# DASH: A Duplication-aware Flash Cache Architecture in Virtualization Environment

Xian Chen, Wenzhi Chen, Shuiqiao Yang, Zhongyong Lu, Zonghui Wang
College of Computer Science and Technology
Zhejiang University, Hangzhou, China
Email: {chenxiancool, chenwz, yangshuiqiao, lzy6032, zhwang}@zju.edu.cn

***Abstract*—With the rapid development of multi-core and multi-threading technologies, the performance gap between CPU and storage system is widening year by year, causing the storage system to be the bottleneck of the whole system performance. To alleviate this situation, flash memory has been used as the caching device of HDDs. On the other hand, cloud computing is becoming more and more popular and mature in industry field. As the key building block of it, virtualization technology allows several virtual machines (VMs) running on one single physical machine simultaneously, most of which usually run the same or similar operating systems and applications. In this scenario, flash cache will be occupied by many duplicate data blocks. However, existing flash cache architectures and replacement policies don't take this observation into consideration, which greatly limits the efficient use of the flash cache.**

**In this paper, we propose a new duplication-aware flash cache architecture (DASH). In this architecture, flash cache is organized to cache only one copy of the duplicate data blocks, which can notably expand the effective cache capacity, making more I/O requests hit in the cache. Moreover, this architecture can reduce the amount of data written to flash cache, and thus the life span of flash device can be significantly prolonged. Experiments based on realistic applications show that, in some situations, our cache architecture can improve the cache hit ratio by 5 times, reduce the average I/O latency by 63% and eliminate flash cache writes by 81%.**

***Keywords—flash cache architecture; virtualization technology; data deduplication; replacement strategy***

## I. Introduction

In recent years, with the rapid development of multi-core and multi-threading technologies, CPU processing power has experienced a great growth, while the performance improvement of disk I/O system has not kept pace. As a result, the performance gap between CPU and storage system is widening year by year, and consequently the storage system has become the key bottleneck of the whole system performance. In order to alleviate this situation, many technologies have been applied, such as RAID, larger cache capacity and deeper I/O stack. In addition, there is some work using new emerging non-volatile devices to replace traditional mechanical disks or as the cache devices of hard disk drives, e.g. PCM, FeRAM and Flash memory. While these emerging non-volatile devices are very promising in the long run, most of them are not matured enough to be used in production environment. By contrast, Flash memory is the most mature technology, and has been widely used in industry field. For performance and price reasons, flash device is usually placed between memory and hard disk drive, used as the cache device of HDD. And a lot of research work has been done to improve the performance of flash cache from several aspects, e.g. cache organization [9, 10, 11], cache replacement algorithm [14], flash memory endurance [15], and data consistency [16].

On the other hand, cloud computing and virtual desktop infrastructure (VDI) are more and more popular in industry. As the key building block of them, virtualization technology (e.g. KVM, XEN) can allow one single physical machine to host several virtual machines simultaneously, which can greatly increase the hardware utilization, reducing the overall hardware cost. Cloud service vendors such as Amazon, Rackspace and IBM Compute Cloud have been providing commercial cloud offerings, customers can build their own cloud platforms by using the off-the-shelf templates provided by the cloud vendors or deploy from scratch.

Prior work [1, 2, 3] has shown that there is a large degree of similarity among virtual machine images in real cloud data centers. This is mainly caused by that most of the virtual machines run the same or similar operating systems, libraries and applications. As a result, this similarity has introduced much duplicate data in some layers of storage I/O stack as described in [4, 5], and the flash cache layer is no exception. But existing flash cache architectures and cache replacement algorithms do not consider this situation, resulting in the flash device being not fully utilized. In addition, work in [6] points out that the data locality at the disk I/O level is very poor and content is reused more frequently than address. Exploring the similarity of disk I/O content is a very promising way to increase the cache hit ratio, reduce disk access latency, and prolong the lifetime of flash cache device. Therefore, to design a new duplication-aware flash cache architecture is reasonable and necessary.

Combing these aspects presented above, we propose a new duplication-aware flash cache architecture in this paper. First we will introduce the motivation of this work in section Ⅱ. Then, the detailed architecture, data management and I/O request processing flow will be presented in section Ⅲ. Section Ⅳ will give a detailed evaluation on DASH's performance. Related work and the difference with ours will be shown in section Ⅴ. Finally, we will conclude our work and present the future work in section Ⅵ.

## II. Motivation

The crucial premise of this work is that there are sufficient duplicate data blocks in the flash cache layer. And prior work in [1, 2, 3] also points out that the degree of similarity among virtual machine images in real cloud data centers is really high, up to 80% in some situations. But this similarity is based on the static content of VM images, can't reflect the real dynamic similarity of the I/O request flow. Work in [4] makes up for this inadequacy well, it conducted an extensive experiment study on the duplication ratio in host-side caches in virtualized data center environments. Experimental results show that duplication can reduce the data footprint inside host-side caches by as much as 67%, which can be translated into increased effective cache hit ratio.

In order to understand the realistic duplication situation in the flash cache layer in virtualization environment, we also conduct several experiments based on realistic applications. Because existing public disk I/O traces used in prior work are address-based, and don't contain the requested content, thus they can't be used in our experiments. Instead, we choose several common application scenarios in virtualization environment as listed in Table 1.

TABLE 1. MOTIVATING EXPERIMENTS

| NAME | DESCRIPTION |
|---|---|
| VM Boot | Boot three virtual machines simultaneously, this scenario is common for customers to deploy their own cloud platform composed of several virtual machines, and this is also used in [4, 7]. |
| Virus Scan | Conduct virus scanning on three virtual machines, this is common if we have high requirement on cloud security, and it's also used in [4]. |
| Kernel Compilation | Compile Linux kernel (the version is 2.6.6) on two virtual machines, kernel compilation can simulate a mixed workload, and is used in many research works [6, 7]. |
| Data Analytics | Data Analytics is one benchmark of Bigdatabench [8]. This benchmark relies on using the Hadoop Mapreduce framework to perform data analysis on large-scale datasets. In our experiment we use the Wikipedia data set, and perform the word count workload on two virtual machines. |

In addition, we also build a tool to get the content signature of each I/O request. This tool is placed in the general block layer of Linux I/O stack, and the content signature is based on SHA-1.

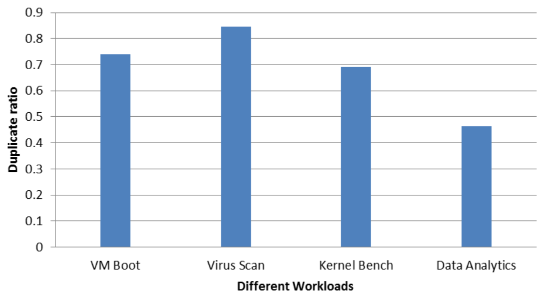


**Figure 1. Duplication ratios under different workloads**

Figure 1 presents the duplication ratios under different workloads. From an overall perspective, the duplication ratio is really high, even in the worst case it's still up to 46.4%. This result agrees well with work in [4].

Another consideration is the computational complexity introduced by deduplication. After careful analysis (details are in section III), we find that this overhead is very small and can be optimized by multi-threading technology. Moreover, the CPU processing power is becoming stronger and stronger. It's not a bad idea to trade computation power for faster disk access speed as mentioned in [9].

Under the above preconditions, we think that to design a duplication-aware flash cache architecture is reasonable and necessary. By storing only one copy of the duplicate data blocks, we can notably increase the effective space of the flash cache, making more I/O requests hit in the cache. Furthermore, the write traffic to flash cache device can be greatly reduced, which can prolong the flash devices' lifetime. In addition, the idea of applying deduplication in flash cache can be combined with prior work for further optimization.

## III. System Architecture of DASH

In this section, we will describe the details in DASH from several aspects, including cache organization, I/O request processing flow, cache replacement algorithm and persistency of data.

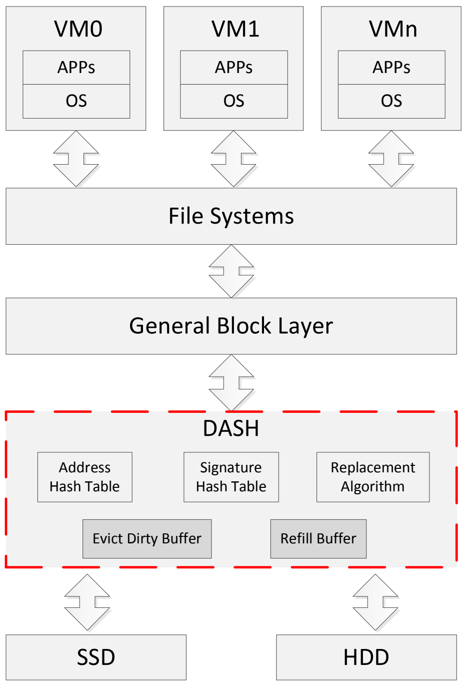


**Figure 2. System Architecture Of DASH**

As depicted in figure 2, we chose to place DASH beneath the general block layer in Linux I/O stack as FlashCache [10]. The main reason is that the semantics of this layer is simple. We can easily intercept the information of every I/O request with the granularity of block, and then some flexible and effective I/O block management strategies can be designed. To some extent, our solution is fully software-based. All the strategies are realized in software, and there is no restriction on storage devices.

### *A. Cache Organization*

In traditional flash cache architecture each cache block only cache one I/O request regardless of the duplication of the content (figure 3(a)). While in DASH, duplication is taken into consideration. If two or more I/O requests have the same content, then we link them to the same flash cache block, saving one or more flash cache blocks. For example, as depicted in figure 3(b), I/O request a and c have the same content, so they are linked to the same flash cache block (A), and the same to d and e. After this operation, the total number of flash cache blocks occupied by I/O requests is three (A, B and D), while for tradition architecture the total number is five, 40% reduction in the used space of flash cache.

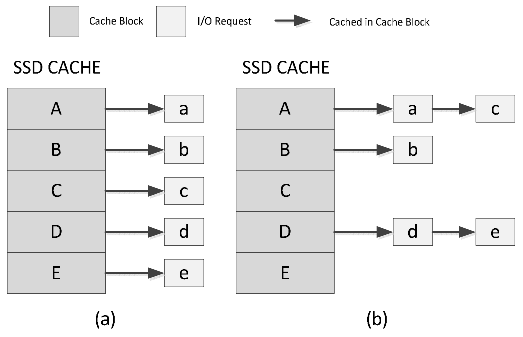


**Figure 3. Comparison of different cache organizations. (a) traditional flash cache organization; (b) duplication-aware flash cache organization**

In the implementation of DASH, the two most important data structures are address hash table (AHT) and signature hash table (SHT). Here, signature refers to the digest extracted from the cache block content by hash function SHA-1. The address hash table is used to find the corresponding flash cache block for a given I/O request if it's already cached in the cache. If we can't find the corresponding cache block, then a cache miss occurs. The signature hash table maps the digest of block content to the corresponding cache blocks whose content can be used to generate the same digest. For a given requested block, we can use its signature to search the signature hash table to find whether or not there is a flash cache block containing the same content with the requested block. If such a cache block exists, we say that there is a replica of the requested block in cache.

To further reduce the latency of the I/O requests, we add two small buffers: Evict Dirty Buffer (EDB) and Refill Buffer (RFB). EDB is used to buffer the evicted dirty cache blocks which need to be written back to HDD, and RFB is used to buffer the fetched data from HDD which will be inserted into SSD. When there is no empty cache block for a write request or a refill request in read miss scenario, we have to use the replacement strategy to select one cache block, if the content is dirty, we have to do a SSD read operation to fetch the cache block and put it into EDB. Then the selected cache block can be used by the incoming request. The data blocks in EDB will be written back to HDD by a separate thread. When a read miss occurs, we have to fetch the data from HDD, and then insert the data into SSD. Like EDB, when processing a read miss, we just put the fetched data into RFB, and use another separate thread to write the data into SSD. With EDB and RFB, we can remove time-consuming write operations from the critical processing path.

### *B. I/O Request Processing Flow*

**Write request**. We first check if the requested data block has already been cached by searching AHT with the requested address. If hit in AHT, then the requested address may be cached in one of the three places: EDB, RFB and SSD cache. If in EDB, the accessed data block may be accessed again in the near future, then we need to move the data block from EDB to RFB, and write the new data to the block. If in RFB, this means the data block fetched in prior read miss processing has not been inserted into SSD cache, then we can directly write the new data to the block, eliminating one SSD write operation. If in SSD cache, we need to use SHA-1 to generate the signature of the new data, and check if there is already a replica of the new data by searching SHT with the signature. If hit in SHT, we just need to change the corresponding map item in AHT to map the requested address to the replica and increase the reference count of the replica, eliminating one SSD write. If miss in SHT, this means the new data can't contribute to the deduplication in SSD cache, then we need to write the data to SSD. Because the requested cache block may be referenced by other addresses, so we should check its reference count before further operation. If it's referenced only by one address, then we directly write the new data to SSD; if not, we should use replacement algorithm to pick a victim cache block and move it to EDB if it's dirty. At last, we write the new data to the victim cache block and modify the corresponding map item in AHT.

If we can't find the corresponding map item in AHT for the write request, we need to use SHA-1 to generate the signature of the new data and search SHT with it. If hit in SHT, we only need to create a new map item for the write request in AHT, and map it to the replica, reducing one SSD write operation. If miss in SHT, we need to select a victim cache block and move it to EDB if it's dirty. When the victim cache block is usable, we write the new data to it and create a new item map in AHT.

**Read Request**. Read request processing flow is relatively simple. When a read request comes, we first search AHT with the requested address. If hit in AHT, the requested data block may be cached in one of the three places: EDB, RFB and SSD cache. If in EDB or RFB, we directly copy the data from EDB or RFB. For EDB and RFB are all in memory, the request processing is very fast. It should be noted that if the data is in EDB, we also need to move the data block from EDB to RFB for further requests. If in flash cache, one SSD read operation is conducted to fetch the data. If there is no corresponding map item in AHT, then a read miss occurs, we need to fetch the requested data from HDD, and insert it into RFB.

### *C. Replacement Algorithm*

Existing flash cache architectures are address-based, and their replacement algorithms only consider the recency and frequency of I/O requests. While our flash cache architecture is based on both address and content, which introduces a new metric for cache replacement strategies, thus former algorithms are no longer suitable here. An effective cache replacement algorithm, which not only considers the recency and frequency but also takes the reference count of cache block into account, is needed for DASH. For space reason, we only focus on the overall architecture and I/O request processing flow in this paper. All the following experiments are based on traditional replacement algorithms (LRU, ARC), and even in this case the performance improvement is very obvious compared with traditional flash cache architecture which does not consider the duplication among different cache blocks. We leave the design of the new cache replacement algorithm described above as our future work.

### *D. Persistency of Metadata and Data*

In DASH, there are four components in memory: AHT, SHT, EDB and RFB. EDB and RFB contain the data blocks which will be written to HDD or SSD. AHT and SHT contain the map information and signature of cache blocks in SSD. When we unload DASH module, we first process all the data blocks in EDB and RFB. After all the data blocks have been processed, we store AHT and SHT in a NVRAM or a specific place in SSD for further reuse.

## IV. EVALUATION

To evaluate the performance of DASH, we construct a trace-driven simulator based on the I/O processing flow presented in section Ⅲ. Because existing public I/O traces used in prior work are based on sector address, and the content of each I/O request is not in the traces, thus they can't be used in our experiments. Traces used here are grabbed in the realistic application scenarios depicted in Table 1. All the applications except data analytics are deployed on a machine with one Intel I7-3770 processor, 8GB RAM and 500GB SATA Seagate disk. As for data analytics, more memory is needed, so we deploy it on a more powerful machine with two Intel Xeon E5620 processors, 32GB RAM and 1TB SAS Seagate disk. Table 2 lists the detailed parameter configuration used in the following experiments, among which the values of DiskRead, DiskWrite, SSDRead, SSDWrite are also used in work [11]. HashTime are based on the realistic test results on our hardware platform (the first one described above). As for the time used to find a specific item in hash table, we directly add the realistic consumed time for every request to the total time overhead.

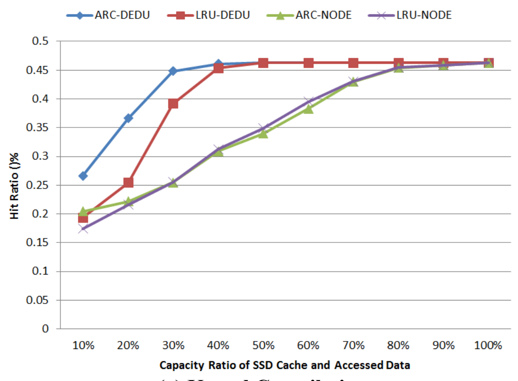
In order to better understand DASH's performance, we choose a traditional flash cache architecture, which doesn't consider the duplication among different cache blocks, as our baseline architecture. For better comparison, LRU-based and ARC-based replacement algorithms are implemented in both the baseline architecture and DASH, which are respectively indicated by LRU-NODE, ARC-NODE, LRU-DEDU and ARC-DEDU in the following performance figures. One thing needs to be noted is that we also implement the EDB and RFB in the baseline architecture to remove the performance influence caused by the additional memory space occupied by EDB and RFB.

TABLE 2: PARAMETER CONFIGURATION

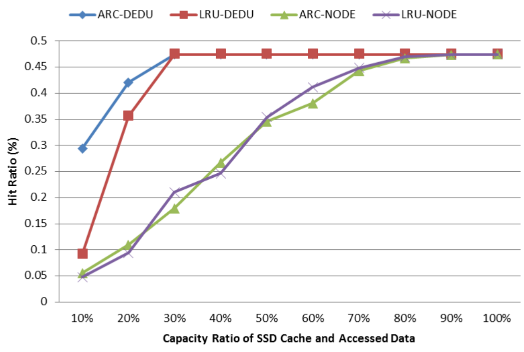
| Operation Type | Description | Value |
|---|---|---|
| DiskRead | Time for 4KB read to HDD | 3.4ms |
| DiskWrite | Time for 4KB write to HDD | 3.9ms |
| SSDRead | Time for 4KB read to SSD | 50us |
| SSDWrite | Time for 4KB write to SSD | 900us |
| HashTime | Time for extracting the signature of a 4KB data block by SHA-1 | 25us |

For a given cache architecture, there are two main performance metrics: cache hit ratio and average I/O time. But if flash memory is used as the cache device, another metric must be taken into consideration for its limited erase lifecycle (about 100000 for SLC and less than 10000 for MLC). The third metric is the amount of data written to flash cache. In the following sub-sections we will discuss the DASH's performance from the above three aspects. For space reason, we only present the performance of kernel compilation workload and virus scan workload.

### *A. Cache Hit Ratio*



**(a) Kernel Compilation**

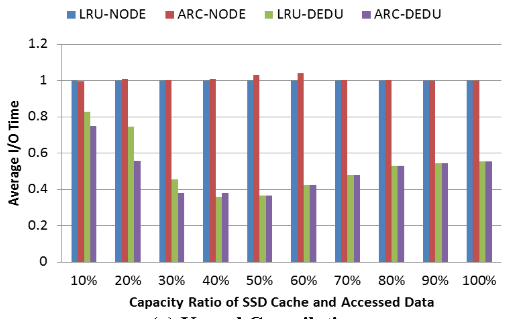


**(b) Virus Scan**

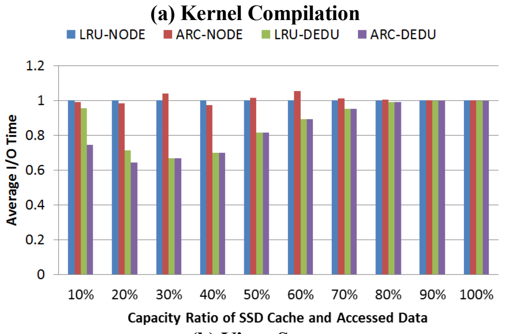
**Figure 4. Cache hit ratio**

Figure 4 depicts the cache hit ratios of the two workloads. The x-axis shows the capacity ratio of SSD cache and the accessed address space. The accessed address space refers to the number of unique addresses requested when the corresponding application is running. The y-axis represents the cache hit ratio. In order to better compare the performance of DASH and the baseline, we set the capacity ratio from 10% to 100%. As we all know, if the cache capacity is big enough to cache most of the accessed data, then the difference between different replacement strategies and cache management schemes will be decreased, which explains why all the lines get closer and closer when the cache capacity increases. From figure 4, we can find: **1)** DASH can reach the highest hit ratio much faster than the baseline architecture. With DASH, we can get the highest cache hit ratio when the capacity ratio is 40% (30%) for kernel compilation(virus scan), while with the baseline architecture, the capacity ratio needs to be 80%. **2)** DASH can significantly improve the cache hit ratio especially when the cache capacity is relatively small. For instance, in virus scan scenario, when the cache capacity ratio is 10% and ARC is used as the replacement algorithm, the cache hit ratio can reach as high as 29.4% while the baseline is just about 5%, about 5X improvement. This is because many duplicate data blocks are removed in DASH, and then the effective cache capacity is expanded. **3)** ARC-based replacement algorithm is more suitable in DASH. This is because that the recency and frequency of content are better than that of address.

## B. *Average I/O Latency*



**(a) Kernel Compilation**

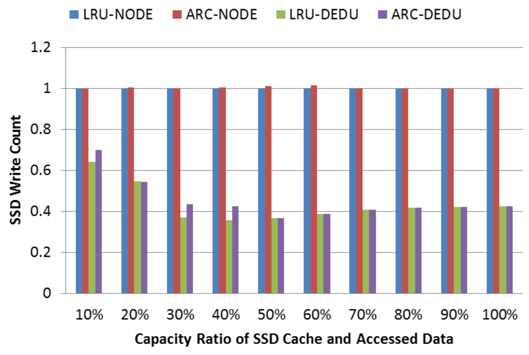


**(b) Virus Scan**

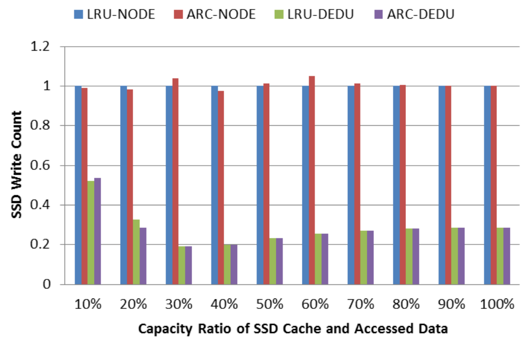
**Figure 5. Average I/O Time**

Figure 5 describes the situation of average access latency. For better comparison, the latency is normalized to the baseline architecture with LRU-based algorithm. From this figure, we can find **1)** the difference between LRU and ARC in baseline architecture is very small, and in some situations the performance of ARC is worse than LRU. This is because the data locality at the disk I/O level is very poor. **2)** DASH can provide notable performance improvements over the baseline architecture in most cases. As depicted in figure 5(a), in kernel compilation scenario, when the capacity ratio is 50%, the average access latency in DASH is only 37% of the baseline, about a reduction of 63%. **3)** For virus scan workload, the difference between DASH and the baseline architecture is very small when the capacity ratio is high. This is caused by two factors. One is that about 96% of the requests are read requests, and read requests are processed in the same way in DASH and the baseline architecture. The other factor is that when the capacity ratio is high, most of the requests can be cached in SSD cache, and then the performance difference will be small.

## C. *Flash Memory Write Count*



**(a) Kernel Compilation**



**(b) Virus Scan**

**Figure 6. SSD Write Count**

As presented in figure 6, DASH can significantly reduce the SSD write count for the two workloads. Especially for virus scan workload, when the capacity ratio is 30%, the SSD write count can be reduced to 19.1% of the baseline, about a reduction of 81%. With DASH, another interesting finding is that the SSD write count of ARC is larger than that of LRU in kernel compilation workload, while ARC can provide better I/O performance than LRU as showed in figure 5. This is because that ARC can generate a much higher read hit ratio and a slightly lower write hit ratio than LRU. Thus more SSD writes will be performed in ARC.

To better understand the potential of DASH in reducing SSD write count, we conduct a further analysis on the evicted cache blocks, and we find some evicted blocks have a large reference count. This is because all the replacement strategies implemented in DASH are based on frequency or recency, none of them take the reference count of cache block into consideration.

## V. RELATED WORK

## A. *Flash Cache Architecture*

For the advantages of high I/O performance and low power consuming, flash memory has been widely adopted in the storage system. One promising direction is to use flash memory as the cache device of HDDs, because this architecture can provide SSD-like access latency and HDD-like storage capacity.

Kgil et al. proposed to split the cache space into separate read and write regions in [12], and a programmable flash memory controller can be used to adjust the cache's working pattern. In [9], Jin et al. described a novel disk I/O architecture, in which a SSD and a HDD were intelligently coupled by a special algorithm. The SSD stores the reference data blocks and the HDD stores the log of deltas between currently accessed I/O blocks and their corresponding reference blocks in SSD. This architecture could significantly improve the I/O performance and reduce the write traffic to HDD.

To improve the flash cache endurance, Pei et al. presented a scalable set-associative flash-cache management design in [11], and this management scheme was demonstrated better than traditional N-way set-associative method. In addition, Liang et al. [13] attempted to combine phase change memory (PCM) and flash memory to construct a hybrid nonvolatile disk cache for good disk I/O performance and better cache endurance.

All these research works presented above don't take the duplication inside flash cache into consideration, and are orthogonal to our work in this paper.

### *B. Redundancy in Vritualization Environment*

As explained in section Ⅰ and Ⅱ, virtualization technology can introduce a lot of redundancy in memory, flash cache and secondary storage. Data deduplication in memory and secondary storage has been extensively studied, and there are many classic methods in these fields, while little work has been done in flash cache layer.

In [2], Jayaram et al. conducted an empirical study on the similarity among 525 VM images from a production cloud. The results show that the duplication ratio among VM images is very high, up to 80% in some situations. In [4], Jing et al. analyzed I/O traces from six VDI applications and two long-term CIFS workloads, and the results suggest that if data deduplication is implemented in host-side flash cache, the total data footprint in cache can be reduced by as much as 67%. But they just presented this observation, and didn't give a feasible cache architecture. Motivated by this work, we conducted several experiments based on realistic applications to validate the correctness of this observation, and further proposed a duplication-aware flash cache architecture.

## VI. CONCLUSION AND FUTURE WORK

In virtualization environment, much of the flash cache capacity is occupied by duplicated data blocks, and this greatly limits the efficient use of flash cache. To solve this problem, we propose a new duplication-aware flash cache architecture, in which the flash cache is organized to store only one copy of the duplicated data blocks. Thus, the effective cache capacity will be significantly expanded, and more requests can be cached. Furthermore, by deduplication in flash cache, the overall cache writes can be notably reduced and the lifetime of flash device can be prolonged. Experiments based on real applications show that, in some situations, our flash cache architecture can improve the cache hit ratio by 5 times, decrease the average I/O request latency by 63% and eliminate flash cache writes by 81%.

In future, we will try to design a more effective cache replacement strategy for DASH. This strategy should consider not only the frequency and recency of I/O requests, but also the reference count of flash cache block.

## ACKNOWLEDGMENT

We thank the anonymous reviewers for their hard work for this conference, and also we'd like to thank all the members of Linux Interest Group of ARC Lab at Zhejiang University for their constructive feedback. This work was supported by the key state science and technology project 2013ZX03003010-002.

## REFERENCES

[1] Jin, Keren, and Ethan L. Miller. "The effectiveness of deduplication on virtual machine disk images." Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference. ACM, 2009.

[2] Jayaram, K. R., Peng, C., Zhang, Z., Kim, M., Chen, H., and Lei, H. "An empirical analysis of similarity in virtual machine images." Proceedings of the Middleware 2011 Industry Track Workshop. ACM, 2011.

[3] Kochut, Andrzej, and Alexei Karve. "Evaluation of redundancy driven provisioning for hypervisors with locally attached storage." Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on. IEEE, 2011.

[4] Feng, Jingxin, and Jiri Schindler. "A deduplication study for host-side caches in virtualized data center environments." Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on. IEEE, 2013.

[5] Sharma, Prateek, and Purushottam Kulkarni. "Singleton: system-wide page deduplication in virtual environments." Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing. ACM, 2012.

[6] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand. "KSM++: Using I/O-based hints to make memory-deduplication scanners more efficient." RESoLVE.2012

[7] Barker, S. K., Wood, T., Shenoy, P. J., and Sitaraman, R. K. "An Empirical Study of Memory Sharing in Virtual Machines." USENIX Annual Technical Conference. 2012.

[8] Wang, Lei, et al. "Bigdatabench: A big data benchmark suite from internet services." arXiv preprint arXiv:1401.1406 (2014).

[9] Yang, Qing, and Jin Ren. "I-CASH: Intelligently coupled array of SSD and HDD." High Performance Computer Architecture (HPCA), 2011.

[10] Facebook Corporation. https://github.com/facebook/flashcache/

[11] Suei, P., M. Yeh, and T. Kuo. "Endurance-Aware Flash-Cache Management for Storage Servers." (2013): 1-1

[12] Kgil, Taeho, David Roberts, and Trevor Mudge. "Improving NAND flash based disk caches." Computer Architecture, 2008. ISCA 2008.

[13] Shi, L., Li, J., Jason Xue, C., and Zhou, X. "Hybrid nonvolatile disk cache for energy-efficient and high-performance systems." ACM Transactions on Design Automation of Electronic Systems (TODAES) 18.1 (2013): 8.

[14] Huang, S., Wei, Q., Chen, J., Chen, C., and Feng, D. "Improving flash-based disk cache with Lazy Adaptive Replacement." Mass Storage Systems and Technologies (MSST), 2013

[15] Liu, R. S., Yang, C. L., Li, C. H., and Chen, G. Y. "DuraCache: A durable SSD cache using MLC NAND flash." Proceedings of the 50th Annual Design Automation Conference. ACM, 2013

[16] Saxena, Mohit, Michael M. Swift, and Yiying Zhang. "Flashtier: a lightweight, consistent and durable storage cache." Proceedings of the 7th ACM european conference on Computer Systems. ACM, 2012.