Intelligent storage system with machine learning

Amy Yen
Department of Electrical Engineering
Stanford University
Stanford, U.S.A.
htyen@stanford.edu

I. INTRODUCTION

Storage system is the key component in big data. As data size grows, we need to scale storage system efficiently. Majority of today's storage devices present a simple linear block address space to the operating system. This simplistic addressing scheme treats each block address as a generic bucket of opaque data. While this architecture allows the operating system to easily abstract the stored data for upper layer application to use, it offers the devices no insight into the content of the data nor the relationship among data at various addresses. Enabling the devices to intelligently assign data storage location and plan data movement across devices in different tiers of the entire storage system can optimize the system's performance and cost.

This paper explores using machine learning (ML) to construct a model to design a storage system to achieve minimum access latency at lowest cost. The ML model will coordinate interactions across different devices in the system and within each device, it will direct the location to store the data. The input to our ML model is IO trace file. The trace file lists all IO activities to a storage device. With each IO record a training example, our data set consists of all IO records from the input trace file. They are fed into ML-clustering algorithm with access time and offset as features. Outputs are clusters of IO records indicating which accesses were accessed with temporal and spatial locality.

II. RELATED WORK

A. *Caching and prefetching*

A storage system consists of tiers of different devices with varying cost and performance. Top tier is the smallest capacity, fastest, and most expensive. Fully utilizing the top tier is the key to minimize latency while keeping cost down. Kroeger *et al.* proposed caching frequently accessed data and prefetch data to be accessed in near future into the top tier [1].

B. *Data sementic aware devices*

Physical devices that understand the specific applications issuing IO requests to them can organize data to minimize number of accesses and each access' latency. Sivathaunu *et al.* proposed database aware storage [2]. The storage system snooped write-head log of the data base system to accurately infer the evolving access pattern. It also gathered statics such as access time of queries, correlation between table/indexes, and number of queries on tables over a duration of time to devise caching scheme.

Arpacii-Dusseau *et al.* applied similar idea to construct a file system aware disk, which utilized similar statistics as proposed by Sivathaunu *et al.* and directory/inode structure specific to file system to infer applications' view of data blocks [3].

More transparency between upper level application and physical devices is the most direct method to construct an intelligent storage system, but it incurs significant costs: complication in design and extra hardware overhead.

C. *Device characteristics aware applications*

This approaches tackles the opacity issue between application and device from the opposite direction as described in section B. Upper level applications are developed with assumptions on specific hardware's behaviors. Schindler *et al.* proposed embedding knowledge of disk's physical geometry information in applications' algorithm to align related accesses within a disk track to minimize access latency [4]. This works well until the hardware design obsoletes.

D. *Machine learning approach*

We can apply machine learning to allow storage system to infer data semantic without the transparency between higher level application and physical devices as described in section B. Wildani *et al.* applied k-means on IO accesses history to identify access upper level applications' working sets [5]. Our proposal also uses a clustering algorithm to learn applications' access pattern, but incorporates an additional inferred feature. We also explored multiple runs of clustering algorithm on the same data set to learn time domain and spatial domain separately.

III. DATASET AND FEATURES

Our dataset consists of ~2 million IO records collected by Microsoft on their MSN Storage file server over a duration of 6 hours [6]. Each IO record describes an IO access by its time issued, type (read or write), location addressed (offset), data size, and retrieval latency. The raw trace files include accesses to multiple disks. The time, offset, and size are in units of microseconds, bytes respectively. Preprocessing separates IOs from different disks to different datasets and converts the features to storage standard format/units. Offset and size are transformed from bytes into number of blocks. A block is the smallest addressable unit for any block device, such as SSD and disk. Industry standard block size is 512 bytes. Access time given in microseconds is converted to milliseconds to prevent overflow of representation in signed 32-bit data types. This is acceptable because a typical IO access is on the order of milliseconds [8]. Figure 1 plots the read accesses and figure 2 plots the write accesses captured during the first 10 minutes of disk0 IO trace.

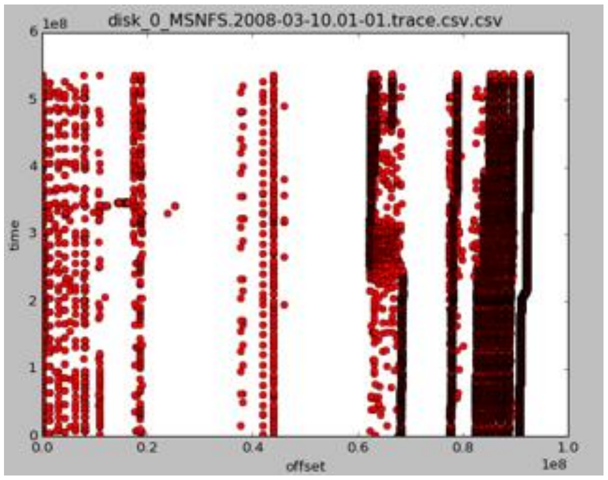


Figure 1: write accesses of first 10 minutes of disk0

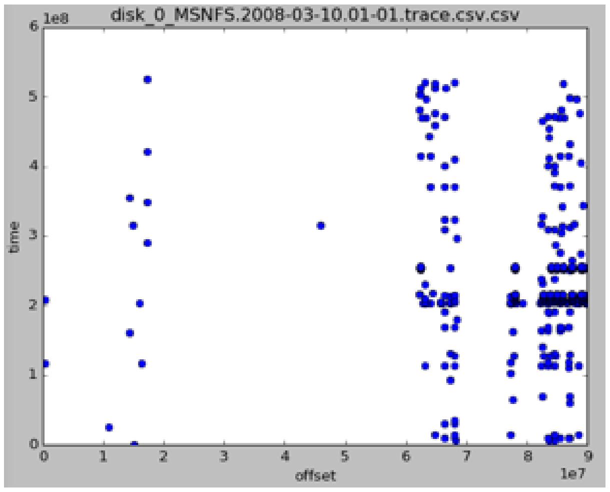


Figure 2: read accesses of first 10 minutes of disk0

Based on the result of B. Wildani *et al.*, we selected access time and offset as features for our first pass clustering run. This pass of clustering is made to bias towards time domain (y-axis direction) grouping by dividing offset with a factor α. Figure 3 shows an example clustering output. Data points in the same color are in one cluster. The stars indicate the cluster centroids. The output of this first pass clustering is showing what the ML algorithm learned as an "epoch" duration. An epoch defines the unit of time during which temporally related accesses took place. The duration of an epoch evolves as time progresses and applications' access pattern changes.

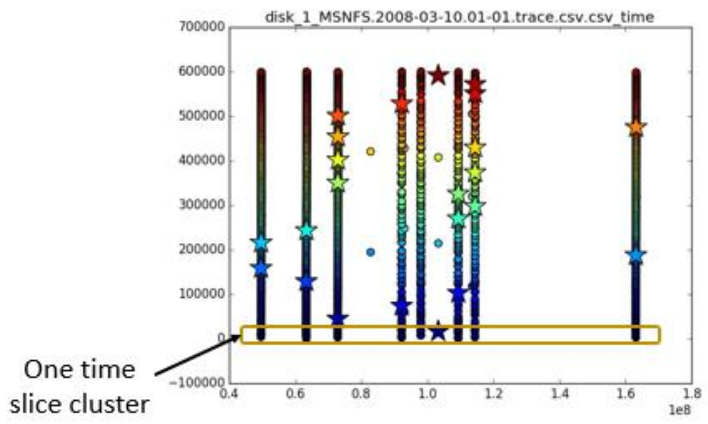


Figure 3: first pass time centric clustering result of first 10-minute trace from disk1

Once we divided the time domain into slices of epochs, we iterate through all the epochs and run clustering on each of them. The features used in second pass clustering are offset and an inferred feature called extent. An extent is a series of related consecutive blocks on the storage device [7]. An extent is usually accessed by multiple IO accesses that have very tight temporal locality. For example, as shown in figure 4, we have 2 IO accesses; both with size 10. First access happened at time t to offset 0. Second access time was at t + x to offset 10. If the difference in access time is much less than typical disk access latency (15 milliseconds) [8], they are highly likely to belong in the same extent.

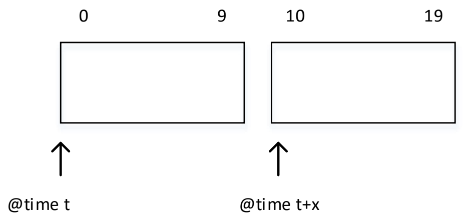
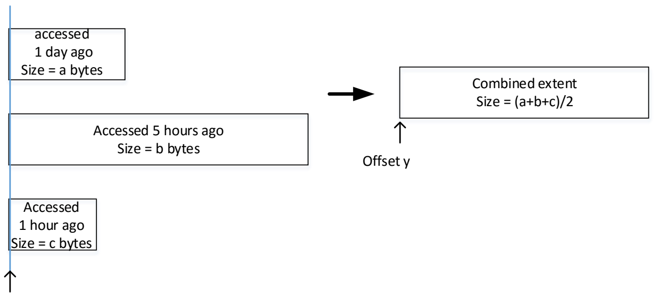


Figure 4: extent example

We identified the possible extents by building a list of closely timed sequential IO accesses from our traces. Then we combine the identified extents with the same offset by taking an average of their sizes using time as weights (demonstrated below).


accessed 1 day ago Size = a bytes
Accessed 5 hours ago Size = b bytes
Accessed 1 hour ago Size = c bytes
Combined extent Size = (a+b+c)/2
Offset y

We used the identified extents with their offsets and size as features for support vector regression (SVR) and locally weighted linear regression in attempt to learn the relationship between offset and extent size. However, as shown in figure 5 and 6, the association between offset and extent size turns out to be decidedly non-linear and not easily distinguished. Both results had unacceptably large test error. We think this could be explained by the lack of intrinsic relationship between offset and extent size, as a file systems have no reason to group similarly sized extents together by offsets. Thus, we directly used our inferred extent map as a lookup table instead of trying to learned this feature through ML.

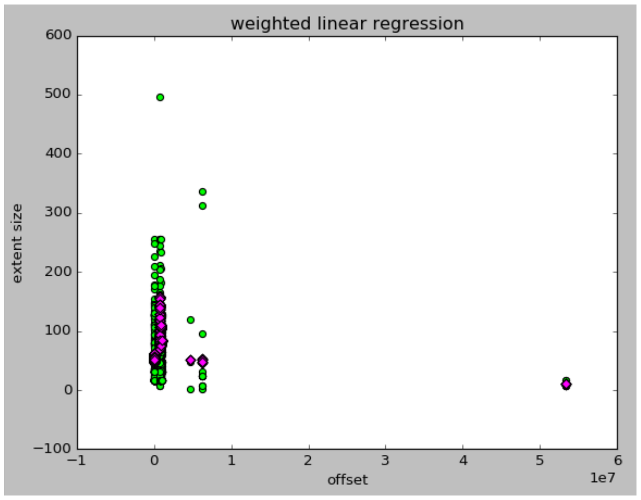


Figure 5: locally weighted linear regression result

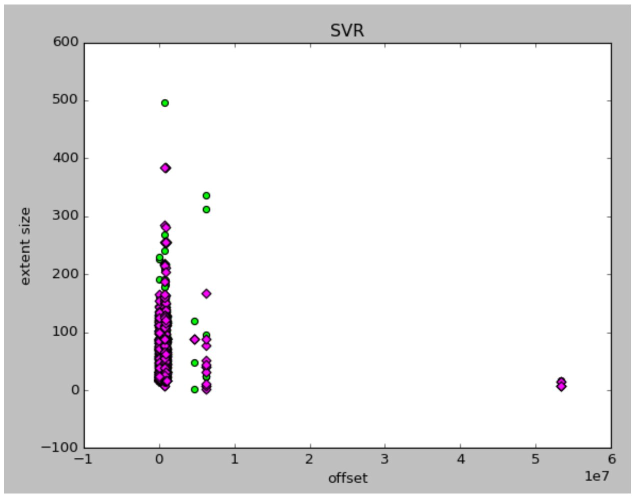


Figure 6: SVR result

The goal of the second pass is to identify spatial locality. Figure 7 shows the result of second pass clustering applied on figure 3.

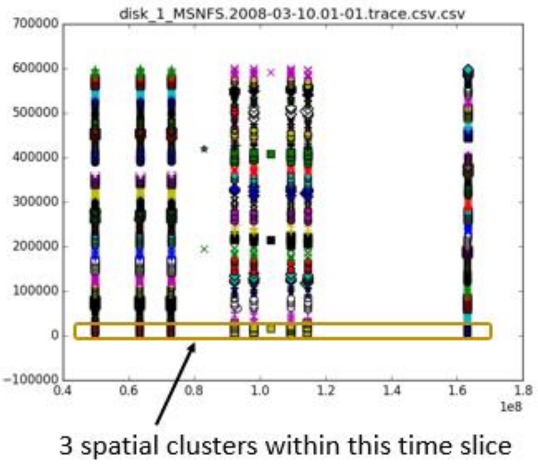


Figure 7: second pass clustring applied on top of figure 3

As shown above, each time slice is further divided into groups of spatially related accesses. The result is the model we used to predict the working set of an incoming access. The training for model creation and application of the model process is overlapped, similar to online machine learning. For example, we train for the first minute to construct a model. Apply this model in second minute, at the same time training this second minute to prepare a evolved model for third minute to use.

IV. Methods

Before we finalized on the 2-pass clustering ML approach, we explored using reinforcement learning to learn and manage the working sets. In MDP [9] context, we would define the state space to be all possible configurations of the address space. Each block in the address space can be in one of 3 conditions: not accessed yet, in upper tier, or in lower tier. So, if there are N addressable blocks, there will be 3^N states. The huge number of states is the major deterrent of MDP approach. While there might be more advanced MDP algorithm such as factored MDP proposed by Osband *et al.*[10] to resolve the large state space issue, formulating our state transition probabilities is another daunting task. We would need a huge number of traces to have sufficient number of accesses to hit all possible states frequently enough to construct the complete state transition probability.

We used mean shift clustering algorithm to decompose the training data into working sets along temporal and spatial dimensions. Mean shift is an iterative, non-parametric mode-seeking algorithm. It was first presented in 1975 by Fukunaga and Hostetler [11]. Intuitively, the algorithm iteratively 'shifts' each data point towards the direction with the highest density of neighboring data points. Thus, with each iteration, each data point will 'shift' closer and closer together towards a local cluster centroid that is marked by the highest data point density. All points associated with the same centroid belong in the same cluster.

In more detail, mean shift uses kernel density estimation (KDE) [12] to estimate the underlying probability density function (PDF) of the data set. If (x_1, x_2… x_n) are n independent and identically distributed data points sampled from an unknown distribution D in $\mathbb{R}^d$, KDE allows us to estimate the PDF of D using the kernel density estimator:

$$f(x) = \frac{1}{nh^d}\sum_{i=1}^{n} K(\frac{x-x_i}{h})$$

where K(·) is a kernel function and h is the bandwidth parameter which we will discussion more below. The values of x which satisfies $\nabla f(x) = 0$ are then the modes of the density function, towards which we want to 'shift' the neighboring data points. This results in the update rule:

$$\frac{\sum_{i=1}^{n} K(\frac{x-x_i}{h})x_i}{\sum_{i=1}^{n} K(\frac{x-x_i}{h})} \rightarrow x$$

which mean shift applies repeatedly to each data point until convergence. The bandwidth parameter is an important value to consider as it determines the resultant density function. If it is too small, we will have a bumpy density estimator which can result in too many little clusters. On the other hand, if the bandwidth is overly large, we will have a very smooth density estimator resulting in all points belonging in one giant cluster.

V. Experiments and Results

We measured the effectiveness of our ML approach by simulating a two-tier storage system. The upper tier has limited capacity (e.g. cache or main memory) but better performance. The lower tier has unlimited capacity (e.g. disk) but slow access time. In the rest of the paper, we will use cache to refer to the upper tier. The goal is to have as much useful data residing in cache during the time of access without incurring large latency to retrieve it from lower tier. We quantify this by hit rate.

$$\text{hit rate} = \frac{\text{number of blocks hit in upper tier}}{\text{number of blocks hit in upper tier} + \text{number of blocks need to be retrieved from lower tier}}$$

The simulator implementing ML takes the dataset (IO traces) as input. Iterate through time ordered IO accesses in group of δ seconds. Run our clustering algorithm on the examples within each δ-second time interval group for 2 passes as described in section III to obtain a model of the most recent view of temporally and spatially related accesses. We will follow B. Wildani *et al.*'s convention and called these related accesses to be in one working set. These working sets are represented by an undirected graph in our simulator, with nodes representing offsets. Edges connect two nodes when the offsets they represent belong in the same working set. Each edge has an initial weight of ω, it is incremented by 1 whenever the two end node offsets appears in the same working set again, and are aged periodically by decrementing the weight by 1 at rate ɣ. When the weight reaches zero, the edge and the any orphaned nodes are removed from the graph. This graph is updated after each δ-second interval training. As soon as an updated working set model is ready, the simulator applies it on new incoming accesses. If an incoming access is a part of a working set and not already in upper tier, the simulator will attempt to bring in the entire working set. If the upper tier does not have enough capacity, simulator will evict sufficient data in first-in first-out order to make room for the incoming working set. This is the testing phase of our ML approach. While the testing is in progress in one δ-second interval, we simultaneously train on this same interval to update our model for the next δ-second interval.

To compare the performance of our ML based two-tier storage system, we constructed a second simulator implementing the same system with least recently used (LRU) cache as upper tier. LRU-based simulator caches just the incoming access if it is a miss, and evict enough least recently accessed data to make room for this new access.

The simulators have several common parameters: input trace density (number of accesses within a fixed time period), input trace duration (time period over which all accesses in our dataset span), and upper tier capacity (how many blocks of data the upper tier storage device can hold). Specific to ML-based simulator are the parameters of working set aging rate ɣ, time interval δ of each training/testing phase, and bandwidth parameter for mean shift clustering as described in section IV.

The aging rate ɣ and training time interval δ are selected through cross validation on our dataset. Big δ means coarser grain level of training. If δ is too big, our dataset might include data points which are no longer temporally related. However, δ cannot be too small either because we might end up with too little data points to construct complete working sets. ɣ has to be large enough to retain enough useful working sets but not too large such that we bring in too much data and overflow the cache constantly.

After working set aging rate and training interval are decided, we naturally selected the longest duration trace our compute environment can handle. It is a 6 hour run from a Microsoft production server accessing 6 disks. We selected disk1's trace in combination with a 10 megabyte cache size after experimenting with several different combinations of disk trace and cache size. The traces show that different disks can have very different IO access density. Given our limited amount of data point and compute power, we must select a combination that can yield meaningful result. For example, figure 8 shows that the hit rate using disk4 trace is too low in both LRU and ML simulators using the same size cache as disk1 trace because the IO density in disk4 is much greater than disk1. Since, we cannot enlarge the cache size to accommodate this larger density due to compute limitation, disk1 trace was selected as our dataset.

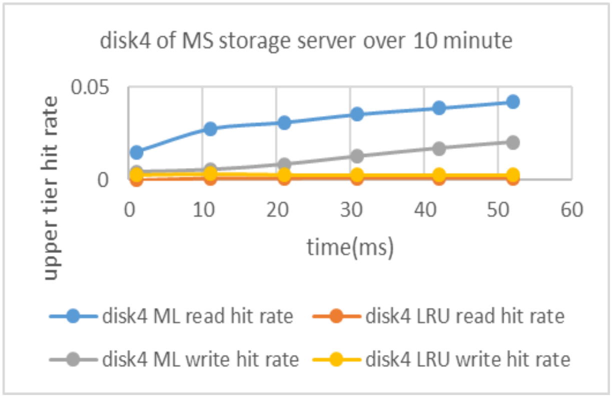


Figure 8: MSNFS.2008-03-10.01-01.trace.csv (IO density)

Once disk1 trace is selected, we then choose an appropriate cache size, the minimal size able to fit ML's working sets.

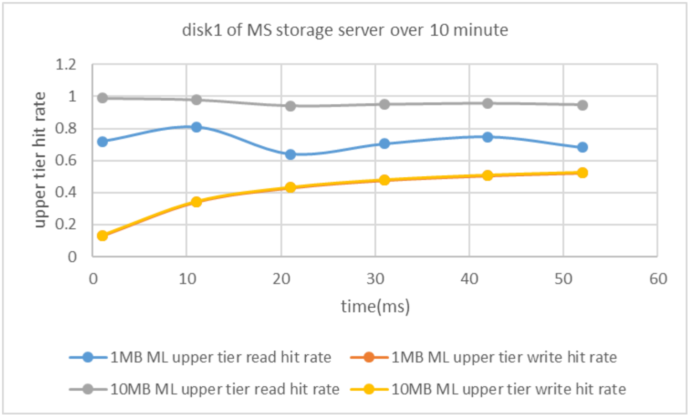


Figure 9: MSNFS.2008-03-10.01-01.trace.csv (cache size)

Figure 9 shows that 1 megabyte seems too small for our ML model to operate on the IO density of disk1 data. Even though it allows faster simlation time, 10 megabyte was selected instead.

Figure 10 shows the simulation result of the 6-hour trace with the selected parameters described before. The result shows our ML-based two tier storage system has performance advantage over the LRU-based version.

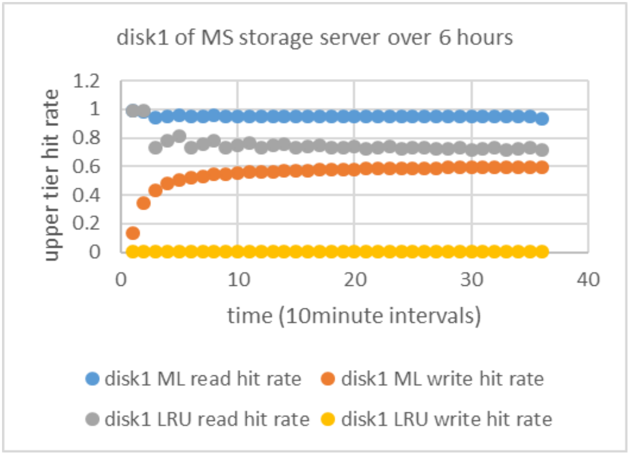


Figure 10: MSNFS.2008-03-10.01-01 ~ 10.12-51.trace.csv

Our ML simulator has an average read hit rate of 95.1%, while LRU's average read hit is 75.2%. This means our ML approach has 26.4% higher read hit rate. On the write side, ML's performance (54.9% write hit rate) is significantly better than LRU's (0.178%). The overall advantage of ML over LRU demonstrates our working set model is effective. Refering back to figure 3, we can see the amount of data been accessed in a temporal slice is more than the upper tier's capacity. In addition, the accesses within a time slice are in multiple spatial locality clusters. The caches will not be able to hold all the working sets. Thus, some of the older accesses are evicted before we advance to the next time slice with similar access pattern. In LRU, each "revisit" of an evicted offset results in a miss. With ML, the entire working set of the evicted offset are brought in together. Any subsequent accesses to the offsets within this same working set will result in hits. The drastic advantage of ML over LRU on the write access front is surprising. More investigation is needed to confirm if it is a phenomenon specific to the Miscrosoft production server trace or there is a more fundamental reason behind it.

VI. CONCLUSION AND FUTURE WORK

Our work shows that we can distinguish effective working sets using spatiotemporal features and ML clustering algorithms such as mean shift. Our experimental data further shows that a working set oriented caching policy out performs traditional LRU approach in both read and write cache hits. We believe this is the result of file system design which keeps related data close spatially and our extraction and usage of extent information.

We used mean shift clustering instead of k-means clustering for the following reasons. First, we believe our data pattern of sporadic timed but spatial close IOs are better modeled by a mode-seeking algorithm. Second, we want the algorithm to find natural clustering of working sets through time instead of always using a rigid parameter number. Finally, we don't want to constraint our cluster shapes to be spherical or elliptical. However, this comes at a cost of greater computing complexity.

While we were successful with applying ML-clustering techniques. We also run into limit on what our ML tools can be used to for. The lack of intrinsic relationship prevented us from using SVR/locally weighted linear regression to predict extent size from an access offset.

Our ML-based simulation has very simple eviction policy chosen for the ease of simulator's implementation. As an extension, we can design a more sophisticated eviction policy to improve performance. Other future work includes extending the two tier into multi-tier storage system to more accurately reflect modern data center architectures. We can also experiment with longer traces as well as traces from other types of systems such as Linux servers or a single user PC. We might find single user trace yielding better result than what we have shown because its access pattern maybe more "predictable" than multi-tenant systems. Different types of traces will also help with the investigation of why we observed drastic better write hit rate of ML over LRU approach as shown in section V.

VII. References

[1] T.M. Kroeger and D.D.E. Long. "Predicting file system actions from prior events," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, 1996.

[2] M. Sivathanu, L. Bairavasundaram, A. Dusseau, and R. Dusseau. "Database-Aware Semantically-Smart Storage," in *USENIX Symposium on File and Storage*, 2005.

[3] A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, L.N. Bairavasundaram, T.E. Denehy, F.I. Popovici, V. Prabhakaran, and M. Sivathanu. "Semantically-Smart Disk Systems," in *Second USENIX Symposium on File and Storage (FAST'03)*, 2003.

[4] J. Schindler, J.L. Griffin, C.R. Lumb, and G.R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. In Conference on File and Storage Technologies, 2002.

[5] A. Wildani, E. Miller, and L. Ward. "Efficiently identifying working sets in block I/O streams," in *SYSTOR '11 Proceedings of the 4th Annual International Conference on Systems and Storage*, 2011.

[6] "Microsoft production Server Traces," [Online]. Available: http://iotta.snia.org/tracetypes/3.

[7] Wikipedia, "Extent (file systems)," [Online]. Available: https://en.wikipedia.org/wiki/Extent_(file_systems).

[8] Wikipedia, "," [Online]. Available: https://en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics#Seek_time

[9] Andrew Ng, 'Reinforcement Learning and Control', Stanford University, 2016.

[10] Ian Osband and Benjamin Van Roy. "Near-optimal Reinforcement Learning in Factored MDPs," in Advances in Neural Information Processing Systems 27 (NIPS), 2014.

[11] K. Fukunaga and L. Hostetler. "The estimation of the gradient of a density function, with applications in pattern recognition," in IEEE Transactions on Information Theory (Volume: 21, Issue: 1), Jan 1975.

[12] Emanuel Parzen. "On Estimation of a Probability Density Function and Mode," in The Annals of Mathematical Statistics Vol. 33, No. 3, Sep., 1962.