Just-in-time Intelligent Tiering for Emerging Storage Systems

Lu Pang[1], Anis Alazzawe[1], Madhurima Ray[2], Krishna Kant[1], and Jeremy Swift[3]

[1]Temple University, [2]Intel Corporations, [3]Dell EMC Inc.

***Abstract*—Multi-tiered storage is used in enterprises to reduce data access latency. In these storage systems, data is moved from a lower to a higher tier in an effort to reduce the average latency. The decision used to determine which data elements to move to the higher tier is usually done on a nightly basis. In this paper, we explore a Just-in-time Intelligent Tiering (JIT) mechanism that operates on a much shorter time scale than traditional tiering. The JIT mechanism generates a set of candidate movements and uses a control method to further refine the candidates. Based on extensive simulations in a 3-tier system consisting of HDD, SSD, and Intel Optane drives, we show that the proposed scheme can reduce the average latency by up to 76%.**

***Index Terms*—Storage System, Tiering, Data migration**

I. Introduction

Modern storage systems need to deal with an increasing amount of data volume, yet, only a very small fraction of the data is needed frequently for the duration of a task. With many established and emerging storage technologies that provide different levels of capacities and performance, it is natural to organize a storage system as a hierarchy where the slowest (but usually the cheapest and hence largest capacity) devices lie at the bottom and the fastest (and most expensive, and hence smallest capacity) devices at the top. Intelligent movement of data across these *tiers* is crucial for achieving the best possible performance at the lowest cost. Traditionally, tiering has been done on a very coarse time granularity (e.g., nightly). In this paper, we consider a much more agile (and hence finer grain) intelligent tiering that can take advantage of the emerging high-speed storage technologies.

In particular, we introduce a novel mechanism called **J**ust-in-time **I**ntelligent **T**iering (JIT) and show how it can lead to substantially lower IO latencies. The JIT mechanism is a tiering mechanism that makes migration decisions, in fixed short-term intervals, throughout the day. It works on the block-level user requests and has no visibility to file-level details. JIT performs this migration in two steps: (a) providing a set of data movement candidates by learning to predict the short-term relative order of accesses to data elements, and (b) estimating the long-term benefit of migrating a subset of the proposed movements to a different tier and making those migrations if it is advantageous. Since JIT learns end-to-end, it avoids the need of manually developing and tweaking features to feed the model.

The rest of the paper is organized as follows. Section III discusses the related works. Section IV describes the sophisticated simulation environment that we built for multi-tiered storage systems. Section V describes the challenges and essential details of the JIT method. Section VI then presents the implementation details. Section VII discusses the evaluation results using several storage traces. Finally, section VIII concludes the paper.

II. Tiering in Storage Systems

A. Traditional Tiering

In current systems, the hard disk drives (HDDs) form the bottom layer of the tiering hierarchy, and the solid-state drives (SSDs), based on NAND "flash" technology, form the top layer (i.e., 2-layer hierarchy). The emergence of SSDs as increasingly inexpensive and faster storage makes them ideal as a high tier in SSD-HDD tiering arrangements; however, this hierarchy is still limited by the rather high latency and low transfer rates of HDD if the popularity of the data changes frequently.

Traditionally, tiering decisions are made infrequently, commonly on a nightly basis. In particular, the system observes the popularity of IO blocks or objects over the entire day, and during low-traffic late night hours, the items are moved up or down the hierarchy based on a statistical method that records their popularity, such as least recently used (LRU) or least frequently used (LFU). One key reason for relegating tiering related data movement to low traffic periods has been the very low IO bandwidth and high latency for HDDs. For 4KB IOs, a typical HDD can provide $\approx 25K$ IO/sec (IOPS) for sequential transfers and as low as 1/100th as much for random transfers, whereas the latencies can be as high as ≈ 5ms (depending on the seek and rotational latency components). HDD can achieve higher bandwidth by striping data across many drives.

With the emergence of newer storage technologies, most notably the Intel Optane based disk drives, it is attractive to use this technology as the top tier, because of its ability to provide far lower latency than an SSD ($\approx .10\mu s$ vs. $> 100\mu s$ for SSDs). The highest two tiers have comparable but hugely higher IO rates than HDD (e.g., $\approx 1M$ 4KB IOPS, and no substantial difference between sequential and random IO). However, because of the much higher cost of Optane, this tier must necessarily be much smaller.

B. Motivation for JIT

Modern tiering systems consist of devices with very high data rates and must accommodate highly dynamic workloads. The adequacy of traditional tiering needs to be questioned.

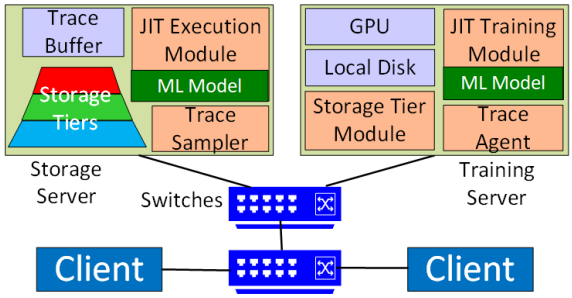


Fig. 1. Illustration of JIT Deployment in a Storage System

In particular, simply identifying complex patterns of access and using them for traditional tiering overnight is inadequate since the patterns themselves tend to change at much shorter time scales. This provides the motivation for a much shorter time scale tiering, that we call **J**ust-in-time **I**ntelligent **T**iering (JIT). Fortunately, the enormous IO bandwidth provided by solid-state storage technologies can easily accommodate the much higher tiering frequencies.

The key idea of JIT is to monitor the popularity of data, and use it for making movement decisions over a short observation time window, henceforth called the JIT Window. The JIT window can be on the order of minutes or hours versus the traditional tiering window which happens over 24 hours. While this seems like a simple change, it has profound implications for the storage system and requires study. The key issues to consider are (a) the in-line data movement and its management may place substantial stress on the storage server and may interfere with the normal IO; and (b) since the popularity of data over the rather short JIT window can be unreliable, a simplistic frequency based movement no longer suffices.

The basic unit of IO is typically a "block" of size 512B or 4,096B, addressed using a sequential number called LBA (Logical Block Address). However, for tiering purposes, keeping track of individual LBAs is not practical in a large storage system. Instead, it is more practical to use a much larger unit typically of several MB, which we call a "chunk". Unnecessary movement of large chunks can be costly, not only in terms of IO bandwidth used but also could delay normal user requests.

The optimal JIT window duration is a complex function of many parameters and is beyond the scope of this paper; however, we shall discuss the considerations in our choice.

C. Architecture of JIT System and Scope of Work

JIT mechanism employs a migration policy that consists of two parts. The first one generates chunk movement suggestions based on the expected popularity of the chunks that are learned from the user IO traffic and the condition of the storage system. The suggestions are fed to the second component which refines the proposed movements, based on the long-term benefits, and issues the actual movement commands. The features required for both parts are hard to engineered directly and we opt to learn those features by having each part interacts with the storage system (as discussed later). This would need extensive training using the real user IO traffic to make good decisions. However, live training in the storage system poses three potential adverse impacts: (a) poor movement decisions until much of the training is done, (b) overhead of capturing the IO requests to feed to the algorithm, and (c) CPU, memory, and storage consumption associated with the training itself. Given the criticality of IO performance in production systems, none of these overheads are likely to be acceptable.

We thus propose a two-stage process for operating our mechanism as shown in Fig. 1. In addition to the storage server (SS) we have a training server (TS) located on the same network segment as the SS. We envision the TS to be a self-contained appliance that is equipped with a local disk, CPU, memory, and a GPU to handle the training load. The TS uses a *storage tier module* that simulates the real storage hierarchy of SS. This module captures the queuing delays experienced by the user IO and chunk transfer requests. It also represents each device simplistically in terms of its observed read/write latency and bandwidth. The storage tier module only keeps the status of each chunk (and requested LBAs) but does not keep the actual data of the chunks. The SS collects the trace segments using a sampling strategy and periodically send them to the TS so the trace space requirements at SS remain small.

Such a system would initially use traditional tiering in the SS while the model is training. The trained model is then transferred to SS which then uses it with the JIT execution module for making real tiering decisions. Querying the trained model is not expensive and does not have any appreciable impact on the SS. Meanwhile, the model in TS could continue to train further, possibly with occasional updates to the simulation parameters (e.g. IO bandwidth), if appropriate. The updated model could be transferred to SS occasionally (e.g., nightly or once a week) so that the JIT can easily track any evolution in the workload.

The purpose of this paper is only to examine the operation of the TS and confirm the benefits of JIT over the traditional tiering with respect to overall latency experienced by the user request. Building out the entire solution as depicted in Fig. 1 is out of the scope of this paper and will be reported in later works. For this reason, this paper only concerns the simulation of the tiering system using publicly available storage traces.

III. Related Works

To the best of our knowledge, the technique proposed in this paper is novel and has not been reported elsewhere, even though specific elements (i.e., reinforcement learning, tiering, etc.) are well explored.

Curator [1] comes closest to our work. It applies reinforcement learning (RL) to decide a hotness threshold for moving data from HDD to SSD. To identify cold data (to be kept in HDD), it uses a mechanism similar to map-reduce to build global visibility. In this RL application, the states, rewards, and actions represent the resource usage, the amount of latency reduction, and whether or not to perform tiering respectively. In contrast, we use the RL to decide not just when to move, it also decides what to move and to which tier.

Herodotou, et.al. [4] use an automated approach to upgrade and downgrade files based on access to a storage tier of a dis-

tributed file system like HDFS. Authors use machine learning (ML) i.e. gradient boosted trees for file access tracking and prediction; uses file-level metadata and access for decision making; and migrates the entire file. In contrast, we migrate fixed size chunks, since the storage system does not have any file-level information.

Likewise, Vengerov [12] uses RL to learn the utility functions in the context of dynamic data migration in a hierarchical storage system. The author uses RL to tune the parameters of the tier cost function on which the migration decision depends. The goal of the work is to design an automated migration policy to minimize the average response time of the system.

Tsai, et.al [11] provide a resource allocation strategy to deal with heterogeneity across storage servers. A score is projected for every access based on the degree of randomness, frequency, and location in the storage tier. The authors then formulate an ILP to minimize response time and use the heuristic to get the resource allocation map. Based on the system load their technique performs a mix of minimal and full migration across tiers. In a cloud storage environment solving a complete placement problem and applying those outcomes seems unrealistic.

Noel, et.al [8] suggest ML based adaptation process for load balancing and migration where the system tunes its parameters as workload changes. The method detects possible hotspots, workload-interference and uses a stochastic policy gradient based RL to learn long-term policies to fix these bottlenecks. But long-term policies do not capture short-term changes in the workloads. Hence, we consider both short and long-term workload behaviors for migration decisions.

Ziggurat [15] uses an intelligent data placement policy that places small, synchronous writes to NVM and large asynchronous writes to the disks for better space usage and performance. It uses two separate predictors for synchronicity and size to predict the future access behavior and accordingly steer the writes to the appropriate tier. Eventually, it coalesces cold data from NVM to free up space and also move read dominated traffic from HDD to NVM tier, but the priority here is the placement of the data and not migration.

AutoTiering [13] is used for virtual machine file (VMDK) placement and migration in a multi-tier all flash storage array to optimize the resource usage, performance, and migration overhead by reducing the number of VMDK file migration as IOPS increase. In our work, we migrate only necessary data at a granularity much smaller than that of the VDMK files which are usually large in size.

IV. Detailed JIT Architecture

In this section, we discuss the details of the JIT training module, storage tier module, and trace agent (see Fig 1). The JIT execution module uses the trained model to migrate data to the appropriate tier in the real storage hierarchy. Each device in the storage tier model is simulated by a separate process. The trace agent is responsible for replaying the historical traces to train the JIT system.

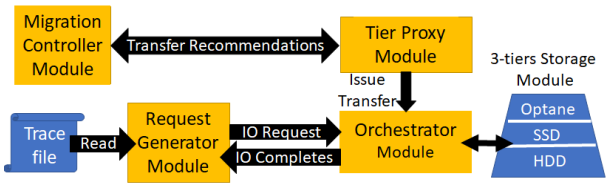


Fig. 2. JIT Architecture

In our simulator, the functionality of the three modules in the JIT deployment is simulated with five primary modules. The trace agent and storage tier module are simulated by the request generator module and storage module respectively. The functionality of the JIT training module is simulated across three modules: an orchestrator module, a migration controller module, a tier proxy module that facilitates transactions between the controller and the orchestrator. We built our discrete event simulator (DES) using SimPy [9].

Request Generator Module: The request generator module plays a similar role to the trace agent in the JIT deployment. As shown in Fig 2, the request generator module reads the trace files, generates IO requests, and sends these requests to the orchestrator. An IO request can be described with a 5-tuple: (request id, issuing timestamp, logical block address, request size, request type). As a request completes, the completion comes back to the request generator which then captures the completion time of the IO. This concludes the life cycle of an IO request.

Storage Module: In the storage module, we assume that each tier contains one type of device. The devices in each tier have different latency and capacity characteristics. The storage system we consider consists of three tiers. The top tier is Intel Optane, the middle tier is SSD, and the lowest tier is HDD. For the HDD tier, we assume that the chunk is striped across 8 HDDs, so that while serving a transfer request, it can be read from or written to all 8 drives in parallel.

Migration Controller Module: The migration controller module consists of two components. The first is the proposal component which predicts which chunks are most likely to be accessed in near future. The other is the control component that is responsible for refining suggestions to determine which data chunks should be migrated to which tier. The proposal component generates potential movements based on the estimation of the most likely accessed chunks in the next JIT window. The control component refines the proposed movements to ensure that long-term improvements are not sacrificed for short-term hits on the higher tier. For simplicity, we work with a fixed data chunk size, which is assumed to be 8MB. The optimal chunk size could be environment-dependent, but we do not study that aspect here. Periodically, the migration controller determines whether the proposed transfers should be made, and if so which chunks should be transferred to which tier. It determines this by looking at the environment and the effects that previous migrations had on the environment. During this interaction, the tier proxy module passes several pieces of information extracted from the environment to the migration controller, and the migration controller uses the

information to determine how effective its policy is.

Orchestrator Module: The orchestrator module is designed to receive both the I/O requests and the migration commands from the request generator module and the migration controller module respectively, and forward them to the corresponding devices. While forwarding the requests, the orchestrator follows the priority of the requests where the IOs get higher priority than the transfer commands. Upon receiving a transfer command from the orchestrator, the storage tiers serve the command immediately if and only if there are no pending IO commands directed to the source and the destination tiers of the transfer request. For each transfer, a chunk is first read from the device where it currently resides, then written to the tier where it is transferred to.

Tier Proxy Module: The tier proxy is the module that communicates with the migration controller and issues the transfer commands to the orchestrator. Periodically, the tier proxy asks the migration controller for the transfer recommendations. The migration controller gives a list of the chunks that should be moved to higher tiers, i.e. the chunks that should be moved to Optane and the chunks that should be moved to SSD. The proxy will then decide the chunks that need to be evicted or transferred back to the lower tier. The proxy will first select the candidates for eviction, write them back to the lower tier and finally start the chunk transfer from low to high tier.

In this simulation, we assume that the HDD tier is inclusive i.e. contains a copy of all the data present in Optane and SSD tiers. Hence during evictions from Optane or SSD to the HDD; we only write back the dirty chunks. Whereas a clean chunk is discarded and assumed to be transferred in zero time.

V. Methodology of JIT

A. Challenges in JIT

JIT can potentially outperform traditional tiering since it can account for the current workload characteristics in making tiering decisions. There are a lot of different challenges which need to be addressed in JIT, we discuss three efficiency challenges.

One challenge is selecting the chunks that should be moved to each tier from all the chunks in a storage system. One may argue for an exhaustive method that enumerates the value of all the possible chunk combinations. However, searching all the combinations through the entire chunk space is a combinatorial problem and is not tractable. Additionally, the value of each combination changes over time. Therefore, we introduce a proposal component to provide the list of possible movements, based on the estimated highest usage in the next JIT window with combined chunk size equal to available space in the higher tiers. Although we reduce the search space a lot by the proposal component, deciding which movements from the proposed list also has a large search space when considering all the possible partitions. Since JIT needs to make movement decisions based on the value of selected movements over time, keeping track of the different decisions for each time window makes the search space of the decision sequences difficult to deal with. Thus we use neural networks in the control component to generate estimate values for each action which explores the space more efficiently.

Ideally, if we can learn the phase shifts whereby the hitherto popular chunks are likely to become unpopular (or vice versa), we can account for that in deciding on evictions from and transfers to the higher tier. To address these aspects, we can generate a set of candidate chunks, for migration to higher tiers, based on their expected popularity in the near future. Then from amongst these candidates, we select a subset based on the current load, available space, and other information extracted from the environment. Since some of the preceding is not directly observable for the tiering system, we developed a machine learning model that should implicitly capture the relevant environmental information.

B. Estimating accesses

To get an initial list of chunks to move, JIT needs to have an understanding of what the current workload may look like. This is not an easy challenge because there are factors that JIT cannot measure and also because of the stochastic nature of the problem. One element that would be relevant to predicting workload is the number of data requests to chunks over the next time period. We must predict those values and use those predictions as part of deciding what chunks to move. We do not want to provide an expertly engineered set of features, as these may work for only some workloads and not others. This means that we must learn to estimate the number of accesses from the raw data available to the JIT system.

The raw data, which is available, is the information that is readily acquirable or can be easily kept track of in a storage system. Examples of these include (see section V-D) read accesses to each chunk, chunk location in the tier hierarchy, and time period of access. As part of determining the chunks to migrate, we would like to transform the raw data to some sort of prediction on what will be used in the future and the consequences of the migration. We simplify the problem, by noting that if we only consider one time period, the most important element of predicting the accesses for that time period is the relative order of chunks. So the primary job of the proposal component is to take the raw data and output the relative order. Then based on the relative order and the location of each chunk, the proposal component generates the potential movements which transfer the expected most accessed chunks to the higher tiers if the chunks are located elsewhere currently.

C. Searching through the control space

The control component receives the set of potential movements from the proposal component and identifies which chunks are ready for migration. To do this, the control component needs a method that searches efficiently through the control space by interacting with and observing the environment. Reinforcement Learning (RL) [10] is well suited for the control component since it allows the control component to explore the space of possible migrations and learn from experience without the need to try every possible migration. Through exploration and exploitation, the control component

decides what possible migration it should take across the various situations the environment may be in. Since the goal of the migration controller module is to reduce the average latency of requests over the long term, the control component will learn from experience what portion of the proposed data movements it should actually move. For each migration, the control component receives a reward or gets a penalty. The control component learns which action to take, that is the proportion of data to move, by estimating which action returns the highest reward.

We use Q-Learning [10] to map the set of actions, in any state the environment may be in, to values that represent the expected latency from taking those actions. Since the number of possible states is extremely large, we need to get the values of different regions of the state space efficiently. To make this tractable, we need a way to approximate those values. We use neural networks to perform this function approximation. The merging of reinforcement learning with deep learning is referred to as Deep Reinforcement Learning (DRL) [6]. Specifically, we use DQN [6], which allows us to approximate the value of actions in a given state.

D. State Construction for Migration Controller Module

The purpose of the state is to represent the current condition of the environment. The migration controller module uses the state as a proxy for learning about the workload. Since keeping around the history of all requests, chunk locations, and other details about accesses will be infeasible as time goes on, the state s_t at time t ($t=0,1,2,...$) only captures the observations of the environment at t.

State at time window t	
R_t, W_t, R_t+W_t	read, write, both read & write request accesses
X_t	order of sorted long-term accesses for each chunk
O_t, P_t	chunks located in Optane & SSD tier
Y_t, Z_t	sine & cosine value of the relative time

Table I
STATE COMPONENT

We provide several chunk-level features to characterize the state in a storage system. The requests that the storage system receives are 4-tuple block-level messages. We convert these requests into chunk-level information. Each request contains the LBA, I/O type (read or write), timestamp, and the size of this request. Let n_l be the total number of LBAs in the storage system and let all the LBAs be represented as the set $L = \{L^1,...,L^{n_l}\}$. Let S_l denote the LBA size and S_c the chunk size. We define $|C| = \frac{S_c}{S_l}$, be the number of LBAs per chunk. For each chunk, $C^i = L^{(i-1)*|C|},\cdots,L^{i*|C|}$. The total number of chunks is $n = \left\lceil \frac{n_l}{|C|} \right\rceil$. Table I shows the state components we used. The following paragraphs describe the construction of these components.

R_t, W_t, R_t+W_t: The read and write accesses are not always in lockstep. Therefore, we include both read and write as part of the state representation. Fig. 3 shows the heatmap of read and write accesses collected on a Friday for the usr workload in the MSR dataset. As shown in Fig. 3, the read and write accesses tend to display different patterns across workloads. We divide the trace into JIT windows of size τ. To calculate

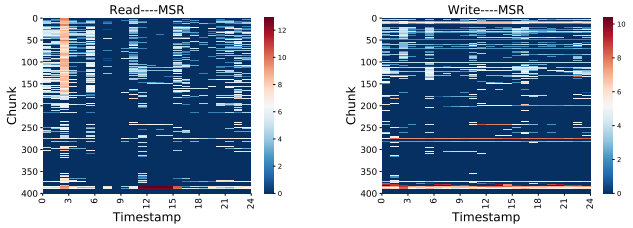


Fig. 3. Heatmap of read and write accesses of Friday for MSR workload

the access frequency of each chunk, during a JIT window, we aggregate all the accesses to LBAs within that chunk. Let $R_t, W_t \in \mathbb{R}^n$ denote the total number of read and write requests at time window t. The vector $[R_t, W_t, R_t+W_t]$ is then used to build part of the state.

X_t: Besides the short-term chunk accesses, we also capture long-term chunk accesses to build the state. Long-term accesses to the chunks are collected based on the chunks' LBA accesses over the previous 24 hours. The migration controller module does not need the exact chunk frequency for migration decision, instead, it only requires the relative frequency order of the chunk accesses. We construct a vector X_t of length n to represent the relative frequency of each chunk in time step t. For $i \in \{1,..,n\}$, x_t^i is the order of chunk C^i in the sorted list.

O_t, P_t: We can use the environment to capture the tier that each chunk resides on. As stated above, that HDD contains the set of all chunks. The set of chunks in Optane and SSD are disjoint. Hence we only store the chunk to tier mapping for the Optane and SSD tiers in two binary vectors of the length n, denoted as O_t and P_t respectively. If chunk C^i is in Optane, $o_t^i = 1$, otherwise, $o_t^i = 0$. P_t has the same structure. For a given chunk C^i, o_t^i and p_t^i cannot both be equal to 1.

Y_t, Z_t: The time that the request is made is also useful to capture since the workload may follow a pattern based on frequency. For example, the peak of the requests coming to the storage may be at the same time of the day (e.g., 10 am). We apply sine and cosine functions to the relative time and get two numbers back. Then we build one vector Y_t of length n where the element's value is set to $\sin\left(\frac{t}{T_d} \cdot 2\pi\right)$ and another vector Z_t of the same length n where the element's value is set to $\cos\left(\frac{t}{T_d} \cdot 2\pi\right)$, T_d is the total time we have in a day.

Finally, for each time step t, the state is constructed to be an array of shape $n \times 8$ using the above information, $s_t = [O_t, P_t, R_t, W_t, R_t+W_t, X_t, Y_t, Z_t]$.

E. Formalization of Migration Controller Module

As stated above, our chunk movement decision starts with the proposal component generating a list of possible movements to higher tiers, henceforth denoted as M. First, the proposal component takes in the states s_t and outputs a list D_t that contains the relative order of the estimated chunk accesses in time t. Then we form the possible movements based on the corresponding values of O_t and P_t which provides the chunk location information. Let N_O and N_P denote the capacity of Optane and SSD respectively. If the element d_t^i is the top N_O values of D_t and $o_t^i = 0$, we add a movement to M_t where the

destination tier is Optane. The chunk number of the movement is C_i, and the source tier is SSD ($p_t^i = 1$) or HDD ($p_t^i = 0$). The movement to SSD is formed in the same manner.

Let $A = \{A_1, ..., A_k\}$ denote the set of actions where k is the total number of actions we have. Each item in A is a subset of M. Assume the control component decides to take action $a \in A$ and move the suggested data chunks. We use the total access latency of the requests occurring during time interval t to represent the reward r_t. The higher the latency, the lower the reward; this is achieved by simply multiplying the latency value by -1. The access latency of the requests at time t can be influenced by prior data migrations. Also, the data movements of time step t may influence the access latency of future requests. Therefore, the short-term penalty is sometimes preferred as it may lead to long-term rewards.

Since the purpose of data migration is to decrease the average latency for all the requests, the goal of the control component is to achieve the maximum future reward. In RL, the reward in a future time is discounted by a factor γ to favor rewards of the same value that come earlier and avoid infinite rewards, where $0 \leq \gamma < 1$. Therefore the value that represents the future reward is given by $R(t) = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t} r_T$, where t is the current time step and T is the termination time step, which is the end of the day in our case.

Since there are multiple data movement actions to choose from at each state and the different data movement actions lead to different next states, we get a recursive situation where the action that should be chosen depends on the states that we end up in and what actions to choose in that new state. We get the Q-Learning setting, which defines $Q(s_t, a_t)$ to be the value of the control component taking action a_t in the state s_t. Q^* is used to represent the best-expected value. That is,

$$Q^*(s_t, a_t) = \max_{\pi} Q^{\pi}(s_t, a_t)$$

and π is the policy for taking actions under given states. This makes $Q^*(s_t, a_t)$ the optimal state-action value function over all policies. We can represent the optimal state-action value function as

$$Q^*(s_t, a_t) = \mathbb{E}\left[r + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \mid s_t, a_t \right]$$

where s_{t+1} is the state of next time step and a_{t+1} is the possible action of next time step. In our work, we use neural networks as a non-linear function approximator to estimate the optimal state-action value $Q^*(s_t, a_t)$.

VI. Implementation of JIT

We build the JIT migration controller module using a multi-headed network architecture. Fig. 4 shows the overall scheme of the module. We input the states into both the proposal component which consists of *proposal block 1* and *proposal block 2*, and the control component which consists of *control block 1* and *control block 2*. To help our migration module generalize better, we augmented the data. We randomly selected a certain percentage of the data chunks in the storage system, and swap the request accesses of the selected chunks with a random neighbouring chunk within a certain range. The percentage and the neighbouring range are both small numbers since the storage accesses patterns of the neighbouring chunks can be similar while this rarely happens to the chunks that are located far away from each other.

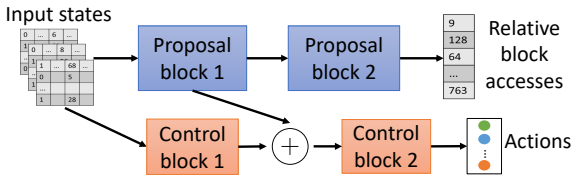


Fig. 4. The overall network architecture of migration controller module.

A. Building Proposal Component

One head of the module is the proposal component which we use to generate data movement proposals for some data chunks based on the expected frequency of those chunks in the next JIT window. The idea is by moving data chunks with expected maximum frequency to higher tiers, JIT minimizes the average latency of future requests. To do this, we first input the state to get a compact representation through *proposal block 1* and then carry out the opposite process through *proposal block 2* to output a list of expected accesses for all the chunks. During training, the mean squared error loss of the output is taken with respect scaled access frequency of the next JIT window. Based on the relative order of the expected accesses for each chunk, we obtain the data movement proposals.

We use Convolutional Neural Network (CNN) layers to build the proposal component. The reason for this is that there is a locality that CNN can take advantage of. We note The input of the proposal component is the state array. A part of the state array is the access frequency number and the relative order for all the chunks. It is possible that the access frequencies of the nearby chunks are related. In addition, the order of the chunks in a sorted frequency list is clearly correlated with the number of accesses it gets. It is also likely that the number of accesses may be correlated with the access timestamp because the workloads may follow a daily pattern. In short, the values of local neighboring elements in the input state are correlated. CNN introduces an architectural bias that can take advantage of these local correlations which often leads to requiring less training time and data.

B. Building Control Component

The other head of the multi-headed network is the control component which we use to filter the data movement proposals. We define the actions as several different possible sets of data movements based on the data movement proposals we obtain from the proposal component. We input the state to *control block 1* and obtain an intermediate representation. This representation is passed to *control block 2* together with the output from *proposal block 1* which is the intermediate representation of the expected access frequency of each chunk. Finally, we output the estimated value of each action with the input state and select the data movements associated with the maximum value. We use an adaptation of the Deep Q-learning (DQN) [6] called Double DQN [3] to train our control component. The Double DQN combines Double Q-learning [2] algorithm with deep RL. By using two Q-learning networks,

one for action selection and one for value function estimation, the method stabilizes the training process and lessens the problem of overestimation in the original DQN. In order to make the valid estimation of Q, we need to be able to explore a large number of states. As we get some estimations for the states, we want to discover if those estimations are good. The transition between these two facets is performed through what is known as exploration vs. exploitation dilemma [10]. In practice, we use the ϵ-greedy method to approach this dilemma, where at the beginning of the training, we focus on exploration and slowly adjust to more exploitation as we go through the training. Thus allowing the control component to find a desirable path through the state space.

The control component observes and reacts to different states of the storage system and collects the data s_t,a_t,r_t,s_{t+1} for each time step. The training data itself is a time-dependent sequence since the action a_t taken at time t influences the state and the action for the next time step $t+1$. This time correlation breaks the assumption of independent and identically distributed (iid) data and clearly the trajectory through a day of the trace is not iid. We apply a method called experience replay [6] in our training process to decorrelate the time dependency. The basic idea of experience replay is to store the data we obtain from exploring the storage system into the replay memory and use the random sampling of the stored data to update the parameters of the deep neural network, such as updating the Q-learning parameters. In addition, when we update the weights of the network (in Fig. 4), the updates from the actions output loss to the weights of the *proposal block 1* is disabled. This is because the data movement choices should not influence the number of accesses each chunk gets in the next time step. That is only the weights of the *control block 2* and *control block 1* in the multi-headed network are updated.

VII. Experimental Setup and Results

In this section, we evaluate our tiering method using two publicly available traces and analyze the results.

A. Datasets Used for Evaluation

We have used two different datasets for our experiments. The first one is the one-week long block I/O trace of enterprise servers at Microsoft Research Cambridge (MSR) [7] consisting of three different workloads: User home directories (*usr*), Project directories (*proj*), and Hardware monitoring (*hm*). Originally an IO request in an MSR trace file contains 7 tuples. But attributes that are useful for our purposes in an IO request are - timestamp, request offset, request type (read or write), and request size.

The second group of datasets is the ten-day long block I/O traces from a Tencent production Cloud Block Storage (CBS) system [14]. The CBS trace records the I/O requests issued by the clients from a forwarding proxy server of the client and storage server. The trace captures the timestamp, offset, size, type (read or write), and cloud disk id of each request. This dataset is collected from thousands of cloud virtual volumes (virtual disks). We evaluated several dozen of these traces, and filtered out the traces that showed little variation in the requested blocks and those that show little locality in the requests. Of those, we narrowed it down to two virtual disk ids and used the corresponding traces over the weekdays of the first week. We call these two workloads TC1 and TC2.

B. Calibration of Storage Tier Model

The simulation model maintains the queues of user requests (512 Bytes per LBA) and chunk transfers, though the storage model only needs to keep track of the basic "service time" of individual requests at each tier. In general, such service time depends on numerous details of the system, but we found that it suffices to calibrate it simply by using an $a+bx$ model where a is the minimum latency for a request, x is the size of the transfer in LBAs, and b is the size-dependent part. For HDD, a is dominated by the seek and rotational latency, but for SSD it includes FTL (flash translation layer) delays, NVMe protocol latency, latency of internal buses, etc. The Optane delays would be even more dominated by protocol delays.

Since we do not have the characteristics or measurements from the actual setups used for generating the MSR or CBS traces, we calibrated the model by considering the characteristics of currently available devices. Obviously, the calibration needs to be done for each system and workload. For our calibration, the a parameter was $4ms$ for HDD and $60\mu s$ for SSD. For HDD, we have $b = 2\frac{\mu s}{LBA}$ for both reads and writes. For SSD, $b = 0.5\frac{\mu s}{LBA}$ for reads. The writes are considered as 2X slower effectively. (The actual flash writes are much slower than that but the write buffer in SSD reduces their effective latency.) For Optane we use $a = 0.2\mu s$, $b = 0.26\frac{\mu s}{LBA}$ for both reads and write.

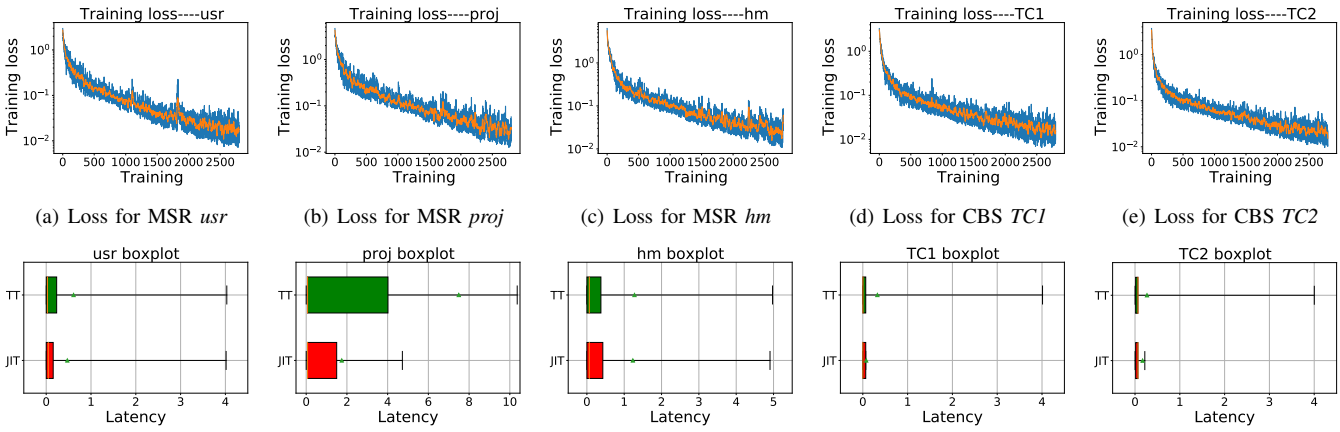
C. Experimental Setup

Since each trace has its own quirks, we needed to perform some simple manipulation on the trace files to have them all in a uniform format. The timestamp of the CBS traces is recorded with a crude 1-sec resolution, which often concentrates too many requests in 1 sec. Therefore, we added a random time to each request in [0..1] sec range to disperse them, but maintained their original order from the trace (since the order is often quite important for storage requests). We randomly selected Thursday, which is in the middle of the week to assess our method.

We used Adam optimizer [5] to update weights iteratively while training the migration controller module using a batch size of 32. We used ϵ-greedy to train the control component to take actions but decreased ϵ in proportion to the percentage of training finished. We defined three actions for the control component. Each action represents a different amount of data movement selected from the proposal.

D. Experimental Results

The traditional tiering (TT) method collects the frequency of each chunk from the previous day and uses the LFU algorithm to perform data migration at midnight. TT places the most accessed chunks in the higher tiers. We evaluate



(a) Loss for MSR *usr* (b) Loss for MSR *proj* (c) Loss for MSR *hm* (d) Loss for CBS *TC1* (e) Loss for CBS *TC2*

(f) Box plot for MSR *usr* (g) Box plot for MSR *proj* (h) Box plot for MSR *hm* (i) Box plot for CBS *TC1* (j) Box plot for CBS *TC2*

Fig. 5. Training loss (log-scale) and Box/Whisker plots for various MSR and CBS traces

the performance based on the average latency of the test day. The latency of each I/O request is calculated after the request has been received from the storage device and considered completed. We use the timestamp when the request is issued by the request generator and when the reply comes back to the request generator to calculate the latency.

Table II shows a comparison between the traditional tiering and JIT methods. It is clear that the JIT reduces the average latency for every workload as compared to TT. The amount of reduction is dependent on the value of learning the short-term access pattern as opposed to the overall long-term popularity. For most cases, the reduction is quite significant and even huge in some cases. For example, the average latency for *proj* is decreased by 76%. However, the *hm* workload appears to be quite stable and therefore is unlikely to have much difference between the long-term popularity (exploited by TT) and short-term popularity (exploited by JIT).

Mechanism	**Workload**				
	usr	**proj**	**hm**	**TC1**	**TC2**
TT	0.611ms	7.493ms	1.275ms	0.325ms	0.267ms
JIT	0.472ms	1.746ms	1.231ms	0.076ms	0.169ms

Table II
AVERAGE LATENCY OF TT AND JIT FOR MSR & CBS TRACES

In Fig. 5 we show two things for each of the 5 workloads (namely MSR usr, proj, hm and CBS TC1, TC2): (a) The loss for training, and (b) the resulting latency distribution for tiering on the test data.

The loss functions show that there is no divergence issue even though augmented the data as described in the previous section. We expect the results to be similar or better when using more training data. The latency distribution is shown via box/whisker plots where the filled boxes show the boundaries of 25% and 75% latency values, and the rightmost tick gives the 95% value over all the access latencies. In the *proj* workload, we also see a significant decrease in the tail latency, which has important ramifications for user experience. As stated above, for some workloads, such as for *hm*, tiering on shorter time scales does not help because the workload is quite stable. Similar characteristics are observed for the CBS traffic as well. Overall, we expect JIT to work better for workloads with higher variability, since for those workloads a simple long-term frequency based tiering does not suffice.

VIII. CONCLUSIONS

In this paper, we present the just-in-time intelligent tiering mechanism and evaluate its effectiveness using several workloads from two publicly available datasets. We introduced a simulator that is able to capture the essential characteristics of multi-tier storage systems; from user access to migration movement. We are able to instrument, test, and record statistics about different migration policies. JIT makes migration decisions by first providing movement candidates of chunks that will be accessed soon. Then JIT chooses a subset to migrate from these candidates based on their long-term benefits. The primary goal of JIT is to reduce the overall access latency, and we show that in most cases, it is able to reduce the latency significantly.

REFERENCES

[1] Cano, I., et al.: Curator: Self-managing storage for enterprise clusters. In: 14th USENIX NSDI (2017)
[2] Hasselt, H.: Double q-learning. In: Lafferty, J., et al. (eds.) Advances in Neural Information Processing Systems (2010)
[3] van Hasselt, H., et al.: Deep reinforcement learning with double q-learning (2015)
[4] Herodotou, H., Kakoulli, E.: Automating distributed tiered storage management in cluster computing. arXiv preprint arXiv:1907.02394 (2019)
[5] Kingma, D.P., et al.: Adam: A method for stochastic optimization (2017)
[6] Mnih, V., et al.: Playing atari with deep reinforcement learning (2013)
[7] Narayanan, D., et al.: Write off-loading: Practical power management for enterprise storage. ACM Transactions on Storage (TOS) (2008)
[8] Noel, R.R., et al.: Towards self-managing cloud storage with reinforcement learning. In: IEEE IC2E (2019)
[9] SimPy, T.: Documentation for SimPy, `https://simpy.readthedocs.io`
[10] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. The MIT Press, second edn. (2018)
[11] Tsai, C.H., et al.: Value-based tiering management on heterogeneous block-level storage system. In: 4th IEEE Intl. conf on Cloud Computing Technology & Science. pp. 393–398. IEEE (2012)
[12] Vengerov, D.: A reinforcement learning framework for online data migration in hierarchical storage systems. The Journal of Supercomputing **43**(1), 1–19 (2008)
[13] Yang, Z., et al.: Autotiering: automatic data placement manager in multi-tier all-flash datacenter. In: IEEE IPCCC (2017)
[14] Zhang, Y., et al.: Osca: An online-model based cache allocation scheme in cloud block storage systems. In: USENIX ATC (2020)
[15] Zheng, S., et al.: Ziggurat: A tiered file system for non-volatile main memories and disks. In: 17th USENIX FAST (2019)