proposes various techniques that aim to accurately identify performance-critical data to place it in a "best-fit" storage device. Unfortunately, most of these techniques are rigid, which (1) limits their adaptivity to perform well for a wide range of workloads and storage device configurations, and (2) makes it difficult for designers to extend these techniques to different storage system configurations (e.g., with a different number or different types of storage devices) than the configuration they are designed for. Our goal is to design a new data placement technique for hybrid storage systems that overcomes these issues and provides: (1) *adaptivity*, by *continuously learning* from and adapting to the workload and the storage device characteristics, and (2) *easy extensibility* to a wide range of workloads and HSS configurations.

We introduce *Sibyl*, the first technique that uses reinforcement learning for data placement in hybrid storage systems. Sibyl observes different features of the running workload as well as the storage devices to make system-aware data placement decisions. For every decision it makes, Sibyl receives a reward from the system that it uses to evaluate the long-term performance impact of its decision and continuously optimizes its data placement policy online.

We implement Sibyl on *real* systems with various HSS configurations, including dual- and tri-hybrid storage systems, and extensively compare it against four previously proposed data placement techniques (both heuristic- and machine learning-based) over a wide range of workloads. Our results show that Sibyl provides 21.6%/19.9% performance improvement in a performance-oriented/cost-oriented HSS configuration compared to the best previous data placement technique. Our evaluation using an HSS configuration with three different storage devices shows that Sibyl outperforms the state-of-the-art data placement policy by 23.9%-48.2%, while significantly reducing the system architect's burden

in designing a data placement mechanism that can simultaneously incorporate three storage devices. We show that Sibyl achieves 80% of the performance of an oracle policy that has complete knowledge of future access patterns while incurring a very modest storage overhead of only 124.4 KiB.

CCS CONCEPTS

• **Hardware → Communication hardware, interfaces and storage**; • **Computing methodologies → Reinforcement learning**.

KEYWORDS

solid-state drives (SSDs), reinforcement learning, hybrid storage systems, data placement, hybrid systems, machine learning

1 INTRODUCTION

Hybrid storage systems (HSS) take advantage of both fast-yet-small storage devices and large-yet-slow storage devices to deliver high storage capacity at low latency [1–48]. The key challenge in designing a high-performance and cost-effective hybrid storage system is to accurately identify the performance-criticality of application data and place data in the "best-fit" storage device [22].

Past works [49–73] propose many different data placement techniques to improve the performance of an HSS. We identify two major shortcomings of prior proposals that significantly limit their performance: lack of (1) adaptivity to workload changes and the storage device characteristics, and (2) extensibility.

(1a) Lack of adaptivity to workload changes. To guide data placement, past techniques consider only a limited number of workload characteristics [49–57, 74, 75]. Designers statically tune the parameters values for all considered workloads at design time based on empirical analysis and designer experience, and expect those statically-fixed values to be equally effective for a wide range of dynamic workload demands and system configurations seen in the real world. As a result, such data placement techniques cannot easily adapt to a wide range of dynamic workload demands and significantly underperform when compared to an oracle technique that has complete knowledge of future storage access patterns (up to 41.1% lower performance, ref. §3).

(1b) Lack of adaptivity to changes in device types and configurations. Most prior HSS data placement techniques (e.g., [49–59]) do not adapt well to changes in the underlying storage device characteristics (e.g., changes in the level of asymmetry in the read/write latencies, or the number and types of storage devices). As a result, existing techniques cannot effectively take into account the cost of data movement between storage devices while making data placement decisions. This lack of adaptivity leads to highly inefficient data placement policies, especially in HSSs with significantly-different device access latencies than what prior techniques were designed for (as shown in §3).

(2) Lack of extensibility. A large number of prior data placement techniques (e.g., [49–51, 53–57]) are typically designed for an HSS that consists of only two storage devices. As modern HSSs already incorporate more than two types of storage devices [49, 59, 76], system architects need to put significant effort into extending prior techniques to accommodate more than two devices. We observe that a state-of-the-art heuristic-based data placement technique optimized for an HSS with two storage devices [76] often leads to suboptimal performance in an HSS with three storage devices (up to 48.2% lower performance, ref. §8.7).

Our goal is to develop a new, efficient, and high-performance data placement mechanism for hybrid storage systems that provides (1) *adaptivity*, by *continuously learning* from and adapting to the workload and storage device characteristics, and (2) *easy extensibility* to a wide range of workloads and HSS configurations.

Key ideas. To this end, we propose Sibyl, a reinforcement learning-based data placement technique for hybrid storage systems.[1] Reinforcement learning (RL) [78] is a goal-oriented decision-making process in which an autonomous agent learns to take optimal actions that maximize a reward function by interacting with an environment. The key idea of Sibyl is to design the data placement module in hybrid storage systems as a reinforcement learning agent that *autonomously learns* and adapts to the best-fit data placement policy for the running workload and the current hybrid storage system configuration. For every storage page access, Sibyl observes different features from the running workload and the underlying storage system (e.g., access count of the current request, remaining capacity in the fast storage, etc.). It uses the features as **state** information to take a data placement **action** (i.e., which device to place the page into). For every action, Sibyl receives a delayed **reward** from the system in terms of per-request latency. This reward encapsulates the internal device characteristics of an HSS (such as read/write latencies, latency of garbage collection, queuing delays, error handling latencies, and write buffer state). Sibyl uses this reward to estimate the long-term impact of its action (i.e., data placement decision) on the overall application performance and continuously optimizes its data placement policy online to maximize the long-term benefit (i.e., reward) of its actions.

Benefits. Formulating the data placement module as an RL agent enables a human designer to specify only *what* performance objective the data placement module should target, rather than designing and implementing a new data placement policy that requires explicit specification of *how* to achieve the performance objective.

The use of RL not only enables the data placement module to *autonomously* learn the "best-fit" data placement strategy for a wide range of workloads and hybrid storage system configurations but also significantly reduces the burden of a human designer in finding a good data placement policy.

Challenges. While RL provides a promising alternative to existing data placement techniques, we identify two main challenges in applying RL to data placement in an HSS.

(1) Problem formulation. The RL agent's effectiveness depends on how the data placement problem is cast as a reinforcement learning-based task. Two key issues arise when formulating HSS data placement as an RL problem: (1) taking into account the latency asymmetry within and across storage devices, and (2) deciding which actions to reward and penalize (also known as the *credit assignment problem* [79]). First, we need to make the agent aware of the asymmetry in read and write latencies of each storage device and the differences in latencies across multiple storage devices. Real-world storage devices could have dynamic latency variations due to their complex hardware and software components (e.g., internal caching, garbage collection, error handling, multi-level cell reading, etc.) [80–87]. Second, if the fast storage is running out of free space, there might be evictions in the background from the fast storage to the slow storage. As a result, when we reward the agent, not only there is a variable and delayed reward, but it is also hard to properly assign *credit* or *blame* to different decisions.

(2) Implementation overhead. A workload could have hundreds of thousands of pages of storage data, making it challenging to efficiently handle the large data footprint with a low design overhead for the learning agent.

To address the first challenge, we use two main techniques. First, we design a *reward* structure in terms of *request latency*, which allows Sibyl to learn the workload and storage device characteristics when continuously and frequently interacting with a hybrid storage system. We add a negative penalty to the reward structure in case of eviction, which helps with handling the credit assignment problem and encourages the agent to place only performance-critical pages in the fast storage. Second, we perform thorough hyper-parameter tuning to find parameter values that work well for a wide variety of workloads. To address the second challenge, we use two main techniques. First, we divide states into a small number of bins that reduce the state space, which directly affects the implementation overhead. Second, instead of adopting a traditional table-based RL approach (e.g., [88, 89]) to store the agent's state-action information (collected by interacting with an HSS), which can easily introduce significant performance overheads in the presence of a large state/action space, we use a simple feed-forward neural network [90] with only two hidden layers of 20 and 30 nodes, respectively.

Key results. We evaluate Sibyl using two different dual-HSS configurations and two different tri-HSS configurations. We use fourteen diverse storage traces from Microsoft Research Cambridge (MSRC) [91] collected on real enterprise servers. We evaluate Sibyl on workloads from FileBench [92] on which it has never been trained. We compare Sibyl to four state-of-the-art data placement techniques. We demonstrate four key results. First, Sibyl provides 21.6%/19.9% performance improvement in a performance-oriented/cost-oriented HSS configuration compared to the best previous

---

[1]In Greek mythology, Sibyl is an oracle who makes accurate prophecies [77].

data placement technique. Second, Sibyl outperforms the best-performing supervised learning-based technique on workloads it has never been trained on by 46.1% and 54.6%, on average, in performance-oriented and cost-oriented HSS configurations, respectively. Third, Sibyl provides 23.9%-48.2% higher performance in tri-hybrid storage systems than a state-of-the-art heuristic-based data placement technique demonstrating that Sibyl is easily extensible and alleviates the designer's burden in finding sophisticated data placement mechanisms for new and complex HSS configurations. Fourth, Sibyl's performance benefits come with a low storage implementation overhead of only 124.4 KiB.

This work makes the following **major contributions**:

- We show on real hybrid storage systems (HSSs) that prior state-of-the-art HSS data placement mechanisms fall short of the oracle placement due to: lack of (1) adaptivity to workload changes and storage device characteristics, and (2) extensibility.
- We propose Sibyl, a new self-optimizing mechanism that uses reinforcement learning to make data placement decisions in hybrid storage systems. Sibyl dynamically *learns*, using both multiple workload features and system-level feedback information, how to continuously adapt its policy to improve its long-term performance for a workload.
- We conduct an in-depth evaluation of Sibyl on real systems with various HSS configurations, showing that it outperforms four state-of-the-art techniques over a wide variety of applications with a low implementation overhead.
- We provide an in-depth explanation of Sibyl's actions that show that Sibyl performs dynamic data placement decisions by learning changes in the level of asymmetry in the read/write latencies and the number and types of storage devices.
- We freely open-source Sibyl to aid future research in data placement for storage systems [93].

2 BACKGROUND

2.1 Hybrid Storage Systems (HSSs)

Figure 1 depicts a typical HSS consisting of a fast-yet-small storage device (e.g., [94, 95]) and a large-yet-slow storage device (e.g., [96–99]). Traditional hybrid storage systems [51, 100, 101] were designed with a smaller NAND flash-based SSD and a larger HDD. Nowadays, hybrid storage systems come with emerging NVM devices (e.g., [102–105]) coupled with slower high-density NAND flash devices [49, 106–108]. The storage management layer can be implemented either as system software running on the host system or as the firmware of a hybrid storage device (e.g., flash translation layer (FTL) in flash-based SSDs [84, 109]), depending on the configuration of the HSS. In this work, we demonstrate and implement our ideas in the storage management layer of the operating system (OS), but they can be easily implemented in firmware as well. The storage management layer in the OS orchestrates host I/O requests across heterogeneous devices, which are connected via an NVM Express (NVMe) [110] or SATA [111] interface. The storage management layer provides the operating system with a unified logical address space (like the multiple device driver (md) kernel module in Linux [112]). As illustrated in Figure 1, the unified logical address space is divided into a number of logical pages (e.g., 4 KiB pages). The pages in the logical address space are assigned to an application. The storage management layer translates a read/write performed

on a logical page into a read/write operation on a target storage device based on the data placement policy. In addition, the storage management layer manages data migration between the storage devices in an HSS. When data currently stored in the slow storage device is moved to the fast storage device, it is called *promotion*. Promotion is usually performed when a page in the slow storage device is accessed frequently. Data is moved from the fast storage device to the slow storage device during an *eviction*. Eviction typically occurs when the data in the fast storage device is infrequently accessed or when the fast storage device becomes full.
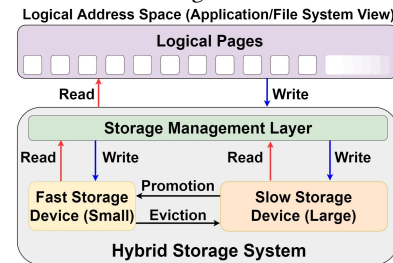

Figure 1: Overview of a hybrid storage system

The performance of a hybrid storage system highly depends on the ability of the storage management layer (Figure 1) to effectively manage diverse devices and workloads [49, 59]. This diversity presents a challenge for system architects when they design an intelligent data placement policy. A desirable policy has to effectively utilize the low latency characteristics of the fast device while making optimal use of its small capacity and should provide easy extensibility to a wide range of workloads and HSS configurations.

3 MOTIVATION

To assess the effectiveness of existing HSS data placement techniques under diverse workloads and hybrid storage configurations, we evaluate state-of-the-art heuristic-based (*CDE* [49] and *HPS* [113]) and supervised learning-based (*Archivist* [59]) techniques. We also implement an RNN-based data placement technique (RNN-HSS), adapted from hybrid main memory [58]. To evaluate the effect of underlying storage device technologies, we use three different storage devices: high-end (H) [94], middle-end (M) [96], and low-end (L) [98], configured into two different hybrid storage configurations: a performance-oriented HSS (H&M) and a cost-oriented HSS (H&L). Table 3 provides details of our system and devices. We restrict the fast storage capacity to 10% of the working set size of a workload, which ensures eviction of data from fast storage to slow storage when fast storage capacity is full.

CDE [49] allocates hot or random write requests in the faster storage, whereas cold and sequential write requests are evicted to the slower device. HPS [113] uses the access count of pages to periodically migrate cold pages to the slower storage device. Archivist [59] uses a neural network classifier to predict the target device for data placement. *RNN-HSS*, adapted from [58], is a supervised learning-based mechanism that exploits recurrent neural networks (RNN) to predict the hotness of a page and place hot pages in fast storage. We compare the above policies with three extreme baselines: (1) Slow-Only, where all data resides in the slow storage device (i.e., there is no fast storage device), (2) Fast-Only, where all data resides in the fast storage device, and (3) Oracle [113], which exploits complete knowledge of future I/O-access patterns to

perform data placement and to select victim data blocks for eviction from the fast device.

We identify two major shortcomings of the state-of-the-art baseline data placement techniques: lack of (1) adaptivity to workload changes and the storage device characteristics, and (2) extensibility. **(1a) Lack of adaptivity to workload changes.** Figure 2 shows the average request latency of all policies, normalized to Fast-Only, under two different hybrid storage configurations. We make the following three observations. First, all the baseline techniques are only effective under specific workloads, showing significantly lower performance than Oracle in most workloads. CDE, HPS, Archivist, and RNN-HSS achieve comparable performance to Oracle for specific workloads (e.g., hm_1 for HPS in H&M, usr_0 for CDE in H&L, hm_1 for Archivist in H&M, and RNN-HSS in proj_2 for CDE in H&L). Second, the baselines show a large average performance loss of 41.1% (32.6%), 37.2% (55.5%), 39.7% (66.7%), and 34.4% (47.6%) compared to Oracle's performance, under the H&M (H&L) hybrid storage configuration, respectively. Third, in H&M, the baseline techniques provide a performance improvement of only 1.4%, 7.4%, 3.5%, and 11.3% compared to Slow-Only.
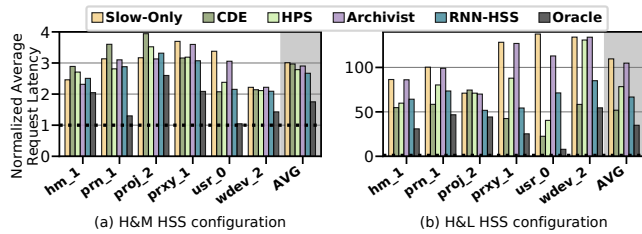

Figure 2: Average request latency normalized to Fast-Only policy

We conclude that all four baselines consider only a limited number of workload characteristics to construct a data placement technique, which leads to a significant performance gap compared to the Oracle policy. Thus, there is no single policy that works well for all the workloads.

To further analyze the characteristics of our evaluated workloads, we plot the average hotness (y-axis) and randomness (x-axis) in Figure 3. We provide details on these workloads in Table 4. In these workload traces, each storage request is labeled with a timestamp that indicates the time when the request was issued from the processor core. Therefore, the time interval between two consecutive I/O requests represents the time the core has spent computing. We quantify a workload's hotness (or coldness) using the average access count, which is the average of the access counts of all pages in a workload; the higher (lower) the average access count, the hotter (colder) the workload. We quantify a workload's randomness using the average request size in the workload; the higher (lower) the average request size, the more sequential (random) the workload. From Figure 3, we make the following two observations. First, the average hotness and randomness vary widely between workloads. Second, we observe that each of our evaluated workloads exhibits highly dynamic behavior throughout its execution. For example, in Figure 4, we show the execution timeline of rsrch_0, which depicts the accessed addresses and request sizes. We conclude that an efficient policy needs to incorporate continuous adaptation to highly dynamic changes in workload behavior.
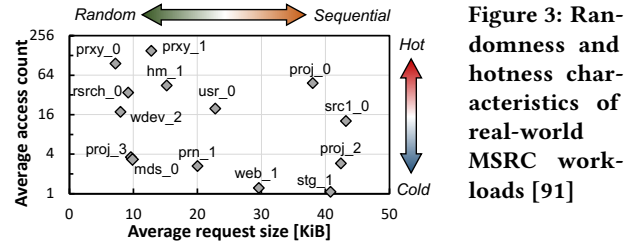

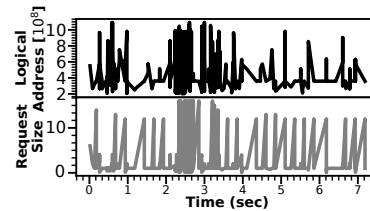Figure 3: Randomness and hotness characteristics of real-world MSRC workloads [91]

Figure 4: Timeline of accessed logical addresses and request sizes during the execution of rsrch_0 workload

(1b) Lack of adaptivity to changes in device types and configurations. There are a wide variety and number of storage devices [49, 94–99, 102–108, 114–117] that can be used to configure an HSS. The underlying storage technology used in an HSS significantly influences the effectiveness of a data placement policy. We demonstrate this with an example observation from Figure 2. In the H&M configuration (Figure 2(a)), we observe that for certain workloads (hm_1 and prn_1), both CDE and HPS provide rather low performance even compared to Slow-Only. Similarly, Archivist and RNN-HSS provide lower performance for hm_1 and proj_2 in H&M compared to Slow-Only. While in the H&L configuration (Figure 2(b)), we observe that CDE, HPS, Archivist, and RNN-HSS result in lower latency than Slow-Only for the respective workloads. Thus, we conclude that both heuristic-based and learning-based data placement policies lead to poor performance due to their inability to holistically take into account the device characteristics. The high diversity in device characteristics makes it very difficult for a system architect to design a generic data-placement technique that is suitable for all HSS configurations.

(2) Lack of extensibility. As modern HSSs already incorporate more than two types of storage devices [1, 49, 59, 76], system architects need to put significant effort into extending prior data placement techniques to accommodate more than two devices. In §8.7, we evaluate the effectiveness of a state-of-the-art heuristic-based policy [76] for different tri-HSS configurations, comprising of three different storage devices. This policy is based on the CDE [49] policy that divides pages into hot, cold, and frozen data and allocates these pages to H, M, and L devices, respectively. A system architect needs to statically define the hotness values and explicitly handle the eviction and promotion between the three devices during design-time. Through our evaluation in §8.7, we conclude that such a heuristic-based policy (1) lacks extensibility, thereby increasing the system architect's effort, and (2) leads to lower performance when compared to an RL-based solution (up to 48.2% lower).

Our empirical study shows that **the state-of-the-art heuristic- and learning-based data placement techniques are rigid and far from optimal,** which strongly motivates us to develop a new data placement technique to achieve significantly higher performance than existing policies. The new technique should provide (1) *adaptivity* to better capture the features and dynamic changes in

I/O-access patterns and storage device characteristics, and (2) *easy extensibility* to a wide range of workloads and HSS configurations. Our goal is to develop such a technique using reinforcement learning.

4 REINFORCEMENT LEARNING
4.1 Background

Reinforcement learning (RL) [78] is a class of machine learning (ML) algorithms that involve an *agent* learning to achieve an objective by interacting with its *environment*, as shown in Figure 5. The agent starts from an initial representation of its environment in the form of an initial state[2] $s_0 \in S$, where S is the set of all possible states. Then, at each *time step t*, the agent performs an *action* $a_t \in A$ in state s_t (A represents the set of possible actions) and moves to the next state s_{t+1}. The agent receives a numerical reward $r_{t+1} \in R$, which could be *immediate* or *delayed* in time, for action a_t that changes the environment state from s_t to s_{t+1}. The sequence of states and actions starting from an initial state to the final state is called an *episode*. The agent makes decisions and receives corresponding rewards while trying to maximize the *accumulated* reward, as opposed to maximizing the reward for only *each* action. In this way, the agent can optimize for the long-term impact of its decisions.

Figure 5: Main components of general RL

The policy π governs an agent's action in a state. The agent's goal is to find the optimal policy that maximizes the cumulative reward[3] collected from the environment over time. The agent finds an optimal policy π^* by calculating the optimal action-value function (Q^*), also known as the **Q-value** of the state-action pair, where $Q(S, A)$ represents the expected cumulative reward by taking an action A in a given state S.

Traditional RL methods (e.g., [88, 89, 118–120]) use a tabular approach with a lookup table to store the Q-values associated with each state-action pair. These approaches can lead to high storage and computation overhead for environments with a large number of states and actions. To overcome this issue, *value function approximation* was proposed. [121–124]. Value function approximation replaces the lookup table with a supervised-learning model [121–126], which provides the capability to generalize over a large number of state-action pairs with a low storage and computation overhead.

4.2 Why Is RL a Good Fit for Data Placement in Hybrid Storage Systems?

We choose RL for data placement in HSS due to the following advantages compared to heuristic-based (e.g., [49, 113]) and supervised learning-based (e.g., [59]) techniques.

(1) Adaptivity. As discussed in §1 and §3, a data placement technique should have the ability to adapt to changing workload demands and underlying device characteristics. This adaptivity requirement of data placement makes RL a good fit to model data

placement. The RL agent works autonomously in an HSS using the provided state features and reward to make data placement decisions without any human intervention.

(2) Online learning. Unlike an *offline* learning-based approach, an RL agent uses an *online* learning approach. Online learning allows an RL agent to continuously adapt its decision-making policy using system-level feedback and specialize to the current workload and system configuration.

(3) Extensibility. RL provides the ability to easily extend a mechanism with a small effort required to implement the extension. As shown in §8.7, unlike heuristic-based mechanisms, RL can be easily extended to different types and number of storage devices. Such extensibility reduces the system architect's burden in designing sophisticated data placement mechanisms.

(4) Design Ease. With RL, the designer of the HSS does not need to specify a data placement policy. They need to specify *what* to optimize (via reward function) but not *how* to optimize it.

(5) Implementation Ease. RL provides ease of implementation that requires a small computation overhead. As shown in §8, *function approximation*-based RL techniques can generalize over all the possible state-action pairs by using a simple feed-forward neural network to provide high performance at low implementation overhead (compared to sophisticated RNN-based mechanisms).

5 SIBYL: RL FORMULATION

Figure 6 shows our formulation of data placement as an RL problem. We design Sibyl as an RL agent that learns to perform accurate and system-aware data placement decisions by interacting with the hybrid storage system. With every storage request, Sibyl observes multiple workload and system-level features as a *state* to make a data placement decision. After every *action*, Sibyl receives a *reward* in terms of the served request latency that takes into account the data placement decision and internal storage system state. Sibyl's goal is to find an optimal data placement policy that maximizes overall performance for the running workload and the current system configuration. To reach its performance goal, Sibyl needs to minimize the average request latency of the running workload by maximizing the use of the fast storage device while avoiding the eviction penalty due to non-performance critical pages.
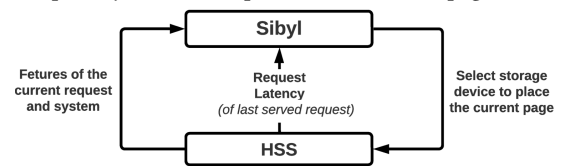

Figure 6: Formulating data placement as an RL problem

Reward. After every data placement decision at time-step[4] t, Sibyl gets a reward from the environment at time-step $t + 1$ that acts as a feedback to Sibyl's previous action. To achieve Sibyl's performance goal, we craft the reward function R as follows:

$$R = \begin{cases} \dfrac{1}{L_t} & \text{if no eviction of a page from the} \\ & \text{fast storage to the slow storage} \\ max(0, \frac{1}{L_t} - R_p) & \text{in case of eviction} \end{cases} \quad (1)$$

where L_t and R_p represent the last served request latency and eviction penalty, respectively. If the fast storage is running out of free space, there might be evictions in the background from the fast

---

[2]State is a representation of an environment using different features.
[3]The total cumulative reward is also known as the *return* [78].

[4]In HSS, a time-step is defined as a new storage request.

storage to the slow storage. Therefore, we add an eviction penalty (R_p) to guide Sibyl to place only performance-critical pages in the fast storage. We empirically select R_p to be equal to $0.001{\times}L_e$ (L_e is the time spent in evicting pages from the fast storage to the slow storage), which prevents the agent from aggressively placing all requests into the fast storage device.

L_t (request latency) is the time taken to service the last read or write I/O request from the OS. Request latency can faithfully capture the status of the hybrid storage system, as it significantly varies depending on the request type, device type, and the internal state and characteristics of the device (e.g., such as read/write latencies, the latency of garbage collection, queuing delays, and error handling latencies). Intuitively, if L_t is low (high), i.e., if the agent serves a storage request from the fast (slow) device, the agent receives a high (low) reward. However, if there is an eviction, we penalize the agent so as to encourage the agent to place only performance-critical pages in the fast storage device. We need the eviction penalty to be large enough to discourage the agent from evicting and small enough not to deviate the learned policy too much on a placement decision that leads to higher latency.

State. At each time-step t, the state features for a particular read/write request are collected in an *observation* vector. We perform feature selection [127] to determine the best state features to include in Sibyl's observation vector.

We use a limited number of features due to two reasons. First, a limited feature set allows us to reduce the implementation overhead of our mechanism (see §10). Second, we empirically observe that our RL agent is more sensitive to the reward structure than to the number of features in the observation vector. Specifically, using the request latency as a reward provides indirect feedback on the internal timing characteristics *and* the current state (e.g., queueing delays, buffer dependencies, effects of garbage collection, read/write latencies, write buffer state, and error handling latencies) of the hybrid storage system. Our observation aligns with a recent study [128] that argues that the reward is the most critical component in RL to find an optimal decision-making policy.

In our implementation of Sibyl, the observation vector is a 6-dimensional tuple:
$$O_t = (size_t, type_t, intr_t, cnt_t, cap_t, curr_t). \qquad (2)$$
Table 1 lists our six selected features. We quantize the representation of each state into a small number of bins to reduce the storage overhead of state representation. These features can be captured in the block layer of the storage system and stored in a separate metadata table (§10). $size_t$ represents the size of the current request in terms of the number of pages associated with it. It indicates whether the incoming request is sequential or random. $type_t$ (request type) differentiates between read and write requests, important for data placement decisions since storage devices have asymmetric read and write latencies. $intr_t$ (access interval) and cnt_t (access count) represent the temporal and spatial reuse characteristics of the currently requested page, respectively. Access interval is defined as the number of page accesses between two references to the same page. Access count is defined as the total number of accesses to the page. These metrics provide insight into the dynamic behavior of the currently requested page.[5] cap_t is a global counter that

tracks the remaining capacity in the fast storage device, which is an important feature since our agent's goal is to maximize the use of the limited fast storage capacity while avoiding evictions from the fast storage device. By including this feature, the agent can learn to avoid the eviction penalty (i.e., learn to restrain itself from placing in fast storage non-performance critical pages that would lead to evictions). $curr_t$ is the current placement of the requested page. Since every data placement decision affects the decision for future requests, $curr_t$ guides Sibyl to perform past-aware decisions.

Table 1: State features used by Sibyl

Feature	Description	# of bins	Encoding (bits)
$size_t$	Size of the requested page (in pages)	8	8
$type_t$	Type of the current request (read/write)	2	4
$intr_t$	Access interval of the requested page	64	8
cnt_t	Access count of the requested page	64	8
cap_t	Remaining capacity in the fast storage device	8	8
$curr_t$	Current placement of the requested page (fast/slow)	2	4

Action. At each time-step t, in a given state, Sibyl selects an action (a_t in Figure 6) from all possible actions. In a hybrid storage system with two devices, possible actions are: placing data in (1) the fast storage device or (2) the slow storage device. This is easily extensible to N storage devices, where $N \geq 3$.

6 SIBYL: DESIGN

We implement Sibyl in the storage management layer of the host system's operating system. Figure 7(a) shows a high-level overview of Sibyl. Sibyl is composed of two parts, each implemented as a separate thread, that run in parallel (1) the *RL decision thread*, where Sibyl decides the data placement ❹ of the current storage request while collecting information ❼ about its decisions ❹ and their effects ❻ in an *experience buffer* ❺, and (2) the *RL training thread*, where Sibyl uses the collected *experiences*[6] ❽ to update its decision-making policy online ❾. Sibyl continuously learns from its past decisions and their impact. Our two-threaded implementation avoids that the learning (i.e., training) interrupts or delays data placement decisions for incoming requests. To enable the parallel execution of the two threads, we duplicate the neural network that is used to make data placement decisions. While one network (called the *inference network* ❷) is deployed (i.e., makes decisions) the second network (called the training network ❾), is trained in the background. The inference network is used *only* for inference, while the training network is used *only* for training. Therefore, Sibyl does *not* perform a separate training step for the inference network and instead periodically copies the training network weights to the inference network ❿.

For every new storage request to the HSS, Sibyl uses the state information ❶ to make a data placement decision ❹. The inference network predicts the Q-value for each available action given the state information. Sibyl policy ❸ selects the action with the maximum Q-value or, with a low probability, a random action for exploration and performs the data placement.

6.1 Sibyl Data Placement Algorithm

Algorithm 1 describes how Sibyl performs data placement for an HSS. Initially, the experience buffer is allocated to hold e_{EB} entries (line 1), and the training and the inference network weights are initialized to random values (lines 2 and 3). When a storage request is

---

[5]We did not use the reuse distance as a locality metric due to its high computation overhead during online profiling [129].

[6]Experience is a representation of a transition from one time step to another, in terms of ⟨$State, Action, Reward, NextState$⟩.
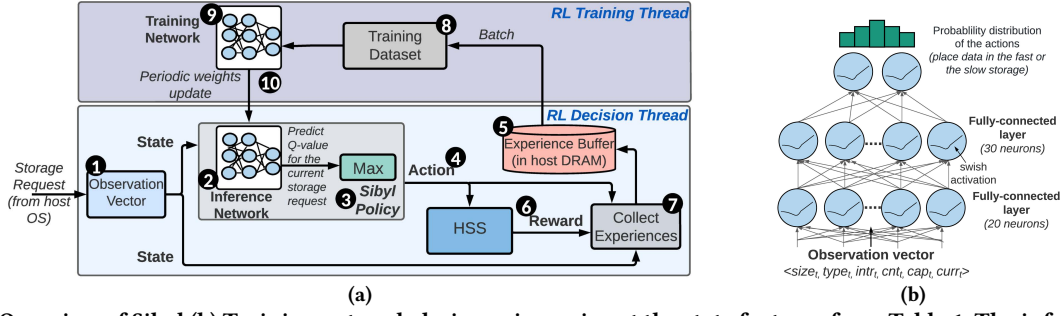
Figure 7: (a) Overview of Sibyl (b) Training network design using as input the state features from Table 1. The inference network is identical except it is used only for inference

received (line 4), Sibyl policy (❸ Figure 7(a)) either (1) randomly selects an action with ϵ probability (lines 6-7) to perform exploration in an HSS environment, or (2) selects the action that maximizes the Q-value, based on information stored in the inference network (lines 8-9). After performing the selected action (line 10), Sibyl collects its reward, whose value depends on whether an eviction is needed from fast storage (lines 11-14). The generated experience is stored in the experience buffer (line 15). Once the experience buffer has e_{EB} entries (line 16), Sibyl trains the training network. During training, the training network samples a batch of experiences from the experience buffer (line 17) and updates its weights using stochastic gradient descent (SGD) [130] (line 18). Sibyl does *not* perform a separate training step for the inference network. Instead, the training network weights are copied to the inference network (line 19), which removes the training of the inference network from the critical path of decision-making.

Algorithm 1 Sibyl's reinforcement learning-based data placement algorithm

```
1: Intialize: the experience buffer EB to capacity e_EB
2: Intialize: the training network with random weights θ
3: Intialize: the inference network with random weights θ̂
4: Intialize: the observation vector O_t=O(s_1) with storage request s_1={req_t}, and
   host and storage features
5: for all storage requests do
6:     if (rand() < ε) then              ▷ with probability ε, perform exploration
7:         random action a_t
8:     else                              ▷ with probability 1-ε, perform exploitation
9:         a_t = argmax_a Q_t(a)  ▷ select action with the highest Q_t value from inference network
10:    execute a_t                       ▷ place the requested page to fast or slow storage
11:    if no eviction then
12:        r_t ← 1/L_t            ▷ reward, given no eviction of a page from fast to slow storage
13:    else
14:        r_t ← max(0, 1/L_t − R_P)   ▷ reward with an eviction penalty in case of an eviction
15:    store experience (O_t, a_t, r_t, O(t+1)) in EB
16:    if (num requests in EB==e_EB) then   ▷ train training network when EB is full
17:        sample random batches of experiences from EB, which are in format
           (O_j, a_j, r_j, O(j+1))     ▷ where O_j represents an observation at a time instant j from EB
18:        Perform stochastic gradient descent   ▷ update the training network weights
19:        θ̂ ← θ                      ▷ copy the training network weights to the inference network
```

6.2 Detailed Design of Sibyl

6.2.1 RL Decision Thread. In this thread, Sibyl makes data placement decisions while storing *experiences* in an experience buffer. Sibyl extracts the observation vector ❶ from the attributes of the incoming request and the current system state (e.g., access count, remaining capacity in the fast storage) and uses the inference network ❷ to predict the Q-values for each possible action with the given state vector. While making data placement decisions, Sibyl balances the random *exploration* of the environment (to find a better policy without getting stuck at a suboptimal one) with the

exploitation of its current policy (to maximize its reward based on the current inference network weights).

Sibyl policy. For every storage request, Sibyl policy selects the action that leads to the highest long-term reward ❻. We use a Categorical Deep Q-Network (also known as C51) [131] to update $Q(s, a)$. C51's objective is to learn the distribution of Q-values, whereas other variants of Deep Q-Networks [121–126] aim to approximate a single value for $Q(s, a)$. This distribution helps Sibyl to capture more information from the environment to make better data placement decisions [132].

For tracking the state, we divide each feature into a small number of bins to reduce the state space (see §5), which directly affects the implementation overhead of Sibyl. We select the number of bins (Table 1) based on empirical sensitivity analysis. Our state representation uses a more relaxed encoding of 40 bits (than using only 20 bits for the observation vector) to allow for future extensions (e.g., features with more bins). Similarly, we use a relaxed 4-bit encoding for the action to allow extensibility to a different number of storage devices. For the reward structure, we use a half-precision floating-point (16-bit) representation.

Experience buffer. Sibyl stores *experiences* it collects while interacting with the HSS in an *experience buffer* [133]. The experience buffer is allocated in the host main memory (DRAM). To minimize its design overhead, we deduplicate data in the stored experiences. To improve the training quality, we perform batch training where each batch consists of randomly sampled experiences. This technique of randomly sampling experiences from the experience buffer is called *experience replay* [133].

Figure 8 shows the effect of different experience buffer sizes on Sibyl's performance in the H&M configuration. We observe that Sibyl's performance saturates at 1000 entries, which we select as the experience buffer size. Since the size of our state representation is 40 bits, to store a single experience tuple, we need 40-bit+4-bit+16-bit+40-bit, i.e., 100 bits. In total, for 1000 experiences, the experience buffer requires 100 KiB in the host DRAM.
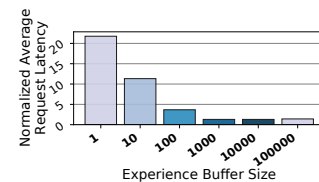

Figure 8: Effect of different experience buffer sizes on the average request latency (normalized to Fast-Only)

Exploration vs. exploitation. An RL agent needs to *explore* the environment to improve its policy to maximize its long-term reward beyond local maxima [78]. At the same time, the agent needs to *exploit* what it has already *experienced* so that it can take advantage of its learning so far. To balance exploration and exploitation, we use the ϵ-greedy policy [134]: the best-known action based on the agent's experience is selected with (1-ϵ) probability, and otherwise, i.e., with ϵ probability, another action is chosen randomly. Exploration allows Sibyl to experience states it may not otherwise get into [78] and thus avoid missing higher long-term rewards. To perform exploration, Sibyl randomly chooses to place data to the fast or the slow storage device, so that it can get more information about the HSS and the workload. Based on the received reward, Sibyl updates its training network. Such exploration helps Sibyl to avoid making suboptimal data placement decisions in the long run.

6.2.2 RL Training Thread. This thread uses a batch of collected experiences ❽ from the experience buffer to train the training network ❾. The updated weights of the training network are transferred to the inference network after every 1000 requests ❿.

Training and inference networks. The training and inference network allows the parallel execution of decision and training threads. We use an identical neural network structure for the training and inference networks. A deep neural network can be prohibitive due to the long time it requires for training and convergence, preventing Sibyl to adapt to new state-action pairs in a timely manner. Based on experiments, we find that a simple feed-forward network [135] with only two hidden layers [136] provides good performance for Sibyl's data placement task. Figure 7(b) shows the structure of our training network.[7] The network takes the observation vector O_t as its input and produces a probability distribution of Q-values as its output. Before feeding the data to the network, we preprocess the data by normalizing and casting the data to low precision data types, which allows us to reduce memory in the experience buffer. Next, we apply two fully-connected hidden layers of 20 and 30 neurons, respectively. We select these neurons based on our extensive design space exploration with different numbers of hidden layers and neurons per layer. After the two hidden layers, we have an output layer of 2 neurons, one for each action. Sibyl policy ❸ selects the action with the maximum Q-value. All fully-connected layers use the swish activation function [137], a non-monotonic function that outperforms ReLU [138].

During the training of the training network, the inference network's weights are fixed. After every 1000 requests, the weights of the training network are copied to the inference network, which removes the training of the inference network from the critical path. We set the number of requests to 1000 based on our empirical evaluation of the experience buffer size (Figure 8). Each training step is composed of 8 batches of experiences from an experience buffer of 1000 experiences with a batch size of 128. We perform the training on the host CPU rather than on a dedicated hardware accelerator because (1) the network size is small and the weights perfectly fit in on-chip caches of the CPU in our evaluated system, and (2) to avoid continuous weight transfer overhead between the host CPU and the accelerator over the external interface.

---

[7]The inference network is identical in shape to the training network.

Hyper-parameter tuning. We improve Sibyl's accuracy by tuning its hyper-parameters. Hyper-parameters are sets of RL algorithm variables that can be tuned to optimize the accuracy of the RL agent [139, 140]. For hyper-parameter tuning, we perform cross-validation [141] using different hyper-parameter values. During cross-validation, we randomly select one workload for hyper-parameter tuning and use the other thirteen workloads for validation. On the selected workload, we use different hyper-parameter configurations that we choose using the *design of experiments* (DoE) [142]. DoE allows us to minimize the number of experiments needed to find the best hyper-parameter values without sacrificing the quality of the information gathered by the experiments. Unlike traditional supervised learning methods, we do *not* train Sibyl *offline* using a training dataset before deploying it for data placement. All training happens online in Sibyl. For every evaluated workload, Sibyl starts with no prior knowledge and gradually learns to make data placement decisions online by interacting with the hybrid storage system. Sibyl needs only one-time offline hyper-parameter tuning.

Table 2 shows the hyper-parameters considered in Sibyl's design as well as their chosen values after the tuning process. The discount factor (γ) determines the balance between the immediate and future rewards. At $\gamma=0$ ($\gamma=1$), Sibyl gives importance only to the immediate (long-term) reward. The learning rate (α) determines the rate at which neural network weights are updated. A lower α makes small updates to the neural network weights, which could take more training iterations to converge to an optimal policy. While a higher α results in large updates to the neural network weights, which could cause the model to converge too quickly to a suboptimal solution. The exploration rate (ϵ) balances exploration and exploitation for Sibyl. We also explore different batch sizes (i.e., the number of samples processed in each training iteration) and experience buffer sizes to train our training network.

Table 2: Hyper-parameters considered for tuning

Hyper-parameter	Design Space	Chosen Value
Discount factor (γ)	0-1	0.9
Learning rate (α)	$1e^{-5} - 1e^0$	$1e^{-4}$
Exploration rate (ϵ)	0-1	0.001
Batch size	64-256	128
Experience buffer size (e_{EB})	10-10000	1000

7 EVALUATION METHODOLOGY

Evaluation setup. We evaluate Sibyl using real systems with various HSS configurations. The HSS devices appear as a single flat block device that exposes one contiguous logical block address space to the OS, as depicted in Figure 1. We implement a lightweight custom block driver interface that manages the I/O requests to storage devices. Table 3 provides our system details, including the characteristics of the three storage devices we use. To analyze the sensitivity of our approach to different device characteristics, we evaluate two different hybrid storage configurations (1) *performance-oriented HSS: high-end device (H) [94] and middle-end device (M) [96], and (2)* cost-oriented HSS: high-end device (H) [94] and low-end device (L) [98]. We also evaluate two tri-hybrid HSS configurations consisting of (1) H&M&L and (2) H&M&L$_{SSD}$ devices. We run the Linux Mint 20.1 operating system [143] with the Ext3 file system [144]. We use the TF-Agents API [145] to develop Sibyl. We evaluate Sibyl using two different metrics: (1) *average request latency*, i.e., average

of the latencies of all storage read/write requests in a workload, and (2) *request throughput (IOPS)*, i.e., throughput of all storage requests in a workload in terms of completed I/O operations per second.

Table 3: Host system and storage devices used in hybrid storage configurations

Host System	AMD Ryzen 7 2700G [146], 8-cores@3.5 GHz, 8×64/32 KiB L1-I/D, 4 MiB L2, 8 MiB L3, 16 GiB RDIMM DDR4 2666 MHz
Storage Devices	**Characteristics**
H: Intel Optane SSD P4800X [94]	375 GB, PCIe 3.0 NVMe, SLC, R/W: 2.4/2 GB/s, random R/W: 550000/500000 IOPS
M: Intel SSD D3-S4510 [96]	1.92 TB, SATA TLC (3D), R/W: 550/510 MB/s, random R/W: 895000/21000 IOPS
L: Seagate HDD ST1000DM010 [98]	1 TB, SATA 6Gb/s 7200 RPM, Max. Sustained Transfer Rate: 210 MB/s
L_{SSD}: ADATA SU630 SSD [99]	960 GB, SATA 6 Gb/s, TLC, Max R/W: 520/450 MB/s
HSS Configurations	**Fast Device** | **Slow Device**
H&M (Performance-oriented)	high-end (H) | middle-end (M)
H&L (Cost-oriented)	high-end (H) | low-end (L)

Baselines. We compare Sibyl against two state-of-the-art heuristic-based HSS data placement techniques, (1) cold data eviction (CDE) [49] and (2) history-based page selection (HPS) [113], (3) a state-of-the-art supervised learning-based technique (Archivist) [59], and (4) a recurrent neural network (RNN)-based data placement technique (RNN-HSS), adapted from Kleio [58], a data placement technique for hybrid memory systems. RNN-HSS provides a state-of-the-art ML-based data placement baseline. We compare the above policies with three extreme baselines: (1) Slow-Only, where all data resides in the slow storage (i.e., there is no fast storage), (2) Fast-Only, where all data resides in the fast storage, and (3) Oracle [113], which exploits complete knowledge of future I/O-access patterns to perform data placement and to select victim data blocks for eviction from the fast device.

Workloads. We use fourteen different block-I/O traces from the MSRC benchmark suite [91] that are collected from real enterprise server workloads. We carefully select the fourteen traces to have distinct I/O-access patterns, as shown in Table 4, in order to study a diverse set of workloads with different randomness and hotness properties (see Figure 3). We quantify a workload's randomness using the average request size of the workload; the higher (lower) the average request size, the more sequential (random) the workload. The average access count provides the average of the access counts of all pages in a workload; the higher (lower) the average access count, the hotter (colder) the workload. Table 4 also shows the number of unique requests in a workload. To demonstrate Sibyl's ability to generalize and provide performance gains across *unseen traces*, i.e., traces that are *not* used to tune the hyper-parameters of Sibyl, we evaluate Sibyl using four additional workloads from FileBench [92].

8 RESULTS

8.1 Performance Analysis

Figure 9 compares the average request latency of Sibyl against the baseline policies for H&M (Figure 9(a)) and H&L (Figure 9(b)) HSS configurations. All values are normalized to Fast-Only. We make five major observations. First, Sibyl consistently outperforms all the baselines for all the workloads in H&L and all but two workloads in H&M. In the H&M HSS configuration (Figure 9(a)), where

Table 4: Characteristics of 14 evaluated workloads

Workload	Write %	Read %	Avg. request size	Avg. access count	No. of unique requests
hm_1	4.7%	95.3%	15.2	44.5	6265
mds_0	88.1%	11.9%	9.6	3.5	31933
prn_1	24.7%	75.3%	20.0	2.6	6891
proj_0	87.5%	12.5%	38.0	48.3	1381
proj_2	12.4%	87.6%	42.4	2.9	27967
proj_3	5.2%	94.8%	9.6	3.6	19397
prxy_0	96.9%	3.1%	7.2	95.7	525
prxy_1	34.5%	65.5%	12.8	150.1	6845
rsrch_0	90.7%	9.3%	9.2	34.7	5504
src1_0	43.6%	56.4%	43.2	12.7	13640
stg_1	36.3%	63.7%	40.8	1.1	3787
usr_0	59.6%	40.4%	22.8	19.7	2138
wdev_2	99.9%	0.1%	8.0	17.7	4270
web_1	45.9%	54.1%	29.6	1.2	6095

the latency difference between two devices is relatively smaller than H&L, Sibyl improves average performance by 28.1%, 23.2%, 36.1%, and 21.6% over CDE, HPS, Archivist, and RNN-HSS, respectively. In the H&L HSS configuration (Figure 9(b)), where there is a large difference between the latencies of the two storage devices, Sibyl improves performance by 19.9%, 45.9%, 68.8%, and 34.1% over CDE, HPS, Archivist and RNN-HSS, respectively. We observe that the larger the latency gap between HSS devices, the higher the expected benefits of avoiding the eviction penalty by placing only performance-critical pages in the fast storage. Second, in the H&M HSS configuration, CDE and HPS are ineffective for certain workloads (hm_1, prn_1, proj_2, proj_3, and src1_0) even when compared to Slow-Only. In contrast, Sibyl consistently and significantly outperforms Slow-Only for all workloads because it can learn the small latency difference between the two storage devices in H&M and dynamically adapts its data placement decisions, which is difficult for CDE and HPS due to their inability to holistically take into account the underlying device characteristics. Third, Sibyl provides slightly lower performance than other baselines in only two workloads: Slow-Only, HPS, Archivist, and RNN-HSS for hm_1 and CDE and HPS for prxy_0 in the H&M HSS configuration. We observe that such workloads are write-intensive and have many random requests (in terms of both access pattern and request size). Therefore, such workloads would benefit from more frequent retraining of Sibyl's training network. We experimentally show in §8.3 that using a lower learning rate during the training of the training network helps to improve Sibyl's performance for such workloads. Fourth, Sibyl achieves, on average, 80% of the performance of the Oracle, which has complete knowledge of future access patterns, across H&M and H&L. Fifth, RNN-HSS provides higher performance than heuristic-based policies (2.1% and 8.9% than CDE and HPS, respectively, in H&M and 9.8% than HPS in H&L), but Sibyl outperforms it by 27.9%. Unlike Sibyl, the two machine learning-based policies, Archivist and RNN-HSS, do *not* consider any system-level feedback, which leads to their suboptimal performance.

Figure 10 compares the request throughput (IOPS) of Sibyl against other baseline policies. We make two observations. First, in the H&M (H&L) HSS configuration (Figure 10), Sibyl improves throughput by 32.6% (22.8%), 21.9% (49.1%), 54.2% (86.9%), and 22.7% (41.9%) over CDE, HPS, Archivist, and RNN-HSS, respectively. Second, Sibyl provides slightly lower performance than Slow-Only, CDE, HPS, Archivist, and RNN-HSS for only hm_1 in H&M HSS configuration. We draw similar observations for throughput results as we did for latency results (Figure 9) because as Sibyl considers the request size in state features and request latency in the reward, it also indirectly captures throughput (size/latency).
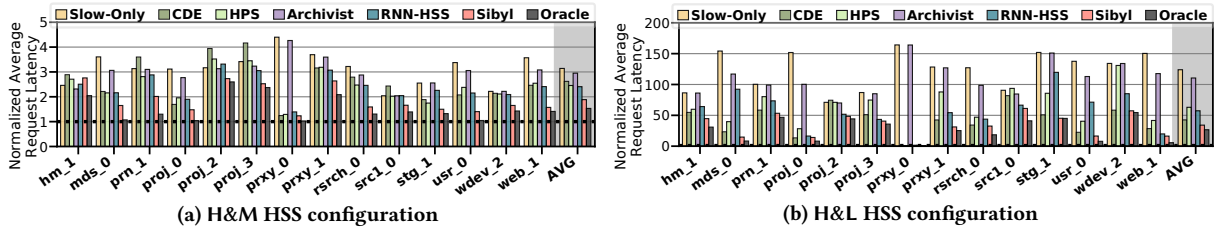
(a) H&M HSS configuration

(b) H&L HSS configuration

Figure 9: Average request latency under two different hybrid storage configurations (normalized to Fast-Only)

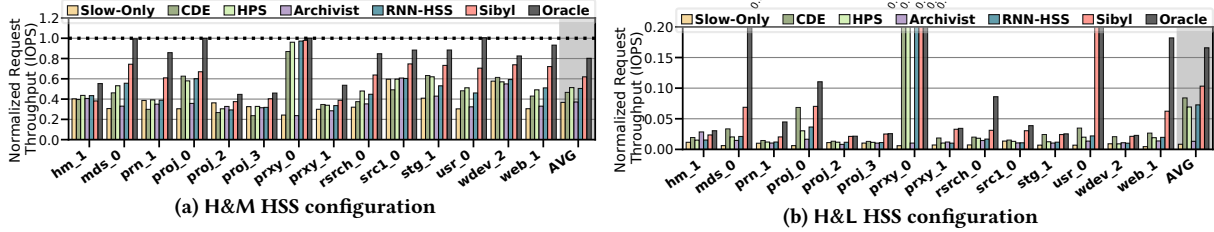(a) H&M HSS configuration

(b) H&L HSS configuration

Figure 10: Request throughput (IOPS) under two different hybrid storage configurations (normalized to Fast-Only)

We conclude that Sibyl consistently provides higher performance than all five baselines and significantly improves both average request latency and request throughput.

8.2 Performance on Unseen Workloads

To demonstrate Sibyl's ability to generalize and provide performance gains across *unseen* workloads that are *not* used to tune the hyper-parameters of the data placement policy of Sibyl, we evaluate Sibyl using four additional workloads from FileBench [92]. No data placement policy we evaluate, including Sibyl, is tuned on these workloads. Figure 11 shows the performance of these unseen workloads. We observe the following observations. First, in H&M (H&L) HSS configuration, Sibyl outperforms RNN-HSS and Archivist by 46.1% (54.6%) and 8.5% (44.1%), respectively. Second, Sibyl may misplace some pages during the online adaptation period, but it provides significant performance benefits over existing ML-based data placement techniques. We conclude that Sibyl provides high performance benefits on unseen workloads for which it has not been tuned.
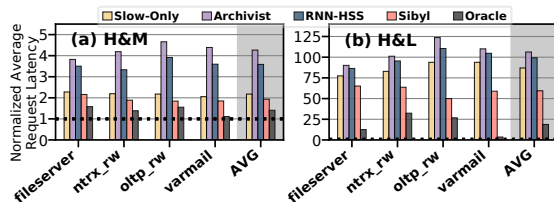

Figure 11: Average request latency on unseen workloads (normalized to Fast-Only) under two HSS configurations

8.3 Performance on Mixed Workloads

We evaluate mixing two or more workloads at the same time while randomly varying their relative start times. Table 5 describes the characteristics of these mixed workloads. These workloads are truly independent of each other, potentially creating more evictions from the fast storage device than a single workload. Such a scenario (1) leads to unpredictable execution where requests arrive at different, unpredictable timesteps, (2) mimics distributed workloads,

and (3) further tests the ability of Sibyl to dynamically adapt its decision-making policy.

Figure 12 shows average request latency for mixed workloads. We use two different settings for Sibyl: (a) $Sibyl_{Def}$, where we use our default hyper-parameters (§6.2.2), and (b) $Sibyl_{Opt}$, where we optimize the hyper-parameters for these mixed workloads and use a lower learning rate (α) of $1e^{-5}$. A lower learning rate performs smaller updates to the training network's weights in each training iteration, thus requiring more training to converge to an optimal solution.

Table 5: Characteristics of mixed workloads

| Mix	Workloads	Description
mix1	prxy_0 [91] and ntrx_rw [92]	Both prxy_0 and ntrx_rw are write-intensive
mix2	rsrch_0 [91] and oltp_rw [92]	rsrch_0 is write-intensive and oltp_rw is read-intensive
mix3	proj_3 [91] and YCSB_C [147]	Both proj_3 and YCSB_C are read-intensive
mix4	src1_0 [91] and fileserver [92]	Both src1_0 and fileserver have nearly equal numbers of reads and writes
mix5	prxy_0 [91], oltp_rw [92] and fileserver [92]	prxy_0 is write-intensive, oltp_rw is read-intensive, and fileserver has nearly equal numbers of reads and writes
mix6	src1_0 [91], YCSB_C [147] and fileserver [92]	src1_0 and fileserver have nearly equal numbers of reads and writes while YCSB_C is read-intensive
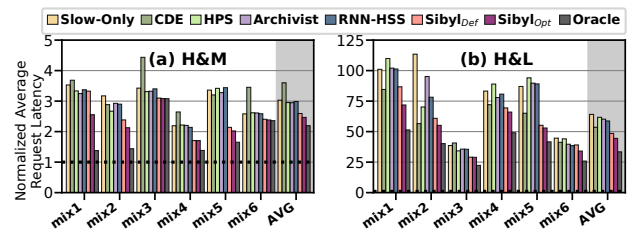
Figure 12: Average request latency on mixed workloads (normalized to Fast-Only) and two HSS configurations

We make two observations. First, $Sibyl_{Def}$ consistently outperforms CDE, HPS, Archivist, and RNN-HSS by 27.9%, 12.2%, 12.1%, and 12.9%, respectively, in the H&M HSS configuration and 9.4%, 21.3%, 19.4%, and 17.1%, respectively, in H&L HSS configuration. Second, with a lower learning rate and optimized hyper-parameters, $Sibyl_{Opt}$ provides 5.2% (9.3%) higher average performance for H&M (H&L) HSS configuration than $Sibyl_{Def}$. Third, for mix_1, HPS

provides comparable performance to $Sibyl_{Def}$ in H&M, and CDE provides slightly better performance in H&L. As discussed in §8.1, prxy_0 is write-intensive and has random requests (with an average request size of 7.2) within every 1000 requests, which is the experience buffer size to train the training network. Such a workload requires more frequent retraining of Sibyl's training network to achieve higher performance. We conclude that Sibyl can effectively adapt its data placement policy online to highly dynamic workloads.

8.4 Performance with Different Features

Figure 13 compares the use of some of the most useful features for the *state* of Sibyl in our H&L HSS configuration. All represents using all the six features in Table 1. Sibyl autonomously decides which features are important to maximize the performance of the running workload.
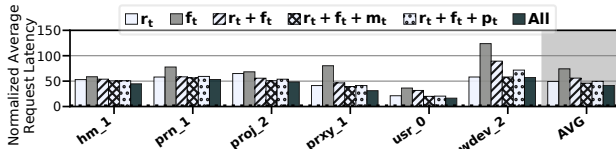

Figure 13: Average request latency when using different features (see Table 1) for the state space of Sibyl in the H&L HSS configuration (normalized to Fast-Only)

We make two key observations from Figure 13. First, Sibyl consistently achieves the lowest latency (up to 43.6% lower) by using all the features mentioned in Table 1 (All in Figure 13). Second, by using the same features as in baseline heuristic-based policies, Sibyl is able to perform better data placement decisions. For example, r_t and f_t configurations of Sibyl (in Figure 13) use only one feature, just like CDE and HPS do. These two Sibyl configurations outperform CDE and HPS policies by 4.9% and 5.5%, respectively (*ref.* Figure 9(b)). Using the same features as a heuristic-based policy, Sibyl autonomously finds a higher-performance dynamic policy that can maximize the reward function, which heuristic-based policies cannot possibly do. We conclude that Sibyl uses a richer set of features that can capture multiple aspects of a storage request to make better data placement decisions than a heuristic-based policy. RL reduces the design burden on system architects, as Sibyl autonomously learns to use the provided features to achieve the highest cumulative reward. In contrast, traditional heuristic-based policies use features to make rigid data placement decisions without any system-level feedback, and thus they underperform compared to Sibyl.

8.5 Performance with Different Hyper-Parameters

Figures 14(a), 14(b), and 14(c) show the effect of three critical hyper-parameters (discount factor, learning rate, and exploration rate) on Sibyl's throughput in H&M HSS configuration. Figure 14(a) shows that Sibyl's throughput drops sharply at $\gamma = 0$. At $\gamma = 0$, Sibyl gives importance only to the immediate reward and not at all to the long-term reward, leading to lower performance. We use $\gamma = 0.9$, where Sibyl is more forward-looking, giving enough weight to long-term rewards. Figure 14(b) shows that at a learning rate of $\alpha = 1e^{-4}$, Sibyl provides the best performance. The learning rate determines the rate at which training network weights are updated. Both too

slow and too fast updates are detrimental for adaptive learning and stable exploitation of a learned policy, respectively. Third, Figure 14(c) shows that the performance of Sibyl drops sharply if it performs exploration too frequently (i.e., $\epsilon = 1e^{-1}$) and thus does not sufficiently exploit its learned policy. Sibyl achieves the highest performance improvements for $1e^{-5} \leq \epsilon \leq 1e^{-2}$.
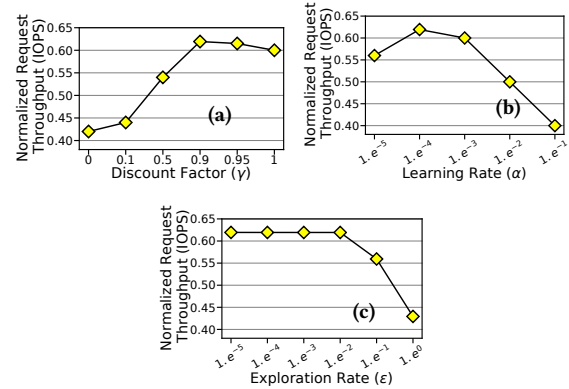

Figure 14: Sensitivity of Sibyl throughput to: (a) the discount factor (γ), (b) the learning rate (α), (c) the exploration rate (ϵ), averaged across 14 workloads (normalized to Fast-Only)

8.6 Sensitivity to Fast Storage Capacity

Figure 15 shows the average request latency of Sibyl and baseline policies as we vary the available capacity in the fast storage. The x-axis denotes a range of fast storage device sizes available for data placement and represented in terms of percentages of the entire fast storage device capacity, where 100% represents the size where all pages of a workload can fit in the fast storage.
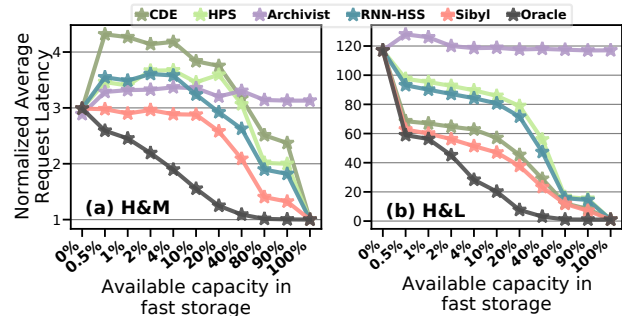

Figure 15: Average request latency for various fast storage device sizes (normalized to Fast-Only)

We make two observations. First, for all fast storage sizes, Sibyl performs better than the baseline heuristic- and supervised learning-based policies for both H&M and H&L HSS configurations. Even when the fast storage size is as small as 1%, Sibyl outperforms CDE, HPS, Archivist, RNN-HSS by 47.2% (11.5%), 17.3% (58.9%), 12.3% (110.1%), 21.7% (50.2%), respectively, in H&M (H&L) . Second, at a larger (smaller) fast storage device size, the performance approaches that of the Fast-Only (Slow-Only) policy, except for Archivist. Archivist classifies pages as hot or cold at the beginning of an epoch and does not change its placement decision throughout the execution of that epoch. It does not perform any promotion or eviction of data. We observe that Archivist often mispredicts

the target device for a request and classifies the same number of requests for the fast and slow storage device under different fast storage sizes.

As we vary the size of the fast storage device, a dynamically adaptable data placement policy is required, which considers features from both the running workload and the underlying storage system. We conclude that Sibyl can provide scalability by dynamically and effectively adapting its policy to the available storage size to achieve high performance.

8.7 Tri-Hybrid Storage Systems

We evaluate two different tri-HSS configurations, H&M&L and H&M&L$_{SSD}$ (Table 3), implemented as a single flat block device. The H&M&L$_{SSD}$ configuration has a low-end SSD (L$_{SSD}$), whose performance is lower than the H and M devices but higher than the L device. We restrict the available capacity of H and M to 5% and 10%, respectively, of the working set size of a given workload. This ensures data eviction from H and M devices once they are full. We compare the performance of Sibyl on a tri-hybrid system with a state-of-the-art heuristic-based policy [49, 76] that divides data into *hot*, *cold*, and *frozen* and places them respectively into H, M, and L devices.[8] Figure 16 shows the performance of the heuristic-based and Sibyl data placement policies.

(a) H&M&L configuration

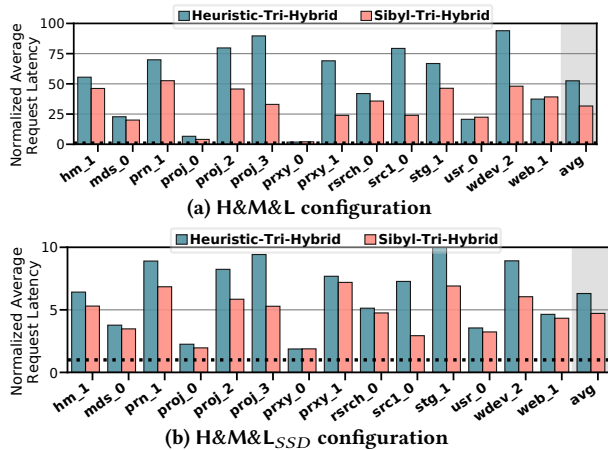(b) H&M&L$_{SSD}$ configuration

Figure 16: Average request latency for the tri-hybrid HSS (normalized to Fast-Only)

We observe that Sibyl outperforms the heuristic-based policy by, on average, 43.5% (48.2%) and 23.9% (25.2%) for H&M&L (H&M&L$_{SSD}$). This is because Sibyl is much more dynamic and adaptive to the storage system configuration due to its RL-based decision-making than the baseline heuristic-based policy, which is rigid in its decision-making. To extend Sibyl for three storage devices, we had to only (1) add a new action in Sibyl's action space, and (2) add the remaining capacity in the M device as a state feature. We conclude that Sibyl provides ease of extensibility to new storage system configurations, which reduces the system architect's burden in designing sophisticated data placement mechanisms.

9 EXPLAINABILITY ANALYSIS

We perform an explainability analysis to understand our results further and explain Sibyl's decisions. We extract Sibyl's actions

[8]CDE, HPS, Archivist, and RNN-HSS do *not* consider more than two devices and are not easily adaptable to a tri-hybrid HSS.

for different workloads under H&M and H&L HSS configurations and analyze the page placements for each workload. Figure 17 shows Sibyl's preference for the fast storage device over the slow storage device, measured as the ratio of the number of fast storage placements to the sum of the number of placements in both fast and slow storage devices (i.e., Preference=$\frac{\text{\#fast placements}}{\text{\#fast+\#slow placements}}$).
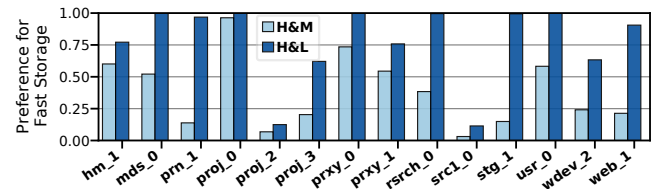
Figure 17: Sibyl's preference for the fast storage device under different HSS configurations

We make the following four observations. First, in the H&L configuration, where the latency difference is large between the two storage devices, Sibyl prefers to place more data in the fast storage device. Sibyl learns that despite the eviction penalty, the benefit of serving more requests from the fast storage device is significant. On the other hand, in the H&M device configuration, where the latency difference between two devices is smaller compared to H&L, Sibyl places only performance-critical pages in the faster storage device to avoid the eviction penalty.

Second, in the H&M configuration, Sibyl shows less preference to place pages from mds_0, prn_1, proj_2, proj_3, src1_0, stg_1, and web_1 in the fast storage device. These workloads are cold and sequential (Table 4) and thus are less suitable for the fast storage device. Therefore, for such workloads, Sibyl shows more preference for the slow storage device. In contrast, for hot and random workloads (prxy_0 and prxy_1), Sibyl shows more preference to place pages in the fast storage device.

Third, for rsrch_0, wdev_2, and web_1, Sibyl places ≤40% of pages in the fast storage device. Such requests have random access patterns, while pages with cold and sequential accesses are placed in the slow storage.

Fourth, in the H&L setting, Sibyl shows more preference to place requests in the fast storage device, except for proj_2 and src1_0 workloads. We observe that these two workloads are highly random with a low average access count (Table 4). Therefore, aggressive placement in the fast storage is not beneficial for long-term performance.

We also measure the number of evictions (as a fraction of all storage requests) that occur while using Sibyl and other baseline policies, as shown in Figure 18. We make two observations. First, in the H&M HSS configuration, Sibyl leads to 68.4%, 43.2%, 19.7%, and 29.3% fewer evictions from the fast storage than CDE, HPS, Archivist, and RNN-HSS, respectively. Second, CDE places more data in the fast storage, which leads to a large number of evictions in both HSS configurations. However, if the latency difference between the two devices is large (e.g., H&L configuration), CDE provides higher performance than other baseline policies (see Figure 9(b)). Therefore, in the H&L HSS configuration, we observe that Sibyl follows a similar policy, leading to more evictions compared to other baselines.
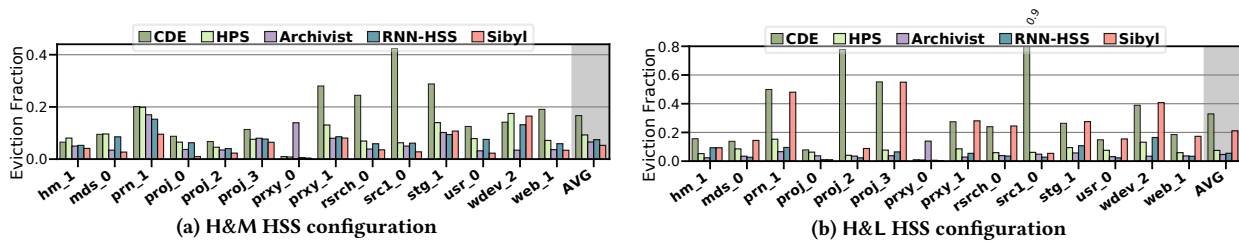
(a) H&M HSS configuration   (b) H&L HSS configuration

Figure 18: Comparison of evictions from the fast storage to the slow storage (normalized to the total number of storage requests)

10 OVERHEAD ANALYSIS

10.1 Inference and Training Latencies

The input layer of the training and inference networks consists of six neurons, equal to the number of features listed in Table 1. Each feature is normalized to transform the value range of different features to a common scale. The size of one state entry is 40 bits (32 bits for state features and 8 bits for the counter used for tracking the remaining capacity in the fast storage device). We make use of two hidden layers with 20 and 30 neurons each. The final output layer has neurons equivalent to our action space, i.e., two for dual-HSS configurations and three for the tri-HSS configurations.

Inference latency. Our inference network has 52 inference neurons (20+30+2) with 780 weights (6×20+20×30+30×2). As a result, Sibyl requires 780 MAC operations per inference (1×6×20+1×20×30 +1×30×2). On our evaluated CPU, we can perform these operations in ~10ns, which is several orders of magnitude smaller than the I/O read latency of even a high-end SSD (~10us) [94, 95]. Sibyl's inference computation can also be performed in the SSD controller.

Training latency. For each training step, Sibyl needs to compute 1,597,440 MAC operations, where each batch requires 128×6×20+ 128×20×30+128×30×2 MAC operations. This computation takes ~2us on our evaluated CPU. This training latency does not affect the benefits of Sibyl because (1) training occurs asynchronously with inference, and (2) training latency is ~5× smaller than the I/O read latency of even a high-end SSD.

We conclude that Sibyl's performance benefits come at small latency overheads that are easily realizable in existing CPUs.

10.2 Area Overhead

Storage cost. We use a half-precision floating-point format for the weights of the training and the inference networks. With 780 16-bit weights, each neural network requires 12.2 KiB of memory. Since we use the same network architecture for the two networks, we need 24.4 KiB of memory. In total, with an experience buffer of 100 KiB (§6.2), Sibyl requires 124.4 KiB of DRAM overhead, which is negligible compared to the memory size of modern computing systems.

Metadata cost. HSSs need to maintain the address mapping information for the underlying storage devices [148]. Sibyl requires 40 bits to store state information (i.e., the per-page state features; see Table 1). This overhead is ~0.1% of the total storage capacity when using a 4-KiB data placement granularity (5-byte per 4-KiB data). We conclude that Sibyl has a very modest cost in terms of storage capacity overhead in main memory (DRAM).

11 DISCUSSION

Cost of generality. We identify two main limitations of using RL for data placement. First, currently, RL is largely a *black-box policy*. Our explainability analysis (§9) tries to provide intuition into Sibyl's internal mechanism. However, providing rigorous explainability to reinforcement learning-based mechanisms is an active field of research [149–154], a problem that is beyond the scope of this paper. Perfectly finding worst-case workloads against an RL policy is, therefore, very difficult, in fact, impossible, given the state-of-the-art in reinforcement learning. There are many dynamic decisions that the agent performs, which cannot be easily explained or modeled in human-understandable terms. Second, Sibyl requires *engineering effort* to (1) thoroughly tune the RL hyper-parameters, and (2) implement and integrate Sibyl components into the host OS's storage management layer. This second limitation is not specific to Sibyl and applies to any ML-based storage management technique. As quantified in §10, Sibyl's storage and latency overheads are small.

Sibyl's implications. Sibyl (1) provides performance improvements on a wide variety of workloads and system configurations (our evaluations in §8 show that Sibyl outperforms all evaluated state-of-the-art data placement policies under all system configurations), (2) provides extensibility by reducing the designer burden when extending data placement policies to multiple devices and different storage configurations, and (3) enables reducing the fast storage device size by taking better advantage of the fast-yet-small storage device and large-yet-slow storage device to deliver high storage capacity at low latency.

Adding more features and optimization objectives. An RL-based approach simplifies adding new features (such as bandwidth utilization) in the RL state and optimization objectives (such as endurance) using the RL reward function. This flexibility allows an RL-based mechanism to self-optimize and adapt its decision-making policy to achieve an objective without the designer explicitly defining *how* to achieve it. We demonstrate and evaluate example implementations of Sibyl using a reward scheme that is a function of request latency and eviction latency. We find that request latency in the reward structure best encapsulates system conditions since latency could vary for each storage request based on complex system conditions. To optimize for a different device-level objective, one needs to define a new reward function with appropriate state features, e.g., to optimize for endurance, one might use the number of writes to an endurance-critical device in the reward function. Another interesting research direction would be to perform multi-objective optimization, e.g., optimizing for both performance and energy. We leave the study of different objectives and features to future work.

Necessity of the reward. RL training is highly dependent upon the quality of the reward function and state features. Using an incorrect reward or improper state features could lead to severe performance degradation. Creating the right reward is a human-driven effort that could benefit from design insights. We tried two other reward structures to achieve our objective to improve system performance:

- **Hit rate of the fast storage device:** Maximizing the hit rate of the fast storage device is another potentially plausible objective. However, if we use the hit rate as a reward, Sibyl (1) tries to aggressively place data in the fast storage device, which leads to unnecessary evictions, and (2) cannot capture the asymmetry in the latencies present in modern storage devices (e.g., due to read/write latencies, latency of garbage collection, queuing delays, error handling latencies, and write buffer state).
- **High negative reward for eviction:** We also tried a negative reward for eviction and a zero reward in other cases. We observe that such a reward structure provides suboptimal performance because Sibyl places more pages in the slow device to avoid evictions. Thus, with such a reward structure, Sibyl is not able to effectively utilize the fast storage.

We conclude that our chosen reward structure works well for a wide variety of workloads §8, as reinforced by our generality studies using unseen workloads in §8.2.

Managing hybrid main memory using RL. The key idea of Sibyl can be adapted for managing hybrid main memory architectures. However, managing data placement at different levels of the memory hierarchy has its own set of challenges [1, 155–167] that Sibyl would need to adapt to, such as the low latency decision-making and control requirements in main memory. Even with the use of hybrid main memories, many systems continue to benefit from using hybrid storage devices due to much lower cost-per-bit of storage, which accommodates increasingly larger datasets. Therefore, we focus on hybrid storage systems and leave it to future work to study RL to manage hybrid main memories.

12 RELATED WORK

To our knowledge, this is the first work to propose a reinforcement learning-based data placement technique for hybrid storage systems. Sibyl can continuously learn from and adapt to the running application and the storage configuration and device characteristics. We briefly discuss closely-related prior works that propose data management techniques for hybrid memory/storage systems and RL-based system optimizations.

Heuristic-based data placement. Many prior works [1, 14, 18, 20, 25, 49–57, 62, 65, 68, 70, 74, 76, 100, 155–157, 162, 168–176] propose heuristic-based techniques to perform data placement. These techniques rely on statically-chosen design features that usually favor certain workloads and/or device characteristics, leading to relatively rigid policies. In §3 and §8, we show that Sibyl outperforms two state-of-the-art works, CDE [49] and HPS [113].

ML-based data placement. Several works [58–60, 177, 178] propose ML-based techniques for data placement in hybrid memory/storage systems. These works 1) are based on supervised learning techniques that require frequent and very costly retraining to adapt to changing workload and device characteristics, and 2) have not been evaluated on a real system. We evaluate RNN-HSS, which is inspired by the state-of-the-art data placement technique in hybrid

main memory [58]. It uses sophisticated recurrent neural networks (RNNs) for data placement and shows promising results compared to heuristic-based techniques. However, it has two major limitations that make it impractical or difficult to implement: it (1) trains an RNN for each page, which leads to large computation, storage, and training time overheads, and (2) requires offline application profiling. Our evaluation (ref. §8.1) shows that Sibyl outperforms two state-of-the-art ML-based data placement techniques, RNN-HSS [58] and Archivist [59], across a wide variety of workloads.

RL-based techniques in storage systems. Recent works (e.g., [179–183]) propose the use of RL-based approaches for managing different aspects of storage systems. These works cater to use cases and objectives that are very different from Sibyl's. Specifically, Liu *et al.* [179] (1) propose data placement in cloud systems and *not* hybrid storage systems, (2) consider devices with unlimited capacity, sidestepping the capacity limitations, (3) *emulate* a data center network rather than use a real system for design and evaluation, and (4) focus only on data-analytics workloads. Yoo *et al.* [180] do *not* focus on data placement; they instead deal with dynamic storage resizing based on workload characteristics using a trace-based simulator. Wang *et al.* [181] (1) focus on cloud systems to predict the data storage consumption, and (2) do *not* consider hybrid storage systems. Sibyl is the first RL-based mechanism for data placement in hybrid storage systems.

RL-based system optimizations. Past works [88, 89, 184–195] propose RL-based methods for various system optimizations, such as memory scheduling [88, 192], data prefetching [89, 190], cache replacement [185], and network-on-chip arbitration [184, 186]. Along with Sibyl, designed for efficient data placement in hybrid storage systems, this body of work demonstrates that RL is a promising approach to designing high-performance, and highly-adaptive self-optimizing computing systems.

13 CONCLUSION

We introduce Sibyl, the first reinforcement learning-based mechanism for data placement in hybrid storage systems. Our extensive real-system evaluation demonstrates that Sibyl provides adaptivity and extensibility by continuously learning from and autonomously adapting to the workload characteristics, storage configuration and device characteristics, and system-level feedback to maximize the overall long-term performance of a hybrid storage system. We interpret Sibyl's policy through our explainability analysis and conclude that Sibyl provides an effective and robust approach to data placement in current and future hybrid storage systems. We hope that Sibyl and our open-sourced implementation of it [93] inspire future work and ideas in self-optimizing storage and memory systems.

REFERENCES

[1] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory," in *WEED*, 2013.

[2] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, "Exploring Storage Class Memory with Key Value Stores," in *SOSP*, 2013.

[3] C. W. Smullen, J. Coffman, and S. Gurumurthi, "Accelerating Enterprise Solid-State Disks With Non-Volatile Merge Caching," in *IGSC*, 2010.

[4] N. Lu, I.-S. Choi, S.-H. Ko, and S.-D. Kim, "A PRAM Based Block Updating Management for Hybrid Solid State Disk," in *ELEX*, 2012.

[5] M. Tarihi, H. Asadi, A. Haghdoost, M. Arjomand, and H. Sarbazi-Azad, "A Hybrid Non-Volatile Cache Design for Solid-State Drives Using Comprehensive I/O Characterization," in *TC*, 2015.

[6] W. Xiao, H. Dong, L. Ma, Z. Liu, and Q. Zhang, "HS-BAS: A Hybrid Storage System Based on Band Awareness of Shingled Write Disk," in *ICCD*, 2016.

[7] C. Wang, D. Wang, Y. Chai, C. Wang, and D. Sun, "Larger, Cheaper, but Faster: SSD-SMR Hybrid Storage Boosted by a New SMR-Oriented Cache Framework," in *MSST*, 2017.

[8] Z.-W. Lu and G. Zhou, "Design and Implementation of Hybrid Shingled Recording RAID System," in *PiCom*, 2016.

[9] D. Luo, J. Wan, Y. Zhu, N. Zhao, F. Li, and C. Xie, "Design and Implementation of a Hybrid Shingled Write Disk System ," in *TPDS*, 2015.

[10] M. Srinivasan, P. Saab, and V. Tkachenko, "Flashcache," in *Facebook*, 2010.

[11] D. Reinsel and J. Rydning, "Breaking the 15K-rpm HDD Performance Barrier with Solid State Hybrid Drives," in *IDC*, 2013.

[12] S. Lee, Y. Won, and S. Hong, "Mining-Based File Caching in a Hybrid Storage System," in *JISE*, 2014.

[13] W. Felter, A. Hylick, and J. Carter, "Reliability-Aware Energy Management for Hybrid Storage Systems," in *MSST*, 2011.

[14] K. Bu, M. Wang, H. Nie, W. Huang, and B. Li, "The Optimization of the Hierarchical Storage System Based on the Hybrid SSD Technology," in *ISDEA*, 2012.

[15] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "SSD Bufferpool Extensions for Database Systems," in *VLDB*, 2010.

[16] T. Bisson and S. A. Brandt, "Reducing Hybrid Disk Write Latency with Flash-Backed I/O Requests," in *MASCOTS*, 2007.

[17] M. Saxena, M. M. Swift, and Y. Zhang, "FlashTier: A Lightweight, Consistent and Durable Storage Cache," in *EuroSys*, 2012.

[18] K. Krish, B. Wadhwa, M. S. Iqbal, M. M. Rafique, and A. R. Butt, "On Efficient Hierarchical Storage for Big Data Processing," in *CCGrid*, 2016.

[19] D. Zhao, K. Qiao, and I. Raicu, "Towards Cost-Effective and High-Performance Caching Middleware for Distributed Systems," in *IJBDI*, 2016.

[20] L. Lin, Y. Zhu, J. Yue, Z. Cai, and B. Segee, "Hot Random Off-Loading: A Hybrid Storage System with Dynamic Data Migration," in *MASCOTS*, 2011.

[21] X. Chen, W. Chen, Z. Lu, P. Long, S. Yang, and Z. Wang, "A Duplication-Aware SSD-Based Cache Architecture for Primary Storage in Virtualization Environment," in *ISJ*, 2015.

[22] J. Niu, J. Xu, and L. Xie, "Hybrid Storage Systems: A Survey of Architectures and Algorithms," in *IEEE Access*, 2018.

[23] Y. Oh, E. Lee, C. Hyun, J. Choi, D. Lee, and S. H. Noh, "Enabling Cost-Effective Flash Based Caching with an Array of Commodity SSDs," in *Middleware*, 2015.

[24] Y. Liu, X. Ge, X. Huang, and D. H. Du, "MOLAR: A Cost-Efficient, High-Performance SSD-Based Hybrid Storage Cache," in *CLUSTER*, 2013.

[25] J. Tai, B. Sheng, Y. Yao, and N. Mi, "SLA-Aware Data Migration in a Shared Hybrid Storage Cluster," in *CC*, 2015.

[26] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen, "Improving Flash-Based Disk Cache with Lazy Adaptive Replacement," in *TOS*, 2016.

[27] T. Kgil and T. Mudge, "FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers," in *CASES*, 2006.

[28] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND Flash Based Disk Caches," in *ISCA*, 2008.

[29] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Caching Less For Better Performance: Balancing Cache Size and Update Cost of Flash Memory Cache in Hybrid Storage Systems," in *FAST*, 2012.

[30] J. Yang, N. Plasson, G. Gillis, N. Talagala, S. Sundararaman, and R. Wood, "HEC: Improving Endurance of High Performance Flash-Based Cache Devices," in *SYSTOR*, 2013.

[31] J. Ou, J. Shu, Y. Lu, L. Yi, and W. Wang, "EDM: An Endurance-Aware Data Migration Scheme for Load Balancing in SSD Storage Clusters," in *IPDPS*, 2014.

[32] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Cache, Cache Everywhere, Flushing All Hits Down the Sink: On Exclusivity in Multilevel, Hybrid Caches," in *MSST*, 2013.

[33] Y. Cheng, W. Chen, Z. Wang, X. Yu, and Y. Xiang, "AMC: An Adaptive Multi-Level Cache Algorithm in Hybrid Storage Systems," in *CCPE*, 2015.

[34] Y. Chai, Z. Du, X. Qin, and D. A. Bader, "WEC: Improving Durability of SSD Cache Drives by Caching Write-Efficient Data," in *TC*, 2015.

[35] N. Dai, Y. Chai, Y. Liang, and C. Wang, "ETD-Cache: An Expiration-Time Driven Cache Scheme to Make SSD-Based Read Cache Endurable and Cost-Efficient,"

in *CF*, 2015.

[36] F. Ye, J. Chen, X. Fang, J. Li, and D. Feng, "A Regional Popularity-Aware Cache Replacement Algorithm to Improve the Performance and Lifetime of SSD-Based Disk Cache," in *NAS*, 2015.

[37] H.-P. Chang, S.-Y. Liao, D.-W. Chang, and G.-W. Chen, "Profit Data Caching and Hybrid Disk-Aware Completely Fair Queuing Scheduling Algorithms For Hybrid Disks," in *SPE*, 2015.

[38] M. Saxena and M. M. Swift, "Design and Prototype of a Solid-State Cache," in *TOS*, 2014.

[39] Y. Li, L. Guo, A. Supratak, and Y. Guo, "Enabling Performance as a Service For a Cloud Storage System," in *CLOUD*, 2014.

[40] Z. Zong, R. Fares, B. Romoser, and J. Wood, "FastStor: Data-Mining-Based Multilayer Prefetching for Hybrid Storage Systems," in *CC*, 2014.

[41] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson, "Turbocharging DBMS Buffer Pool Using SSDs," in *SIGMOD*, 2011.

[42] D. Lee, C. Min, and Y. I. Eom, "Effective SSD Caching For High-Performance Home Cloud Server," in *ICCE*, 2015.

[43] S. H. Baek and K.-W. Park, "A Fully Persistent and Consistent Read/Write Cache Using Flash-Based General SSDs for Desktop Workloads," in *ICEIS*, 2016.

[44] Y. Liu, J. Huang, C. Xie, and Q. Cao, "RAF: A Random Access First Cache Management to Improve SSD-Based Disk Cache," in *NAS*, 2010.

[45] Y. Liang, Y. Chai, N. Bao, H. Chen, and Y. Liu, "Elastic Queue: A Universal SSD Lifetime Extension Plug-in for Cache Replacement Algorithms," in *SYSTOR*, 2016.

[46] G. Yadgar, M. Factor, K. Li, and A. Schuster, "Management of Multilevel, Multi-client Cache Hierarchies with Application Hints," in *TOCS*, 2011.

[47] Z. Zhang, Y. Kim, X. Ma, G. Shipman, and Y. Zhou, "Multi-level Hybrid Cache: Impact and Feasibility," in *ORNL Tech. Rep*, 2012.

[48] Y. Klonatos, T. Makatos, M. Marazakis, M. D. Flouris, and A. Bilas, "Azor: Using Two-Level Block Selection to Improve SSD-Based I/O Caches," in *NAS*, 2011.

[49] C. Matsui, C. Sun, and K. Takeuchi, "Design of Hybrid SSDs With Storage Class Memory and NAND Flash Memory," in *Proc. IEEE*, 2017.

[50] C. Sun, K. Miyaji, K. Johguchi, and K. Takeuchi, "A High Performance and Energy-Efficient Cold Data Eviction Algorithm for 3D-TSV Hybrid ReRAM/MLC NAND SSD," in *CAS*, 2013.

[51] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems," in *SC*, 2011.

[52] E. Vasilakis, V. Papaefstathiou, P. Trancoso, and I. Sourdis, "Hybrid2: Combining Caching and Migration in Hybrid Memory Systems," in *HPCA*, 2020.

[53] Y. Lv, X. Chen, G. Sun, and B. Cui, "A Probabilistic Data Replacement Strategy for Flash-Based Hybrid Storage System," in *APWeb*, 2013.

[54] Z. Li, "GreenDM: A Versatile Tiering Hybrid Drive for the Trade-Off Evaluation of Performance, Energy, and Endurance," Ph.D. dissertation, Stony Brook University, NY, 2014.

[55] J. Guerra, H. Pucha, J. S. Glider, W. Belluomini, and R. Rangaswami, "Cost Effective Storage Using Extent Based Dynamic Tiering," in *FAST*, 2011.

[56] A. Elnably, H. Wang, A. Gulati, and P. J. Varman, "Efficient QoS for Multi-Tiered Storage Systems," in *HotStorage*, 2012.

[57] H. Wang and P. Varman, "Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation," in *FAST*, 2014.

[58] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska, "Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence," in *HPDC*, 2019.

[59] J. Ren, X. Chen, Y. Tan, D. Liu, M. Duan, L. Liang, and L. Qiao, "Archivist: A Machine Learning Assisted Data Placement Mechanism for Hybrid Storage Systems," in *ICCD*, 2019.

[60] P. Cheng, Y. Lu, Y. Du, Z. Chen, and Y. Liu, "Optimizing Data Placement on Hierarchical Storage Architecture via Machine Learning," in *NPC*, 2019.

[61] A. Raghavan, A. Chandra, and J. B. Weissman, "Tiera: Towards Flexible Multi-Tiered Cloud Storage Instances," in *Middleware*, 2014.

[62] R. Salkhordeh, H. Asadi, and S. Ebrahimi, "Operating System Level Data Tiering Using Online Workload Characterization," in *JSC*, 2015.

[63] J. Hui, X. Ge, X. Huang, Y. Liu, and Q. Ran, "E-HASH: An Energy-Efficient Hybrid Storage System Composed of One SSD and Multiple HDDs," in *ICSI*, 2012.

[64] J. Xue, F. Yan, A. Riska, and E. Smirni, "Storage Workload Isolation via Tier Warming," in *ICAC*, 2014.

[65] G. Zhang, L. Chiu, C. Dickey, L. Liu, P. Muench, and S. Seshadri, "Automated Lookahead Data Migration in SSD-enabled Multi-tiered Storage Systems," in *MSST*, 2010.

[66] X. Zhao, Z. Li, and L. Zeng, "FDTM: Block Level Data Migration Policy in Tiered Storage System," in *NPC*, 2010.

[67] H. Shi, R. V. Arumugam, C. H. Foh, and K. K. Khaing, "Optimal Disk Storage Allocation for Multitier Storage System," in *TMAG*, 2013.

[68] X. Wu and A. N. Reddy, "Data Organization in a Hybrid Storage System," in *ICNC*, 2012.

[69] S. Ma, H. Chen, Y. Shen, H. Lu, B. Wei, and P. He, "Providing Hybrid Block Storage for Virtual Machines using Object-based Storage," in *ICPADS*, 2014.

[70] I. Iliadis, J. Jelitto, Y. Kim, S. Sarafijanovic, and V. Venkatesan, "ExaPlan: Queueing-Based Data Placement and Provisioning for Large Tiered Storage Systems," in *MASCOTS*, 2015.

[71] X. Wu and A. N. Reddy, "Managing Storage Space in a Flash and Disk Hybrid Storage System," in *MASCOTS*, 2009.

[72] X. Wu and A. N. Reddy, "Exploiting Concurrency to Improve Latency and throughput in a Hybrid Storage System," in *MASCOTS*, 2010.

[73] D. Park and D. H. Du, "Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters," in *MSST*, 2011.

[74] Y. Lv, B. Cui, X. Chen, and J. Li, "Hotness-Aware Buffer Management For Flash-Based Hybrid Storage Systems," in *CIKM*, 2013.

[75] D. Montgomery, "Extent Migration For Tiered Storage Architecture," in *USPTO*, 2014.

[76] C. Matsui, T. Yamada, Y. Sugiyama, Y. Yamaga, and K. Takeuchi, "Tri-Hybrid SSD with Storage Class Memory (SCM) and MLC/TLC NAND Flash Memories," *Proc. IEEE*, 2017.

[77] Wikipedia, "Sibyl, https://en.wikipedia.org/wiki/Sibyl."

[78] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2018.

[79] M. Minsky, "Steps Toward Artificial Intelligence," in *Proc. IRE*, 1961.

[80] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *HPCA*, 2015.

[81] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in *MICRO*, 2009.

[82] M. Jung, E. H. Wilson, D. Donofrio, J. Shalf, and M. T. Kandemir, "NANDFlashSim: Intrinsic Latency Variation Aware NAND Flash Memory System Modeling and Simulation at Microarchitecture Level," in *MSST*, 2012.

[83] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, O. Unsal, A. Cristal, and K. Mai, "Neighbor-Cell Assisted Error Correction for MLC NAND Flash Memories," in *SIGMETRICS*, 2014.

[84] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives," in *Proc. IEEE*, 2017.

[85] J. Cui, Y. Zhang, W. Wu, J. Yang, Y. Wang, and J. Huang, "DLV: Exploiting Device Level Latency Variations for Performance Improvement on Flash Memory Storage Systems," in *TCAD*, 2017.

[86] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, "Reliability Issues in Flash-Memory-Based Solid-State Drives: Experimental Analysis, Mitigation, Recovery," in *Inside Solid State Drives (SSDs)*, 2018.

[87] J. Park, M. Kim, M. Chun, L. Orosa, J. Kim, and O. Mutlu, "Reducing Solid-State Drive Read Latency by Optimizing Read-Retry," in *ASPLOS*, 2021.

[88] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," in *ISCA*, 2008.

[89] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning," in *MICRO*, 2021.

[90] A. Zell, *Simulation neuronaler netze*. Addison-Wesley Bonn, 1994.

[91] "MSR Cambridge Traces., http://iotta.snia.org/traces/388."

[92] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A Flexible Framework for File System Benchmarking," 2016.

[93] CMU-SAFARI, "Sibyl, https://github.com/CMU-SAFARI/Sibyl."

[94] Intel, "Intel Optane SSD DC P4801X Series, https://ark.intel.com/content/www/us/en/ark/products/149365/intel-optane-ssd-dc-p4801x-series-100gb-2-5in-pcie-x4-3d-xpoint.html."

[95] Samsung, "Ultra-Low Latency with Samsung Z-NAND SSD, https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf."

[96] Intel, "Intel SSD D3-S4510 Series, https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/d3-series/d3-s4510-series/d3-s4510-1-92tb-2-5inch-3d2.html."

[97] Intel, "Intel SSD 660p Series, https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/consumer-ssds/660p-series-brief.html."

[98] Seagate, "Seagate Barracuda Datasheet, https://www.seagate.com/www-content/datasheets/pdfs/3-5-barracuda-3tbDS1900-10-1710US-en_US.pdf."

[99] ADATA, "ADATA Ultimate Series: SU630, https://shop.adata.com/adata-ultimate-series-su630-960gb-sata-iii-internal-2-5-solid-state-drive/."

[100] Z. Feng, Z. Feng, X. Wang, G. Rao, Y. Wei, and Z. Li, "HDStore: An SSD/HDD Hybrid Distributed Storage Scheme for Large-Scale Data," in *WAIM*, 2014.

[101] R. Micheloni, L. Crippa, and M. Picca, "Hybrid Storage Systems," in *Inside Solid State Drives (SSDs)*, 2018.

[102] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches," in *FAST*, 2014.

[103] K. Tsuchida, T. Inaba, K. Fujita, Y. Ueda, T. Shimizu, Y. Asao, T. Kajiyama, M. Iwayama, K. Sugiura, S. Ikegawa *et al.*, "A 64Mb MRAM with Clamped-Reference and Adequate-Reference Schemes," in *ISSCC*, 2010.

[104] A. Kawahara, R. Azuma, Y. Ikeda, K. Kawai, Y. Katoh, Y. Hayakawa, K. Tsuji, S. Yoneda, A. Himeno, K. Shimakawa *et al.*, "An 8Mb Multi-Layered Cross-Point ReRAM Macro with 443MB/s Write Throughput," in *ISSCC*, 2012.

[105] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo *et al.*, "A 20nm 1.8V 8Gb PRAM with 40MB/s Program Bandwidth," in *ISSCC*, 2012.

[106] Y. Oh, E. Lee, J. Choi, D. Lee, and S. H. Noh, "Hybrid Solid State Drives for Improved Performance and Enhanced Lifetime," in *MSST*, 2013.

[107] H. G. Lee, "High-Performance NAND and PRAM Hybrid Storage Design for Consumer Electronics," in *TCE*, 2010.

[108] S. Okamoto, C. Sun, S. Hachiya, T. Yamada, Y. Saito, T. O. Iwasaki, and K. Takeuchi, "Application Driven SCM & NAND Flash Hybrid SSD Design for Data-Centric Computing System," in *IMW*, 2015.

[109] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," in *CSUR*, 2005.

[110] N. Express, "Everything You Need to Know About the NVMe 2.0 Specifications and New Technical Proposals," https://nvmexpress.org/.

[111] S. A. W. Group, "Serial ATA: High Speed Serialized AT Attachment," https://web.archive.org/web/20161009182351/http://www.ece.umd.edu/courses/enee759h.S2003/references/serialata10a.pdf.

[112] Linux Source Code (v5.10.16), https://elixir.bootlin.com/linux/latest/source.

[113] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-stacked and Off-package Memories," in *HPCA*, 2015.

[114] W. Cheong, C. Yoon, S. Woo, K. Han, D. Kim, C. Lee, Y. Choi, S. Kim, D. Kang, G. Yu, J. Kim, J. Park, K. Song, K. Park, S. Cho, H. Oh, D. D. G. Lee, J. Choi, and J. Jeong, "A Flash Memory Controller for 15μs Ultra-Low-Latency SSD Using High-Speed 3D NAND Flash with 3μs Read Time," in *ISSCC*, 2018.

[115] Micron, "3D XPoint Technology, https://www.micron.com/products/advanced-solutions/3d-xpoint-technology."

[116] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform Storage Performance With 3D XPoint Technology," in *Proc. IEEE*, 2017.

[117] Intel, "Intel SSD DC P4610 Series, https://ark.intel.com/content/www/us/en/ark/products/140103/intel-ssd-dc-p4610-series-1-6tb-2-5in-pcie-3-1-x4-3d2-tlc.html."

[118] C. J. C. H. Watkins, "Learning From Delayed Rewards," Ph.D. dissertation, 1989.

[119] C. J. Watkins and P. Dayan, "Q-learning," in *ML*, 1992.

[120] G. A. Rummery and M. Niranjan, *On-line Q-Learning Using Connectionist Systems*. Citeseer, 1994.

[121] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the Game of Go With Deep Neural Networks and Tree Search," in *Nature*, 2016.

[122] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the Game of Go Without Human Knowledge," in *Nature*, 2017.

[123] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," in *NIPS*, 2013.

[124] S. Liang and R. Srikant, "Why Deep Neural Networks for Function Approximation?" in *arXiv*, 2016.

[125] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy Gradient Methods for Reinforcement Learning With Function Approximation," in *NIPS*, 1999.

[126] L. Baird, "Residual Algorithms: Reinforcement Learning With Function Approximation," in *ML*, 1995.

[127] K. Kira and L. A. Rendell, "A Practical Approach to Feature Selection," in *ML*, 1992.

[128] D. Silver, S. Singh, D. Precup, and R. S. Sutton, "Reward is Enough," *AI*, 2021.

[129] Y. Zhong, X. Shen, and C. Ding, "Program Locality Analysis Using Reuse Distance," in *TOPLAS*, 2009.

[130] L. Bottou, "Stochastic Learning," in *Advanced Lectures on Machine Learning*, 2003.

[131] M. G. Bellemare, W. Dabney, and R. Munos, "A Distributional Perspective on Reinforcement Learning," in *arXiv*, 2017.

[132] D. J. Harrold, J. Cao, and Z. Fan, "Data-Driven Battery Operation For Energy Arbitrage Using Rainbow Deep Reinforcement Learning," in *Energy*, 2021.

[133] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-Level Control Through Deep Reinforcement Learning," in *Nature*, 2015.

[134] M. Tokic and G. Palm, "Value-Difference Based Exploration: Adaptive Control between Epsilon-Greedy and Softmax," in *AAAI*, 2011.

[135] G. Bebis and M. Georgiopoulos, "Feed-Forward Neural Networks," in *IEEE Potentials*, 1994.

[136] J. De Villiers and E. Barnard, "Backpropagation Neural Nets with One and Two Hidden Layers," in *IEEE Trans. Neural Netw. Learn. Sys*, 1993.

[137] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for Activation Functions," in *arXiv*, 2017.

[138] A. F. Agarap, "Deep Learning using Rectified Linear Units (ReLU)," in *arXiv*, 2018.

[139] T. L. Paine, C. Paduraru, A. Michi, C. Gulcehre, K. Zolna, A. Novikov, Z. Wang, and N. de Freitas, "Hyperparameter Selection For Offline Reinforcement Learning," in *arXiv*, 2020.

[140] G. Singh, J. Gómez-Luna, G. Mariani, G. F. Oliveira, S. Corda, S. Stuijk, O. Mutlu, and H. Corporaal, "NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning," in *DAC*, 2019.

[141] S. Arlot and A. Celisse, "A Survey of Cross-Validation Procedures for Model Selection," *SS*, 2010.

[142] D. C. Montgomery, "Design and Analysis of Experiments," 2017.

[143] Linux Mint 20.1 "Ulyssa", https://linuxmint.com/edition.php?id=284.

[144] S. C. Tweedie *et al.*, "Journaling the Linux ext2fs Filesystem," in *The Fourth Annual Linux Expo*, 1998.

[145] S. Guadarrama, A. Korattikara, O. Ramirez, P. Castro, E. Holly, S. Fishman, K. Wang, E. Gonina, N. Wu, E. Kokiopoulou, L. Sbaiz, J. Smith, G. Bartók, J. Berent, C. Harris, V. Vanhoucke, and E. Brevdo, "TF-Agents: A Library for Reinforcement Learning in TensorFlow, https://github.com/tensorflow/agents," 2018.

[146] AMD, "AMD Ryzen™ 7 PRO 2700 Processor, https://www.amd.com/en/products/cpu/amd-ryzen-7-2700."

[147] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems With YCSB," in *SOCC*, 2010.

[148] S. Tsukada, H. Takayashiki, M. Sato, K. Komatsu, and H. Kobayashi, "A Metadata Prefetching Mechanism for Hybrid Memory Architectures," in *COOL CHIPS*, 2021.

[149] A. Verma, V. Murali, R. Singh, P. Kohli, and S. Chaudhuri, "Programmatically Interpretable Reinforcement Learning," in *ICML*, 2018.

[150] G. Liu, O. Schulte, W. Zhu, and Q. Li, "Toward Interpretable Deep Reinforcement Learning With Linear Model u-Trees," in *ECML PKDD*, 2018.

[151] Z. Juozapaitis, A. Koul, A. Fern, M. Erwig, and F. Doshi-Velez, "Explainable Reinforcement Learning via Reward Decomposition," in *IJCAI-ECAI*, 2019.

[152] P. Madumal, T. Miller, L. Sonenberg, and F. Vetere, "Explainable Reinforcement Learning Through a Causal Lens," in *AAAI*, 2020.

[153] P. Sequeira and M. Gervasio, "Interestingness Elements For Explainable Reinforcement Learning: Understanding Agents' Capabilities and Limitations," in *AI*, 2020.

[154] E. Puiutta and E. Veith, "Explainable Reinforcement Learning: A Survey," in *CD-MAKE*, 2020.

[155] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu, "Utility-Based Hybrid Memory Management," in *CLUSTER*, 2017.

[156] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs Within Heterogeneous Memory Systems," in *ASPLOS*, 2015.

[157] N. Agarwal and T. F. Wenisch, "Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory," in *ASPLOS*, 2017.

[158] B. Goglin, "Exposing the Locality of Heterogeneous Memory Architectures to HPC Applications," in *MEMSYS*, 2016.

[159] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated Control for Energy-Efficient and Heterogeneous Memory Systems," in *HPCA*, 2013.

[160] F. X. Lin and X. Liu, "memif: Towards Programming Heterogeneous Memory Asynchronously," in *ASPLOS*, 2016.

[161] K. T. Malladi, U. Kang, M. Awasthi, and H. Zheng, "DRAMScale: Mechanisms to Increase DRAM Capacity," in *MEMSYS*, 2016.

[162] M. Pavlovic, N. Puzovic, and A. Ramirez, "Data Placement in HPC Architectures With Heterogeneous Off-Chip Memory," in *ICCD*, 2013.

[163] A. J. Pena and P. Balaji, "Toward the Efficient Use of Multiple Explicitly Managed Memory Subsystems," in *CLUSTER*, 2014.

[164] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *ISCA*, 2009.

[165] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *ICCD*, 2012.

[166] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," in *IEEE CAL*, 2012.

[167] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems," in *MICRO*, 2015.

[168] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated Control for Energy-Efficient and Heterogeneous Memory Systems," in *HPCA*, 2013.

[169] Z. Yang, M. Hoseinzadeh, A. Andrews, C. Mayers, D. T. Evans, R. T. Bolt, J. Bhimani, N. Mi, and S. Swanson, "AutoTiering: Automatic Data Placement Manager in Multi-Tier All-Flash Datacenter," in *IPCCC*, 2017.

[170] C. Chou, A. Jaleel, and M. K. Qureshi, "BATMAN: Maximizing Bandwidth Utilization of Hybrid Memory Systems." Citeseer, 2015.

[171] Y. Kim, A. Gupta, B. Urgaonkar, P. Berman, and A. Sivasubramaniam, "HybridStore: A Cost-Efficient, High-Performance Storage System Combining SSDs and HDDs," in *MASCOTS*, 2011.

[172] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu, "Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories," in *PLDI*, 2019.

[173] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *ICS*, 2011.

[174] L. Liu, S. Yang, L. Peng, and X. Li, "Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems," in *TPDS*, 2019.

[175] Y. Luo, P. Jin, and S. Wan, "Optimal Data Placement for Data-Centric Algorithms on NVM-Based Hybrid Memory," in *DSAA*, 2020.

[176] T. D. Doudali, D. Zahka, and A. Gavrilovska, "Cori: Dancing to the Right Beat of Periodic Data Movements over Hybrid Memory Systems," in *IPDPS*, 2021.

[177] M. Shetti, B. Li, and D. Du, "Machine Learning-based Adaptive Migration Algorithm for Hybrid Storage Systems," in *TOS*, 2019.

[178] S. Sen and N. Imam, "Machine Learning Based Design Space Exploration for Hybrid Main-Memory Design," in *MEMSYS*, 2019.

[179] K. Liu, J. Peng, J. Wang, B. Yu, Z. Liao, Z. Huang, and J. Pan, "A Learning-Based Data Placement Framework for Low Latency in Data Center Networks," in *TCC*, 2019.

[180] S. Yoo and D. Shin, "Reinforcement Learning-Based SLC Cache Technique for Enhancing SSD Write Performance," in *USENIX HotStorage*, 2020.

[181] H. Wang, H. Shen, Q. Liu, K. Zheng, and J. Xu, "A Reinforcement Learning Based System for Minimizing Cloud Storage Service Cost," in *ICPP*, 2020.

[182] W. Kang, D. Shin, and S. Yoo, "Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD," in *TECS*, 2017.

[183] W. Kang and S. Yoo, "Dynamic Management of Key States for Reinforcement Learning-assisted Garbage Collection to Reduce Long Tail Latency in SSD," in *DAC*, 2018.

[184] T.-R. Lin, D. Penney, M. Pedram, and L. Chen, "A Deep Reinforcement Learning Framework for Architectural Exploration: A Routerless NoC Case Study," in *HPCA*, 2020.

[185] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An Imitation Learning Approach for Cache Replacement," in *ICML*, 2020.

[186] J. Yin, Y. Eckert, S. Che, M. Oskin, and G. Loh, "Toward More Efficient NoC Arbitration: A Deep Reinforcement Learning Approach," in *AIDArc*, 2018.

[187] Q. Fettes, M. Clark, R. Bunescu, A. Karanth, and A. Louri, "Dynamic Voltage and Frequency Scaling in NoCs With Supervised and Reinforcement Learning Techniques," in *TC*, 2018.

[188] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae *et al.*, "Chip Placement With Deep Reinforcement Learning," *Nature*, 2021.

[189] O. Mutlu, "Intelligent Architectures for Intelligent Computing Systems," in *DATE*, 2021.

[190] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic Locality and Context-based Prefetching Using Reinforcement Learning," in *ISCA*, 2015.

[191] J. F. Martinez and E. Ipek, "Dynamic Multicore Resource Management: A Machine Learning Approach," in *IEEE Micro*, 2009.

[192] J. Mukundan and J. F. Martinez, "MORSE: Multi-Objective Reconfigurable Self-Optimizing Memory Scheduler," in *HPCA*, 2012.

[193] R. Jain, P. R. Panda, and S. Subramoney, "Machine Learned Machines: Adaptive Co-optimization of Caches, Cores, and On-chip Network," in *DATE*, 2016.

[194] H. Zheng and A. Louri, "Agile: A Learning-Enabled Power And Performance-Efficient Network-On-Chip Design," in *TETCI*, 2020.

[195] S. M. PD, H. Yu, H. Huang, and D. Xu, "A Q-Learning Based Self-Adaptive I/O Communication for 2.5D Integrated Many-Core Microprocessor and Memory," in *IEEE TOC*, 2015.