Locality-Driven Dynamic Flash Cache Allocation

Liang Xu, Qianbin Xia and Weijun Xiao
Department of Electrical and Computer Engineering
Virginia Commonwealth University
Email: {xul4, xiaq2, wxiao}@vcu.edu

Abstract—**Flash-based SSDs are widely deployed as the storage caches to boost the system performance. However, unlike the traditional cache devices such as SRAM and DRAM, SSDs have internal garbage collection activities, which can severely degrade the cache performance. Moreover, SSD can only sustain limited P/E cycles. Therefore, traditional cache hit ratio oriented optimizations might not obtain the optimal performance for the SSD cache and can even shorten the device lifetime, especially for the SSD write cache, which could introduce more internal garbage collection processes and lifetime concerns. In this paper, we propose a reuse distance aware cache management to improve both the performance and lifetime of SSD-based write cache by compromising the cache hit ratio and the internal garbage collection overhead.**

I. Introduction

Nowadays, both personal laptops and large data centers are equipped with large main memory to bridge the huge performance gap between the high-performance processors and slow storage systems. This large main memory can be effective for hiding the latency of read-intensive workloads [1], especially for workloads with good locality. However, for write requests, the main memory is much less effective due to its volatile nature, which means that data could be lost during power failure. Therefore, unlike read cache, which follows the popular cache algorithms like LRU and ARC, write buffering are using the write through policy or the high-low watermark to guide the data flushing processes. Whenever the number of dirty data reaches a predefined threshold (high watermark), the dirty data will be flushed back to the underlying hard disks [**?**]. Therefore, the write operations have been identified as the dominate traffic to the storage system and the performance-critical part of the whole systems equipped with large main memory [2, 3].

Nonvolatile memory techniques such as NAND Flash-based SSDs, phase change memory (PCM), spin transfer torque RAM (STT-RAM), and resistive RAM (ReRAM) are possible solutions to improve the performance of IO intensive workloads as caches or replacements of main memory, a comprehensive summary could be found in [8]. Currently, NAND Flash-based SSDs are the only mature, widely produced, and deployed technique. STT-RAM and ReRAM are still under the research stage and currently not in volume production. Although PCM has been existed for a long time with some real-world products, PCM has never been widely deployed due to its high production cost and high write power consumption, which is even higher than DRAM. Nowadays, energy consumption has become a critical design issue for servers and big data centers, where the low power consumption advantage of SSDs has been explored to save the energy [17, 27]. Moreover, Kim et al. [9] observed that although PCM has a better physical write performance, Flash-based SSDs can achieve a better system level write performance. The major reason is that PCM has limited write parallelism due to its high write energy consumption. While, SSD can be easily scaled to obtain high write bandwidth. In this paper, we are interested in the adopting of Flash memory as a write cache for disks because of its huge capacity, low energy consumption and high achievable write bandwidth. Previous work have presented the benefits of using SSDs as write caches for write-dominated tasks [30]. SSDs have several merits that make it a good candidate for write cache devices between the main memory and hard disk. First, the performance of SSDs are two to three orders of magnitude higher than the hard disk. Second, SSDs are cheaper than DRAM. Third, SSD is nonvolatile (never losing data with power off and no cold cache misses). Due to the nonvolatile nature of SSDs, it could be used as a write back cache and follow the normal cache algorithms like LRU.

Despite all the above merits, SSDs have several limitations, especially for the internal garbage collection processes and limited lifetime. In order to update a Flash page in-place, the whole Flash block (usually consists of 64-256 Flash pages) need to be erased together, which will introduce a remarkable latency for the write operation. In order to hide this latency, SSD adopts the out-of-place update by relocating the new data to other pre-reserved free space and marking the old data as invalid. To support the out-of-place update, part of a SSD capacity will be reserved as the over-provisional space, which is typically 7%-35% of the total SSD capacity. When the accumulation of the invalid data reduces the available free space to a predefined threshold, a garbage collection process will be trigged to reclaim the invalid Flash pages. The garbage collection is a time consume process, which could significantly degrade the performance of SSDs. A higher over-provisional configuration could delay and reduce the internal garbage collection processes.

Another major concern of applying Flash-based SSDs as write caches is the limited endurance. For example, the limited program/erase(P/E) cycles for SLC Flash memory is about 10,000 from the manufacturers' datasheets. In reality, the block P/E limitation is defined to meet the retention and reliability specifications in industrial standards, e.g., the JEDEC standard JESD47G.01 [18] specifies that NAND Flash blocks cycled to 10% of the P/E limitation must have a retention time of ten years, and fully cycled blocks must have a one-year retention time. However, when SSDs are used as write caches, the cold data will be quickly evicted out, while the hot data will be updated within limited reference intervals. Therefore, the requirement of retention time could be highly relaxed and the endurance of SSD write cache will be notably extended. Previous work have shown the possibility of extending the SSD endurance by the relaxing of retention time [19, 20]. Besides, the advancement of error correction codes like low-density parity-check (LDPC) can also help to extent the lifetime of SSDs [25, 26] without noticeable sacrificing of SSD write performance.

Most existing advanced cache algorithms like LIRS [34] and ARC [36] all target the traditional cache devices such as DRAM and SRAM, and use the cache hit ratio as the performance metric. Even the latest cache optimizations, which target the SSDs, still try to capture the maximum cache hit ratio [33, 38]. However, due to the internal garbage collection processes, blindly try to maximum the cache hit ratio may lead to suboptimal performance as showed in [32, 41]. For a SSD write cache, there is a compromise between the cache space and over-provisional space to obtain the optimal cache performance. On one hand, more cache space means higher cache hit ratio, but less over-provisional space and higher garbage collection overhead. On the other hand, less cache space means more over-provisional space and lower garbage collection cost, but at the same time lower cache hit ratio. Hence, how to make the best compromise between the cache space and over-provisional space is of crucial importance to obtain the optimal cache performance.

Miss ratio curve (MRC) is a powerful performance metric representing the relationship between the cache miss ratio and cache capacity. Previously, the MRC is used as an offline modeling due to the extremely high memory and computing resource demands. However, recent advances like [4, 5] make it possible to generate a lightweight and continuously-updated miss ratio curves online. In this paper we will show how MRCs can be leveraged to guide the SSD capacity allocation between the cache space and over-provisional space to achieve the optimal write cache performance.

The rest of this paper is organized as follows. In Section II, we describe the background on SSD and MRCs. Section III presents the details of our proposed dynamically SSD allocation scheme. Section IV presents the evaluation methodology and the experimental results. The related work is included in Section V. Section VI presents concluding remarks.

II. Background and Motivation

A. NAND Flash-Based SSD

Currently, there are three widely used types of Flash memories in the market: SLC (single-level cell, storing one bit information per cell), MCL (multi-level cell, storing two bits information per cell), and TLC (Tri-level cell, storing three bits information per cell). MLC and TLC could significantly increase the capacity and density of Flash memory with the penalty of sacrificed performance and lifetime. Due to the relative higher performance and better endurance, SLC-based Flash memory is a more suitable candidate for the storage cache, especially for the enterprise workloads. Therefore, in this paper, we only consider the SLC-based Flash memory which has been widely used in the industry. Flash-based SSDs have two noticeable special properties compared with traditional cache devices like DRAM. First, SSDs can only support out-of-place update. Second, each Flash block can only sustain limited P/E cycles. To make SSDs compatible with existing file systems designed for traditional in-place update devices, an Flash Translation Layer (FTL) is deployed inside SSDs. A typical FTL has three major function units: address translator, garbage collector, and ware-leveler. The address translator is responsible for maintaining an address mapping table between the logical page number and physical page number and performing the address translation. The address mapping methods could be coarsely classified into three major categories based on the mapping granularity: page-level mapping, block-level mapping, and hybrid mapping. A page-level mapping can obtain the best performance by maintaining the mapping information for each individual Flash page, but it also has the highest memory overhead. In contrast to the page mapping scheme, a block-level mapping [10] builds the mapping table in the unit of Flash blocks and has the least memory consumption, but it will lead to space wastage and performance degradation. To make a compromise, several hybrid schemes [11, 12] combining the page-level mapping scheme with the block-level mapping scheme have been proposed. To update a data block, the new data will be redirected to the pre-reserved over-provisioning space and the old data will be marked as invalid by updating the mapping table. The task of a garbage collector is to reclaim the invalid Flash pages and make room for the coming request in the future. Whenever, the pre-reserved free over-provisioning space is below a predefined threshold, a garbage collection process is trigged to reclaim the space occupied by the invalid data. The garbage collection will first migrate the valid pages in the victim block to other free block and then erase the whole victim block. The garbage collection is a time-consuming process and can block the Flash packages or planes from servicing the coming requests. Many different garbage collection schemes have been proposed, the greedy garbage collection policy [13] is one of the most widely used, which always try to generate the largest number of free space during the garbage collection process. The objective of wear-leveler is to evenly distribute the erase operations over the Flash blocks to prolong the whole lifetime of SSDs. Due to the internal garbage collection process and limited lifetime issue of SSDs, the traditional cache algorithms that use the cache hit ratio as the only metric may not achieve the consistent performance gain for SSD cache and may even shorten the lifetime. For simplicity, we assume the page-level mapping, greedy garbage collection, and no wear-leveler are used in our design.

B. Miss Ratio Curves

The reuse distance (reuse distance is defined as the number of distinctive data elements accessed between two consecutive uses of the same element) of workloads is of prominent importance for the performance prediction and optimization of storage and CPU caches. Miss ratio curve (MRC) is an visualized representation of the reuse distances of the workloads. Figure 1 shows an example of the miss ratio curve of Financial1, where x-axis is the cache size and y-axis is the cache miss ratio. The figure shows that MRC is a diminishing curve (the cache miss ratio decreases or stays the same with the increasing of the cache size). MRC could have several important applications. First, MRC can be used as an off-line optimal cache performance analysis like MIN [39]. Second, MRC of workloads could be used to predict cache performance in future. Third, the MRC can help the system administrator determine the size of the cache needed to meet the system performance requirement. Finally, for a system running multi workloads concurrently, an automated cache manager can generate separate MRC for each individual workloads and leverage the MRCs to optimize the cache space allocation among all these different workloads to achieve the optimal system performance [21, 22]. Although MRC is a powerful tool and has many vital application values, precise MRC measurement in the past requires $O(NlogM)$ time and $O(M)$ space for a trace of N accesses to M distinct elements [23]. The expensive memory and computing resources overhead has restricted MRC to the offline applications. Thanks to the recent advances like SHARDs [4] and AET [5], which reduce the memory overhead to $O(1)$ for the fixed-size MRC construction scheme and $R \times O(M)$ for a fixed-rate scheme, where R is the sampling rate. Besides, SHARDs and AET reduce the computing overhead of constructing the whole MRC for a trace of N accesses to $O(N)$, which means the computing overhead of processing each individual request is merely $O(1)$. Moreover, Niu et al. proposed a parallel algorithm to compute the MRC, which could further reduces the computing overhead by 13-50 times. All the above mentioned advancements have removed the offline restriction of MRC and made it possible to be deployed as an powerful online workloads analysis tool. For example, variants of SHARDS [4] have been implemented and deployed as a key component in real prototype implementations to help the data cache management in big data centers [14, 15].

III. Performance Modeling

Equation (1) shows the definition of the over-provision, where C_{cache} and C_{total} is the capacity for caching data and the total capacity of SSD, respectively. Equation (2) gives the average cost of garbage collection, where U is the average utilization of each block (valid pages ratio) during garbage collections. Therefore $U \times N$ is the total number of Flash pages needed to be migrated, where N is number of pages per block. During valid page migration, a valid page should be first read from its physical location and then rewritten to other free space. Hence, the total cost of valid pages migration is cost of

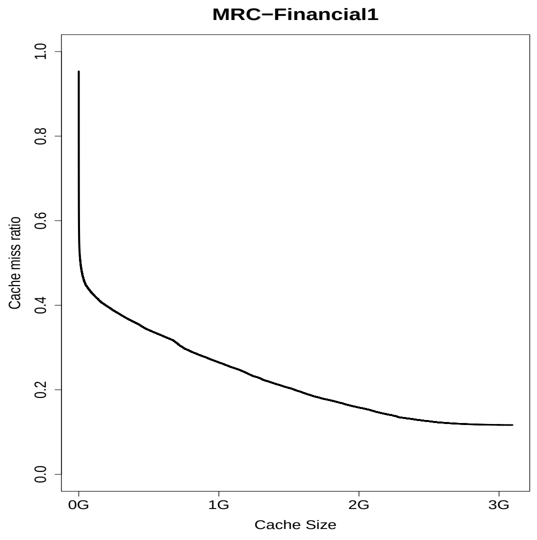


Fig. 1: Miss ratio curve of Financial1.

$U \times N$ Flash reads and writes. The total GC cost is valid page migration cost plus the following block erase cost as depicted in Equation (2). $Cost_{fr}$, $Cost_{fw}$, and $Cost_{erase}$ are the Flash read, write, and block erase overheads, respectively.

$$OP = \frac{C_{cache}}{C_{total}} \tag{1}$$

$$Cost_{gc} = U \times N \times (Cost_{fr} + Cost_{fw}) + Cost_{erase} \tag{2}$$

Each victim block can obtain extra $(1-U) \times N$ free pages after garbage collection, so the real average Flash write cost can be depicted by Equation (3), which evenly splits the garbage collection overhead to the extra $(1-U) \times N$ free pages. If we assume that the valid pages are evenly distributed among the Flash blocks, then U equals OP. However, due to the skewness of the real-world workloads and different garbage collection policies adopted inside SSDs, there could be striking difference between U and OP. Some previous work tried to modeling the relationship between U and OP through a static equation [42], which can not work for all different workloads and SSD configurations. In this paper, we propose to get the U dynamically with training process according to changing of the workloads, which will be presented in Section V in detail. For a write hit in the SSD cache, we only need to write the new data to the SSD cache and invalidate the old-version of the data. While, for a write miss, a cache entry will be evicted out and written to the Disk at first, which involves a Flash read and Disk write operations. Then the new data will be inserted into the SSD cache with an additional Flash write operation. Equation (4) shows the whole latency of this hybrid storage system. What's more, Equation (4) could be further transformed into Equation (5), where MR is the miss ratio equals $1 - HR$, HR is the hit ratio.

$$Cost'_{fw} = Cost_{fw} + \frac{Cost_{gc}}{(1-U) \times N} \tag{3}$$

$$Latency = HR \times Cost'_{fw} + MR \times (Cost_{fr} + Cost'_{fw} + Cost_{hd}) \quad (4)$$

$$Latency = Cost'_{fw} + MR \times (Cost_{fr} + Cost_{hd}) \quad (5)$$

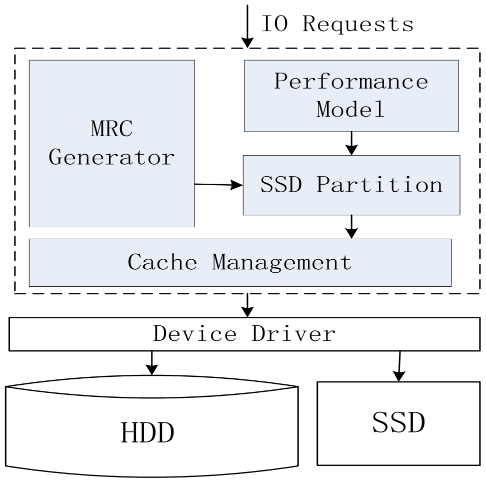


Fig. 2: System Architecture.

IV. Design and Implementation

In this paper, we propose a method to utilize the workload behavior in the past as a reference to guide the following Flash cache space management by combining our derived performance model with Equation (5) and the MRC. Figure 2 shows the system architecture of our hybrid storage system design. Our storage system contains a hard disk as the primary storage and an SSD as the second layer write cache under the device driver. The component above the device driver is our reuse distance aware cache management, which consists four major components: MRC generator, performance model, SSD partition, and normal cache management.

The MRC generator is deployed to dynamically generate the miss ratio curve for the incoming IO requests periodically. Our proposed scheme tries to leverage the locality of the workloads in the history to guide the Flash cache allocation in the future. Hence a time window T is introduced in our design to define how much information in the past should be used to guide the behavior in the future and what's the frequency should we dynamically change our configuration. Initially, the MRC generator is empty and receives the incoming requests to record the reuse distances. After time window T, a miss ratio curve will be generated based on all the requests in the previous time window. The generated miss ratio curve will be leveraged to guide the allocation of the cache space. After the allocation of the cache space, a new time window starts and the miss ratio curve for the previous time window will be cleared. How to choose a proper time window T may have significant impacts on the cache performance. A large time window T may introduce too much valueless information long time ago and can't quickly responses to the changing trends of the workloads. While, a small time window T may generate a pessimistic estimation of locality of the workloads and can't fully utilize the SSD space to achieve the optimal performance. Currently, we configure the time window T as the logical time to process a fixed number of Flash page write requests. For 4 GB SSD buffer with 4 KB Flash pages, the time window T is configured as processing 8GB write requests, which equals to 2097152 Flash writes, while for a 16 GB SSD buffer, the time window is set as 40GB write requests that equal to 10485760 Flash write requests based on our observations of sensitive analysis in experimental result section. As discussed in Section II, the memory overhead is $R \times O(M)$ for a fixed-rate MRC scheme. If we assume R is 0.0001 and each bucket of MRC requires 12 bytes memory space, then for our small-scale and large-scale cache configurations, the memory overheads are merely 24 KB and 120KB, respectively. Even modern laptops are equipped with 8GB or 16GB memory, let alone the big data centers, so the memory overhead of constructing the MRC is totally affordable. The computing overhead of updating the MRC for each individual request is $O(1)$. For a CPU that works at 2 GHz, the computing cost is only about 0.5ns, which is negligible compared with the latencies of accessing Flash memory and hard disks. Therefore, the computing overhead of the MRC construction is ignored in our evaluations.

Based on the generated miss ratio curve from the MRC generator, the performance model component calculates the optimal SSD capacity allocation scheme for current workloads based on equation (5) that we derived in the previous section, which means how much SSD capacity should be used as the data cache and how much should be reserved for over-provision. To make our performance model work properly, an accurate estimation of the utilization U for the victim blocks during garbage collection processes is of utmost importance. Some previous work proposed to use a static equation [42] to catch the relationship between U and OP, which can not work for all different workloads and SSD configurations. In our implementation, we propose to use a training stage to get U dynamically. At the beginning, we will configure our SSD cache with different OP values and get the corresponding U. Since it is impractical to explore all the possible OP values, our training only performs on several discrete OP values (15%, 25%, 35%, 45%, 55%, 65%, 75%, 85% in our design). Based on the training results, a mapping table between the OP and the U will be builded and stored in the performance model. Since during the training stage, the SSD cache doesn't work at the its optimal configuration in majority of the time. Therefore, we will not perform the training process in ever time window, only when we detect a big shift of the workload properties like the reuse distance distributions, a retraining process will be trigged to update the OP to U mapping table.

The SSD allocator receives the optimal allocation result from the performance model and then adjusts the ratio between the SSD data cache and over-provisional space. In our current implementation, the widely used classic least recently updated policy (LRU) is deployed to evaluate our design. There are three possible cases for the cache space allocation. First, the optimal cache configuration is the same with our current cache configuration, then nothing need to be changed. Second, the optimal data cache space is larger then the current data cache space, then we increase the maximum size of the LRU queue to the optimal value by inserting more distinctive data in the cache. Third, the optimal data cache capacity is less than the current data cache size, then we need to reduce the maximum size of the LRU queue by evicting the data from the LRU positions of the cache queue. Since the SSD is used as the write cache, all the data inside SSD are dirty. Therefore, during the data eviction process, the data will be written back to the primary hard disk. Besides, the Flash pages inside SSD that contains the data to be evicted also need to be marked as invalid so that it could be reclaimed during the garbage collection processes in the future. Here, the inherent trim command of SSD could be utilized to inform the SSD of the corresponding data evictions.

TABLE I: Configuration of Our Simulator

Flash Page Size	4KB
Flash Block Size	256KB
GC Threshold	5%
Cache Size	4GB, 16GB
Page Read Latency	25us
Page Write Latency	200us
Block Erase Latency	1.5ms
Disk Access Latency	5ms

TABLE II: Characteristics of I/O workloads traces

Type	Workloads Size (GB)	Working Set Size (KB)	Avg. Req. Size (KB)	Request Amount (GB)
Small Scale	Financial1	3.6	7.2	28.8
	Homes	5	3.9	66.8
Large Scale	Exchange	23.29	12.4	131.69
	MSNFS	23.03	11.12	74.01

Initially, we set the preserved space as 35% as the default configuration. Then for every request, we update the miss ratio curve. Whenever the number of requests reaches a pre-defined period T, we leverage the MRC in the past period to find the optimal C_{user} that achieves the best performance. Then the cache space will be configured as this optimal value. Then the MRC will be reset and updated by the coming request in the next period. In this way, optimal C_{user} could be dynamically adjusted according to the changing trends of the workloads.

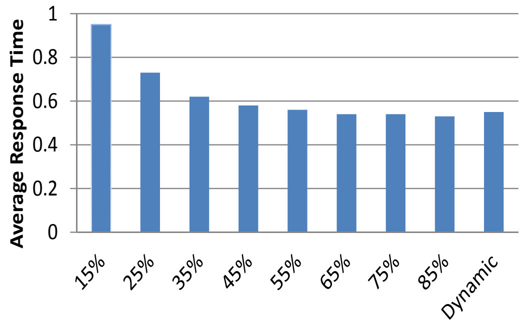
V. Experimental Methodology and Results

To verify the efficiency of our proposed cache design, we modified the Disksim with SSD extension [28] to implement our proposed design. Table I lists the main parameters of our simulator. Flash page size is 4KB and each Flash block consists of 64 Flash pages. The Flash page read and write latencies are configured as 25us and 200us, respectively. The Flash block erase cost is 1.5ms and the garbage collection threshold is set as 5%. The average hard disk access latency is defined as 5ms. The workloads used as our input are from [6] and [7]. The properties of these workloads are presented in Table II. Basically, these four workloads could be divided into two categories based on the working set size: small scale (Financial1 and homes) and large scale (Exchange and MSNFS). Accordingly, our simulator is configured with two different SSD physical sizes: 4GB for the small scale workloads, while 16GB for the large scale workloads.

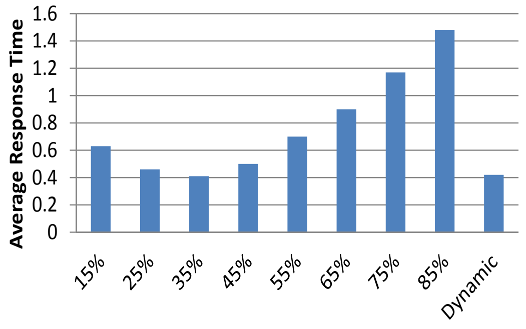
A. Performance

In this section, we evaluate the performance of our dynamical SSD capacity allocation design. For comparison, we also implement the static SSD cache allocation schemes, which include the following over-provisioning configurations: 15%, 25%, 35%, 45%, 55%, 65%, 75%, and 85%. For the static cache allocation schemes, the over-provision of SSD cache is configured as a fixed value like 35% throughout the whole simulation process. While, our dynamic cache allocation scheme will dynamically adjust the ratio between the data cache capacity and over-provisional space to achieve the optimal performance. Figure 3 and Figure 4 show the average response time and cache hit ratio of the static allocation scheme and our dynamic allocation scheme, respectively. From the results, we can observe that the cache hit ratio decrease or keep the same with the increasing of the over-provisioning space. The reason is as follows. Higher over-provision means more SSD space being reserved for out-of-place update and less capacity for caching data, which will lead to lower cache hit ratio. For traditional cache devices like DRAM and SRAM, lower cache hit ratio means lower cache performance. However, due to the internal garbage collection processes, higher cache hit ratio can not always guarantee higher cache performance for SSD-based cache. Figure 3 shows the cache performance variations with the increasing of the over-provision from 15% to 85%. The results show that the cache performance with the increasing of the over-provision is a concave curve, which first increases with the increasing of the over-provision and then decreases with the continuously increasing of the over-provision. When the over-provision is low, the frequent garbage collection processes are the dominant latency contributor. Therefore, increasing the over-provisioning space can reduce the garbage collection activities and improve the system performance. However, when the over-provision reaches the inflection point, the garbage collection overhead become the secondary contributor to the system latency and the long-latency disk accesses due to the cache misses is dominated, further increasing the over-provisioning space will lead to worse system performance due to the lower cache hit ratio.

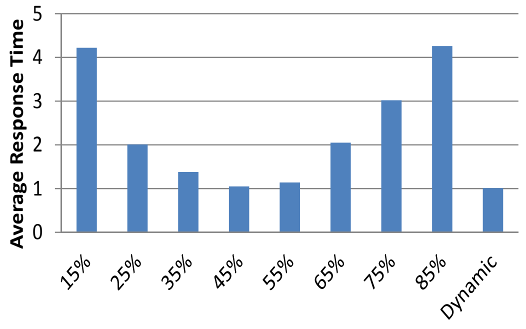
What's more, the results also show that different workloads has different inflection points or optimal static over-provisioning space, for example, the optimal static over-provisions for Financial1, Homes, Exchange, and MSNFS are 85%, 35%, 45%, and 25%, respectively. Therefore, static cache allocation scheme can not always achieve the best system performance. The rightmost bar in the figures shows the performance of our reuse distance aware dynamical cache allocation scheme. The results strongly indicates the effectiveness of our reuse distance aware dynamic cache allocation scheme,



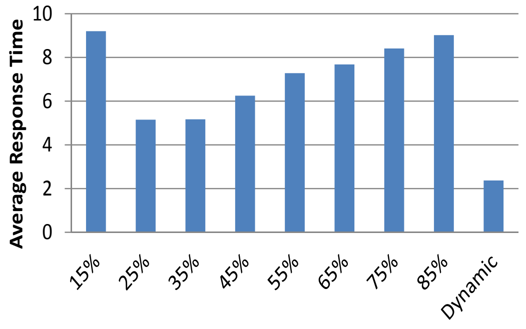
(a) Financial1



(b) Homes

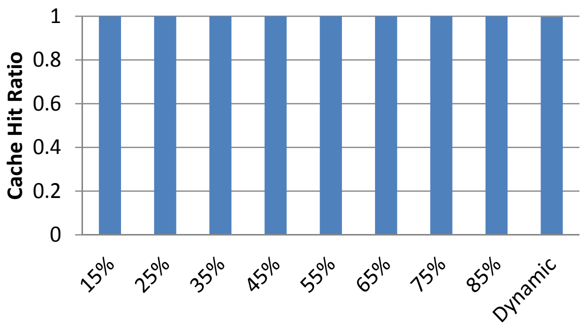


(c) Exchange

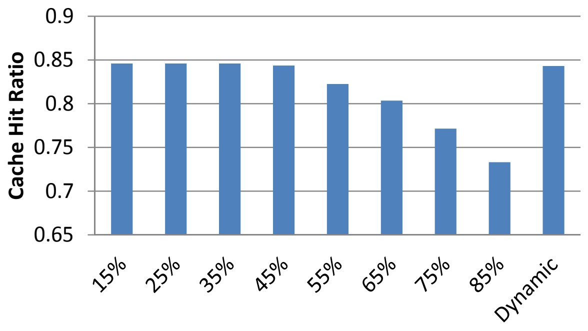


(d) MSNFS

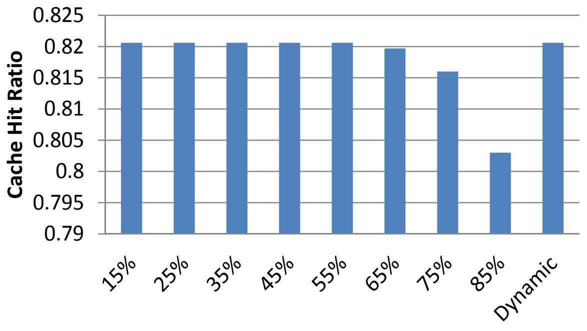
Fig. 3: Average response time of different static over provision configurations and our dynamic allocation scheme.



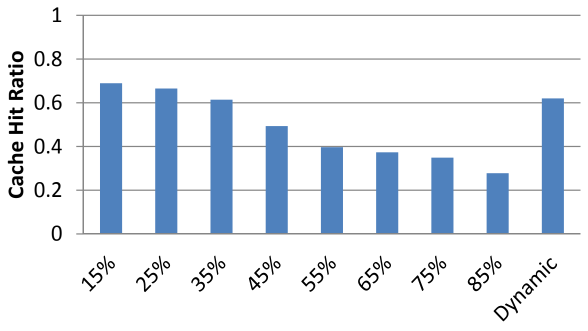
(a) Financial1



(b) Homes

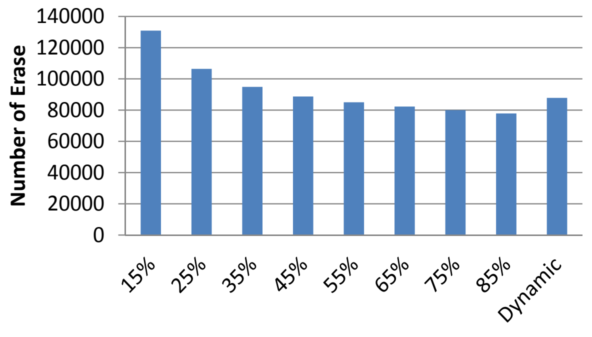


(c) Exchange

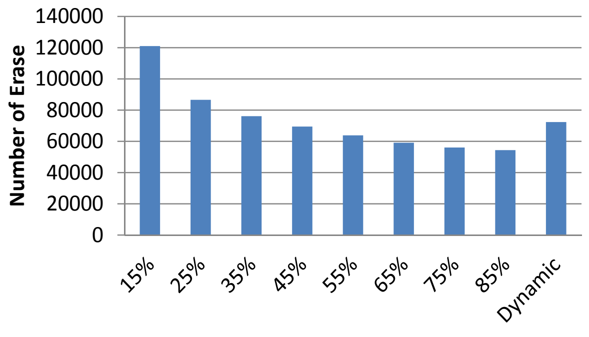


(d) MSNFS

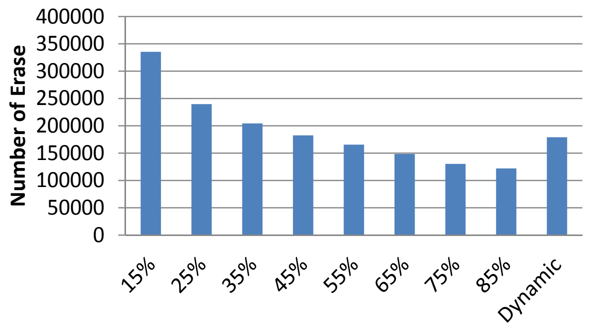
Fig. 4: Cache hit ratios of different static over provision configurations and our dynamic allocation scheme.



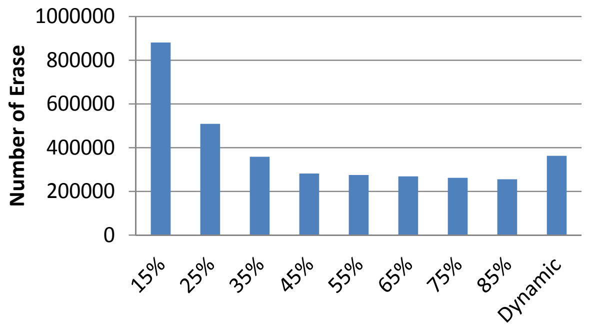
(a) Financial1



(b) Homes

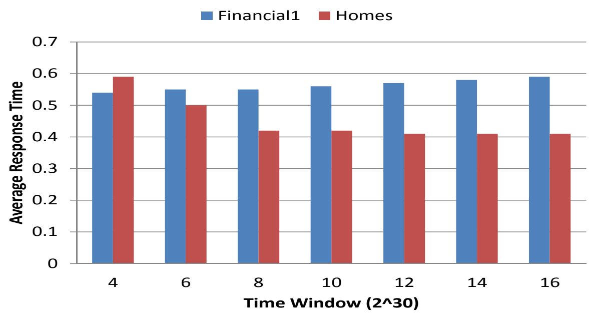


(c) Exchange



(d) MSNFS

Fig. 5: Erase counts of different static over provision configurations and our dynamic allocation scheme.



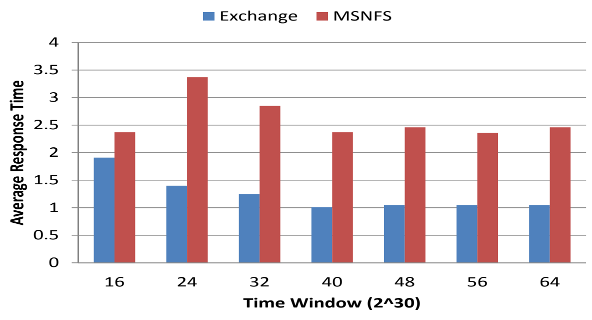


Fig. 6: Effect of different time window sizes on the cache performance.

which can always obtain the system performance close to the static optimal allocation scheme for Financial1, Homes, and Exchange or even better than the static optimal allocation scheme for MSNFS.

B. Endurance

Number of erases is the widely used metric to evaluate the lifetime of SSDs. Figure 5 presents the number of erase for different static cache allocations and our dynamic allocation scheme. For all the four workloads, the number of erase will decrease with the increasing of the over-provisioning space. The reason is straightforward, higher over-provisioning space can delay and reduce the garbage collection activities and improve the garbage collection efficiency. Compared to the typical over-provision configurations that ranges from 7% to 35%, our dynamic cache space allocation scheme not only enhances the system performance, but also reduces the number of erase operations and hence prolongs the lifetime of the cache devices. Besides maximizing the system performance, our dynamic allocation scheme could also be used to maximize the device lifetime without violation of the system performance requirement. For example, if the system performance requirement for running Exchange workload is 2ms, we then can relax the over-provisioning space to 65%, which can prolong the SSD lifetime by nearly 20% compared with the 35% configuration for the optimal performance.

C. Sensitive Analysis

In this section, we discuss how the variations of the time window size T will affect the system performance. On one hand, a small time window might pessimistically estimate the locality of the workloads and can't effectively leverage the cache space to get the optimal system performance. On the other hand, a large time window can bring in too much old information with little value and can't take quick action to keep up with the changing locality of the workloads. Figure 6 shows the system performance with different time window sizes. For small cache with 4G capacity, we change the time window from 4GB to 16GB with 2GB as the step size. While for the large cache with 16GB capacity, the range of the time window sizes is from 16GB to 64GB and the step size is 8GB. For Homes, when the time window increases from 4GB to 8 GB, the system performance are noticeably improved due to the full exploration of the workload locality. However,the system performance are relatively stable or only with limited reduction after 8GB, which means the locality of the same workloads are relatively stable and the effectiveness of utilizing the locality of the workloads in the past to guide the cache space allocation in the future. Similar conclusions could be made for the Exchange and MSNFS. The only exception is the Financial1, when the time window size increases from 4GB to 8GB, there is no big differences of the system performance. After 8GB, increasing the time window can lead to some sacrifice of the system performance. The possible reason is the limited active working set size of Financial1, which makes 4GB is big enough to capture the locality of the workloads.

VI. Related Work

Although cache algorithm and design optimizations are old topics and many advanced cache algorithms have been proposed to improve the cache performance like LIRS [34] and ARC [36]. However, the optimizations of these advanced cache algorithms are based on the traditional cache devices like DRAM and SRAM and use the cache hit ratio as a performance metric, which can not works consistently for SSD-based cache due to the internal garbage collection processes. When SSDs are used as caches in hybrid storage solutions, many optimizations have been proposed to improve the cache performance and lifetime of cache devices. Kgil et al. [29] proposed to take the asymmetrical read and write performance into account by splitting the Flash cache into separate read and write region with dynamically changeable ECC strength and cell density to improve reliability and lifetime of Flash memory. NetApp used Flash memory as a second layer read cache while used the NVRAM as the second layer write cache [40]. Hystor proposed by Chen et al. [30] verified the efficiency to deploy SSD a write buffer for the performance-critical requests. Based on the highly skewness of real-world IO traces, Pritchett et al. proposed a highly-selective caching scheme for SSD cache [31]. An lazy adaptive replacement (LARC) scheme put forward by Huang et al. [33] tries to delay the replacement of the cache entry to reduce the possible cache pollution and improve the cache hit ratios. Since traditional Belady's MIN [39] only considers the cache hit ratio but not the endurance of the cache device, Cheng et al. proposed a new Flash-aware MIN cache algorithms for SSD cache [38]. However, both LARC and Flash-aware MIN are still using the cache hit ratio as the performance metric, which might not obtain the optimal SSD cache performance. In Flash-aware MIN, data that will never result in cache hit or result less than a pre-defined times cache hit will never be inserted into SSD cache to prolong the SSD endurance. Since small random writes can significantly degrade the performance of SSD cache and bring more serve write amplification problem, RIPQ [37] is proposed to aggregate small writes into a large block buffer to improve the SSD cache performance. Huang et al. [35] proposed FlexECC, which selectively replace ECC with EDC to improve the SSD-based cache performance. The kernel idea of FlexECC is that for clean data in SSD cache with backup in the hard disk, EDC will be applied. While for dirty data without backup in the hard disk, ECC will be applied to guarantee the reliability.

Beside, Oh et al. proposed APS to dynamically split the SSD cache space into read, write, and over-provisional regions [32]. However, many ghost LRU caches with different cache sizes are needed to get the cache hit ratios under different cache capacities, which is impractical due to high complexity and memory overhead. Besides, a static equation are applied to translate the OP to U, which also makes their design inaccurate. Xia and Xiao proposed to leverage the out-of-place update property and the over-provisional space to improve the SSD read cache performance [16, 41]. In their work, they also shows that traditional advanced cache algorithms like ARC might can obtain higher cache hit ratio, but may result in worse real cache performance like the average response time.

VII. Conclusion

Unlike traditional cache devices such as DRAM and SRAM, SSDs have internal garbage collection processes, which could significantly degrade the cache performance, especially when used as write cache. Previous optimizations based on traditional cache devices might not obtain consistent performance for SSD cache and even shorten the device lifetime. Therefore, how to compromise the cache hit ratio with the internal garbage collection overhead is of vital importance to obtain the optimal system performance. In this paper, we propose a locality aware dynamic SSD cache space allocation scheme by utilizing the MRC in the past to guide the SSD capacity allocation to achieve the optimal system performance for SSD write cache. The experimental result clearly demonstrate that our locality aware dynamic SSD cache allocation scheme can always achieve the performance close or even better than the optimal system performance with static allocation schemes. Besides, compared with the typical SSD over-provisioning configurations, our dynamic scheme also has the lifetime advantage.

VIII. Acknowledgements

This research is sponsored in part by U.S. National Science Foundation grants CCF-1547804 and CSR-1526190. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. Ousterhout and F. Douglis, "Beating the i/o bottleneck: A case for log-structured file systems," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 1, pp. 11–28, 1989.

[2] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "Borg: Block-reorganization for self-optimizing storage systems." in *FAST*, vol. 9. Citeseer, 2009, pp. 183–196.

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

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

[5] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic modeling of data eviction in cache," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016, pp. 351–364.

[6] "UMass Trace Repository," http://traces.cs.umass.edu.

[7] "SNIA-Block I/O Traces," http://iotta.snia.org/tracetypes/3.

[8] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, 2016.

[9] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches," *ACM Transactions on Storage (TOS)*, vol. 10, no. 4, p. 15, 2014.

[10] A. Ban, "Flash file system," Apr. 4 1995, uS Patent 5,404,485.

[11] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 3, p. 18, 2007.

[12] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "Last: locality-aware sector translation for nand flash memory-based storage systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 36–42, 2008.

[13] W. Bux and I. Iliadis, "Performance of greedy garbage collection in flash-based solid-state drives," *Performance Evaluation*, vol. 67, no. 11, pp. 1172–1186, 2010.

[14] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-defined caching: Managing caches in multi-tenant data centers," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 174–181.

[15] I. A. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska, "sroute: Treating the storage stack like a network." in *FAST*, 2016, pp. 197–212.

[16] Q. Xia and W. Xiao, "Flash-aware high-performance and endurable cache," in *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on*. IEEE, 2015, pp. 47–50.

[17] M. Song and M. Kim, "Solid state disk (ssd) management for reducing disk energy consumption in video servers," in *Proc. of the FAST*, 2011.

[18] D. JEDEC, "Sdram standard," *JESD79-3F, JEDEC SOLID STATE TECHNOLOGY ASSOCIATION*, 2010.

[19] P. Huang, G. Wu, X. He, and W. Xiao, "An aggressive worn-out flash block management scheme to alleviate ssd performance degradation," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 22.

[20] Y. Pan, G. Dong, Q. Wu, and T. Zhang, "Quasi-nonvolatile ssd: Trading flash memory nonvolatility to improve storage system performance for enterprise applications," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012, pp. 1–10.

[21] 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.

[22] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "Lama: Optimized locality-aware memory allocation for key-value cache." in *USENIX Annual Technical Conference*, 2015, pp. 57–69.

[23] F. Olken, "Efficient methods for calculating the success function of fixed-space replacement policies," Lawrence Berkeley Lab., CA (USA), Tech. Rep., 1981.

[24] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "Parda: A fast parallel reuse distance analysis algorithm," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 1284–1294.

[25] Q. Xia and W. Xiao, "Improving mlc flash performance with workload-aware differentiated ecc," in *Parallel and Distributed Systems (ICPADS), 2016 IEEE 22nd International Conference on*. IEEE, 2016, pp. 545–552.

[26] K. Zhao, W. Zhao, H. Sun, T. Zhang, X. Zhang, and N. Zheng, "Ldpc-in-ssd: making advanced error correction codes work effectively in solid state drives." in *FAST*, vol. 13, 2013, pp. 244–256.

[27] H. J. Lee, K. H. Lee, and S. H. Noh, "Augmenting raid with an ssd for energy relief," in *Proceedings of the 2008 conference on Power aware computing and systems*. USENIX Association, 2008, pp. 12–12.

[28] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance." in *USENIX Annual Technical Conference*, 2008, pp. 57–70.

[29] T. Kgil, D. Roberts, and T. Mudge, "Improving nand flash based disk caches," in *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*. IEEE, 2008, pp. 327–338.

[30] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: making the best use of solid state drives in high performance storage systems," in *Proceedings of the international conference on Supercomputing*. ACM, 2011, pp. 22–32.

[31] T. Pritchett and M. Thottethodi, "Sievestore: a highly-selective, ensemble-level disk cache for cost-performance," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 163–174.

[32] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems." in *FAST*, vol. 12, 2012.

[33] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen, "Improving flash-based disk cache with lazy adaptive replacement," *ACM Transactions on Storage (TOS)*, vol. 12, no. 2, p. 8, 2016.

[34] S. Jiang and X. Zhang, "Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1, pp. 31–42, 2002.

[35] P. Huang, P. Subedi, X. He, S. He, and K. Zhou, "Flexecc: Partially relaxing ecc of mlc ssd for better cache performance." in *USENIX Annual Technical Conference*, 2014, pp. 489–500.

[36] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache." in *FAST*, vol. 3, 2003, pp. 115–130.

[37] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li, "Ripq: advanced photo caching on flash for facebook," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. USENIX Association, 2015, pp. 373–386.

[38] Y. Cheng, F. Douglis, P. Shilane, M. Trachtman, G. Wallace, P. Desnoyers, and K. Li, "Erasing beladyâĂŹs limitations: In search of flash cache offline optimality," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 2016, pp. 379–392.

[39] T. E. Anderson, "The performance of spin lock alternatives for shared-money multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, 1990.

[40] M. Woods, "White paper: Optimizing storage performance and cost with intelligent caching," NetApp, Tech. Rep. WP-7107, August 2010.

[41] Q. Xia and W. Xiao, "High-performance and endurable cache management for flash-based read caching," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3518–3531, 2016.

[42] H. Kwon, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Janus-ftl: finding the optimal point on the spectrum between page and block mapping schemes," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2010, pp. 169–178.