GREM: Dynamic SSD Resource Allocation In Virtualized Storage Systems With Heterogeneous IO Workloads

Zhengyu Yang*, Jianzhe Tai*, Janki Bhimani*, Jiayin Wang‡, Ningfang Mi*, and Bo Sheng‡
*Dept. of Electrical & Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115
‡Dept. of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA

***Abstract*—In a shared virtualized storage system that runs VMs with heterogeneous IO demands, it becomes a problem for the hypervisor to cost-effectively partition and allocate SSD resources among multiple VMs. There are two straightforward approaches to solving this problem: equally assigning SSDs to each VM or managing SSD resources in a fair competition mode. Unfortunately, neither of these approaches can fully utilize the benefits of SSD resources, particularly when the workloads frequently change and bursty IOs occur from time to time. In this paper, we design a Global SSD Resource Management solution - GREM, which aims to fully utilize SSD resources as a second-level cache under the consideration of performance isolation. In particular, GREM takes dynamic IO demands of all VMs into consideration to split the entire SSD space into a long-term zone and a short-term zone, and cost-effectively updates the content of SSDs in these two zones. GREM is able to adaptively adjust the reservation for each VM inside the long-term zone based on their IO changes. GREM can further dynamically partition SSDs between the long- and short-term zones during runtime by leveraging the feedbacks from both cache performance and bursty workloads. Experimental results show that GREM can capture the cross-VM IO changes to make correct decisions on resource allocation, and thus obtain high IO hit ratio and low IO management costs, compared with both traditional and state-of-the-art caching algorithms.**

***Keywords*—Solid State Drives, Resource Allocation, Virtualized Storage Systems, Caching Algorithms, Bursty Detection, I/O Workload Characterization**

I. INTRODUCTION

Virtualized systems nowadays are a basic supporting infrastructure in commercial cloud computing environments. In such a virtualized system, multiple virtual machines (VMs) often share storage services, and each VM has its own workload pattern and IO requirements. It then becomes very important to provide high performance and availability to virtual machines. Therefore, flash-based Solid-State Drives (SSDs) are widely being deployed as a per-virtual disk, second-level cache of Hard Disk Drives (HDDs) in virtualized systems to improve IO access performance (e.g., increasing IO throughput and reducing IO latency), and achieve low power consumption. In most of these shared virtualization platforms, SSD is statically pre-allocated to each virtual disk (VMDK) for simplicity, and the caching algorithm decides the cache admission and eviction for each VM only based on IO requests from that particular VM regardless of IOs from the others. It is difficult for the hypervisor to cost-effectively partition and allocate SSD resources among multiple VMs, particularly under diverse IO demands, because it lacks a global view of the cluster-wide disk IO activities. Therefore, in this paper, we focus on addressing a critical design problem for a virtualized storage system, i.e., *how to dynamically partition flash-based SSDs among multiple VMs and cost-effectively update the content of SSDs according to VM workload changes?* The goal of this design is to fully leverage the outstanding performance of shared SSD resources under the global view of caching management.

Typically, there are two straightforward approaches that allocate SSD resources among VMs by either equally or proportionally assigning SSDs to each VM or managing SSD resources in a fair competition mode. In the former approach, all VMs are purely isolated in using their own SSDs and the caching management is fully affected by their own workload changes. While, the second approach allows all VMs to freely use or share the entire SSDs, such that the caching management is centrally interfered by the intensity of all workload changes. However, we found that neither of these approaches can fully utilize the benefits of SSDs, especially when the workloads frequently change and bursts or spikes of IOs occur from time to time. For instance, if SSDs are equally reserved and assigned to all VMs, then VMs with bursty IOs cannot obtain more SSD resources. On the other hand, the second approach solves this issue by allowing all VMs to preempt or compete SSD resources based on their present IO demands. As a result, VMs with higher IO demands can occupy more SSDs by evicting less-accessed data from other VMs. However, under this approach, VMs with bursty IOs might occupy almost all the SSD resources, and then pollute the critical caching of other VMs. It is even worse that bursty workloads usually have less re-accesses in the long term.

In this paper, we strive to solve the above problems by designing a new Global SSD Resource Management solution, named "GREM", which takes dynamic IO demands of all VMs into consideration to split the entire SSD space into a long-term zone and a short-term zone and update the content of SSDs in these two zones cost-effectively. Intuitively, the long-term zone is designed for reserving SSD resources for each VM, in order to cache their own hottest data without any pollution from other VMs. Such a long-term zone is expected to guarantee high hit ratios from VMs that have cache-friendly workloads. On the other side, the short-term zone is used to absorb and handle bursty IOs (mostly from VMs with cache-unfriendly workloads) by being fairly competed among VMs according to their data popularities. In addition, we use a coarse temporal (e.g., 5min) and spatial (e.g., $1MB$) granularity to update the

This work was partially supported by National Science Foundation grant CNS-1552525, National Science Foundation Career Award CNS-1452751, and AFOSR grant FA9550-14-1-0160.

contents of SSDs in the two zones for reducing the cost of managing and operating SSD resources.

An important issue in the design of this new global Flash manager is "*how to dynamically partition SSD resources into two zones?*" and "*how to further dynamically allocate SSD resources in the long-term zone to different VMs?*". It is very challenging to effectively address this issue because we often have VMs with heterogeneous IO workloads (e.g., with a mix of cache-friendly and cache-unfriendly workloads) to share SSD resources and their IO access patterns can frequently change across time. Equally partitioning SSDs in the long-term zone to each VM can only improve the hit ratio for each VM to some extent, but cannot best utilize the reserved SSD resources in the long-term zone. Not all VMs keep fully utilizing their reserved SSDs during their lifetime, as their working data sets might be smaller than its reserved SSD space, or IO popularities of their data blocks decrease during some periods. Thus, GREM dynamically reserves SSDs in the long-term zone for each VM based on its runtime workload changes. Similarly, evenly splitting SSDs into the long- and short-term zones does not consider the diversity and dynamics in IO workloads. Therefore, we further develop D_GREM, a dynamic version of GREM, which online monitors the changes in IO demands of all VMs as well as the SSD allocation performance (e.g., IO hit ratios) and uses this information to dynamically partitioning SSD resources between two zones, and reserving different amounts of SSD resources to each VM.

We conduct trace-driven experiments by replaying real enterprise IO workloads, and evaluate the effectiveness of our new global resource management scheme with respect to IO hit ratio and IO cost. Experimental results show that our GREM well utilizes the benefits of SSDs with the improvement of IO hit ratios across different IO workloads. By dynamically partitioning SSDs into two functional zones, D_GREM further increases IO hit ratio, especially for those VMs with cache-unfriendly workloads. Meanwhile, both GREM and D_GREM are able to save the operational cost of SSDs by up to 85%.

The remainders of this paper is organized as follows. Section II presents our understanding of IO access patterns in real production systems, and discusses the limitations of existing SSD resource allocation solutions. Section III introduces the details of our algorithms that can dynamically assign SSD resources to multiple VMs, considering both performance isolation and workload changes. Section IV evaluates the effectiveness of our algorithms. Section V describes the related work. Finally, we summarize our findings and discuss the future work in Section VI.

II. MOTIVATION

In a virtualized storage system, SSDs are commonly shared by multiple VMs with heterogeneous IO workloads and caching requirements. An effective resource management scheme should absorb hot data in SSDs, ensure good performance isolation across VMs, and maintain high resource utilization of SSDs. To achieve this goal, we need to thoroughly understand IO access patterns of heterogeneous VM workloads and dynamically allocate SSDs among these VMs according to IO workload changes. Therefore, in this section, we present our analysis of IO access patterns in real production systems, and discuss the limitations of two straightforward approaches.

A. Understanding IO Access Patterns

We studied a suite of real IO traces to analyze and understand IO access patterns in enterprise production systems.

[MSR Cambridge] One week IO block traces collected by MSR Cambridge in 2007 [1]. In these IO traces, each data entry describes an IO request, including timestamp, disk number, logical block number (LBN), number of blocks and the type of IO (i.e., read or write). There are 36 traces from MSR-Cambridge, which includes a variety of workloads.

[FIU] Two sets of IO block traces collected by Florida International University (FIU) [2]. *FIU IODedup* contains collected downstream of an active page cache for three weeks in 2008. *FIU SRCMap* covers IO accesses from an email server, a virtual machine monitor running two web servers, and a file server workload during 2008-2009.

[UMASS] Two financial IO traces (*Fin1* and *Fin2*) from OLTP applications running at large financial institutions and three IO traces (*WebSch1*, *WebSch2* and *WebSch3*) from a web search engine [3].

Table I shows some statistical results of selected IO traces, including:

- *Hit Ratio:* the percentage of IOs that are hit in SSDs under the LRU algorithm with a fully associative cache of $4KB$ cache line and $1GB$ cache size.
- *Working Volume (WV) Size:* the total amount of data (in bytes) accessed in the disk.
- *Working Set (WS) Size:* the total address range (in bytes) of accessed data, which is the unique set of WV. A large working set covers more disk space. If the cache size is larger than or equal to a workload's WS, then the IO hit ratio of this workload can be close or equal to 100% under the LRU caching algorithm [4].
- *Sequential Ratio (Seq):* the amount (in bytes) of total sequential IOs (both read and write) divided by the total IO amount (in bytes). In general, SSDs have better performance under sequential IOs than under random IOs.
- *Write IO Ratio (Wr):* the number of write IOs divided by the total number of IOs.
- *Peak Throughput (IOPS):* the peak throughput of the IO workload (with the sampling window of 5 min).

As shown in Table I, high variance can be found in IO hit ratios across different IO workloads. For example, the IO hit ratio is more than 90% under the `MSR-hm0` workload while the IO hit ratio under the `MSR-src21` workload is less than 3%. We thus coarsely classify these IO workloads into two categories: (1) *cache-friendly* workloads (e.g., `MSR-hm0`, `UMASS-Fin1`) always obtain high IO hit ratios, while (2) *cache-unfriendly* workloads (e.g., `MSR-web2`, `FIU-hm2i1`) have relatively low IO hit ratios. We observe that the working set (i.e., the unique data blocks) in cache-friendly workloads (denoted as "F" in Table I) is usually small, which indicates high spatial locality (i.e., high reuse ratio, defined as $\frac{WV}{WS}$), and thus is highly likely to be cached and hit in SSDs. In contrast, cache-unfriendly workloads (denoted as "U" in Table I) often have large working volume sizes (see the WV column in Table I) and working set sizes (see the WS column in Table I). This observation motivates that we should differentiate these two classes of workloads by reserving a particular amount of SSD resources for VMs that have cache-friendly workloads to hold their popular data blocks. Moreover, the reserved SSDs

TABLE I. STATISTICS FOR SELECTED MSR-CAMBRIDGE, FIU AND UMASS WORKLOADS.

Group	Trace Name	Hit (%)	WV (GB)	WS (GB)	Seq (%)	Wr (%)	$Peak$ $IOPS$
MSR-F1	mds0	90.84	67.83	6.43	32.50	88.11	207.02
	stg0	89.28	21.63	13.21	42.43	84.81	187.01
	usr0	88.25	31.81	7.49	64.38	59.58	138.28
	src12	85.63	16.00	5.14	42.45	74.63	143.51
MSR-F2	hm0	91.34	27.88	9.03	33.76	64.50	271.65
	prn0	85.16	132.65	32.74	38.71	89.21	254.55
	web0	77.14	67.82	14.91	40.37	70.12	249.67
	web1	54.25	135.66	8.68	84.57	45.89	146.44
MSR-U	stg1	34.59	203.47	162.03	85.64	36.25	197.75
	usr2	19.48	1060.78	763.12	77.64	18.87	584.50
	web2	6.2	339.16	152.65	85.43	85.43	0.75
	src21	2.82	339.15	41.63	89.50	2.14	303.64
FIU-F1	wbusr1	83.78	15.22	0.32	62.29	99.97	1.83
	wbrsh5	74.92	15.75	0.41	73.23	100.00	31.64
	wbmal4	70.8	36.48	5.87	48.28	85.20	136.25
	wbmal5	70.18	36.35	4.39	49.89	87.26	156.63
FIU-F2	hm4t3	83.38	283.40	1.65	87.22	99.91	467.58
	hm4t1	76.97	264.81	1.67	41.95	92.66	118.98
	wbusr3	75.98	15.75	3.55	69.11	87.48	129.10
	mal1c1	72.04	557.74	36.27	94.96	88.35	488.93
FIU-U	hm2i1	41.41	258.32	15.68	61.77	76.31	284.06
	hm2i2	28.04	391.50	5.27	74.19	91.45	371.74
	hm3m3	26.79	37.25	0.48	82.65	99.17	39.36
	hm3m2	17.23	36.49	0.04	80.27	99.85	1.88
Umass-F	Fin1	99.07	1289.06	1.08	32.52	76.84	218.59
	Fin2	98.51	1.16	1.11	11.79	17.65	159.94
Umass-U	WebSch1	6.08	33.35	18.37	1.78	0.02	355.38
	WebSch2	6.3	33.35	18.98	3.71	0.02	375.02
	WebSch3	6.15	33.35	19.21	14.62	0.03	245.09

do not need to be too large to guarantee high hit ratios of VMs with cache-friendly workloads since their working set sizes are usually small.

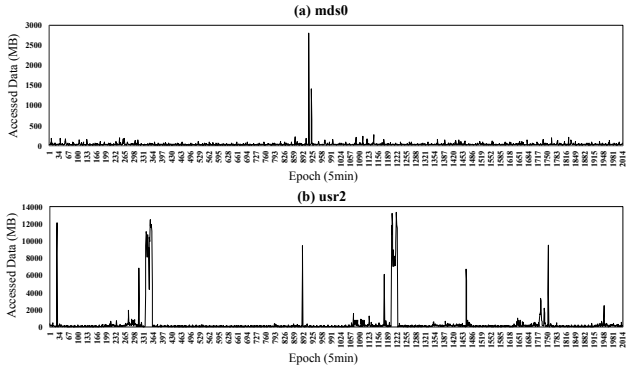

Fig. 1. Examples of bursty IOs (e.g., runtime working set sizes) under (a) cache-friendly workload `mds0` and (b) cache-unfriendly workload `usr2` .

Fig. 1 further shows the runtime working set sizes (WS) of two MSR workloads during every 5min epoch. We observe that cache-unfriendly workloads (see plot (b) in Fig. 1) have more IO spikes (i.e., a large amount of unique data blocks accessed during a short period of time) than cache-friendly workloads (see plot (a) in Fig. 1). These IO spikes in cache-unfriendly workloads are much more frequent, which can dramatically degrade IO hit ratios due to the first-time cache miss and even worse pollute the critical data in SSDs. This observation implies that VMs with cache-unfriendly workloads need to be assigned with a large amount of SSDs during their bursty periods to absorb and handle their bursty IOs for improving their hit ratios. However, to avoid severe cache pollution, allocated SSDs should not overlap the reserved SSDs for cache-friendly workloads. Therefore, a new SSD resource management scheme is needed to discriminate cache-friendly and cache-unfriendly workloads and improve IO performance for both types of workloads.

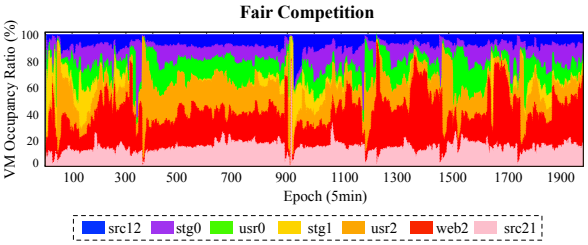

Fig. 2. Runtime SSD allocation among VMs under *fair competition*.

B. Limitations of Straightforward Approaches

Two straightforward approaches can be used to allocate SSD resources among multiple VMs. The first approach (referred to as "*performance isolation*") is to proportionally reserve SSD resources for each VM in the system such that all VMs are purely isolated in using their own assigned SSD resources. Different cache replacement algorithms (e.g., LRU [5], CAR [6], DellFluid [7], Mercuy [8], and SCE [9]) can be used by each VM to cache their recently accessed data blocks and the caching management is fully affected by their own workload changes. In contrast, the second approach [10], [11] (referred to as "*fair competition*") manages SSD resources in a fair competition mode by allowing all VMs to freely use or share the entire SSDs. A caching algorithm is usually used to centrally decide which data blocks should be held in SSDs for all VMs. Consequently, the caching management is inevitably interfered by the intensity of all workload changes. However, we found that neither of these approaches can fully utilize the benefits of SSDs when some of VMs have bursty IO workloads during runtime. To understand the limitations of these two approaches, we run trace-driven simulations by replaying a mix of real IO workloads and investigate the assignment of SSD resources to each VM (or each workload) across time duration about one week. We find that although the first approach (i.e., *performance isolation*) is able to avoid performance interference, VMs with bursty IOs unfortunately have no chance to obtain more SSD resources during their bursty periods. Each VM keeps the fixed amount of SSDs during their runtime. On the other hand, the second approach solves this issue by allowing all VMs to compete SSDs based on their present IO demands. As shown in Fig. 2, VMs (e.g., cache-unfriendly workloads `web2` and `usr2`) occupy more SSD resources when there are IO spikes in their workload. Thus, their IO hit ratios are improved and the overall utilization of SSD resources is increased as well. However, we notice that under this approach, VMs with bursty IOs might occupy a large amount of the SSD resources (e.g., `web2` at epoch 900 in Fig. 2) during their bursty periods by evicting other cached data, which might pollute critical caching of VMs with cache-friendly workloads and then degrade their IO hit ratios.

III. DESIGN AND ALGORITHMS

A. Basic Idea of GREM

As observed in Section II, VMs with cache-friendly workloads can usually achieve high hit ratios when only a small amount of their critical data blocks (i.e., their working sets) are cached in SSDs. However, VMs with cache-unfriendly workloads often have spikes (i.e., a large amount of unique data blocks) of IOs across time, which can incur a significant amount of cache misses and further pollute the critical caching

of other VMs. In order to ensure all VMs benefit from SSDs, we design a new resource management scheme, named GREM, which strives to discriminate different workload types (e.g., cache-friendly and cache-unfriendly workloads) by splitting SSDs into two zones (denoted as "Z_L" and "Z_S" see Fig. 3) such that one zone is designed for reserving SSD resources for each VM and the other zone is used to absorb and handle bursty IOs. GREM manages SSD resources underlying the VM page buffer layer, thus it does not need to (reversely) interact with the VM page buffers.

In particular, we refer "Z_L" to as a "*Long-term Zone*", which is expected to cache the most popular data blocks for each VM based on their IO access frequency during a long period. Furthermore, SSD resources in "Z_L" are reserved for each VM such that their critical and popular data blocks can be kept in SSDs without any pollution from other VMs, which thus guarantees a high hit ratio from VMs with cache-friendly workloads. We refer "Z_S" to as a "*Short-term Zone*", where SSD resources are fairly competed among VMs based on the popularities of their recently accessed data during a short period. Consequently, VMs with cache-unfriendly workloads can have a high chance to get SSD resources in Z_S to cache data for their bursty IOs and achieve an improved IO hit ratio. Given the total capacity of SSD resources C_T, we have

$$C_T \quad = \quad C_{Z_L} + C_{Z_S}, \tag{1}$$

where C_{Z_L} and C_{Z_S} denote the capacities of Z_L and Z_S, respectively. Furthermore, given m VMs running in the system, each VM i will be assigned with $C_{Z_L}(i)$ SSD resources in Z_L such that $\sum_{i=1}^{m} C_{Z_L}(i) = C_{Z_L}$. Therefore, the design goal of GREM is to determine the proper values of C_{Z_L}, C_{Z_S}, and $C_{Z_L}(i)$ in order to maximize the overall IO hit ratio and minimize the IO cost for operating IO access and updating SSD content.

B. Dynamically Partition of the Long-term Zone

As introduced in Sec. III-A, GREM attempts to reserve SSD resources in Z_L for each VM in order to ensure each VM have their own private SSDs to cache their critical hot data. One straightforward approach is to partition zone Z_L among VMs equally or proportionally, i.e., $C_{Z_L}(i) = C_{Z_L} \cdot w_i$, where w_i is a fixed weight based on the Service-Level Agreement (SLA) for VM i and $\sum_{i=1}^{m} w_i = 1$, and reserve a fixed amount (i.e., $C_{Z_L}(i)$) of SSDs in Z_L for VM i. However, we found that this approach is ineffective when workloads frequently change and spikes of IOs occur across time. Reserved SSD resources cannot be fully utilized when some VMs start to have a low IO access rate and meanwhile other VMs may not be able to obtain sufficient SSDs when they experience bursty IOs that need to access a large amount of unique data blocks (i.e., large working sets). Therefore, we develop a dynamic partitioning algorithm for GREM to dynamically decide the capacity (i.e., $C_{Z_L}(i)$) for each VM's reserved SSDs in Z_L based on not only each VM's access history in a long term but also their IO workload changes. Here, we assume that the capacities of two zones are fixed, e.g., $C_{Z_L} = C_{Z_S} = \frac{C_T}{2}$. Later, we present a new version of GREM that adjusts the zone sizes in an online mode.

[Dynamic Z_L Partition and Cache Replacement Solution] Alg. 1 describes the procedure of GREM, including how GREM periodically updates the content in both Z_L and Z_S, and how GREM online adjusts the amount of reserved SSDs in Z_L for each VM. $historyBin$ is a dictionary in which key is bin IDs of all VMs and value is the relative access count for each bin. $currEpochBin$ is the accessed bins of all VMs in current epoch. $shortBin$ and $longBin$ are the cached bins of all VMs in Z_S and in Z_L, respectively. $flashBin$ is those bins need to be cached in Flash. Fig. 4 further shows the procedures of GREM for cache admission and cache eviction. The key idea of GREM is that the amount of reserved SSDs in Z_L for each VM should be proportional to their long-term access behaviors. When the distribution of data bin popularities changes, GREM dynamically adjusts the reservation of SSD resources for each VM in order to fully utilize the Z_L zone. In detail, GREM maintains a long-term IO access history for all running VMs (i.e., "$historyBin$" in Fig. 4) to record the accumulative IO popularity statistics for their bins (e.g., each bin size is $1MB$). This historical information can be saved in the RAM. Similarly, these long-

Algorithm 1: Dynamic Partition in Z_L

```
1  Procedure GREM()
2      UpdateLongTermZone();
3      UpdateShortTermZone();
4      flashBin = shortBin + longBin;
5      return flashBin;
6  Procedure UpdateLongTermZone()
7      if size(historyBin) ≤ size(longBin) then
8          longBin = historyBin.keys;
9      else
10         /* the max number of bins to be cached in Z_L */
             j = size(longBin);
11         /* the top j popular bins */
12         itemH = number of j bins in historyBin.keys
             with highest historyBin.values;
13         /* the evicted bins due to newly cached bins */
             evictBin = bins of longBin which are not in
             itemH;
14         longBin = itemH;
15     return;
16 Procedure UpdateShortTermZone()
17     if size(longBin) < size(historyBin) ≤ C_T then
18         /* Case 1: warming up period */
19         shortBin = the remaining bins of
             historyBin.keys which are not in longBin;
20     else if size(historyBin) > size(flashBin) then
21         /* Case 2: historyBin, evictBin, currEpochBin
             and existing bins in shortBin compete in the Z_S */
22         shortBin −= bins of shortBin which are also in
             longBin;
23         currEpochBin −= bins of currEpochBin which
             are also in longBin;
24         if size(currEpochBin) ≥ size(shortBin) then
25             j = size(shortBin);
26             shortBin = number of j bins in
                 currEpochBin with highest IO popularity;
27         else
28             shortBin += evictBin;
29             shortBin −= bins of shortBin which are also
                 in currEpochBin;
30             j = size(shortBin) − size(currEpochBin);
31             shortBin = number of j bins in shortBin
                 with highest IO popularity;
32             shortBin += currEpochBin;
33     return;
```

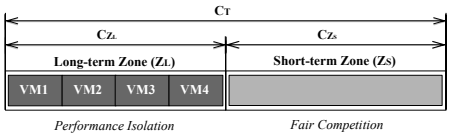

Fig. 3. Basic structure of GREM.

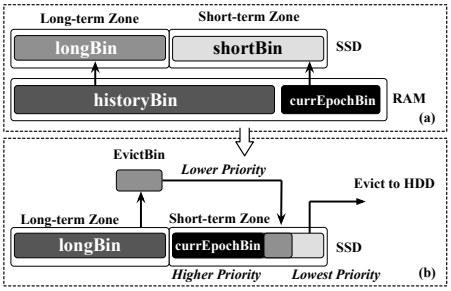

Fig. 4. High level idea of GREM: (a) cache admission and (b) cache eviction.

term IO access records are incrementally updated every time epoch (e.g., every 5 min). An aging function is also used to capture the variation of IO popularities over time. Recent IO popularities (i.e., collecting IO activities in recent time epochs) are assigned with higher weights for contributing more to the accumulative IO popularity statistics. Once the IO popularity statistics are updated, GREM selects the most popular bins with their overall size is equal to C_{Z_L}, and then sets the amount of reserved SSDs for VM i, i.e., $C_{Z_L}(i)$, to the total size of its popular bins that have been selected and cached in Z_L (see lines 6-15 in Alg. 1 and Fig. 4(a)). By this way, GREM also updates the content of SSDs in Z_L by caching the most popular bins with total size of C_{Z_L}. On the other hand, GREM attempts to leverage the recent data access information (e.g., the current epoch) to update the short-term zone Z_S (see lines 16-33 in Alg. 1 and Fig. 4(b)). In particular, GREM also records the IO popularity statistics for all bins that were accessed in the last epoch (i.e., "$lastEpochBin$" in Fig. 4) in the RAM. The most popular bins that have not been cached in Z_L are then cached in Z_S. The total size of these cached bins is bounded by C_{Z_S}. We notice that it is possible that all accessed bins in the "$lastEpochBin$" might have the total size less than C_{Z_S}. In such a case, GREM further considers to cache the bins that are just evicted from Z_L.

C. Dynamical Partition for Z_L and Z_S: D_GREM

GREM statically and equally partitions SSDs into two zones, i.e., $C_{Z_L} = C_{Z_S} = \frac{C_T}{2}$. However, we find that such a partitioning unfortunately may not be optimal to general cases. For example, if workloads have a large number of bins being popular only during a short period, then Z_S may not be large enough to handle bursty IOs that access those bins. This can cause a very low IO hit ratio and increase the operational costs for caching new bins in Z_S. To solve this problem, we design a bursty-detection based partition algorithm that allows GREM to dynamically adjust the sizes of Z_L and Z_S. We refer this new version of GREM to as D_GREM. Fig. 5(a) depicts the high level sketch of D_GREM consisting of two main components, i.e., *bursty detector* and *strategy switcher*. The *bursty detector* takes the feedbacks of workload changes and cache performance (e.g., IO hit or IO miss) as the input to determine if the current workload is bursty or non-bursty. Based on the detected result, the *strategy switcher* will make different partitioning decisions for improving the SSD resource utilization.

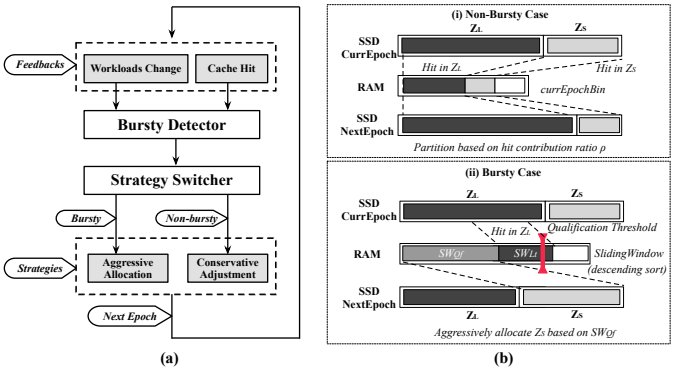

Fig. 5. (a) The high level sketch of D_GREM. (b) Dynamic partition of Z_L and Z_S of D_GREM.

1) Bursty Detection: Basically, bursts or spikes contain a relatively high number of IO accesses on a large amount of working set data. They often occur within a short time period. The bursty detector is expected to help D_GREM decide (1) when to adjust the partition of SSDs, and (2) how to divide SSD resources between two zones. We find that the capacity adjustment of two zones is needed when the number of popular bins or the working set size significantly changes. Intuitively, the new capacity of each zone should be related with the number of popular bins in the current spike, the bin reuse rates, the bin access history during previous epochs, and the hit counts in the Z_L and Z_S. Therefore, D_GREM uses a sliding window (SW), e.g., 5 min, to record IO access history for all VMs in recent epochs. Again, this historical information can be kept in the RAM. D_GREM tracks the changes in the working set sizes between the current (i.e., $|WS_{curSW}|$) and previous (i.e., $|WS_{prevSW}|$) sliding windows. The relative difference between these two sliding windows is then defined as bursty degree (denoted as B_d) as follows.

$$B_d = \Delta(WS_{curSW}, WS_{prevSW}) = \frac{||WS_{curSW}| - |WS_{prevSW}||}{|WS_{curSW}|}. \quad (2)$$

The value of B_d is a good indicator of working set changes across time. The bursty detector claims the arrival of bursty IOs when the value of B_d is beyond a predefined threshold β. We set β to 0.6 in our experiments.

2) Non-Bursty Case Strategy: When there is no burstiness in current IO workloads, i.e., $B_d < \beta$, D_GREM tunes the splitting point between two zones (i.e., Z_L and Z_S) by leveraging the feedback of each zone's caching performance under the present partition (as shown in Fig. 5(b)(i)). In particular, we evaluate the importance of two zones (i.e., their contributions to the overall IO performance) by recording the total IO hit volumes (i.e., the amount of all cached data that are hit by one or multiple IOs) in each zone during the recent epoch (e.g., 5 min). We define a contribution ratio ρ as follows:

$$\rho = \alpha \times \frac{HV_L}{HV_S}, \quad (3)$$

where $\alpha \in [0, 1]$ is a tunable parameter of importance, and HV_L and HV_S represent the IO hit volumes of Z_L and Z_S, respectively. By default, $\alpha = 0.35$. Once ρ is updated at the end of an epoch, D_GREM adjusts the capacities of two zones for next epoch using Eq.(4).

$$\begin{cases} C_{Z_S} = \frac{C_T}{1+\rho}, \\ C_{Z_L} = C_T - C_{Z_S}, \end{cases} \tag{4}$$

where C_T is the capacity of SSDs and C_{Z_S} and C_{Z_L} are the new anticipated capacities of Z_L and Z_S, respectively. Intuitively, the zone that contributes more to the overall hit ratio is highly likely to get more SSD resources and the allocation of SSDs is proportional to the contribution ratio.

3) Bursty Case Strategy: When bursty IOs are identified by the bursty detector, D_GREM turns to aggressively shift SSD resources from one zone to the other. Using the contribution ratio as a feedback to reset the capacities of two zones unfortunately does not work well under this case because this approach cannot quickly adapt to workload changes. The corresponding delay can further incur "cascade effect" (also called "thrashing effect") of insufficient capacity in one of the zones. For example, when bursty IOs that access new bins arrive, the IO hit volume of Z_S in the current epoch might not be large enough to get more SSDs for handling those bursty IOs. Consequently, this zone's IO hit volume becomes even less in the next epoch, which indicates less importance and then keeps reducing the capacity of Z_S if we use Eq.(4). To avoid such a cascade effect, we attempt to dynamically and aggressively assign more SSDs to Z_S when bursty IOs are found in workloads, and further to minimize the penalty on Z_L's caching performance. The general idea of our approach is that if the working set size of accessed bins in the current epoch increases dramatically compared with the previous epoch, then it would also be helpful if we increase the size of Z_S for absorbing the spikes in the near future. Meanwhile, we should ensure those popular bins, which are cached in Z_L and are recently hit, to keep staying in Z_L, as shown in Fig. 5(b)(ii). Therefore, we use the sliding window (SW) to record IO popularity statistics for all bins that are accessed in recent several epochs (instead of the latest one). D_GREM identifies all bins that are recorded in the current sliding window and are also cached in the Z_L zone. We refer this set of bins to as "B_{SWLt}". D_GREM uses the average number of accesses ($\overline{B_{SWLt}}$) of all bins in B_{SWLt} as a criterion to set the threshold (Th_{SW}) for choosing hot bins to be cached in Z_S as follows.

$$Th_{SW} = \gamma \times \overline{B_{SWLt}}, \tag{5}$$

where γ is an adjustment parameter, and is set to 1.2 by default. D_GREM then finds all the bins that have been accessed in the current sliding window more than Th_{SW} times but are not currently cached in Z_L. We refer this set of "qualified" bins to as "B_{SWQf}", as:

$$B_{SWQf} = \{x | x \in SW, x \notin B_{SWLt}, x > Th_{SW}\}. \tag{6}$$

We then set the anticipated capacity of Z_S to the total size of bins in B_{SWQf}.

$$C_{Z_S} = |B_{SWQf}|. \tag{7}$$

4) Dynamic Tuning Procedure: As shown in Fig. 5(a), initially, capacities of both two zones are set to half of the entire SSDs. D_GREM recalculates B_d, the present bursty degree at the end of each sliding window and determines if bursty IOs are arriving by comparing with the threshold β. Under the bursty case, D_GREM aggressively enlarges the capacity of Z_S according to Eq.(7). On the other hand, D_GREM bases on the contribution ratio of two zones to adjust the allocation of SSDs to these zones when there is no burstiness, as shown in lines. A boundary checking is further considered to ensure the minimum capacity for each zone.

IV. EVALUATION

In this section, we conduct trace-driven evaluation by replaying real enterprise IO workloads. We implement two versions of our proposed algorithm: (1) GREM that assigns 50% of the total SSDs to each of two zones, and adaptively adjusts partitions of each VM in Z_L according to the workload change; and (2) D_GREM that further dynamically adjusts the sizes of Z_L and Z_S during runtime. For comparison, we also implement conventional caching algorithms, such as global LRU (GLRU) [5] and global CAR [6], and a recently proposed tiering algorithm VFRM [10]. We evaluate the effectiveness of GREM with respect to our primary goals, i.e., maximizing the IO hit ratio and minimizing the IO cost incurred in managing SSDs.

A. Testbed and Implementation Details

Our evaluation environment is calibrated based on the real testbed specs summarized in TableII. In detail, to deal with the management across physical nodes and multiple tiers, we adopt the same hybrid file developed in our previous work [10], which consists of two "files": a *base file* on the spinning disk tier and a corresponding *peer file* on the Flash tier. Furthermore, we use the following three key techniques to lower the overhead: (1) a "*Pointer Block Cache*" of the peer file is used as the block look up table, which can eliminate the need for an extra lookup table; (2) a "*Heat Map*" is used to represent the IO popularity statistics of each "1MB block" of the files on VMFS, and 1MB block only requires 16 bytes to hold the popularity stats, which is only 0.0015% of the size of the VMDK; and (3) a "*Tiering Map*" is used to represent the placement of the blocks between tiers, whose space overhead is about 0.00001% of the size of the VMDK. Moreover, both the heat and tiering maps do not need to be pinned in memory permanently. Lastly, based our observation in Sec. II and our offline sensitivity analysis results, we set 1MB and 5min as spatial and temporal granularities, respectively, to avoid adverse impact from spikes as well as to reduce the caching management overhead.

TABLE II. TESTBED SPECS.

Component	Specs
Processor	Intel Xeon E5-2430 2.20GHz
Processor Cores	6 Cores
Memory Capacity	48GB ECC DDR3 R-DIMMs
Operating system	Ubuntu 12.04.5
Linux Kernel	3.14 Mainline
SSD Model	Intel DC S3500 Series
SSD Capacity	80 GB
HDD Model	Western Digital 20EURS-63S48Y0 (5400 RPM)
HDD Capacity	2 TB

B. IO Hit Ratio

Fig. 6 shows the overall (i.e., read & write) IO hit ratios as a function of SSD cache size (in GB) under diverse workloads that are mixed with different MSR, FIU and UMASS repositories. For example, "FIU-F1U" is a mixed workload with the IO traces of "FIU-F1" and "FIU-U", where "F" and "U" refer to cache-friendly and cache-unfriendly workloads, as summarized in Table I. In overall, we can see that GREM and D_GREM are superior to other existing caching algorithms. For instance, Fig. 6(a) presents the results under MSR-F1, a cache friendly workload. When the cache size is smaller than $2GB$, VFRM, GREM and D_GREM all have lower IO hit ratios than the conventional algorithms. However, as the

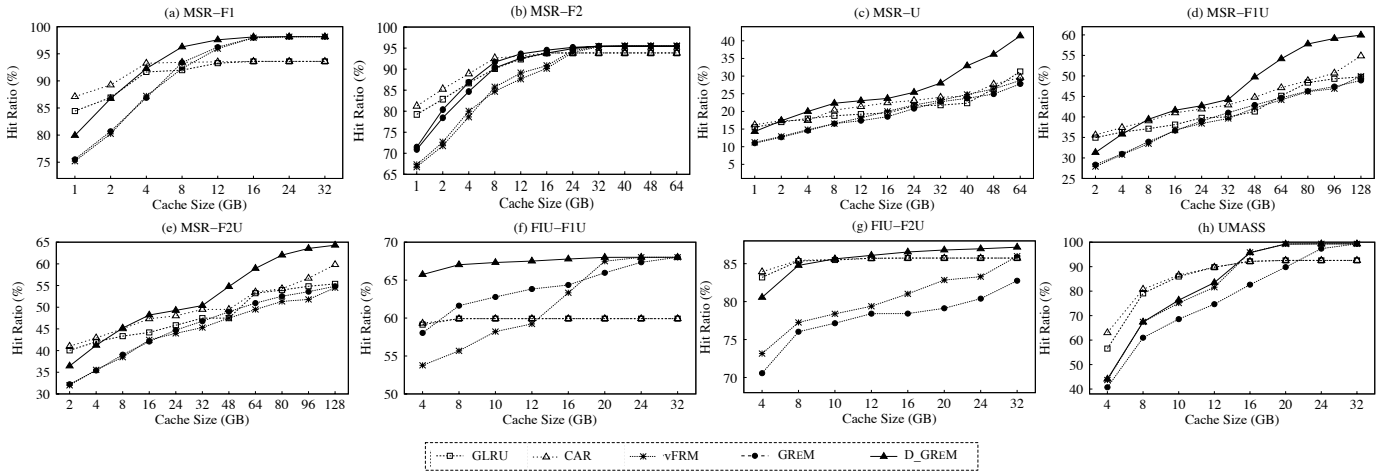

Fig. 6. IO hit ratio results of workload combinations of MSR, FIU and UMASS repositories under different cache sizes and caching algorithms.

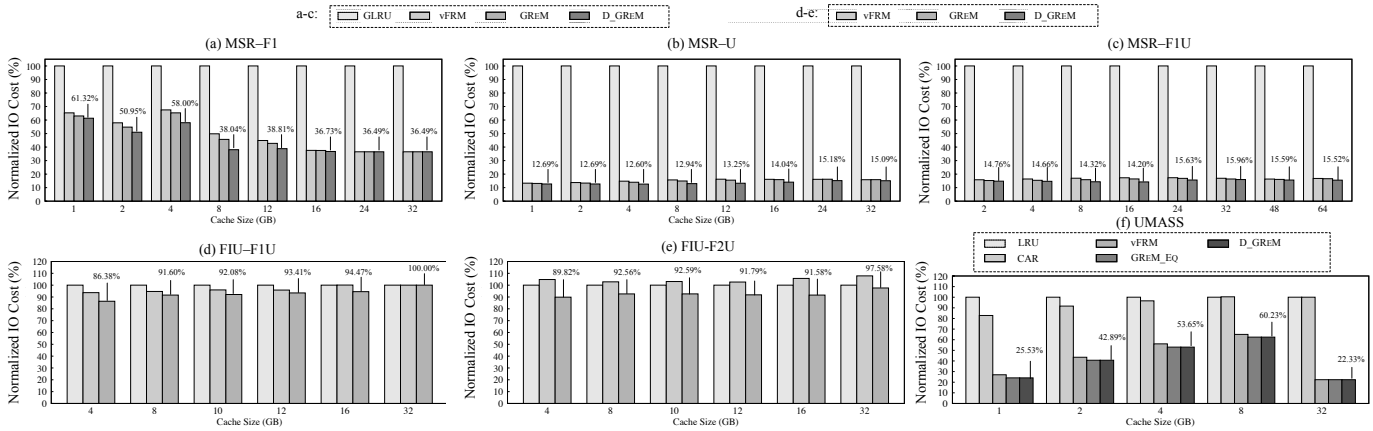

Fig. 7. Normalized IO cost of MSR, FIU and UMASS workloads, which is total latencies of read/write operations of SSD and HDD.

capacity of SSDs increases to $4GB$, D_GREM outperforms GLRU and CAR. When we have more than $8GB$ SSDs, the IO hit ratios under both GREM and D_GREM are beyond those of GLRU and CAR. More importantly, the conventional algorithms cannot take advantage of a large SSD cache. Their IO hit ratios reach to the converging point (i.e., about 94%) when the cache size is 4GB. In contrast, GREM and D_GREM can further use the benefits of the increasing SSD capacity to improve IO hit ratios up to 98%. Moreover, due to large working set sizes and bursty IOs, most caching algorithms, including GREM, cannot achieve high IO hit ratios for cache-unfriendly workloads (e.g., `MSR-U` in Fig. 6(c)) even when we increase the SSD capacity. By dynamically adjusting the size of the short-term zone (Z_S) to absorb bursty IOs, D_GREM keeps improving IO hit ratio up to 40% when we have 64GB SSDs. We further notice that D_GREM still do not converge even with 64GB SSDs,, which indicates that this algorithm is able to further achieve better IO hit ratios.

C. IO Cost

Fig. 7 shows the normalized overall IO costs (SSD/HDD access and SSD-HDD contents updating latencies [11]) based on the measured data from an $80GB$ `Intel DC S3500 Series` SSD and a $2TB$ 5400 RPM `Western Digital WD20EURS-63S48Y0` HDD, e.g., IO latencies of SSD/HDD read and write operations under $4KB$ and $128KB$ cache lines. In our evaluation, the conventional caching algorithms (i.e., GLRU and CAR) use $4KB$ as the cache line size, while other algorithms (i.e., VFRM, GREM and D_GREM) use the cache line size of $128KB$ and group IOs into $1MB$ bins. We observe that by using the coarse-update granularity, all these algorithms (e.g., VFRM and D_GREM) significantly reduce the overall IO costs compared to the conventional caching solutions, especially when there are cache-unfriendly workloads (e.g., `MSR-U` and `MSR-F1U`). Furthermore, D_GREM always achieves the lowest cost in all these cases. For example, as shown in Fig. 7(a), with $8GB$ cache size, the overall IO costs of `MSR` with four cache-friendly workloads under D_GREM are 61.96% lower than GLRU, 11.77% lower than VFRM and 7.62% lower than GREM. Similar observations can be found under cache-unfriendly workloads, see Fig. 7(b).

In Fig. 7(c)-(d), we further investigate the IO costs under different algorithms when we have workloads mixed with both cache-friendly and cache-unfriendly workloads. Again, by dynamically adjusting the sizes of Z_L and Z_S and updating SSD content in the coarse granularity, D_GREM significantly reduces cache pollution due to IO spikes, and avoids too frequent SSD content updates, which thus achieves much lower IO costs compared to the conventional caching algorithms, e.g., GLRU, CAR. Both GREM and D_GREM further slightly reduce the IO costs compared to VFRM, which also adopts the coarse granularity for SSD content updating, but cannot avoid bursty IOs evicting the cached critical data.

V. Related Work

Host-side caches are being widely accepted in modern storage systems. ARC [12] and LRFU [13] are commonly used caching algorithms that consider the frequency and recency of workloads. Inspired by ARC and based on Clock [14], CAR and CART [6] are developed to inherit virtually all advantages of ARC, but not serialize cache hits behind a single global lock. Studies [15], [16] investigated how to model the IO bandwidth performance, and designed a NUMA-aware cache mechanism to align cache memory with local NUMA nodes and threads. mClock [17] follows the proportional-share fairness approach which subjects to minimum reservations and maximum limits on the IO allocations for VMs. VirtualFence [18] is a storage system that provides predictable VM performance and conducts space-partitioning of both the SSD cache and the HDD. Studies [19]–[22] investigated SSD and NVMe storage-related resource management problems, such as how to reduce the total cost of ownership and how to increase the Flash device utilization. A hypervisor-based design "S-CAVE", was presented in [23]. Its optimization is based on runtime working set identification, while GREM explores a different dimension by monitoring changes in data locality, burstiness and IO popularity. vCacheShare [24] presented a dynamic, self-adaptive framework for automated server flash cache space allocation in virtualization environments. However, it only treats SSD as a read-only cache and bypasses write IOs to the disk, which unfortunately degrades the overall hit ratio. Recently, [11] presented a new VMware Flash Resource Manager, named vFRM, which considers of both performance and incurred cost for managing Flash resources, and updates the content of SSDs in a lazy and asynchronous mode.

VI. Conclusion

In this paper, we presented GREM, a new global SSD resource management scheme to allocate a suitable amount of SSDs to heterogeneous VMs. The design goal is to best utilize the SSD resources by maximizing the IO hit ratio and minimizing the IO costs. GREM splits the entire SSD space into the long-term and short-term zones and takes dynamic IO demands of all VMs into consideration for reserving SSD resources in the long-term zones to each VM. We further developed D_GREM to dynamically adjust the partition of SSDs between two zones by leveraging the feedback of workload changes and SSD performance. We show that our new schemes allow VMs with different types of workloads to utilize the benefits of SSDs and thus improve the overall IO hit ratio. We also show that D_GREM successfully detects the changes (or bursts) in IO workloads and quickly adapts to the changes by shifting SSD resources between two zones. The IO hit ratio is further improved under D_GREM.

References

[1] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Transactions on Storage*, vol. 4, no. 3, pp. 10:1–10:23, 2008.

[2] R. Koller and R. Rangaswami, "I/O deduplication: Utilizing content similarity to improve i/o performance," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, p. 13, 2010.

[3] "UMass Trace Repository," http://traces.cs.umass.edu/index.php/Storage/Storage.

[4] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *ACM SIGPLAN Notices*, vol. 38, no. 5. ACM, 2003, pp. 245–257.

[5] E. O'Neil, P. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, Washington, DC, 1993, pp. 297–306.

[6] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement." in *Proceedings of the 2th USENIX Conference on File and Storage Technologies*, vol. 4, 2004, pp. 187–200.

[7] "Dell fluid cache for storage area networks," http://www.dell.com/learn/us/en/04/solutions/fluid-cache-san.

[8] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer, "Mercury: Host-side flash caching for the data center," in *IEEE 28th Symposium on Mass Storage Systems and Technologies*, Pacific Grove, CA, 2012, pp. 1–12.

[9] H. Kim, I. Koltsidas, N. Ioannou, S. Seshadri, P. Muench, C. L. Dickey, and L. Chiu, "Flash-conscious cache population for enterprise database workloads." in *ADMS@ VLDB*, 2014, pp. 45–56.

[10] D. Liu, N. Mi, J. Tai, X. Zhu, and J. Lo, "VFRM: Flash resource manager in vmware esx server," in *Network Operations and Management Symposium (NOMS), 2014 IEEE*. IEEE, 2014, pp. 1–7.

[11] J. Tai, D. Liu, Z. Yang, X. Zhu, J. Lo, and N. Mi, "Improving flash resource utilization at minimal management cost in virtualized flash-based storage systems."

[12] N. Megiddo and D. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 2003, pp. 115–130.

[13] D. Lee, J. Choi, J.-H. Kim, S. Noh, S. L. Min, Y. Cho, and C. S. Kim, "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[14] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[15] T. Li, Y. Ren, D. Yu, S. Jin, and T. Robertazzi, "Characterization of input/output bandwidth performance models in numa architecture for data intensive applications," in *2013 42nd International Conference on Parallel Processing*. IEEE, 2013, pp. 369–378.

[16] Y. Ren, T. Li, D. Yu, S. Jin, and T. Robertazzi, "Design, implementation, and evaluation of a numa-aware cache for iscsi storage servers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 2, pp. 413–422, 2015.

[17] A. Gulati, A. Merchant, and P. J. Varman, "mclock: handling throughput variability for hypervisor io scheduling," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 437–450.

[18] C. Li, Í. Goiri, A. Bhattacharjee, R. Bianchini, and T. D. Nguyen, "Quantifying and improving i/o predictability in virtualized systems," in *Quality of Service (IWQoS), 2013 IEEE/ACM 21st International Symposium on*. IEEE, 2013, pp. 1–6.

[19] Z. Yang, M. Awasthi, M. Ghosh, and N. Mi, "A fresh perspective on total cost of ownership models for flash storage," in *8th IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 2016.

[20] Z. Yang and M. Awasthi, "Online flash resource migration, allocation, retire and replacement manager based on multiple workloads waf tco model," 2015, uS Patent, US15/094971.

[21] J. Roemer, M. Groman, Z. Yang, Y. Wang, C. C. Tan, and N. Mi, "Improving virtual machine migration via deduplication," in *2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE, 2014, pp. 702–707.

[22] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding Performance of I/O Intensive Containerized Applications for NVMe SSDs," in *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[23] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou, "S-CAVE: Effective ssd caching to improve virtual machine storage performance," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 103–112.

[24] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu, "vCacheShare: automated server flash cache space management in a virtualization environment," in *USENIX ATC*, 2014.