**Abstract**—In a shared virtualized storage system that runs heterogeneous VMs with diverse 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 solve this problem: either equally assigning SSDs to each VM or managing SSD resources in a fair competition mode. Unfortunately, they cannot fully utilize the benefits of SSD resources, particularly when the workloads frequently change and bursts or spikes of 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 “smart” 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 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 costs, compared with both traditional and recent caching algorithms.

**Keywords**—Solid State Drives, resource allocation, virtualized storage systems, caching algorithms, bursty workloads

**I. INTRODUCTION**

Virtualized systems nowadays become 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 requirement. It then becomes very important to provide high performance and availability to virtual machines. Flash-based Solid-State Drives (SSDs) are thus widely being deployed as a per-virtual disk, second-level cache in virtualized systems to improve IO access performance (e.g., increasing IO throughput and reducing IO latency) and also achieve low power consumption. In most of such 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 heterogeneous 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 heterogeneous 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 simply allocate SSD resources among VMs by either equally 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. Unfortunately, we found that none 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 thus 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 (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 cost-effectively updates the content of SSDs in these two zones. 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 contents of SSDs in 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 heterogeneous VMs (e.g., with a mix of cache-friendly and cache-unfriendly workloads) to share SSD resources and their IO workloads 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. 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 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 simulations 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%. We further investigate the online reservation of SSDs for each VM as well as the dynamic partitioning of SSDs between two zones. We observe that our GREm and D_GREm can quickly adapt to changes in IO workloads (such as bursty IOs) by allocating different amounts of SSDs to VMs.

The remaining of this paper is organized as follows. Section II presents our understanding of IO access patterns in real production systems and discuss the limitations of existing SSD resource allocation solutions. Section III introduces the details of our algorithm that can dynamically assign SSD resources to multiple VMs, considering both performance isolation and workload changes. Section IV evaluates 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 heterogeneous VMs with different IO workloads and caching requirements. An effective resource management solution should ensure good performance isolation across VMs and high resource utilization of SSDs by thoroughly understanding IO access patterns of heterogeneous VMs and dynamically allocating 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 then discuss the limitations of two straightforward approaches that either equally assign SSDs to each VM or manage SSDs in a fair competition mode.

A. Understanding IO Access Patterns

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

- [MSR Cambridge] One week block IO 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 in total, which includes a variety of workloads.


- [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 [4].

Tables I show the results of our analysis on the selected IO traces, including:

- **Hit Ratio**: the percentage of IOs that are hit in SSDs under the LRU caching algorithm with a fully associative cache with 4KB cache line and 1GB cache size.

- **Working Volume (WV) Size**: the total amount of data accessed in the disk.

- **Working Set (WS) Size**: the total address range of accessed data. A large working set WS covers more area in disk. WS is the unique set of WV.

In our evaluation, we use the following notation to represent the combination of different group of workloads. For example, “FIU-FIU” is a mixed workload with the IO traces of “FIU-FI” and “FIU-U”, where “F” and “U” refer to cache-friendly and cache-unfriendly workloads, respectively.

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 IO workloads into two categories: (1) cache-friendly workloads (e.g., MSR-hm0, FIU-wbusr1 and UMass-Fin1) always obtain high IO hit ratios, while (2) cache-unfriendly workloads (e.g., MSR-web2, FIU-hm21l and UMass-WebSch2) have relatively low hit ratios. We observe that the working set (i.e., the unique data blocks) in cache-friendly workloads is usually small, which has high spatial locality (i.e., high reuse ratio, defined as $WS/WV$) and thus is highly likely to be cached and hit in SSDs. In contrast, cache-unfriendly workloads often have large volume sizes and working set sizes. This observation motivates that we should reserve a particular amount of SSD resources for VMs...
that have cache-friendly workloads to hold their popular data blocks. Moreover, the reserved SSDs 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.

![Graph](image-url)

Fig. 1. Examples of bursty IOs (e.g., runtime working set sizes) under (a) cache-friendly workload mds0 and (b) cache-unfriendly workload src12.

Fig. 1 shows the runtime working set sizes (W/S) of two MSR workloads. We observe that cache-unfriendly workloads (see plot (b) Fig. 1) have more IO spikes (i.e., higher temporal locality) than cache-friendly workloads (see plot (a) 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 for absorbing and handling their bursty IOs and improving hit ratios. However, to avoid severe caching 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 two types of workloads.

### 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]) 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 (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.

![Graph](image-url)

Fig. 2. Runtime allocation ratio of SSDs among multiple VMs under the fair competition approach.

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 ran trace-driven simulations by replaying a mix of real IO workloads and plotted the assignment of SSD resources to each VM (or each workload) across time during about one week. Despite 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. Therefore, VMs (e.g., web2 and usr2) with spikes in their IO demands can occupy more SSD resources. As a result, their IO hit ratios are improved and the overall utilization of SSD resources is increased as well, see Fig. 2. However, we also notice that under this approach, VMs with bursty IOs might occupy a big partition of the SSD resources (e.g., web2 at 900 epoch in Fig. 2) during their bursty periods by evicting cached data, which might pollute critical caching of VMs with cache friendly workloads and then degrade their IO hit ratios.

### III. DESIGN AND ALGORITHMS

In this section, we present a new global SSD resource management scheme (GREM) to dynamically assign SSD
resources to multiple VMs under the consideration of both performance isolation and workload changes.

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 is cached in SSD. However, VMs with cache-unfriendly workloads often have spikes 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, 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) 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, as shown in Fig. 3. Given the total capacity C_T of SSD resources, we then have:

\[ C_{Z_L} + C_{Z_S} = C_T, \]

where \( C_{Z_L} \) and \( C_{Z_S} \) are the capacities of \( Z_L \) and \( Z_S \), respectively. 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 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 popularity 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.

**Fig. 3. Basic idea of GReM.**

1) Reserving SSDs in \( Z_L \): Given \( m \) active VMs, GReM divides the \( Z_L \) zone into \( m \) parts and then reserves \( C_i \) amount of SSDs for VM \( i \) (for \( 1 \leq i \leq m \)) during that VM’s runtime. Then, we have:

\[ \sum_{i=1}^{m} C_i = C_{Z_L}. \]

To determine \( C_i \), one approach is to equally or proportionally partition the zone, i.e., \( C_i = C_{Z_L} \frac{w_i}{\sum_{i=1}^{m} w_i} \), where \( w_i \) is the weight for VM \( i \) and \( \sum_{i=1}^{m} w_i = 1 \), and reserve a fixed amount (i.e., \( C_i \)) of SSDs for VM \( i \). We refer it as to “GReM_Eq” if all VMs have the same weights. However, we found that this static approach becomes ineffective when workloads frequently change and spikes of IOs occur across time. Reserved SSD resources cannot be fully utilized when some VMs have a low IO access rate and meanwhile other VMs may not be able to obtain sufficient SSDs when they experience bursty IO requests.

Therefore, GReM decides the capacity (i.e., \( C_i \)) for each VM’s reserved SSDs in \( Z_L \) dynamically based on not only each VM’s access history in a long term but also their IO workload changes. Algorithm 1 presents the high level idea of such an adjustment of \( C_i \). In detail, GReM maintains a long-term IO access history for all VMs (“historyBin”) to record accumulative IO popularity statistics for their bins (e.g., each bin size is 1MB). An aging function is also used to capture the variation of IO popularity with time passing. Recent IO popularities (i.e., collecting IO activities in recent time epochs) are assigned with higher weights for contributing more to accumulative IO popularity statistics. Once accumulative IO popularity statistics are updated, GReM selects the most popular bins such that their overall size is equal to \( C_{Z_L} \). GReM then sets the reserved amount (\( C_i \)) of SSDs for VM \( i \) to the total size of its bins that has been selected and caches them in \( Z_L \) (see lines 6-14 in Algorithm 1). When the distribution of bin popularities changes, GReM dynamically adjusts the reservation of SSD resources for each VM to fully utilize the \( Z_L \) zone. Meanwhile, GReM caches as many as hot bins from the current epoch (“currEpochBin”) in the \( Z_S \). Here, we give bins from “currEpochBin” higher priority than those bins that are evicted from \( Z_L \) (see lines 15-30 in Algorithm 1).

B. D_GReM: Dynamic Zone Partitions

In GReM, we statically and equally partition SSDs into two zones, i.e., \( C_{Z_L} = C_{Z_S} = \frac{C_T}{2} \), which 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 \( Z_S \) becomes not large enough to handle bursty IOs that access to those bins. This thus incurs a very low IO hit ratio and meanwhile increases 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 dynamically to adjust sizes of \( Z_L \) and \( Z_S \). We refer this new version of GReM to as D_GReM. Fig. 4 depicts the basic structure of our D_GReM. The main component is our bursty detector, which takes the feedback 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. Different partitioning decisions are then made according to the detected result.

1) Bursty Detector: 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 to decide (1) when to adjust the partition of SSDs, and (2) how much SSD should be shifted from one zone to the other. We find that the capacity adjustment of two zones is needed when the number of popular bins or the working set size (WS) significantly changes. Additionally, to improve the sampling accuracy, D_GReM adopts a sliding window (SW) to record IO access history for all VMs in recent epochs. D_GReM tracks the change in working set sizes in the current and previous sliding windows (i.e., \( |SW_{curr}| \) and \( |SW_{prev}| \)). The relative difference between \( |SW_{curr}| \) and \( |SW_{prev}| \) is then defined as bursty degree (denoted as \( B_d \)) as follows.

\[ B_d = \Delta (|SW_{curr}|, |SW_{prev}|) = \frac{|SW_{curr}| - |SW_{prev}|}{|SW_{curr}|}. \]
Algorithm 1: Dynamic Partition in $Z_L$: GREM

**Input:** historyBin: a dictionary in which key is bin IDs of all VMs and value is the relative access count for each bin, currEpochBin: accessed bins of all VMs in last epoch, shortBin: cached bins of all VMs in $Z_S$, longBin: cached bins of all VMs in $Z_L$.

**Output:** flashBin: bins need to be cached in Flash

1. **Procedure** GREM():
   2. UpdateLongTermZone();
   3. UpdateShortTermZone();
   4. flashBin = shortBin + longBin;
   5. return flashBin;

2. **Procedure** UpdateLongTermZone():
   3. if len(historyBin) ≤ len(longBin) then
      4. longBin = historyBin.keys;
   5. else
      6. $j = \text{len}(\text{longBin})$;
      7. itemH = number of $j$ bins in historyBin.keys with highest historyBin.values;
      8. evictBin = bins of longBin which are not in itemH;
      9. longBin = itemH;
   10. return;

3. **Procedure** UpdateShortTermZone():
   11. if len(longBin) < len(historyBin) ≤ len(flashBin) then
      12. shortBin = the remaining bins of historyBin.keys which are not in longBin;
   13. else if len(historyBin) > len(FlashBin) then
      14. shortBin = bins of currEpochBin which are also in longBin;
      15. currEpochBin = bins of currEpochBin which are also in longBin;
      16. if len(currEpochBin) ≥ len(currEpochBin) then
      17. shortBin = number of $j$ bins in currEpochBin with highest IO popularity;
      18. else
      19. shortBin = evictBin;
      20. shortBin = bins of shortBin which are also in currEpochBin;
      21. $j = \text{len}(\text{shortBin}) - \text{len}(\text{currEpochBin})$;
      22. shortBin = number of $j$ bins in shortBin with highest IO popularity;
      23. shortBin = currEpochBin;
   24. return;

The values of $B_d$ are in good agreement with workload changes across time. The bursty detector claims the arrival of bursty I/Os when the value of $B_d$ is beyond a predefined threshold $\beta$ (we set $\beta = 0.6$ in our experiments).

2) Non-Bursty Case Operation: When there is no burstiness in current IO workloads, i.e., $B_d < \beta$, D_GREM tunes the splitting bar between the $Z_L$ and $Z_S$ zones by leveraging the feedback of each zone’s caching performance under the present partition (as shown in Fig. 5(a)). In particular, we evaluate the importance of two zones (i.e., their contributions to 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 I/Os) in each zone during the recent epoch (e.g., 5 minutes). We hereby define contribution ratio $\rho$:

$$\rho = \alpha \times \frac{H_{V_L}}{H_{V_S}},$$

(4)

3) Bursty Case Operation: When bursty I/Os are identified by the bursty detector, D_GREM turns to aggressively shift...
SSD resources from one zone to the other. Using contribution ratio as a feedback to reset two zones unfortunately does not work well under this case. First, we notice that such a cache-feedback-based approach cannot quickly adapt to workload changes. The corresponding delay can further incur a “cascade effect” of insufficient capacity in one of two zones. For example, when bursty I/Os 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 I/Os. As a result, that 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.(5).

To avoid such a cascade effect, we attempt to dynamically and aggressively assign more SSDs to $Z_S$ when bursty I/Os are found in workloads and meanwhile minimize the penalty on $Z_L$’s caching performance. The general idea is that if the working set size of accessed bins in the current epoch increases dramatically compared with the previous epoch, then it would be helpful if we increase the size of $Z_S$ for absorbing the spikes in the near future. Again, to enlarge the sample size, we use a sliding window $SW$ to record the bin popularity statistics in recent several epochs (instead of the latest one). $D_{GR\text{EM}}$ 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 as “$SW_{Lt}$”. Bins in $SW_{Lt}$ are the ones that are frequently accessed in recent epochs and should be kept in $Z_L$. $D_{GR\text{EM}}$ then uses the average number of accesses ($SW_{Lt}$) of all bins in $SW_{Lt}$ as a criterion to set the threshold ($SW_{th}$) for choosing hot bins to be cached in the $Z_S$ zone as follows.

$$SW_{th} = \gamma \times SW_{Lt},$$  \hspace{1cm} (6)

where $\gamma$ is an adjustment parameter. $D_{GR\text{EM}}$ identifies all bins in $SW$ (i.e., accessed in recent epochs) that have been accessed more than $SW_{th}$ times and are not currently cached in $Z_L$.

$$SW_{Qf} = \{x|x \in SW, x \notin SW_{Lt}, x > SW_{th}\}.$$ \hspace{1cm} (7)

We refer this set of bins as “$SW_{Qf}$” and set the anticipated capacity of $Z_S$ to the total size of bins in $SW_{Qf}$.

$$C_{Z_S} = |SW_{Qf}|.$$ \hspace{1cm} (8)

4) Dynamic Tuning Algorithm: Algorithm 2 gives a general idea of how $D_{GR\text{EM}}$ dynamically adjusts the partitioning between $Z_L$ and $Z_S$ for improving overall IO hit ratio. Initially, capacities of both two zones are set to half of the entire SSDs. $D_{GR\text{EM}}$ recalculates $B_d$, the present bursty degree at the end of each sliding window and determines if bursty I/Os are arriving by comparing with the threshold $\beta$, see line 2 in Algorithm 2. Under the bursty case, $D_{GR\text{EM}}$ aggressively enlarges the capacity of $Z_S$ according to Eq.(5), see lines 11-14 in Algorithm 2. On the other hand, $D_{GR\text{EM}}$ bases on the contribution ratio of two zones to conservatively adjust the assignment of SSDs to these zones when there is no burstiness, as shown in lines 6-7 in Algorithm 2. A boundary checking is also further to ensure the minimum capacity for each zone, see line 8 in the algorithm.

IV. EVALUATION

In this Section, we conduct trace-driven simulation for evaluation by replaying real world enterprise IO workloads. We implement three versions of our proposed algorithm: (1) the baseline version $G_{REM_Eq}$ which equally partitions $Z_L$ and $Z_S$ with 50% of the total cache size; (2) $G_{REM}$ also keeps the same size for $Z_L$ and $Z_S$, while each VM’s partition in $Z_L$ is adaptively adjusted according to the workload change. (3) $D_{GR\text{EM}}$ can further dynamically adjust the sizes of $Z_L$ and $Z_S$ during runtime. For comparison, we also implement conventional caching algorithms LRU [5] and CAR [6], and another recently published SSD-HDD tiering algorithm vFRM [7]. We demonstrate and compare the effectiveness of $G_{REM}$ with the respects to our primary goals: maximizing the IO hit ratio and minimizing the IO cost incurred in managing the SSD. We also investigate the effects of the parameters in dynamical partition: (i) each VM’s quota in $Z_L$, and (ii) the sizes of $Z_L$ and $Z_S$.

A. IO Hit Ratio

Overall, our $G_{REM_Eq}$ and $D_{GR\text{EM}}$ are superior to other existing caching algorithms. Fig. 6 shows the IO hit ratio results for different workload combinations among MSR, FIU and UMASS repositories.

First, Fig. 6(a) MSR-F1 represents a cache friendly workload. When the cache size is smaller than 2GB, vFRM, $G_{REM_Eq}$ and $D_{GR\text{EM}}$ all have lower hit ratios than the conventional algorithms, and $D_{GR\text{EM}}$’s is slightly better than vFRM and $G_{REM_Eq}$. This indicates that coarse update granularity and two-zone design do not help much improve the utilization of a small cache. However, as the size of SSD increases to 4GB, $D_{GR\text{EM}}$ first goes cross LRU and CAR, while at 8GB vFRM and $G_{REM_Eq}$ exceed LRU and CAR. Finally vFRM and GReMs converge to 98.15%, which is 4.55% higher than the conventional algorithms’ performance (93.60%). The reason of the converging is that the SSD is sufficiently large to cache all workloads almost without any cross-VM pollution and flushing.

In Fig. 6(c) MSR-U, we observe a cache-unfriendly workload. When the SSD is relatively small, the conventional caching algorithms obtain higher hit ratios than vFRM and
our GREMs, because the cache-unfriendly workloads have relatively large values of $W$. By using a small cache line size (e.g., 4KB) and on-the-fly updating SSD contents for each cache miss, the conventional algorithms can fully utilize the SSD to hold as much as they can. D_GREM starts to exceed all the other algorithms from 2GB, and the gap gets larger when the cache size increases. It is interesting that even with a 64GB SSD, GREMs still do not converge, and they keep the trend of getting better hit ratios. In this experiment, D_GREM is up to 10.59% better than LRU at 40GB, and 12.62% better than vFRM and 13.61% better than GREm both at 64GB. Similar observations can be found in the mixed cases, as well as FIU and UMASS workloads.

B. IO Cost

Fig. 7 shows the normalized overall IO costs based on the measured data from an 80GB Intel DC S3500 Series SSD and a 2TB 5400 RPM Western Digital WD20EERS-63J48Y0 HDD. The conventional caching algorithms, LRU and CAR use 4KB as the cache line size, and all the others update SSD contents using the IO size of 128KB, and group I/Os into 1MB bins. First, we observe that all coarse-update-granularity methods significantly reduce the overall IO costs compared to the conventional caching solutions, especially in cache unfriendly and mixed cases.

Furthermore, D_GREM always yields the lowest cost in all these cases. For instance, as shown in Fig. 7 (a), with 8GB cache size, the overall IO costs of four MSR cache-friendly workloads under D_GREM are 61.96% lower than LRU, 11.77% lower than vFRM and 7.62% lower than GREM_Eq. Later vFRM converges to 36.49% because the cache size is big enough to hold the workloads. Similar results of cache-unfriendly workloads can be seen in Fig. 7(b), where D_GREM’s performance is up to 87.40% lower than LRU at 4GB, 2.97% lower than vFRM at 12GB, and 1.95% lower than GREM at 8GB. For the mixed case, Fig. 7(c) illustrates that D_GREM at most can be 85.80% lower than LRU, 3.06% lower than vFRM, and 2.20% lower than GREM all at 16GB. This observation consolidates that dynamically adjusting the $Z_L$ and $Z_S$ zones according to the incoming workload can help lower the interference across two zones and better utilize the SSD.

C. Dynamically Partition inside $Z_L$

![Fig. 8. Partition inside $Z_L$ for (a) MSR-F1, (b) MSR-U and (c) MSR-FIU workloads under GREM.](image)

We track the SSD occupancy ratio of each VM during the runtime for every 5min of GREM to see how efficiently GREM can dynamically adjust the size allocated to each VM in $Z_L$. We show three representative results from MSR workloads in Fig. 8, where $Z_L$ and $Z_S$ are fixed to be half the cache size: (a) and (b) with 1500MB, and (c) with 3000MB. Cache friendly workloads are relatively stable, therefore the share of each VMs does not change a lot along runtime. As a comparison, for cache-unfriendly workloads in (b), stgl, web2 and usr2 dominate the cache when they include burstiness, while src21 is provisioned less space due to its relatively lighter load. These are ascribed to GREM’s ability of quickly...
responding to workload change by resizing the VM spaces in $Z_L$. Similar observations can be obtained from the mixed workload (MSR-FIU), where the VMs with higher $WS$ and IO accesses receive more allocations than the others.

D. Dynamical Partition of $Z_L$ and $Z_S$

![Graph](image)

**Fig. 7.** Normalized IO Cost of MSR, FIU and UMASS workloads.

![Graph](image)

**Fig. 9.** Bursty detection correctness verification of D_GREM

We first verify the correctness of the bursty detection and operation in Fig. 9, which is a sample piece of D_GREM's detection and reaction for MSR-FIU. We only show $[0, 1]$ here because we are interested in seeking the arrival of each burstiness, i.e., $WS$ varies significantly ($|SW_{prev}|$ is far greater than $|SW_{prev}|$), which means $B_d$ is close to 1. As we can see, during non-bursty periods, D_GREM finds the best partition based on the feedback of cache hit status. When a burstiness comes (threshold 0.6 is exceeded), the bursty-case operation is triggered and aggressively increases $Z_S$'s size guided by the workload feedback. As we mentioned before, only considering the feedback of cache hit status cannot reflect the workload burstiness. When the bursty period ends, D_GREM switches back to the feedback based operation, which can gradually return to the best spot during non-bursty period.

Furthermore, we also evaluate the overall SSD partition over time as shown in Fig. 10. Overall, $Z_S$'s average size of MSR-F1, MSR-U and MSR-FIU are 16.03%, 12.13% and 12.93% of the total SSD size respectively. Cache friendly workload MSR-F1's short term hot bins have higher reaccess possibilities than MSR-U's. Therefore D_GREM assigns less $Z_S$ for it. We also observe that MSR-F1 yields a less aggressive increment in $Z_S$ compared with the other two workloads. All these observations match our previous analysis in Section III-B.

E. Sensitivity Analysis for D_GREM

In this section, we conduct a serial of sensitivity analysis experiments on D_GREM including parameters of bursty detection, bursty/non-bursty operations, and zone boundary. The observations obtained from this section can help us to optimize corresponding parameters for dynamic partition between $Z_L$ and $Z_S$. Table IV-E shows our sensitivity analysis results in terms of IO hit ratio results for MSR-F1, MSR-U and MSR-FIU workloads.
(1) Bursty Detection Parameters: In Table IV-E, $\beta$ is a preset bursty detection threshold. We fix $\alpha = 0.4$, $\gamma = 1$, $SW_s = b$, $B_U = 95\%$ and $B_L = 1\%$, and the results indicate that $\beta = 0.5$ is the best setting for MSR-F1 workload, and $\beta = 0.6$ is the best setting for MSR-F2 and mixed MSR-FIU cases.

$SW_s$ is the size of the sliding window for monitoring the recent workloads (i.e., $SW_s$ epochs are monitored each with 5min duration). A large sliding window can improve the sampling accuracy. However, if we allow $SW_s$ to be infinitely long then it will be similar as historyBin (the only difference is that historyBin has an aging process). In our tests, we fix $\beta = 0.6$, $\alpha = 0.4$, $\gamma = 1$, $B_U = 95\%$ and $B_L = 1\%$. The results show that $SW_s = 10$ is the best choice for all tested workloads.

(2) Bursty and Non-bursty Operation Parameters: We use knob $\gamma$ to control the qualifying threshold. We fix $\beta = 0.6$, $\alpha = 0.4$, $SW_s = 5$, $B_U = 95\%$ and $B_L = 1\%$ and observe that for MSR-F1, it is better to decrease the threshold, and for MSR-U and MSR-FIU, we need to increase the threshold due to the impact of burstiness. $\gamma = 12$ is a good choice for all workloads.

For the non-bursty operation cases, since the majority of hit volume are from $Z_L$, we need to “cool down” the dominating factors from $Z_L$ by adjusting $\alpha$. We tune $\alpha$ from 0.1 to 0.45, and fix $\beta = 0.6$, $\gamma = 1$, $SW_s = 5$, $B_U = 95\%$ and $B_L = 1\%$. We find that 0.35 is the optimal value for all the tested workloads. This means that we can reduce the importance of $HV_L$ to 35% of its value to balance the weights of $HV_L$ and $HV_S$.

(3) Zone Boundary Parameter: Since $Z_L$ contributes more hit volume, we need to avoid the case where short-term burstiness frequently flush and force the $Z_L$ to give up all its space. Therefore, we set up an upper bound of $Z_S$. We observe that 95% is the best setting in our tests. In addition, we limit the minimum size of $Z_S$ to 1% because overall $Z_L$ plays a more important role and it is reasonable to make it more flexible for $Z_L$ to obtain space in the cache.

Above all, we suggest the following settings for D_GRE: $Z_L$ and $Z_S$ to the half of the total cache size at beginning, and let $\alpha = 0.35$, $\beta = 0.6$, $\gamma = 1.2$, $SW_s = 10$, $B_U = 95\%$ and $B_L = 1\%$.

V. RELATED WORK

Effective workload studies can imply the accurate modeling, simulation, development and implementation of storage systems. For example, [8] introduced 12 sets of long-term storage traces from various Microsoft production servers and analyzed trace characterizations in terms of block-level statistics, multi-parameter distributions, file access frequencies, and other more complex analyses. [3] presented an energy proportional storage system by effectively characterizing the nature of IO access on servers using workloads from three production systems. [2] captured IO workload traces from actively-used production storage systems, and all of which revealed surprisingly high levels of content similarity for both stored and accessed data.

Host-side caches are being widely accepted in modern storage systems. ARC [9] and LRFU [10] are commonly used caching algorithms that consider the frequency and recency of workloads. Inspired by ARC and based on Clock [11], CAR and CART [6] are developed which inherit virtually all advantages of ARC, but does not serialize cache hits behind a single global lock. [12] uses SSD as a disk cache and adopt wear-level aware replacement policy based on LRU. A hybrid storage system called “Hystor” [13] is developed to fit the SSD into the storage hierarchy. Hystor identifies the performance- and semantically-critical data and retains these data to SSD. A SSD-based multi-tier solutions called “Extent-DT” [14] is proposed to perform dynamic extent placement utilizing tiering and consolidation algorithms.
Argon [15] partitions the memory cache between different services, providing isolation between the hit rate of each service. The difference with our work is that Argon optimizes the cache hit rate for individual services, while GREM optimizes both the overall cache utilization and hit ratio. Jigsaw [16] is a CPU cache partitioning solution, whose perspective is mainly focusing on resource (i.e., compute and storage) constrains. Albrecht et al. developed Janus [17] which is an optimized flash allocation algorithm based on both the cache-ability of different traces’ IO activities and tiered storage characteristics like speed and price. The major difference here is that GREM targets in block granularity while Janus optimizes file based tiered storage. A hypervisor-based design “S-CAVE”, was presented in [18]. Its optimization is based on runtime working set identification, while GREM explores a different dimension by monitoring changes in locality, burstiness and per VM IO popularity. vCacheShare [19] presented a dynamic, self-adaptive framework for automated server flash cache space allocation in virtualization environments. They introduced an observation window to calculate cache hit ratio estimation (based on reuse distance analysis), and reuse degree to detect spikes. Based on them, it linear-programming-based combinatorial optimization method to conduct partitioning.  

[20] proposed a new allocation model based on the notion of per-device bottleneck sets. In this model, clients that are bottlenecked on the same storage device receive throughputs in proportion to their fair shares while allocation ratios among clients in different bottleneck sets are chosen to maximize overall system utilization. VFRM [7] design new VMware Flash Resource Managers (vFRM) under the consideration of both the performance and the incurred cost for managing Flash resources. They adopt the ideas of thermodynamic heating and cooling to identify data blocks that use coarser temporal and spatial migration granularity between SSD and HDD in a lazy and asynchronous mode.

VI. Conclusion

In this paper, a new global SSD resource management scheme (GREM) is presented to allocate a suitable amount of SSDs to VMs that run heterogeneous workloads. The design goal is to best utilize the SSD by maximizing the IO hit ratio and minimizing the IO costs. GREM takes dynamic IO demands of all VMs into consideration and splits the entire SSD space into long-term and short-term zone.

We further develop D_GREM to dynamically adjust the partition of SSDs between two zones by leveraging the feedback of workload changes and SSD performance. By trace-driven simulations, we demonstrate that our new scheme allows 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 the changes by switching SSDs between two zones. The IO hit ratio is further improved under D_GREM. In the future, we plan to implement our schemes (GREM and D_GREM) in real virtualized storage systems and evaluate the effectiveness of these schemes under different IO workloads.

REFERENCES