OC-Cache: An Open-channel SSD Based Cache for Multi-Tenant Systems

Haitao Wang*†, Zhanhuai Li*†, Xiao Zhang*†, Xiaonan Zhao*†, Xingsheng Zhao‡, Weijun Li§, Song Jiang‡
*School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'An, PR China, 710072
†Key Laboratory of Big Data Storage and Management, Northwestern Polytechnical University,
Ministry of Industry and Information Technology, Xi'An, PR China, 710072
‡Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, USA, TX 76010
§Shenzhen Dapu Microelectronics Co. Ltd, Shenzhen, PR China, 518100
wanght@mail.nwpu.edu.cn, {lizhh, zhangxiao, zhaoxn}@nwpu.edu.cn,
xingsheng.zhao@mavs.uta.edu, liweijun@dputech.com, song.jiang@uta.edu

Abstract—**In a multi-tenant cloud environment, tenants are usually hosted by virtual machines. Cloud providers deploy multiple virtual machines on a physical server to better utilize physical resources including CPU, memory, and storage devices. SSDs are often used as an I/O cache shared among the tenants for large storage systems using hard disk drives (HDDs) as their main storage devices, which can receive much of SSD's performance benefit and HDD's cost advantage. A key challenge in the use of the shared cache is to ensure strong performance isolation and maintain its high utilization at the same time. However, conventional SSD cache management approaches cannot effectively address this challenge. In this paper, we propose OC-Cache, an open-channel SSD cache framework which utilizes SSD'd internal parallelism to adaptively allocate cache to tenants for both good performance isolation and high SSD utilization. In particular, OC-Cache uses a tenant's miss ratio curve to determine the amount of cache space allocation and where the allocation is (in dedicated or shared SSD channels) and dynamically manages cache space according to the workload characteristics. Experiments show that OC-Cache significantly reduces interference among tenants, and maintains high utilization of the SSD cache.**

Index Terms—**multi-tenant system, SSD cache, performance isolation, scheduling framework**

I. Introduction

Multi-tenant systems have become a key infrastructure in the era of big data and cloud computing. In a typical commercial multi-tenant cloud system, each tenant is often served by a dedicated virtual machine (VM) and multiple VMs simultaneously run on the same physical server to achieve a high resource utilization. The data of a VM are stored in the storage system which usually consists of many HDDs to accommodate increasingly growing data due to their low per-gigabyte cost. To bridge the performance gap between main memory and storage system, SSDs are widely deployed as a cache of HDDs to improve I/O performance [1]–[7]. There are two important goals in the design and management of a shared SSD cache, i.e., maximizing utilization of the SSD and minimizing performance interference among different tenants [2]. Data from different tenants are usually striped across the whole SSD to deliver high utilization and bandwidth [8], [9]. However, performance isolation among tenants can be compromised in this scenario. Strong performance isolation among tenants is desired in many multi-tenant systems because they need to observe service level agreements (SLAs) [10], [11]. Therefore, we need to develop techniques to ensure the isolation in the shared cache for multi-tenant systems.

Many software-based isolation techniques use software schedulers to allocate SSD's bandwidth to tenants by selectively sending requests to the SSD to meet performance goals specified by individual tenants, such as rate limiter used in Linux containers [12] and Docker [13]. However, they cannot manage garbage collection (GC) operations inside SSDs. Data from different tenants can be mixed in the SSD blocks. In this scenario, a tenant that issues an excessive number of random write requests can trigger background GC operations and slow down I/O requests of other tenants. Other techniques are based on hardware isolation, such as multi-streamed SSD [14] and OPS (Over-Provisioning Space) isolation [15]. They can separate tenants to different SSD blocks to reduce performance interference introduced by GC operations. Nevertheless, different tenants can still share the same SSD channel and compete with each other for bandwidth in this approach. Therefore, performance interference can still be severe when these tenants intensively access the SSD.

Compared with the above techniques, Open-Channel SSD (OCSSD) enables a new opportunity for implementing strong performance isolation. OCSSD directly exposes internal parallel units (e.g., channels) to users so that external software can manage SSD resources and parallelism to get predictable performance [16], [17]. In this case, a straightforward isolation method is to limit a tenant exclusively in one or a multiple of dedicated channels. For example, FlashBlox [18] pins a tenant in a number of channels to minimize interference among tenants. However, this allocation of space based on channels is inadequate. First, a channel is usually of very large capacity, typically more than a hundred gigabytes. Second, workloads of tenants can be diversified in term of their working set sizes and access patterns. Accordingly, their miss ratio curves (**MRCs**), which measures cache a tenant's cache space demand, can be

978-1-5386-6808-5/18/$31.00 ©2018 IEEE

substantially different [19]. Under the circumstance, simply allocating space at the granularity of channel to tenants, whose working set size can range from a fraction of a channel's capacity to a size of multiple channels, may lead to inefficient use of channel resources, including its space and bandwidth.

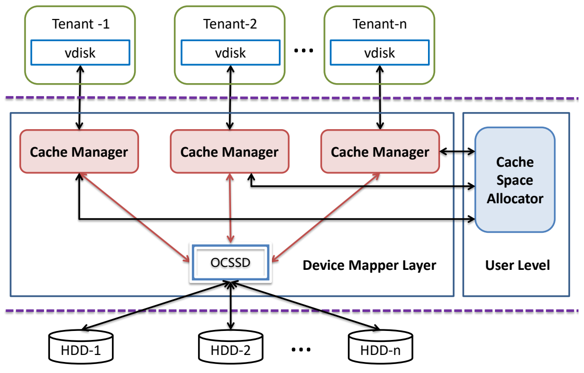
In this paper, we propose OC-Cache, an Open-Channel SSD cache framework to deliver both strong performance isolation among tenants and high SSD utilization. OC-Cache computes a tenant's cache space demand according to its MRC and makes a near-optimal cache partition for them to minimize overall miss ratio of the cache. OC-Cache designates channels in an OCSSD as either dedicated for an individual tenant or shared among all tenants, and determines where to place a tenant's data according to its cache space demand and access pattern. It also manages cache space dynamically in response to changing data access patterns. Experiments using I/O traces from VMware's production systems show that OC-Cache can significantly reduce performance interference among tenants, and maintain relatively high utilization of the SSD cache.

II. Architecture of OC-Cache

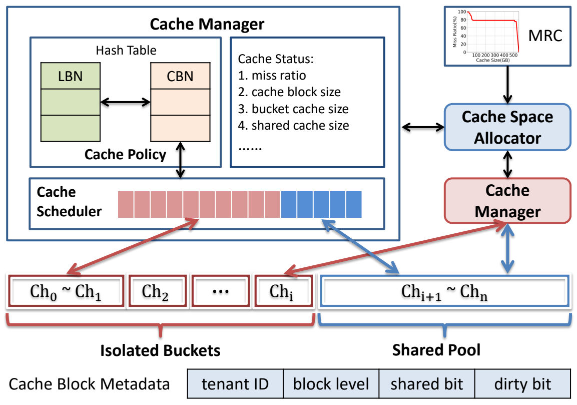
Figure 1(a) depicts a high-level architecture of OC-Cache. For each tenant, OC-Cache launches a module named *cache manager* on device mapper layer to manage its allocated cache space transparently to the tenant. A central module named *cache space allocator* lies on the user level and allocates cache space to tenants accordingly. A tenant's cache manager integrates an HDD with a part of OCSSD to form a virtual disk (i.e., vdisk) for each tenant. I/O requests that sent to a tenant's virtual disk will go to its own cache space or HDD under the control of its cache manager. In this way, I/O requests from different tenants are separated at the host side without modification of virtualization software stack, which makes it easy to plug into existing systems.

Figure 1(b) details the cache manager and its block management. There are two types of channel groups in OC-Cache. One type is named *isolated bucket* where one or multiple channels are exclusively allocated to a tenant to ensure strong performance isolation among tenants. The other is named *shared pool*, whose channels are shared among tenants by striping their data across its channels to improve SSD cache utilization. Each tenant may have its own isolated bucket, and all tenants have a common shared pool. An on-demand cache allocation approach is used to allocate isolated buckets and the shared cache to tenants according to workloads.

The *cache policy* in cache manager maintains a hash table between logical block number (LBN) to cache block number (CBN). It employs a stochastic multiqueue (SMQ) [20] to manage block entries. In short, SMQ maintains a set of LRU lists stacked into multiple levels. HDD block entries are initially at level 0 and move to a higher level when they are accessed. In this way, the more accessed entries (i.e., hotter entries) stay in the top levels and less accessed ones (i.e., colder entries) in the bottom. There are two thresholds to separate blocks into access frequency levels (i.e., *promote level* and *hot level*). These thresholds can be dynamically adjusted



(a) A high level architecture of OC-Cache



(b) Cache manager and cache block management

Fig. 1. Design of OC-Cache.

by the cache policy according to the current I/O patterns of the workloads. Cache policy also manages cache block metadata, which are *tenant ID*, *block level*, *shared bit*, and *dirty bit*. The tenant ID records the ownership of the cache block. The block level represents the level of the block in the SMQ. The shared bit indicates if the block is in the shared pool.

The *cache scheduler* is in charge of data migrations among HDD, isolated buckets, and shared pool according to the level and shared bit of cache blocks (detailed in Section IV). The cache space allocator makes appropriate decisions for cache allocation according to MRCs of tenants' workloads (detailed in Section III). In general, the MRC of a workload can be generated from its history access data, or by some simulation methods such as miniature simulations [19]. During runtime, cache space allocator can collect some cache status as history data through device mapper interface in Linux system periodically to help cache space allocation. When a tenant finishes its work and leaves the system, all of its dirty cache blocks are demoted to its HDD. Cache allocator will then recycle free cache space for future use.

III. Cache Space Allocation

There are two challenges we need to address for an effective cache allocation. The first is to determine how much cache space each tenant should be allocated to minimize the overall miss ratio. The second is to determine the sizes of the shared pool and isolated buckets and where to allocate cache space for each tenant. To address these challenges, we adopt the *convex hull approach* [21] to compute cache partition sizes of tenants according to their MRCs. We then use an *on-demand cache allocation* method to make a partitioning scheme for shared pool and isolated buckets.

A. The Convex Hull Approach

As shown in Figure 2, the convex hull of an MRC is the smallest convex polygon that contains the curve. The convex minorant is the greatest convex curve lying entirely below the polygon. This approach can get a near-optimal cache partitioning to minimize miss ratio of the system, which has been used for partitioning CPU, main memory, and storage system caches [22]–[25].

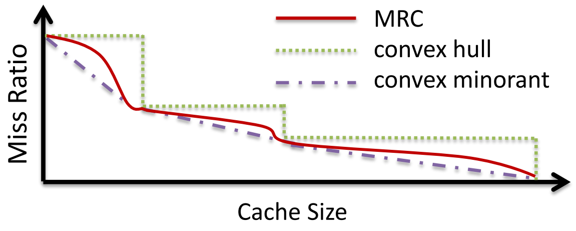


Fig. 2. The convex hull and minorant of a sample MRC.

The convex hull approach has several steps. (1) Initialize partition sizes s_i to zero for each $tenant_i$. (2) Calculate the convex minorant $m_i(s_i)$ of the MRC for each $tenant_i$. (3) Increase s_j by one cache space unit which would reduce the most miss ratio (computed as $m_j(s_j) - m_j(s_j + 1)$) for the tenant. (4) Repeat step (3) until all cache space has been assigned. We name the cache partition size of a tenant computed by the approach *effective cache requirement* (S_r) and corresponding miss ratio *effective miss ratio* (R_e).

B. On-demand Cache Allocation

The crux of the on-demand cache allocation method is two-fold. (1) Allocate most of the cache space to isolated buckets to deliver strong performance isolation, and (2) use the shared pool to accommodate a part of cache to improve cache utilization. Assuming there are multiple tenants online in the system, each of which has its own MRC. Cache space allocation requests from these tenants form a list denoted by $reqList$. There are N_{free} channels in the system and each of which has a capacity of S_cGB. A watermark value W_u is used to indicate the minimum space utilization of each channel. Algorithm 1 describes the allocation algorithm.

At the beginning (Line 3 - 7), we compute S_r and R_e for each tenant via the convex hull approach and sort the request queue according to R_m (i.e., average miss ratio reduction per unit of cache space). We then calculate the number of channels allocated to a bucket ($N_{bucket}[i]$) and shared cache size ($SCache[i]$) for each tenant (Line 8 - 28).

In the process, a request whose cache requirement (S_r) that larger than free space is removed (Line 10), which means we may need to add more OCSSDs into the system. For each $request_i$, we set its shared cache size ($SCache[i]$) to leftover space size (S_{left} in Line 13) if it is below the channel watermark (W_u) (Line 14 - 19), which means we will allocate S_{left} from the shared pool to reduce cache space wastage. Otherwise, we add one more channel to the tenant's bucket (Line 20 - 22), which means all the cache of the tenant will be in its bucket. If there are not enough free channels for the tenant, we add all free channels to its bucket and set $SCache[i]$ to remaining space that cannot fit in the bucket (Line 23 - 28). After the space calculation, we allocate those isolated buckets and shared cache spaces to each tenant (Line 29 - 34). Using Algorithm 1, we can dynamically obtain a near-optimal cache partition scheme to achieve high SSD cache utilization.

Algorithm 1: on-demand cache allocation

```
1  $N_{shared} = 0$; //number of shared channels;
2  $S_{shared} = 0$; //shared cache size;
3  for each $request_i$ in $reqList$ do
4      $S_r[i], R_e[i] = ConvexHullApproach(MRC_i)$;
5      $R_m[i] = (1 - R_e[i])/S_r[i]$;
6      add $[S_r[i], R_e[i], R_m[i]]$ to $request_i$;
7  sort $reqList$ by $R_m$ in descending order;
8  for each $request_i$ in $reqList$ And $N_{free} > 0$ do
9      if $(S_r[i] + S_{shared}) > ((N_{shared} + N_{free}) \times S_c)$ then
10         remove $request_i$ from $reqList$;
11     else if $\lceil S_r[i]/S_c \rceil \leq N_{free}$ then
12         $N_{bucket}[i] = \lfloor S_r[i]/S_c \rfloor$;
13         $S_{left} = S_r[i] \bmod S_c$;
14         if $S_{left} < (W_u \times S_c)$ then
15             $SCache[i] = S_{left}$;
16             $N_{olds} = \lceil S_{shared}/S_c \rceil$;
17             $S_{shared} = S_{shared} + SCache[i]$;
18             $N_{shared} = \lceil S_{shared}/S_c \rceil$;
19             $N_{free} = N_{free} - N_{bucket}[i] - (N_{shared} - N_{olds})$;
20         else
21             $N_{bucket}[i] = N_{bucket}[i] + 1$;
22             $N_{free} = N_{free} - N_{bucket}[i]$;
23     else
24         $N_{bucket}[i] = N_{free}$;
25         $SCache[i] = S_r[i] - N_{free} \times S_c$;
26         $S_{shared} = S_{shared} + SCache[i]$;
27         $N_{shared} = \lceil S_{shared}/S_c \rceil$;
28         $N_{free} = 0$;
29 $N_{shared} = \lceil S_{shared}/S_c \rceil$;
30 create a shared pool with $N_{shared}$ channels;
31 for each $request_i$ in $reqList$ do
32     $Bucket[i] = CreateBucket(N_{bucket}[i])$;
33     allocate $SCache[i]$ to $tenant_i$ from shared pool;
34     allocate $Bucket[i]$ to $tenant_i$;
```

IV. Cache Block Management

During runtime of OC-Cache, tenants who only have isolated bucket are separated from others by channels and performance isolation is guaranteed. The key challenge is to minimize performance interference in the shared pool caused by the GC operations and bandwidth contention among tenants, which can be serious when some tenants are issuing I/O requests intensively to the shared pool. Therefore, we need to keep the shared pool from being accessed too intensively. To this end, we propose the *heat-aware cache replacement* and *adaptive cache migration* approaches to manage the cache.

A. Heat-aware Cache Replacement

In the control of cache policy, when an HDD block's level is higher than the promote level, it is *promoted* into the tenant's cache space in the shared pool if its shared bit is "1" or in the isolated bucket if its shared bit is "0". Some low-level dirty cache blocks are demoted to the HDD when there is not enough cache space. By doing so, we can avoid frequent replacement of cache blocks.

To prevent cache blocks in the shared pool from being intensively accessed, OC-Cache sets the shared bit of a block according to its heat. Specifically, OC-Cache sets a block's shared bit to "1" when its level is higher than the promote level but lower than the hot level. That is, cache blocks in the shared pool are usually colder than blocks in isolated buckets. Therefore, most of the cache accesses are served by isolated buckets. The effect is two-fold. On one hand, I/O requests in the shared pool are unlikely to compete for bandwidth with each other since they are not intensive. On the other hand, GC operations in the shared pool will not be frequent because random write requests are not intensive too. Thus, we achieve strong performance isolation in the shared pool during the cache replacement.

B. Adaptive Cache Migration

The performance isolation in the shared pool can still be an issue with dynamic workloads as the heat of cache blocks may change over time. That is, cache block level in the shared pool may gradually become higher than the hot level (i.e., become hot) which can make the shared pool intensively accessed when there are many hot cache blocks. This can compromise performance isolation. In the meantime, the level of cache blocks in the isolated buckets can become lower than the hot level. In this case, for a tenant who has a bucket and a part of cache space in the shared pool, the bandwidth of its bucket is not fully utilized but I/O requests are slowed down in the shared pool, leading to a low cache utilization.

To address this issue, OC-Cache dynamically migrates cache blocks between isolated buckets and the shared pool for tenants who own cache space in both of the two parts. Specifically, the cache policy monitors the heat of cache blocks by their levels and tracks the number of hot cache blocks (i.e., higher than hot level) in the shared pool. It periodically checks if the ratio of hot cache blocks in the shared pool exceeds the watermark W_{hot}. If so, cache policy will pick some hot blocks in the shared pool from top level of SMQ and inform the cache scheduler to swap them with some cold blocks (i.e., lower than hot level) from isolated buckets in the background until the ratio of hot cache blocks in the shared pool reduces under the W_{hot}. Performance of affected tenants may be slightly lowered during the migration. However, this will not last long since only a small fraction of cache blocks are involved.

V. Evaluation

A. Experimental Setup

We have implemented OC-Cache as a dm-cache module [26] in the Linux kernel. Our experimental platform is configured with two 16-core 2.10GHz Intel Xeon CPUs and 64GB of RAM, running a 64-bit Archlinux 14.04 (Linux kernel 4.14.0-rc2). As for cache device, we use CNEX-8800 LightNVM SDK R1.2, an OCSSD which has 16 channels and each channel has a capacity of 100GB.

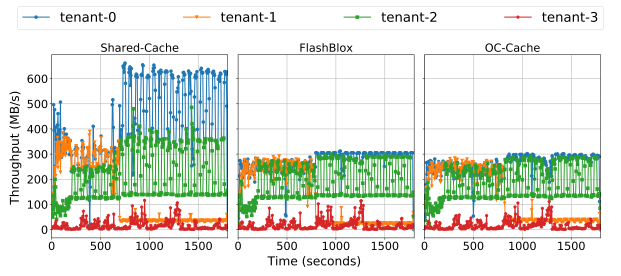
We compare OC-Cache with the conventional solution that all tenants share the whole SSD cache (Shared-Cache for short) and FlashBlox [18] that each tenant exclusively occupies several dedicated channels (FlashBlox for short). In order to avoid interference in the storage, each tenant resides on a dedicated HDD. We create four tenants and choose a representative I/O trace for each tenant in each test. These traces are real-world virtual disk traces from VMware production environments and collected by CloudPhysics [27]. We use the same trace names as they are referred to in [27]. However, MRCs are produced according to the policy in OC-Cache. We set the channel utilization watermark W_u as 50% and cache migration watermark W_{hot} as 20%. We replay these traces to virtual disks of tenants simultaneously with a 4KB I/O block size. In the evaluation, we call a tenant *stable* when it submits I/O requests as fast as possible and *noisy* when it submits I/O requests at a varying rate.

B. Performance Metrics

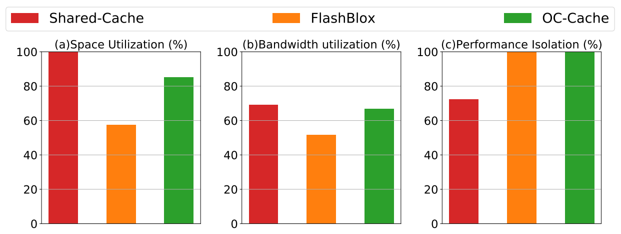
We present three performance metrics to evaluate a SSD cache: (1) **space utilization**, $U_{space} = S_{req}/S_{allo}$ where S_{req} is cache space size required and S_{allo} is cache space size allocated to tenants; (2) **bandwidth utilization**, $U_{band} = B_{tens}/B_{dev}$ where B_{tens} is the aggregate bandwidth of all tenants, $B_{dev} = B_r \times (1 - R_w) + B_w \times R_w$, B_r/B_w is read/write peak bandwidth of cache device and R_w is write ratio of workloads; (3) **performance isolation**, $I_p = C_v/C_v(blox)$ where $C_v = \sum_{i=1}^{n}(\mu_i/\sigma_i)$, μ_i and σ_i are mean value and standard deviation of $tenant_i$'s throughput. $C_v(blox)$ is the C_v of FlashBlox, which is regarded as a baseline. For all of the three metrics, the higher the better. To make a fair comparison, we provide the same number of free channels for Shared-Cache, FlashBlox and OC-Cache.

C. Experimental Results

We first conduct a pure read test, in which tenant-2 is configured to be noisy. In this test, I/O traces for tenant-0 to tenant-3 are t02, t12, t06 and t27 sequentially. According to convex hull approach (Section III-A), the (R_e, S_r) of tenants

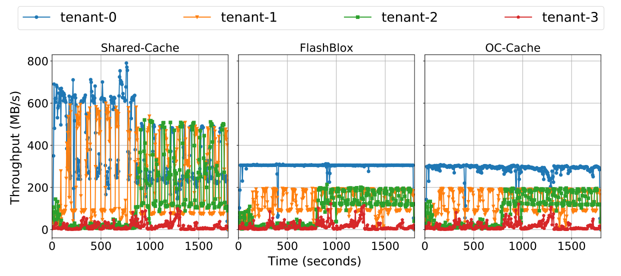


(a) Throughput curves

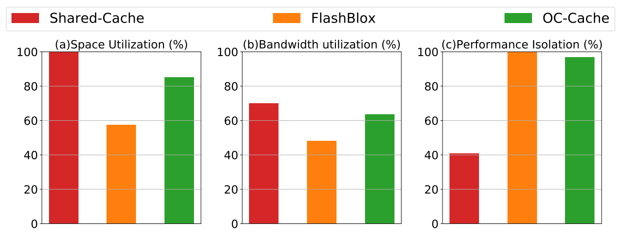


(b) The three metrics

Fig. 3. Performance of all read tenants: tenant-2 is noisy.



(a) Throughput curves



(b) The three metrics

Fig. 4. Performance of mixed workloads: tenant-1 and tenant-2 are noisy and perform write workloads, others are stable and perform read workloads.

are: (1%, 16), (2%, 70), (5%, 114) and (75%, 40) respectively. Note that, a read I/O request to a tenant's virtual disk may cause write operations in cache because a read miss can result in cache replacement. Therefore, GC operations can be triggered by read workloads from tenants.

Figure 3 shows the result. The performance of stable tenants fluctuates much more visibly on Shared-Cache compared with FlashBlox and OC-Cache (Figure 3a). This is mainly because I/O requests from different tenants can easily go to the same channel on Shared-Cache and contend for bandwidth, which makes the performance of all tenants sharing this channel fluctuate even only one is noisy. The performance interference is severer for those tenants whose miss ratio is low (e.g., tenant-0) because they access cache more frequently so that is more likely to contend with others. And GC operations triggered by cache replacement can also aggravate the performance interference. As a contrast, OC-Cache ensures strong performance isolation among tenants through on-demand cache allocation and adaptive cache management according to tenants' workloads. As shown in Figure 3b, take FlashBlox as a baseline, the performance isolation of OC-Cache is 30% higher than Shared-cache.

As for cache space utilization, OC-Cache is 15% lower than Shared-cache because there is a part of free space in the buckets of some tenants (e.g., tenant-1) on OC-Cache which cannot be utilized by others since they are isolated. However, it is 28% higher than FlashBlox since there is more unused cache space on FlashBlox. The bandwidth utilization of OC-Cache is almost the same as Shared-cache and 14% higher than FlashBlox. This is because on FlashBlox, even if a tenant's channel bandwidth is under-utilized (e.g., tenant-3), it cannot share with others since it is isolated. In contrast, OC-Cache improves bandwidth utilization by putting a fraction of cache space of tenants in shared-pool. While on Shared-Cache, cache data tenants are all striped to the whole device so that the bandwidth can be better utilized when dealing with read workloads and cache replacement is not intensive.

The performance interference will become worse when there are more noisy tenants and read/write mixed I/O requests on Shared-Cache. To demonstrate this, we use the same four workloads (i.e., t02, t12, t06 and t27) to conduct a read/write mixed test. In this test, tenant-1 and tenant-2 are noisy tenants who perform write workloads. As shown in Figure 4a, throughput curves of tenants on Shared-Cache fluctuate severely, and stable tenant with low miss ratio (i.e., tenant-0) is drastically interfered by noisy tenants. While on OC-Cache, the performance of tenant-0 is much more stable. As shown in Figure 4b, the performance isolation of OC-Cache is about 2.4× of Shared-Cache with a slightly lower space utilization (15%) and bandwidth utilization (10%) than Shared-Cache. The performance isolation of OC-Cache is 4% lower than FlashBlox because some tenants (i.e., tenant-0, tenant-2, and tenant-3) share cache space in the shared pool which can interfere with each other. But the interference is little because OC-Cache adopts heat-aware cache replacement and adaptive cache migration according to tenants' workload to reduce performance interference in the shared pool.

In conclusion, OC-Cache can significantly improve performance isolation among tenants on a shared SSD cache and deliver a high SSD utilization.

VI. Related Work

In this section, we discuss previous works about SSD cache allocation and performance isolation in multi-tenant systems.

SSD Cache Allocation. There are some previous works about SSD cache allocation in multi-tenant systems. S-CAVE [2] considers the number of reused blocks when estimating a tenant's cache demand, and allocates cache by several heuristics. vCacheShare [4] allocates a read-only cache by maximizing a unity function that captures a tenant's workload characteristics inclduing write ratio, estimated cache hit ratio, disk latency and reuse rate of the allocated cache capacity. CloudCache [6] meets workload's actual cache demand and minimizes the induced wear-out by capturing only the data with good temporal locality. These works can effectively improve cache utilization, but performance isolation between tenants is not adequately considered. In contrast, OC-Cache considers both performance isolation and cache utilization.

SSD Cache Performance Isolation. Some other works try to ensure performance isolation between tenants. Centaur [7] uses SSD cache sizing as a universal knob to meet specific performance goals. It can realize storage QoS but cannot ensure strong performance isolation among tenants since I/O requests from different tenants can go to the same channel. FlashBlox [18] can ensure strong performance isolation between tenants by running them on dedicated channels, but doing so can hurt the cache utilization since tenants' cache requirements are diverse. OC-Cache uses a similar method as FlashBlox to reduce interference among tenants. However, it adopts an on-demand cache allocation method to appropriately allocate isolated channels and shared cache space according to workloads of tenants, delivering both good performance isolation and high cache utilization.

VII. Conclusion

In this paper, we propose OC-Cache, an Open-Channel SSD cache framework for multi-tenant systems. OC-Cache allocates appropriate cache space to tenants according to their MRCs and determines where the cache space should be allocated (i.e., in isolated buckets or the shared pool) by an on-demand allocation method. During runtime, OC-Cache reduces performance interference in the shared pool through adaptive cache block migration between isolated buckets and shared pool according to I/O patterns. Experiments based on production VMware traces show that OC-Cache can significantly reduce performance interference among tenants, and maintain high utilization of the SSD cache.

Acknowledgments

We are grateful to the paper's reviewers who helped to improve the paper's quality. This work was supported by the National Key Research and Development Program of China (2018YFB1004401), the National Natural Science Foundation of China under Grants 61502392 and 61472323. Haitao Wang was supported by the China Scholarship Council (201706290094), and Weijun Li was supported by Shenzhen Peacock Plan (KQTD2015091716453118).

References

[1] S. Byan, J. Lentini, A. Madan, and L. Pabon, "Mercury: Host-side flash caching for the data center," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–12.

[2] 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.

[3] D. Qin, A. D. Brown, and A. Goel, "Reliable writeback for client-side flash caches." in *USENIX Annual Technical Conference*, 2014, pp. 451–462.

[4] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu, "vcacheshare: Automated server flash cache space management in a virtualization environment." in *USENIX Annual Technical Conference*, 2014, pp. 133–144.

[5] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, "Nitro: A capacity-optimized ssd cache for primary storage." in *USENIX Annual Technical Conference*, 2014, pp. 501–512.

[6] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao, "Cloudcache: On-demand flash cache management for cloud computing." in *FAST*, 2016, pp. 355–369.

[7] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side ssd caching for storage performance control," in *Autonomic Computing (ICAC), 2015 IEEE International Conference on*. IEEE, 2015, pp. 51–60.

[8] F. Chen, B. Hou, and R. Lee, "Internal parallelism of flash memory-based solid-state drives," *ACM Transactions on Storage (TOS)*, vol. 12, no. 3, p. 13, 2016.

[9] J. Zhang, J. Shu, and Y. Lu, "Parafs: A log-structured file system to exploit the internal parallelism of flash devices." in *USENIX Annual Technical Conference*, 2016, pp. 87–100.

[10] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in xen," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2006, pp. 342–362.

[11] D. Shue, M. J. Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage." in *OSDI*, vol. 12. Hollywood, CA, 2012, pp. 349–362.

[12] "Control group," https://www.kernel.org/doc/Documentation/cgroup-v2.txt, (Accessed on 08/03/2018).

[13] "docker run," https://docs.docker.com/engine/reference/commandline/run/, (Accessed on 08/03/2018).

[14] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive." in *HotStorage*, 2014.

[15] J. Kim, D. Lee, and S. H. Noh, "Towards slo complying ssds through ops isolation." in *FAST*, 2015, pp. 183–189.

[16] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "Sdf: software-defined flash for web-scale internet storage systems," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 471–484.

[17] M. Bjørling, J. González, and P. Bonnet, "Lightnvm: The linux open-channel ssd subsystem." in *FAST*, 2017, pp. 359–374.

[18] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds." in *FAST*, 2017, pp. 375–390.

[19] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations," in *Proceedings of USENIX ATC*, 2017, pp. 487–498.

[20] "stochastic multiqueue," https://www.kernel.org/doc/Documentation/device-mapper/cache-policies.txt, (Accessed on 08/03/2018).

[21] H. S. Stone, J. Turek, and J. L. Wolf, "Optimal partitioning of cache memory," *IEEE Transactions on computers*, vol. 41, no. 9, pp. 1054–1068, 1992.

[22] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 2006, pp. 423–432.

[23] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *ACM SIGOPS Operating Systems Review*, vol. 38, no. 5. ACM, 2004, pp. 177–188.

[24] D. Thiébaut, H. S. Stone, and J. L. Wolf, "Improving disk cache hit-ratios through cache partitioning," *IEEE Transactions on Computers*, vol. 41, no. 6, pp. 665–676, 1992.

[25] R. Prabhakar, S. Srikantaiah, C. Patrick, and M. Kandemir, "Dynamic storage cache allocation in multi-server architectures," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 8.

[26] "dm-cache," https://www.kernel.org/doc/Documentation/device-mapper/cache.txt, (Accessed on 08/03/2018).

[27] C. A. Waldspurger, N. Park, A. T. Garthwaite, and I. Ahmad, "Efficient mrc construction with shards." in *FAST*, 2015, pp. 95–110.