ChewAnalyzer: Workload-Aware Data Management Across Differentiated Storage Pools

Xiongzi Ge
NetApp, Inc.
xiongzi@netapp.com

Xuchao Xie
NUDT
xuchxie@gmail.com

David H.C. Du
University of Minnesota
du@umn.edu

Pradeep Ganesan
NetApp, Inc.
pradeepg@netapp.com

Dennis Hahn
NetApp, Inc.
denny4@techie.com

***Abstract*—In multi-tier storage systems, moving data from one tier to the next can be inefficient. And because each type of storage device has its own idiosyncrasies with respect to the workloads that it can best support, unnecessary data movement might result. In this paper, we explore a fully connected storage architecture in which data can move from any storage pool to another. We propose a *Ch*unk-level storag*e*-aware *w*orkload *Analyzer* framework, abbreviated as ChewAnalyzer, to facilitate efficient data placement. Access patterns are characterized in a flexible way by a collection of I/O accesses to a data chunk. ChewAnalyzer employs a Hierarchical Classifier [30] to analyze the chunk patterns step by step. In each classification step, the Chunk Placement Recommender suggests new data placement policies according to the device properties. Based on the analysis of access pattern changes, the Storage Manager can adequately distribute or migrate the data chunks across different storage pools. Our experimental results show that ChewAnalyzer improves the initial data placement and that it migrates data into the proper pools directly and efficiently.**

***Index Terms*—Storage, Fully connected storage, Hierarchical Classifier, Data placement, Data movement**

I. Introduction

To reduce total cost of ownership (TCO), enterprise storage vendors are seeking a way to efficiently incorporate different storage technologies and devices into one storage system, where each particular type of device can form a storage pool. Various combinations of flash-based Solid-Stated Drives (SSDs) [18] and Hard Disk Drives (HDDs) have been extensively studied [35]. Today, more storage devices and technologies are emerging such as phase change-memory (PCM), spin-torque transfer RAM (STT-RAM) and memristors [33] —that can serve not only as non-volatile memory, but also as promising high performance non-volatile storage [28] (termed Storage Class Memory (SCM)). The performance of SCM is close to memory. In the foreseeable future, however, the cost per byte of SCM will still be much more than that of traditional storage devices such as flash-based or spinning disks. Different devices and technologies have unique performance characteristics, and such diverse storage technologies in one system make enterprise storage hierarchies more interesting. What used to be a two-tiered hierarchy (dynamic random access memory (DRAM) and HDD) is quickly broadening into multiple tiers.

In the past, storage tiering has been viewed as a method of achieving both performance and affordability. The expected process of data placement is that fast tiers serve a group of intensive workloads for better performance and slow tiers are persistently storing the rest of the data blocks to provide lower cost and higher capacity [36]. Data is migrated from one tier to the next when data access frequency changes. However, moving data between tiers in a predetermined order can be inefficient and it can lead to unnecessary data movement in a diverse multiple storage pool environment.

With proper initial data placement and efficient data migration, storage systems can consist of multiple storage pools that are fully connected with data that is stored in an appropriate storage pool to suit the system's current workload profile. Each storage pool has its own unique price-performance tradeoff and idiosyncrasies with respect to the workload characteristics that it best supports. In addition, the workload profile might change from time to time. Thus, it is challenging to decide how data should be placed in a storage pool initially and when to migrate data from one storage pool to another.

Current commercial multi-tier storage systems from Dell EMC [2], [5], HPE [1], IBM [6] and NetApp [11] provide a block-level interface to customers. Compared with file-level management, block-level management provides finer granularity of control but is challenging because of its limited access information from the upper layers. In this paper we focus on dealing with data blocks at the chunk level. A chunk is a set of consecutive data blocks.

Accurate chunk-level workload characterization can help a system determine which resources are adequate for the associated requests. The traditional hot and cold data classification methodology is employed in the tier-by-tier case [16]. Each data chunk is characterized by a certain access pattern, typically, high or low I/Os per second (IOPS) in a period of time. A greedy migration policy is followed to move data chunks according to their access patterns between different storage pools [20]. However, the employment of different dimensions can generate entirely different access patterns. Workload profiling is closely related to a particular device technology. For example, it is valuable to pay more attention to write requests when both SSDs and SCM are considered. Although both offer better random IOPS than HDDs do, because of the out-place-update and lifetime issues, researchers and developers still prefer to reduce random writes on flash-based SSDs [17], [25]. Furthermore, applications such as big data processing (e.g. Hadoop) have their own characteristics (e.g., streaming or batch processing) [20]. As a result, conventional methods of storage workload analysis that orient to tier-by-tier

TABLE I
STORAGE I/O WORKLOAD CHARACTERIZATION DIMENSIONS.

I/O size
I/O Intensity
Read/Write ratio
Randomness/Sequentiality
Local-access

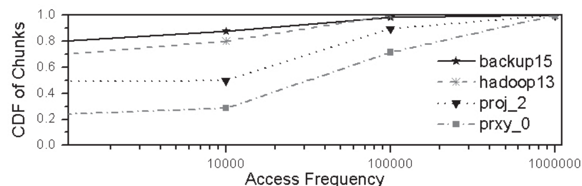

Fig. 1. Chunk access frequency cumulative distribution function (CDF).

cases do not apply to multiple differentiated storage pools.

In this paper, to explore data access patterns in such a fully connected topology, we propose a ***Ch**unk-level storag**e**-aware **w**orkload **Analyzer*** framework, abbreviated as ChewAnalyzer. Specifically, we make the following contributions:

- A detailed experimental study of several enterprise block I/O traces that also shows the shortcomings of the existing methods of profiling block-level workloads
- A framework to conduct cost-effective data management across multiple differentiated storage pools by using a Hierarchical Classifier [30]
- A trace-driven evaluation with SCM, SSD and HDD to show the improvement that ChewAnalyzer contributes

II. I/O WORKLOAD ANALYSIS

Unlike the workload analyses for caching, we focus on data access characteristics over longer durations. Page-level cache replacement policies usually make quick and heuristic decisions that are based on temporal changes to impel memory. The decisions for data movement in large-scale heterogeneous storage systems are more cautious and generally perform at predetermined intervals (by the hour or by the day) or on demand.

A. Preliminary I/O Workload Analyses

To analyze the I/O traces, generally, we divide the entire storage space into fixed-size or variable-size chunks. For our study, we focused on fixed-size chunks. We then characterized the collection of I/O accesses to each chunk in a constant epoch (time window). The major dimensions that we used are summarized in Table I. I/O intensity means the average number of accesses during a time window. The Read/Write ratio is used to classify read- or write-dominated patterns. We study I/O randomness/sequentiality through different sequential detection policies [24]. High local-access ratio [4] describes a scenario in which most of the requests concentrate on certain data blocks in a chunk. For example, an overwritten pattern means a high degree of local access of write requests. The access patterns are defined from one or multiple dimensions. To quantify the internal characteristics, we defined four different classifications of chunk access patterns, as shown in Table II. Taxonomy rule 1 is based on the intensity. Taxonomy rule 2 is based on the degree of the sequentiality and randomness. Taxonomy rule 3 combines the aforementioned two dimensions, sequentiality and randomness, with theread/write ratio. Taxonomy rule 4 differentiates various write patterns, considering sequentiality and local-access simultaneously.

B. I/O Traces

We characterized a set of block-level IO traces that were collected from two kinds of enterprise storage systems. The IO traces that wereused in ChewAnalyzer include:

- **MSRC traces.** The Microsoft Research Cambridge (MSRC) block-level IO traces collected I/O requests on 36 volumes in 13 servers in about one week [9].
- **NetApp traces**. Two kinds of NetApp block-level IO traces were collected on NetApp®E-Series disk arrays [10]. One was from a Hadoop Distributed File System (HDFS) server that ran for about 8 hours. The other came from a backup server that ran for about one day.

C. Observations

Observation 1: Access patterns at the chunk level might be predictable over long periods.

Table III summarizes the I/O traces from the two storage systems. Chunk size was set to 1024MB. We calculated the total I/O count and I/O size in each chunk. Then we sorted them based on the accumulative I/O count and I/O size, respectively. For example, “Access to Top 5% I/O” means the ratio of the number of I/Os in the top 5% of chunks that have the most I/O accesses to the total I/O number in all the chunks. “Access to Top 5% Data” means the ratio of the total I/O size of the top 5% of chunks that have the greatest accumulative I/O sizes to the size of all the I/Os. In most cases, most of the accesses concentrate on a small portion of the data. Thus, we can improve data movement mainly by paying more attention to these active chunks. We set a tuple of thresholds to distinguish the I/O intensities of Taxonomy rule 1 in Table I (i.e., non-access means zero access, inactive means fewer than two accesses in a time window, and the rest are active). Figure 1 shows the function of chunk access frequency cumulative distribution for different I/O workloads. It shows that the inactive data chunks in most cases were rarely accessed (less than twice in a 30-minute time window). Therefore, we can place them on HDDs. More surprisingly, most of the data chunks were never accessed in most of the periods. Therefore, these data chunks can reside on HDDs, they also can be grouped together and can be managed with a bigger chunk size.

The existing work of workload analysis shows that the active data set remains for a long duration in most of the workloads [27]. This case is similar to Taxonomy rule 1 in Table II. We studied the access patterns of different workloads based on Taxonomy rule 3 in Table II. Sequential detection

TABLE II
CHUNK ACCESS PATTERN CLASSIFICATION EXAMPLES.

Taxonomy rule 1	Non-access	Inactive	Active
Taxonomy rule 2	Sequential	Random	
Taxonomy rule 3	Sequential	Random Read	Random Write
Taxonomy rule 4	Sequential write	Fully random write	Overwritten

TABLE III
SUMMARIZATION OF I/O TRACES.

Trace	R/W (GB)	R/W IO Ratio	Access to Top 5% IO (%)	Access to Top 20% IO (%)	Access to Top 5% Data (%)	Access to Top 20% Data (%)
$proj_2$	1015.95/168.68	7.07	21.02	48.24	14.32	45.14
$prxy_0$	3.04/53.80	0.03	65.24	94.43	54.33	89.15
$hadoop13$	189.31/422.31	4.17	13.31	43.27	31.27	56.67
$backup15$	161.98/194.9	1.55	59.79	96.39	59.63	97.67

was implemented by identifying the distance between the Logical Block Address (LBA) of two consecutive requests (512 KB used in [20]). The results of analytical chunk-level characterization of these IO traces show that the access patterns of many chunks and of some of the adjacent chunks can be regular in a period of time, which can potentially be used to optimize data management in a heterogeneous storage system.

On the one hand, repetitive patterns occur in many of the chunks. For traces like $hadoop13$, $backup15$ and $proj_2$, the patterns repeat in a 10-minute, 1-hour, and 1-day duration, respectively. With such information, we can predict the future access patterns and migrate data accordingly. The access patterns we analyzed in this model tended to be stable over long durations. For example, the random write access pattern always concentrated on certain chunks and repeated occasionally. In $prxy_0$, we found that 10% of the chunks stay mostly in a random write pattern. Therefore, if a storage system has enough free SCM space, it is best that we place these chunks on SCM. On the other hand, the correlation between chunk accesses is similar to the spatial locality in a cache study but in a much longer period. This observation inspires us to define proper patterns and to match each of them to a certain type of device.

Observation 2: Chunk access pattern might not change linearly.

A previous study of tiered storage employed a heat-based approach [16] to guide the data movement between tiers. The most common metric is the access frequency (i.e., the heat) of a chunk [20]. Data is moved based on the chunk temperature and data access intensity is supposed to increase gradually. Based on this assumption, data migration is carried out in a slow and conservative way. However, because of bursty I/Os, the access intensity in certain chunks might not increase or decrease linearly. In some cases, the access pattern is only moderately hot and stable for a while without the appearance of obvious intense or cold patterns. Moreover, in many cases, certain access patterns occur suddenly but also disappear quickly. And current big data processing applications have their own access characteristics. On an HDFS data server,

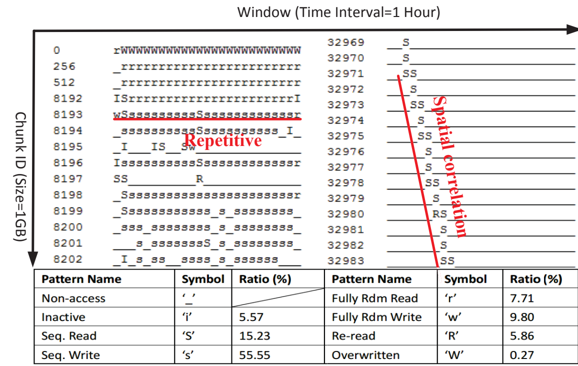

Fig. 2. Diagram of chunk access patterns in consecutive time windows for the $backup15$ trace.

I/O requests are usually read sequentially from disks and are accessed randomly over and over as shown in Figure 2. After analyzing these cases, it is difficult to presume that data access will follow a hot-to-warm-to-cold pattern.

Data flows tier by tier might be inefficient in a storage system that incorporates diverse storage pools. Movement of data from one tier to the next tier not only increases data migration cost, but it also creates potentially unnecessary downstream and upstream data movement. This movement might waste the valuable I/O bandwidth between tiers. For example, if a chunk that is initially in the slowest tier becomes hot, it has to go through a middle tier before it movesg to the fastest tier. Considering the different intrinsic properties of each storage device, there might not be a linear order that can be arranged in these devices.

III. THE FRAMEWORK OF CHEWANALYZER

A. Overview

Let's consider a configuration with multiple storage pools. To overcome the shortcomings of tier by tier data management, the storage pools are fully connected so that data can be moved between any two of them. Suppose that we have n storage pools. Figure 3 presents an overview of ChewAnalyzer in that scenario. The I/O Monitor keeps collecting the necessary

I/O access statistical information in real time. It monitors the incoming I/O requests and tracks the performance results of request executions on the storage devices such as the latency of each request and the queue length of each storage pool. Then ChewAnalyzer analyzes the I/O access information mainly through two major components, a Hierarchical Classifier (HC) and a Chunk Placement Recommender (CPR).

The HC [26], [30] is used to perform flexible access pattern classification for each chunk according to a list of predefined taxonomy rules. ChewAnalyzer uses HC to find the appropriate chunks to place on the fastest pool (e.g. SCM) and then to select chunks to place on the second-fastest pool and so on, until all the data chunks have been placed. The HC process is triggered during selection of data chunks on multiple storage pools (target pools). HC groups data chunks into a set of categories on the basis of one taxonomy rule at each level. A taxonomy rule is defined as a dimension.

We build a benchmark tool to calculate the relative weight values in one dimension for various types of devices. Then we use a threshold to partition chunks based on their weighted values, as shown in Figure 3. After one classification level is finished, CPR looks up the status of the target pools and recommends data placement policies based on the Pattern-to-Pool Chunk Placement Library. If the available storage space and available performance capability in the targeted pools are sufficient for CPR to generate final data placement policies, chunk pattern classification and detection in HC stop at this level. Otherwise, the next hierarchical classification in HC is triggered. After HC finishes, each chunk is selected to be placed in a certain targeted pool. Finally, the placement policies are delivered to the Storage Manager, which makes the final decisions about data movement. In the next section, we describe this process in detail.

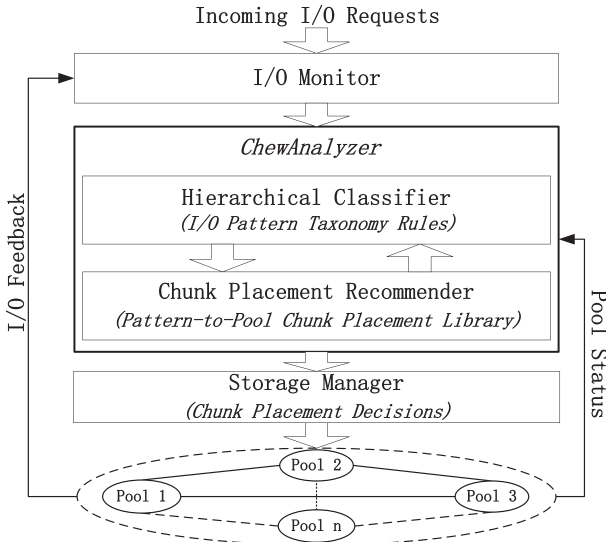

Fig. 3. Overview of ChewAnalyer.

B. ChewAnalyzer Deployment: HC and Chunk Placement

To classify a large set of chunks into different access patterns, ChewAnalyzer uses a Hierarchical Classifier (HC) [26], [30] instead of a flat classifier to group chunks level by level. To explain the HC methodology, we first introduce several notations in Table IV.

TABLE IV
NOTATIONS IN CHEWANALYZER.

Notation	Definition
$chunk(i)$	The $i-th$ chunk
n	The total number of pools
x	The total number of chunks
d	The total number of dimensions for chunk characterization
n	The total number of storage pools
$Size(i)$	The size of chunk i
$FreeSpace(i)$	The total free space on the $i-th$ pool
$Performance(i)$	The current required performance value on the $i-th$ pool [20] (IOPS, throughput)
$Peak(i)$	The peak performance of the $i-th$ pool
$dimension_i$	The $i-th$ dimension for workload profiling
h	The depth of HC
$P_{status}(i)$	The status of the $i-th$ pool

We use the set, D, to represent all the workload characterization dimensions that are used in ChewAnalyzer, denoted as:

$$D = \{dimension_i\}, 1 \leq i \leq d \tag{1}$$

D includes the normal dimensions (e.g. I/O intensity) in Table I for chunk level I/O access characterization. Then at each level, we define a taxonomy rule to partition the chunks into different groups on the basis of their access patterns. The dimension that is used in each taxonomy rule determines the results of the partitions. The idiosyncrasies of each storage pool are respected in defining the taxonomy rules. In this paper, to define a taxonomy rule, we introduce a metric and the relative weight.

To quantify the access differences among diverse storage devices, we develop a benchmark tool with pre-configured access patterns and run it on different devices to measure the performance. For each dimension, we calculate the relative weight for each type of device. This process is known as pattern-to-pool assertion by treating each storage device as a black box. We then compare the performance results to get the performance differences, or weights. For example, if we have two storage pools, SSD and SCM, their read/write accesses perform differently. We run our benchmark tool to find the average read and write performance for SSD and for SCM individually. Based on the average performance for each device, we obtain the weights of a write request and of a read request on SSD and on SCM. Other factors, such as device lifetime, can also be considered. For example, flash-based SSDs are more sensitive and more fragile for write patterns, so we can amplify the weight value by $w0$. The weight of write to read on SSD can be calculated as:

$$Weight_ssd(write) = Weight_ssd(read) * w0 \tag{2}$$

The weight value of each access on a chunk is determined by the dimensions in the current level and in the previous levels as well. We define a weight function of the access, x, in $dimension_i$ as $W_i(x)$. In level 1, if we use $dimension_j$ to characterize the access pattern, the weight of access x on $chunk(i)$ is:

$$Weight(x)(1) = W_j(x) \quad (3)$$

In level $k+1$, if $dimension_q$ is used, the total weight of x on $chunk(i)$ can be calculated as:

$$Weight(x)(k+1) = W_q(x) * Weight(x)(k) \quad (4)$$

The total weight on $chunk(i)$ is accumulated by the total accesses on this chunk.

Chunk placement decisions are closely related to the pool status. If, and only if, the required resources of all the chunks are met in a pool, the placement decisions are considered to be valid. The performance ability of the pool is a factor in achieving acceptable Quality of Services (QoS) for the requests. We employ the approximate performance metric as in [20] by considering both IOPS and throughput. We define the current pool status as an AND operation of both free space and the available performance of this pool, denoted as:

$$P_{status}(i) = FreeSpace(i) \&\& (Peak(i) - Performance(i)) \quad (5)$$

The pool status includes the remaining free space of each pool as $FreeSpace(i)$ and the total available performance ability of each pool. If both are positive numbers, the placement policies are valid.

After we finish the computation of the weight value at a certain level, a set of thresholds based on the storage space of each pool is used to partition the chunks. These thresholds are usually set based on the Pattern-to-Pool Chunk Placement Library.

We build a Pattern-to-Pool Chunk Placement Library to guide ChewAnalyzer to make data placement decisions. This library takes into consideration the idiosyncrasies of each storage pool. For example, an SSD pool might be suitable for random read accesses, but it is also fitting for sequential read I/Os. For a single-level cell (SLC)-based flash pool, random write I/Os are still good candidates. There might be priorities for the CPR to determine which one should be selected first. To set each weight value, we can also consider the penalty of putting the chunks with such patterns into the pool, such as the lifetime impact, power consumption and so on.

When the chunk-level I/O access information arrives, ChewAnalyzer chooses one dimension to divide the chunks into several pattern groups. For example, if HC uses the I/O intensity dimension, it partitions the data chunks into n groups at most (because we have n pools) based on the number of I/O accesses. It can contain fewer than n groups as long as the storage pools can keep all the chunks. If the results are not sufficient for the Storage Manager to make a data placement decision, another dimension is added to partition the chunks in the next level. If HC goes to the last level, ChewAnalyze selects the chunks to place in different pools.

The ordering of taxonomy rules to build the HC plays an important role in classification efficiency. To fully utilize the resources, ChewAnalyzer strives to completely fill the fastest pool with data chunks. Thus, HC first chooses one dimension to sieve the chunks that can be served better on the fastest pool than on the rest of the pools. If the sieved chunks cannot all be kept on the fastest pool, HC goes to the next level, using another dimension to partition further. For example, for the SCM pool, it is preferable to keep as many data chunks as possible that are going to be accessed, so we use I/O intensity to select the most active data chunks for placement into SCM. It is better to place chunks with random write patterns on SCM than to place random read chunks on SCM, if both intensities are almost equal.

ChewAnalyzer employs hierarchical taxonomy rules to classify and to characterize chunks. The major advantage of using HC is to divide the chunks into different categories progressively. When HC has reached the highest level, some chunks might not have not satisfied all the top placement policies from the Pattern-to-Pool library. In that case, the placement decisions start from the fastest pool and move progressively to the slowest pool , and chunks are selected based on the weight in a pool in the remaining list until each successive pool is full. Finally, after the selection process, each chunk is placed in a certain pool. To make ChewAnalyzer more flexible, system designers can indicate more than one pool as the top priority placement decision for certain access patterns.

The placement decision of $chunk(i)$ in the $j-th$ pool is denoted as:

$$Placement(i) = j, 1 \leq i \leq x, 1 \leq j \leq n \quad (6)$$

where x is the total number of chunks and n is the total number of pools.

C. Storage Manager

The placement policies are recommended by CPR and then are delivered to the Storage Manager to carry out the data placements or migrations. This data movement might degrade pool performance temporarily. In the worst case, the migration requests consume extensive amounts of I/O resources, and storage pools cannot complete the application requests. To accommodate the incoming application I/O requests and the pool utilization, Storage Manager makes final placement decisions that avoid such situations. It periodically looks up the current status of all the storage pools, and it also asks for information about workload statistics from the I/O Monitor.

The way that migration is scheduled depends on the current workload burden. To avoid performance degradation of I/O traffic for the foreground requests, the migration requests are first appended into a separate background process. The data migration process is activated when the target pool and the source pool both are available to allow data migration requests to be carried out. A deadline is given to each migration request. If the deadline expires before the next available period, new data migration policies are considered.

IV. Evaluation

We evaluated ChewAnalyzer by replaying real enterprise block I/O traces on a Linux platform. We built a three-pool prototype that was based on Linux device mapper (DM) [3]. We used Linux kernel version 4.4. The heterogeneous storage controller worked as a standard Linux block driver in the DM, which managed the block devices linearly. Our trace replay engine was implemented through libaio [8] on the user level. We enhanced the heterogeneous storage controller to perform chunk-level data migration by using the "kcopyd" mechanism in the DM [7]. The chunk migration manager communicated with ChewAnalyzer through sysfs [13]. Data chunks were triggered to move from one pool to another. To maintain the mapping after migration, we recorded the chunks by using a mapping table in the memory. Each tuple contained the logical chunk ID and the physical address chunk ID on a pool. We run the experiments on a Dell PowerEdge server that was configured with a Seagate 8 TB HDD, a Samsung 850 PRO 512GB SSD, 48GB DDR3 memory (SCM was emulated by changing the memmap kernel parameter [12]), and an Intel Xeon E5-2407 2.20 GHz CPU.

Sequential detection was non-trivial. In our evaluation, sequential I/O patterns were carefully classified and detected as in [24]. The baseline methods that we used to compare with ChewAnalyzer contained the following two existing policies:

- **Greedy IOPS-only Dynamic Tiering (IOPS-only)**: IOPS-only is used widely as a heuristic approach to group hot data. This policy measures the IOPS on each chunk to migrate data chunks tier by tier.
- **Extent-Based Dynamic Tiering (EDT) [20]**: EDT migrates fixed size data chunks tier by tier considering both IOPS and random/sequential accesses.

We ran the four policies by replaying the four traces as summarized in Table III, $prxy_0$, $proj_2$, $hadoop13$, and $backup15$. To carry out migration, we devided the running time into fixed-size epochs (45-minute for $prxy_0$, 1-hour for $proj_2$, and 20-minute for both $hadoop13$ and $backup15$). In addition, the total number of requests in each window was set to less than 20,000, and chunk size is set to 256MB. In the following experiments, the storage configurations as (SCM, SSD) for $hadoop13$, $backup15$, $proj_2$, and $prxy_0$ are (10GB, 100GB), (10GB, 100GB), (10GB, 40GB), and (2GB, 10GB), respectively.

Figure 4 shows the normalized average I/O latency. For all four cases, the IOPS-only policy has the highest latency. With lower I/O latency, ChewAnalyzer outperforms EDT and IOPS-only. Compared with IOPS-only, ChewAnalyzer accelerates I/O access for the four different configurations by 26%, 13%, 24%, and 23%, respectively. ChewAnalyzer's acceleration is exceptionally high in comparison because IOPS-only and EDT do not perform fine-grained access pattern classification. With HC, ChewAnalyzer analyzes more access pattern dimensions.

Figure 5 depicts the normalized total amount of data that was migrated. For the backup trace, both IOPS-only and EDT have almost the same amount of migration load. ChewAnalyzer has 45% less migration load compared with IOPS-only. For the $prxy_0$ trace, ChewAnalyzer reduced the total amount of data that migrated by 46% and 64% compared with EDT and IOPS-only, respectively. Both IOPS-only and EDT perform data movement tier by tier, which can incur extra migration load. In addition, the migration overhead hurts overall performance if storage devices are connected through an external network.

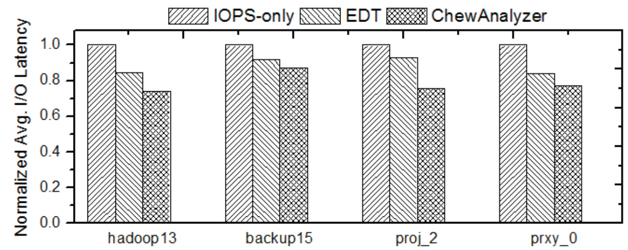

Fig. 4. Normalized average I/O latency for the four policies.

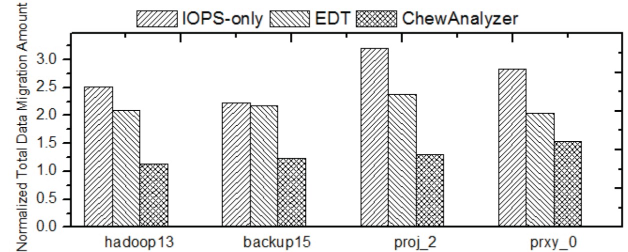

Fig. 5. Normalized total data migration amount

Another factor to consider is that flash-based SSDs have asymmetric read and write speeds. Moreover, flash has a wear-out problem that affects the lifetime of the drives. In the configuration of the Chunk-to-Pattern Chunk Placement Library, ChewAnalyzer prioritizes the write-dominated chunks on SCM to reduce the write load on the SSD. Figure 6 shows the normalized write request distribution on each pool for the $prxy_0$ trace. ChewAnalyzer increases the write load on SCM but reduces the total write times on the SSD. This approach significantly reduces the write load on SSD and lengthens its useful life.

In summary, our experiments show that ChewAnalyzer outperforms the other two policies by lowering the average I/O latency. In addition, it reduces the migration load and the random write load on an SSD.

V. Related work

Following is a summay of related work and a comparison with ChewAnalyzer.

A. Tiered Storage Management

Recent research has focused on improving storage cost and utilization efficiency in different storage tiers. ExaPlan [23] [22] considers access-pattern-aware data placement by using a queueing model to minimize system's mean response time for large tiered storage systems. It mainly supports SSDs

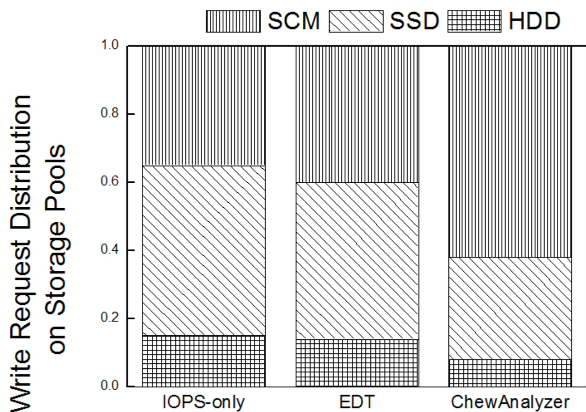

Fig. 6. Distribution of write requests on each pool.

and HDDs, as well as their disk-array counterparts. Guerra et al. [20] built a dynamic tiering system that combines SSDs with SAS and SATA disks to minimize costs and power consumption. IOPS and throughput are considered to carefully control the overhead because of the extent-based migration. CAST [16] provides insights into the design of a tiered storage management framework for cloud-based data analytical workloads. File-level based HSM [21] [31] takes advantage of the fact that data does not have the same value during any given period of time for applications. In HSM, data migration conditions are typically based on the ages of files, the types of files,the popularity of files, and the availability of free space on the storage device [21]. However, the mechanisms that have been used in earlier approaches cannot be directly applied to fully connected differentiated storage pools that including SCM, SSDs, and HDDs, mainly because these mechanisms generally ignore the special read and write properties between SSDs and SCM. ChewAnalyzer uses a different method to determine and to use access patterns to facilitate data placement, and it focuses on multiple storage pools ($\geq$ 3), each of which consists of one type of storage device.

B. I/O Workload Characterization

The theory of I/O workload characterization provides a useful abstraction for describing workloads more concisely, particularly with respect to how they behave in hierarchical storage systems. A large body of work from the storage community explores methods for representing workloads concisely. Chen et al. [29] and Seo et al. [15] exploit workload features through machine learning techniques. Tarasov et al. extract and model large-block I/O workloads with feature matrices [32]. Delimitrou et al. model network traffic workloads by using Markov chains [19]. Bonfire [38] accelerates the cache warm up by using more efficient preload methods. Windows SuperFetch [14] preloads the frequently used system and application information and libraries into memory based on the historical usage pattern to reduce the system boot and application launching time. CAST [16], a storage tiering framework, performs cloud storage allocation and data placement for analytical workloads to achieve high performance cost-effectively. These related studies focus mostly on identifying the data that should be brought into the fast tier for higher efficiency based only on IOPS and intensity of workloads, but ChewAnalyzer concentrates on seeking chunk-level data management to efficiently incorporate differentiated storage pools.

C. SSD and SCM Deployment in Storage Systems

Deployment of NAND flash memory based SSDS in storage systems has increased over the past decade because of its advantages, such as its light weight, high performance and low power consumption. However, the limited P/E cycles may accelerate wear-out of flash chips in SSDs, which is always a potential reliability issue in SSD-based storage systems [25] [17]. The relative high cost of write operations is still the performance bottleneck of flash memory. Hybrid HDD and SSD storage systems have been extensively studied [34]. For example, GREM presents dynamic SSD resource allocation in virtual machine envrioments [37]. Today's PCIe-based SCMs represent an astounding 1000X performance increase compared with traditional spinning disks ($\sim$100,000 IOPS versus $\sim$100) [28]. To maximize the value that is derived from high-cost SCM, storage systems must consistently be able to saturate these devices. Despite the attractive performance of these devices, it is very challenging to effectively slot them into existing systems. Hardware and software must be designed together with an aim of maximizing efficiency. In ChewAnalyzer, we better utilize the unique price-performance tradeoff and idiosyncrasies of SCM, SSDs, and HDDs in a fully connected differentiated storage system.

VI. Conclusion

We have studied an architecture with fully connected differentiated storage pools. To analyze the internal access patterns and to efficiently place data in such a complete topology, we propose a chunk-level storage-aware workload analyzer framework, abbreviated as ChewAnalyzer. In particular, ChewAnalyzer employs a Hierarchical Classifier to analyze the chunk patterns step by step. ChewAnalyzer improves the initial data placement and migrates data into the proper pools directly and efficiently. In this study, we did not consider variable size chunks, the connection cost , or the available bandwidth between different storage pools. These considerations will be included in our future studies.

Acknowledgments

We would like to thank the anonymous MASCOTS reviewers for their helpful comments to improve the previous draft of this paper. We also thank Al Andux, Ajay Bakre, Jerry Fredin, and Art Harkin from NetApp and all the members in the CRIS group for their suggestions and support. This work was partially supported by NSF awards 1305237, 1812537, 1421913, 1439622 and 1525617.

References

[1] Adaptive optimization for HPE 3PAR StoreServ Storage. https://www.hpe.com/h20195/V2/GetPDF.aspx/4AA4-0867ENW.pdf.

[2] Application tiering and tiered storage. https://partnerdirect.dell.com/sites/channel/en-us/documents/cb120_application_tiering_and_tiered_storage.pdf.

[3] Device-mapper resource page. https://www.sourceware.org/dm/.

[4] The DiskSim Simulation Environment (v4.0). http://www.pdl.cmu.edu/DiskSim/.

[5] Fully automated storage tiering (FAST). https://www.emc.com/corporate/glossary/fully-automated-storage-tiering.html.

[6] IBM DS8880 Hybrid Storage. https://www-03.ibm.com/systems/storage/hybrid-storage/ds8000/index.html.

[7] kcopyd. https://www.kernel.org/doc/Documentation/device-mapper/kcopyd.txt.

[8] Kernel Asynchronous I/O (AIO) Support for Linux. http://lse.sourceforge.net/io/aio.html.

[9] MSR Cambridge Traces. http://iotta.snia.org/traces/388.

[10] NetApp E-Series Product Comparison. http://www.netapp.com/us/products/storage-systems/e5400/e5400-product-comparison.aspx.

[11] NetApp Virtual Storage Tier. http://www.netapp.com/us/media/ds-3177-0412.pdf.

[12] Persistent Memory Wiki. https://nvdimm.wiki.kernel.org/how_to_choose_the_correct_memmap_kernel_parameter_for_pmem_on_your_system.

[13] sysfs - the filesystem for exporting kernel objects. https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt.

[14] Windows SuperFetch. https://msdn.microsoft.com/en-us/library/windows/hardware/dn653317(v=vs.85).aspx.

[15] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 43–56. ACM, 2011.

[16] Y. Cheng, M. S. Iqbal, A. Gupta, and A. R. Butt. CAST: Tiering storage for data analytics in the cloud. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 45–56. ACM, 2015.

[17] J. Colgrove, J. D. Davis, J. Hayes, E. L. Miller, C. Sandvig, R. Sears, A. Tamches, N. Vachharajani, and F. Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1683–1694. ACM, 2015.

[18] M. Cornwell. Anatomy of a solid-state drive. *Commun. ACM*, 55(12):59–63, 2012.

[19] C. Delimitrou, S. Sankar, A. Kansal, and C. Kozyrakis. ECHO: Recreating network traffic maps for datacenters with tens of thousands of servers. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 14–24. IEEE, 2012.

[20] J. Guerra, H. Pucha, J. S. Glider, W. Belluomini, and R. Rangaswami. Cost effective storage using extent based dynamic tiering. In *FAST*, volume 11, pages 20–20, 2011.

[21] D. He, X. Zhang, D. H. Du, and G. Grider. Coordinating parallel hierarchical storage management in object-based cluster file systems. In *Proceedings of the 23rd IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2006.

[22] I. Iliadis, J. Jelitto, Y. Kim, S. Sarafijanovic, and V. Venkatesan. Exaplan: queueing-based data placement and provisioning for large tiered storage systems. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on*, pages 218–227. IEEE, 2015.

[23] I. Iliadis, J. Jelitto, Y. Kim, S. Sarafijanovic, and V. Venkatesan. ExaPlan: Efficient Queueing-Based Data Placement, Provisioning, and Load Balancing for Large Tiered Storage Systems. *ACM Transactions on Storage (TOS)*, 13(2):17, 2017.

[24] C. Li, P. Shilane, F. Douglis, D. Sawyer, and H. Shim. Assert (! defined (sequential i/o)). In *HotStorage*, 2014.

[25] Q. Li, L. Shi, C. J. Xue, K. Wu, C. Ji, Q. Zhuge, and E. H.-M. Sha. Access characteristic guided read and write cost regulation for performance improvement on flash memory. In *FAST*, pages 125–132, 2016.

[26] X. Li and D. Roth. Learning question classifiers. In *Proceedings of the 19th international conference on Computational linguistics-Volume 1*, pages 1–7. Association for Computational Linguistics, 2002.

[27] A. Miranda and T. Cortes. CRAID: online RAID upgrades using dynamic hot data reorganization. In *FAST*, pages 133–146, 2014.

[28] M. Nanavati, M. Schwarzkopf, J. Wires, and A. Warfield. Non-volatile storage. *Communications of the ACM*, 59(1):56–63, 2015.

[29] B. Seo, S. Kang, J. Choi, J. Cha, Y. Won, and S. Yoon. IO workload characterization revisited: A data-mining approach. *IEEE Transactions on Computers*, 63(12):3026–3038, 2014.

[30] I. K. Sethi and G. Sarvarayudu. Hierarchical classifier design using mutual information. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (4):441–445, 1982.

[31] T. F. Sienknecht, R. J. Friedrich, J. J. Martinka, and P. M. Friedenbach. The implications of distributed data in a commercial environment on the design of hierarchical storage management. *Performance Evaluation*, 20(1-3):3–25, 1994.

[32] V. Tarasov, S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *FAST*, volume 12, page 22, 2012.

[33] A. Waizy, D. Kasper, J. Schrage, B. Höppner, F. Salfner, H. Schmitz, J.-H. Böse, and S. I. C. Potsdam. Storage Class Memory Evaluation for SAP HANA. *HPI Future SOC Lab: Proceedings 2013*, 88:63, 2015.

[34] L. Wan, Z. Lu, Q. Cao, F. Wang, S. Oral, and B. Settlemyer. SSD-optimized workload placement with adaptive learning and classification in HPC environments. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–6. IEEE, 2014.

[35] H. Wang and P. J. Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *FAST*, pages 229–242, 2014.

[36] J. Xue, F. Yan, A. Riska, and E. Smirni. Storage workload isolation via tier warming: How models can help. In *ICAC*, pages 1–11, 2014.

[37] Z. Yang, J. Tai, and N. Mi. GREM: Dynamic SSD Resource Allocation In Virtualized Storage Systems With Heterogeneous VMs. In *35th IEEE International Performance Computing and Communications Conference (IPCCC). IEEE*, 2016.

[38] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *FAST*, pages 59–72, 2013.

NETAPP, the NETAPP logo, and the marks listed at http://www.netapp.com/TM are trademarks of NetApp, Inc. Other company and product names may be trademarks of their respective owners.